Scheduling Containerized Workflow in Multi-cluster Kubernetes

Danyang Liu[1], Yuanqing Xia[1](✉), Chenggang Shan[2], Guan Wang[1], and Yongkang Wang[1]

[1] School of Automation, Beijing Institute of Technology, Beijing 100081, China
{xia_yuanqing,wang_yk}@bit.edu.cn
[2] School of Artificial Intelligence, Zaozhuang University, Zaozhuang 277160, China

Abstract. Docker and Kubernetes have revolutionized the cloud-native technology ecosystem by offering robust solutions for containerization and orchestration workflows. This combination provides unprecedented speed, scalability, and efficiency in deploying and managing applications in distributed environments. However, when scheduling complex workflows across multi-cluster Kubernetes environments, existing workflow scheduling systems often fail to provide the necessary support. Integrating workflow scheduling algorithms with multi-cluster scheduling algorithms poses a complex and challenging problem. In this paper, we present a comprehensive framework known as the Containerized Workflow Engine (CWE), specifically designed for multi-cluster Kubernetes deployments. The CWE framework employs a two-level scheduling scheme, which combines the benefits of workflow containerization and establishes seamless connections between multi-cluster scheduling algorithms and multi-cluster Kubernetes environments. By integrating workflow scheduling algorithms with Kubernetes schedulers across Kubernetes environments, the CWE framework enables efficient utilization of resources and improved overall workflow performance. Compared to the state-of-the-art Argo workflows, CWE performs better in average task pod execution time and resource utilization.

Keywords: Workflow · Scheduling · Containerized

1 Introduction

Cloud infrastructure is continually evolving due to advancements in cloud-native technologies, hardware capabilities, networking enhancements, and the adoption of industry standards [21]. Cloud-native technologies, including containers, microservices, DevOps, Kubernetes [9], and other transformative practices such as serverless computing, infrastructure as code, and CI/CD, have revolutionized IT operations, maintenance, and development. Docker [5] and Kubernetes have emerged as prominent tools for cloud resource management, playing a significant role in the cloud-native technology ecosystem [13]. However, alternative solutions

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023
E. Chen et al. (Eds.): BigData 2023, CCIS 2005, pp. 149–163, 2023.
https://doi.org/10.1007/978-981-99-8979-9_12

and platforms are available, such as Podma [3] and Apache Mesos [10], catering to specific use cases and requirements.

Kubernetes is an open-source container orchestration and management platform for automating containerized applications' deployment, scaling, and management. It provides a highly scalable, reliable, and user-friendly way to build, deploy, and manage applications across multiple hosts. Kubernetes defines the desired state of an application and automates tasks such as container creation, replication, restarts, and scaling, ensuring that the application runs consistently with the defined state. It also offers powerful service discovery and load balancing capabilities, supports horizontal auto-scaling, and enables developers and operations teams to efficiently build and manage modern distributed applications.

Single-cluster workflow scheduling [12] may face the risk of a single point of failure. For example, when the cluster experiences a failure, the workflow may be interrupted or halted. Furthermore, single-cluster workflow scheduling has limited resource utilization and cannot effectively handle load fluctuations. Multi-cluster workflow scheduling [22] offers better elasticity, scalability, and high availability than single-cluster workflow scheduling. Multi-cluster workflow scheduling can automatically adjust the cluster size based on workflow demands, ensuring elastic allocation and scalability of resources. Additionally, multi-cluster workflow scheduling can achieve resource isolation, meet geographical distribution and data locality requirements, and enhance the performance and efficiency of workflows.

Argo workflows (Argo) [2] is a powerful open-source workflow orchestration engine that focuses on managing and orchestrating containerized workloads in Kubernetes environments. The engine provides rich workflow orchestration capabilities, a visual interface, and close integration with Kubernetes. However, the current Argo scheduler has a significant drawback: it cannot schedule workflows across multi-cluster Kubernetes. This means that when it comes to orchestrating and scheduling workflows across multi-cluster, users need to implement cross-cluster scheduling logic themselves. This problem poses a challenging issue for containerized workflow [16,20] scheduling on Kubernetes and urgently requires an efficient framework closely integrated with Kubernetes to address this problem.

In this paper, we design a Containerized Workflow Engine, referred to as CWE, based on the further development of CWB [19]. The main objective of this system is to support workflow scheduling across multi-cluster Kubernetes and improve the execution efficiency of workflows by implementing a two-level scheduling scheme and containerized execution on Kubernetes. The CWE system consists of two components: Containerized Workflow Controller (CWC) and Containerized Workflow Scheduler (CWS). The CWC component can be deployed on high-performance hosts, and its primary function is to receive workflows and send them to the CWS components in multi-cluster. CWC implements a dual-channel mechanism, where the fast channel is used for quick workflow forwarding, while the slow channel ensures accurate routing of workflows to the schedulers. Additionally, CWC implements load balancing across clusters and within clus-

ters. Specifically, when CWC receives an actual task, it decides which cluster to send the new task to based on the current workload of each cluster. CWC communicates with CWS to understand the load situation of CWS within the cluster, enabling it to decide whether to send the task to an existing CWS or start a new one. This design effectively addresses the pressure of large-scale cloud workflow tasks on the system. It avoids the predicament of a single CWS facing a sharp decline in performance or even failure to function properly due to massive requests. The CWS component is deployed in each Kubernetes cluster. To ensure smooth execution of workflow scheduling, CWS internally employs advanced workflow scheduling algorithms and utilizes the informer component to monitor Kubernetes resources. It also uses the Clint-go package to implement task container creation functionality. CWS utilizes the Goroutine mechanism to create concurrent task containers after the current task is completed for cases with multiple parallel successor tasks in a workflow. Furthermore, CWS handles data dependencies between task containers using the dynamic volume-sharing feature of StorageClass. Experimental results demonstrate that our CWE system exhibits better performance in terms of workflow execution efficiency. Compared to state-of-the-art technologies, CWE achieves a 31.61% improvement in enhancing workflow execution efficiency. Our contributions are summarized as follows:

- Design a framework for effectively managing containerized workflows within a Kubernetes environment. This framework incorporates a two-level scheduling scheme, allowing workflow management across multi-cluster Kubernetes.
- Implement a workflow injection module, CWC, and CWS. The workflow injection module is designed to handle the task injection process into the CWC during experiments. The primary role of the CWC is to transmit workflow tasks to the CWS within the Kubernetes cluster, taking into account the resource status of the cluster. The CWS is responsible for efficiently scheduling workflows within the Kubernetes environment.
- Provides a case study of containerized workflow in simulated production practice and presents a detailed performance analysis of CWE compared to other workflow scheduling solutions.

We have open-sourced the CWE. The source code is publicly available on GitHub at [4]

2 Related Work

As the standard container orchestration tool in the cloud-native era, Kubernetes provides rich and comprehensive support for developing the container application ecosystem. Its emergence offers powerful functionalities for the development and deployment of cloud-native applications and drives the continuous evolution of workflow engines. Airflow [1] is a platform to programmatically author, schedule, and monitor workflows. Airflow provides a user-friendly interface for defining, scheduling, and monitoring workflows as directed acyclic graph (DAG), offering features like task dependencies, error handling, and extensibility. Nextflow [7] is

a bioinformatics workflow manager that enables the development of portable and reproducible workflows. Nextflow simplifies the creation and execution of scalable scientific workflows, supporting large-scale data and computational workloads with its DSL and containerization capabilities. Argo is an open-source container-native workflow engine hosted by Cloud Native Computing Foundation (CNCF). Argo enables the deployment and management of containerized applications in Kubernetes clusters, allowing users to define complex workflows as code with features such as templating and event-driven execution. Volcano [11], born in Huawei Cloud Native, is CNCF's first batch computing project. Volcano optimizes scheduling and resource management for batch and AI workloads on Kubernetes clusters, improving resource utilization and job performance through intelligent resource allocation and prioritization. These platforms empower organizations to streamline and automate their data processing and workflow management tasks, enhancing productivity and scalability.

Containerized workflow scheduling remains a relatively emerging research field. Existing technologies and tools, such as Airflow, Nextflow, Argo, and Volcano, primarily focus on workflow scheduling within a single Kubernetes cluster. However, their support for multi-cluster Kubernetes is not yet comprehensive enough. Consequently, a framework is needed to operate efficiently in a multi-cluster Kubernetes environment. Furthermore, Airflow and Nextflow were not originally intended as native workflow systems for Kubernetes, while Volcano was primarily focused on batch tasks. Presently, Argo is a cloud-native workflow engine specifically designed for Kubernetes. Therefore, in this paper, the experimental evaluation will primarily compare the submission method using Argo.

3 Design

This section provides a detailed explanation of the scientific workflow definition and the two-level scheduling scheme. We present the architectural design of the CWE and subsequently introduce the CWC and the CWS.

3.1 Scientific Workflow

In large-scale data processing tasks, the workflow [14,15] is typically described using a DAG to represent a distributed system application comprehensively. The relationships between tasks can be likened to edges in a DAG graph [23]. In addition to shared files, dependencies between tasks may involve data transfer, message queues, API calls, and other means. Container technologies [18] such as Docker provide a lightweight virtualization solution to encapsulate the execution environment and required resources for workflow tasks. The advantages of container technology include isolation, portability, and repeatability, utilizing container images as static snapshots of containers. In Kubernetes, a Pod is the smallest scheduling unit, serving as a logical deployable entity consisting one or more related containers, providing a shared network and storage environment.

The Kubernetes scheduler determines suitable nodes within the cluster to schedule Pods based on resource requirements, affinity rules, and scheduling policies.

3.2 Two-Level Scheduling Scheme

CWE and Kubernetes combine to implement a two-level scheduling scheme, as shown in Fig. 1. CWE serves as the interface that connects the workflow injector module and Kubernetes. Through the CWC module, the workflow scheduling algorithm is used to make scheduling decisions for workflows and distribute them to the CWS modules of different Kubernetes clusters. The CWS module is responsible for executing workflow tasks containerized to fully utilize cluster resources and improve the execution speed of workflows. The CWS module uses workflow scheduling algorithms to manage cluster resources efficiently, ensuring tasks are scheduled and executed based on task dependencies and resource requirements.

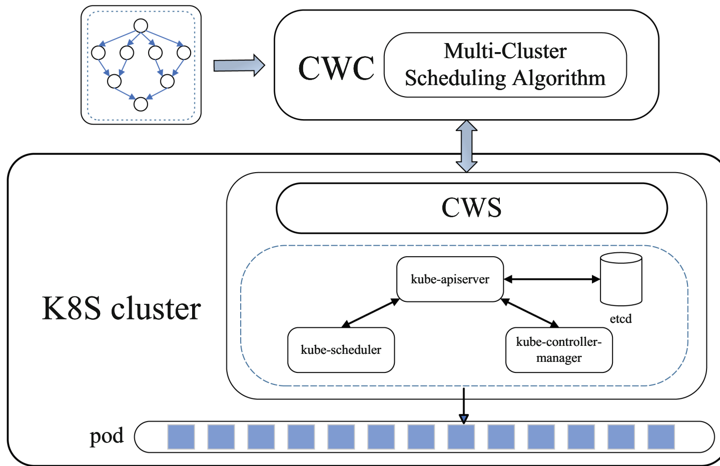

Fig. 1. Two-level scheduling scheme. The two-level scheduling scheme refers to Multi-cluster scheduling of CWC and workflow scheduling of CWS.

3.3 CWC Architecture

As shown in Fig. 2, CWC includes distributor module, pre-selector module, pressure evaluator module and state tracker module. Algorithm 1 shows the details. The function INITIALIZECLUSTER is responsible for the initialization of a cluster, involving the allocation of available resources and the computation of its initial score. To begin, the available resources of the cluster are gathered (line 1). Subsequently, the initial score of the cluster is calculated (lines 2 and 3). This initial score, along with the available resources, is then appended to the registry

table (line 4) for future reference. In the subsequent function, TASKSCHEDULING, the process of task allocation is orchestrated by systematically examining each cluster entry in the registry table. For each cluster under consideration, the algorithm first assesses whether the task's requirements can be accommodated by the cluster's available resources (lines 10 and 11). Following this, a predictive score for the cluster is computed (lines 11 and 12), which aids in the determination of its suitability for the given task. Ultimately, the task is assigned to the cluster that best matches with its needs.

Algorithm 1. Scheduling Algorithm

1: **function** INITIALIZECLUSTER(Cluster n, Available resources c_n, m_n, b_n, Total resources C_n, M_n, B_n)
2: **if** Cluster n is new **then**
3: $P_n = \alpha \frac{c_n}{C_n} + \beta \frac{m_n}{M_n} + \gamma \frac{b_n}{B_n}$ ▷Calculate initial cluster score
4: Add cluster score and available resources to the registry table R
5: **end if**
6: **end function**
7:
8: **function** TASKSCHEDULING(Registry table R, Task t, Task requirements c_t, m_t, b_t)
9: **for** each cluster n in R **do**
10: **if** Task requirements can be met by cluster n **then**
11: $S_t = \alpha \frac{c_t}{C_n} + \beta \frac{m_t}{M_n} + \gamma \frac{b_t}{B_n}$ ▷Calculate task score
12: $P_n = P_n - S_t$ ▷Update cluster score
13: Assign task t to the cluster n
14: **end if**
15: **end for**
16: **end function**

a) Distributor Module: Responsible for sending workflows to CWS in multi-cluster Kubernetes. Through the distributor module, workflow tasks are intelligently allocated to different Kubernetes, resulting in optimized resource utilization and streamlined task execution. The module implements flexible resource allocation and load-balancing strategies, continuously adapting to demand variations. These capabilities enable higher concurrency, enhance system scalability, and improve overall performance.

b) Pre-Selector Module: Responsible for establishing a pre-selection table in advance, utilizing the current Kubernetes resource data obtained from CWS. This table provides precise information to the controller allocator, ensuring facilitating quick routing turnover and accurate workflow routing to the optimal scheduler. After scheduling workflows by the allocator module, the pre-selection table is updated in real-time according to the resource evaluation algorithm to correct each cluster's scoring and CWS load values. By predictive real-time update mechanism enables higher accuracy in pre-selection tables, improving the efficiency and reliability of workflow scheduling for CWC, resulting in improved performance.

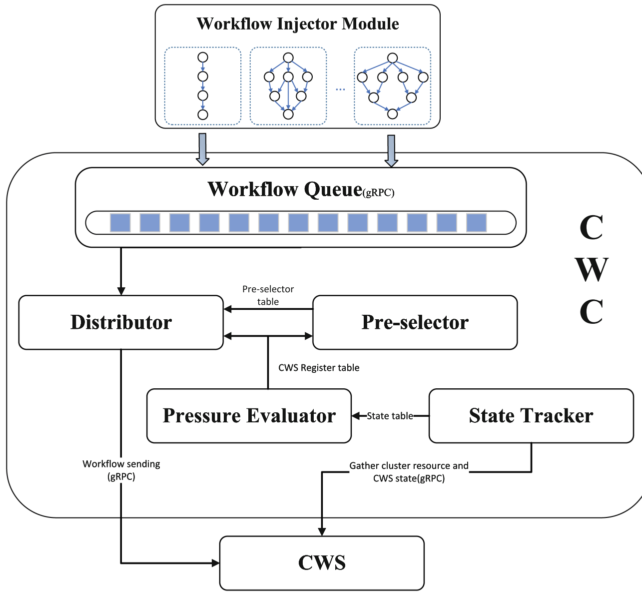

Fig. 2. Architecture of the CWC.

c) Pressure Evaluator Module: Responsible for continuously monitoring and evaluating the workload pressure of each CWS and Kubernetes cluster within the system and creating a corresponding workload registry table. Analyzing real-time and historical data calculates workload pressure scores, which indicate the level of resource utilization for each Kubernetes cluster. This information is used to optimize workload balancing and routing of workflows to ensure efficient task execution, maximize resource utilization, and enhance system performance. The module collaborates with the Pre-Selector and Distributor modules to analyze workload pressure data and make informed decisions regarding workflow distribution.
d) State Tracker module: Responsible for real-time monitoring and management of workflow and CWS statuses. It tracks the progress of workflows, ensuring their successful execution and handling failed workflows by rescheduling them. Furthermore, the module continuously monitors CWS to detect potential issues and updates the state table accordingly. Through active tracking of workflow execution, efficient management of failures, and maintenance of an accurate state table, this module significantly enhances the reliability and effectiveness of the CWS.

3.4 CWS Architecture

As shown in Fig. 3, CWS includes the scheduler, resource allocator, and tracker modules.

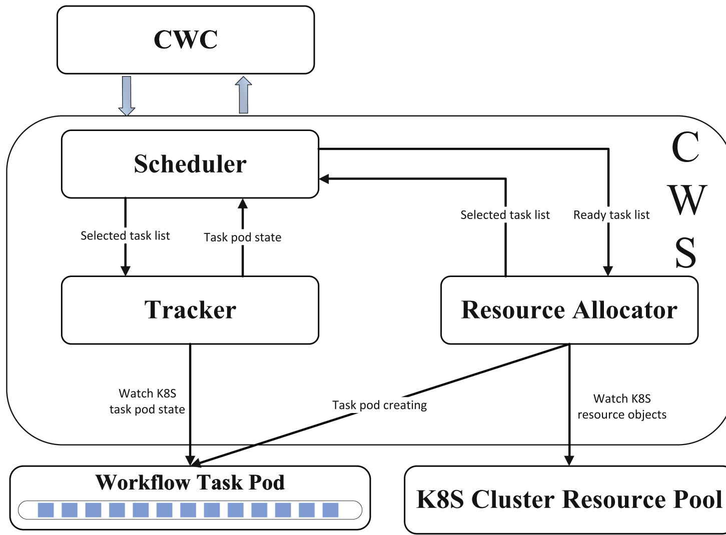

Fig. 3. Architecture of the CWS.

a) Scheduler module: Responsible for implementing critical algorithms for cloud workflow scheduling. The main objective was to efficiently allocate and manage workflow tasks submitted by users, ensuring optimal resource allocation and meeting personalized requirements. The Scheduler module analyzed task dependencies and resource demands, allocating them effectively among available Kubernetes cluster resources to achieve optimal execution efficiency and resource utilization. This module considered factors like task priority, data transmission between tasks, and resource utilization to formulate appropriate scheduling strategies.

b) Resource allocator module: Responsible for implementing the resource allocation functionality for workflow tasks. Its main functions include containerizing workflow tasks, monitoring Kubernetes resources using the informer component, creating task containers with the Clint-go package, creating concurrent task containers using the Goroutine mechanism after the current task is completed, handling data dependencies between task containers through dynamic volume-sharing using StorageClass, caching resource information locally to reduce API access pressure, and generating namespaces for achieving isolated environments for workflow resources. By effectively allocating and managing workflow tasks, optimizing resource utilization, meeting personalized requirements, and enhancing execution efficiency, this module ultimately improves the overall workflow performance.

c) Task Tracker module: Responsible for monitoring the execution status of cloud workflow task containers and providing real-time feedback to the scheduler to support the orderly execution of task containers. It detects the health of containers, collects and stores container log information, records the

execution time of task containers, provides task progress updates, cleans up containers, and releases resources promptly after completing tasks.

3.5 Workflow Injection Module

The workflow injection module is an independent auxiliary module that operates separately from the CWE. Its primary functions include generating workflows, handling input requests from subsequent workflows, and transmitting workflow information to the CWC via gRPC. This module establishes the overall structure of workflow tasks and utilizes the Json method to inject configuration files containing task dependencies into the respective containers.

4 Experimental Evaluation

This section will evaluate the proposed CWE using various evaluation metrics and discuss its benefits compared to Argo.

4.1 Experimental Setup

To assess the performance of CWE, we have designed the workflow injection module. This module is containerized for deployment with CWC and CWE. Effective communication between these modules is facilitated through the gRPC mechanism.

The Kubernetes cluster used in our experiments consists of one master node and five worker nodes. Each node equips with a 6-core AMD EPYC 7742 2.2 GHz CPU and 8 GB of RAM, running Ubuntu 20.04 and Kubernetes v1.19.6 and Docker version 18.09.6 and Argo v3.2.9. CWC and workflow injector module are deployed on a high-performance virtual machine, and CWS is containerized and deployed into the Kubernetes cluster through Service and Deployment. In order to evaluate the performance of CWE across multi-cluster Kubernetes, we utilized a total of nine Kubernetes clusters. Due to Argo's lack of support for multi-cluster scheduling, we established a separate Kubernetes cluster comprising three master nodes and forty-five worker nodes for Argo.

4.2 Workflow Example

In order to validate the application scalability of CWE, we have tailored a customized workflow that encompasses all the node-dependent characteristics of the DAG diagram, accommodating more intricate scenarios. The workflow task program employs resource loads to simulate workflow tasks in real-world production practice.

a) Workflow Topology: We utilize a DAG diagram to represent the workflow, constructing an experimental example encompassing all the typical characteristics of such a diagram. As shown in Fig. 4, this workflow comprises seven

tasks featuring branching, crossover, and merging elements. Based on the interdependencies among task nodes, the scheduling algorithm employed for this workflow follows a top-down approach, ensuring tasks are scheduled topologically.

*b) Workflow Containerization:*Taking inspiration from [17], we adopt a Python application as a workflow task and utilize the Stress tool [6] to emulate CPU and memory usage within a defined timeframe. To facilitate this, we employ the Docker engine to package the Python application into a task Image file. This task Image file is subsequently stored in either a local Harbor [8] or a remote Docker Hub repository, and its image address is initialized within the workflow injection module. Furthermore, container parameters can be imported into the task container, specifying CPU cycles, memory allocation, and duration, all of which contribute to determining the runtime of the task pod. The task involves CPU forking and memory allocation operations, executed over 15 s. Within the JSON file, we specify the task pod's resource requests and resource limit parameters as 1000 milli cores for CPU and 512Mi for memory. It is worth noting that the requests and limits fields share the same parameter values.

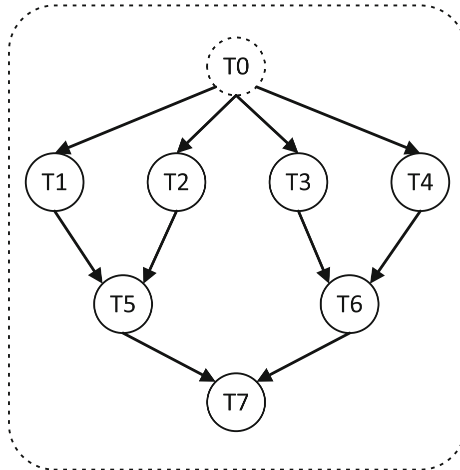

Fig. 4. Workflow topology diagram.

4.3 Results and Analysis

In order to evaluate the effectiveness of CWE, our first step is to verify the workflow execution efficiency on multi-cluster Kubernetes using CWE. Subsequently, we will compare CWE and Argo, focusing on workflow execution efficiency and CPU usage rate. We will now describe the two methods for workflow submission.

- CWE: We employ the containerized method to deploy CWE. CWC and workflow injector module are deployed on a high-performance virtual machine, while CWS is deployed within each Kubernetes cluster. We use a JSON file describing a DAG to represent the workflow task dependency relationship. After going through the workflow injector module, compress the JSON file using Snappy and submit it to CWC via gRPC.
- Argo: We deployed the Argo Workflow image in the Kubernetes cluster using the official YAML file provided by Argo. Similar to the CWE, we employ the same JSON file and leverage a workflow injection module to convert it into a YAML format that Argo can recognize and then submit to Argo.

a) Workflow task execution efficiency: We package Docker images for CWC, CWS, and workflow injection modules. We define YAML files for RBAC, StorageClass, and CWS. CWC and workflow injection modules are containerized and deployed on high-performance virtual machines with a Docker engine. The YAML files are deployed in the Kubernetes cluster, where CWS are scattered and scheduled to the cluster nodes as pods. The components communicate with each other using gRPC.

As is shown in Fig. 5, The execution time for each group of workflows has been averaged across ten experiments. The execution time of the CWE workflow task was determined by subtracting the start time of the workflow injector from the *successedWorkflows* metric. Similarly, the execution time of the Argo workflow task was determined by subtracting the start time of the workflow injector from the *Successfully* metric found in the log of the Argo workflow-controller pod. The CWE takes 133.3 s to receive 100 workflow tasks from the workflows injection module until the workflows pod is execution completed, 350.1 s to execution completed 500 workflow tasks, and 825.8 s to execution completed 1000 workflows tasks. The Argo takes 121.2 s to receive 100 workflow tasks from the workflows injection module until the workflows pod is execution completed, 431.1 s to execution completed 500 workflow tasks and, 1086.89 s to execution completed 1000 workflows tasks. During the initial execution of 100 workflows, CWE and Argo exhibited similar execution times, indicating sufficient resources within the Kubernetes cluster. However, as the workload increased to 500 workflows, CWE experienced a 23.13% decrease in execution time compared to Argo. This disparity can be attributed to inadequate resources within the Kubernetes cluster. When executing 1000 workflows, Argo generated workflow task pods exclusively within its namespace, resulting in a significant accumulation of pods. This accumulation ultimately led to the restart of the Argo workflow-controller pod. Consequently, CWE experienced a 31.61% decrease in execution time compared to Argo.

In addition, upon comparing the execution of 500 workflows with that of 1000 workflows, it becomes evident that CWE outperforms in handling larger-scale workflows. It validates that the CWE is a framework for a large-scale workflow scheduling tool for multi-cluster Kubernetes.

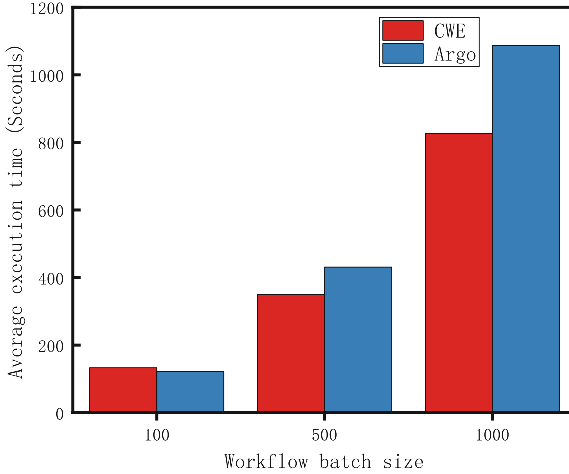

Fig. 5. Average execution time of workflow.

b) Resource Usage Comparison: This section aims to utilize Prometheus to capture the state changes of underlying resources in a Kubernetes cluster under different numbers of workflows to showcase the CPU utilization characteristics of two engines, CWE and Argo. To ensure accurate performance comparisons for CPU usage, it is crucial to address the substantial impact caused by frequent resource fluctuations. In order to mitigate this influence, we meticulously configure our experimental environment to eliminate any additional workloads that could affect performance measurements. To enhance the stability and resource allocation efficiency of the Kubernetes cluster, the Master node is intentionally excluded from participating in pod scheduling and workload, focusing solely on its core administrative tasks.

Figure 6, Fig. 7 show the CPU usage rate of the CWE and Argo over the lifecycle of 100 and 500 workflows. When executing 100 workflows in two Kubernetes clusters with the same number of nodes, the CPU utilization curves of the two workflow engines are similar. However, the CPU utilization curve shows significant fluctuations when executing 500 workflows using the Argo engine. This could be due to the fact that when the Argo workflow engine executes a large number of workflows simultaneously, all the workflow pods are launched in the same namespace, resulting in a drastic drop in system performance and the inability to schedule workflows properly. Regardless of the type of Kubernetes cluster mode, the available number of CPU cores is 270000 milli. Under no load conditions, the CPU utilization of the Kubernetes cluster components is 0.7%. After injecting workflows, the CWS is launched and requires 2 CPU cores. Therefore, after completing workflow execution, the CPU utilization of CWE will remain at 7%. Due to the prolonged time required by Argo to clean up completed pods, which marks the end of workflow execution, there has been a significant performance degradation issue

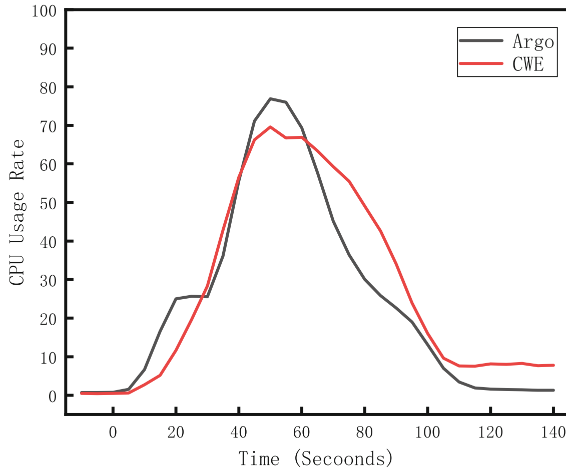

Fig. 6. CPU Usage Rate for 100 Workflows.

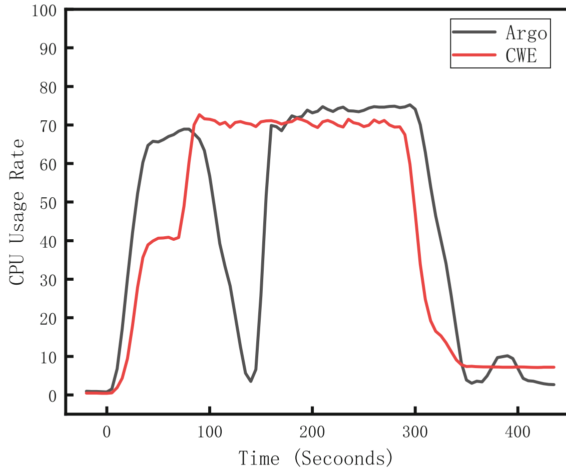

Fig. 7. CPU Usage Rate for 500 Workflows.

when a large number of workflow injections occur. We designed CWE that assigns a separate namespace for each workflow task, enabling resource isolation and more efficient cleanup of completed pods. In a multi-cluster Kubernetes environment, our CWE has better performance.

5 Conclusion

In this paper, our CWE has successfully achieved efficient workflow task scheduling across multi-cluster Kubernetes. CWE offers comprehensive workflow man-

agement functionalities, including task definition, dependency management, and execution sequencing. It also employs intelligent distributed scheduling strategies to allocate tasks to different clusters based on resource availability and workload conditions, thus enhancing the overall system efficiency and performance.

Our experimental results demonstrate a significant improvement in the workflow scheduling throughput of CWE compared to the state-of-the-art single-cluster workflow scheduling engine, Argo, with an approximate increase of around 31.61% in multi-cluster Kubernetes scenarios. This indicates the superior scheduling performance and scalability of CWE in multi-cluster environments.

In conclusion, as a multi-cluster workflow scheduling engine, CWE holds promising prospects for a wide range of applications. By providing flexible workflow management functionalities and intelligent distributed scheduling strategies, CWE significantly improves the efficiency and performance of workflow task scheduling in multi-cluster environments. Future research can focus on further refining the scheduling algorithms, optimizing resource management strategies, and expanding the capabilities of CWE to cater to the growing demands of containerized workflow tasks.

References

1. Apache airflow (2023). https://airflow.apache.org/
2. Argo-workflows - github (2023). https://github.com/argoproj/argo-workflows
3. The best free and open source container tools (2023). https://podman.io/
4. CWE - github (2023). https://github.com/liudy093/CWE
5. Develop faster. Run anywhere (2023). https://www.docker.com/
6. Linux man page (2023). https://linux.die.net/man/1/stress
7. Nextflow (2023). https://www.nextflow.io/
8. Our mission is to be the trusted cloud native repository for kubernetes (2023). https://goharbor.io/
9. Production-grade container orchestration (2023). https://kubernetes.io/
10. Program against your datacenter like it's a single pool of resources (2023). https://mesos.apache.org/
11. Volcano - github (2023). https://github.com/volcano-sh/volcano
12. Adhikari, M., Amgoth, T., Srirama, S.N.: A survey on scheduling strategies for workflows in cloud environment and emerging trends. ACM Comput. Surv. (CSUR) **52**(4), 1–36 (2019)
13. Bernstein, D.: Containers and cloud: from LXC to docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
14. Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.H., Vahi, K.: Characterization of scientific workflows. In: 2008 Third Workshop on Workflows in Support of Large-Scale Science, pp. 1–10. IEEE (2008)
15. Deelman, E., Gannon, D., Shields, M., Taylor, I.: Workflows and e-science: an overview of workflow system features and capabilities. Futur. Gener. Comput. Syst. **25**(5), 528–540 (2009)
16. Hobson, T., Yildiz, O., Nicolae, B., Huang, J., Peterka, T.: Shared-memory communication for containerized workflows. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 123–132. IEEE (2021)

17. Klop, I.: Containerized workflow scheduling (2018)
18. Pahl, C.: Containerization and the PaaS cloud. IEEE Cloud Comput. **2**(3), 24–31 (2015)
19. Shan, C., Wang, G., Xia, Y., Zhan, Y., Zhang, J.: Containerized workflow builder for Kubernetes. In: 2021 IEEE 23rd International Conference on High Performance Computing & Communications; 7th International Conference on Data Science & Systems; 19th International Conference on Smart City; 7th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), pp. 685–692. IEEE (2021)
20. Shan, C., Xia, Y., Zhan, Y., Zhang, J.: KubeAdaptor: a docking framework for workflow containerization on Kubernetes. Futur. Gener. Comput. Syst. **148**, 584–599 (2023)
21. Varghese, B., Buyya, R.: Next generation cloud computing: new trends and research directions. Futur. Gener. Comput. Syst. **79**, 849–861 (2018)
22. Wang, Y.R., Huang, K.C., Wang, F.J.: Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments. Futur. Gener. Comput. Syst. **60**, 35–47 (2016)
23. Zheng, C., Tovar, B., Thain, D.: Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 130–139. IEEE (2017)