sensors

Article

FASDQ: Fault-Tolerant Adaptive Scheduling with Dynamic QoS-Awareness in Edge Containers for Delay-Sensitive Tasks

Ruifeng Wang, Ningjiang Chen *, Xuyi Yao and Liangqing Hu

School of Computer and Electronic Information, Guangxi University, Nanning 530004, China; 1813302007@st.gxu.edu.cn (R.W.); 1913301047@st.gxu.edu.cn (X.Y.); hulq@gxu.edu.cn (L.H.)
* Correspondence: chnj@gxu.edu.cn

Abstract: As the requirement for real-time data analysis increases, edge computing is being implemented to leverage the resources of edge devices to reduce system response times and decrease the latency. However, due to the resource constraints of edge clouds, edge servers are more prone to failures than other systems. Therefore, guaranteeing the reliability of services in edge clouds is critical. In this paper, we propose a fault-tolerant adaptive scheduling mechanism with dynamic quality of service (QoS) awareness (FASDQ), which extends the primary/backup (PB) model by applying QoS on demand to task copies. The aim of the method is to reduce the latency and achieve reliable service for tasks by changing the execution time of task copies. This paper also proposes a container resource-adaptive adjustment mechanism, which adjusts the timing of resources when the available resources cannot meet the task copy requirements. Finally, this paper reports the results of simulation experiments on the EdgeCloudSim platform to evaluate the difference in performance between FASDQ and other methods. The results show that the mechanism effectively reduces the execution time of task copies and outperforms other methods in terms of reliability and general resource utilization.

Keywords: edge container; fault-tolerant scheduling; quality of service; resource management

check for updates

Citation: Wang, R.; Chen, N.; Yao, X.; Hu, L. FASDQ: Fault-Tolerant Adaptive Scheduling with Dynamic QoS-Awareness in Edge Containers for Delay-Sensitive Tasks. *Sensors* **2021**, *21*, 2973. https://doi.org/10.3390/s21092973

Academic Editor: Marco Picone

Received: 14 March 2021
Accepted: 14 April 2021
Published: 23 April 2021

1. Introduction

The goal of edge computing [1] is to obtain real-time responses and provide storage and computation at the edge of the network. This process extends cloud resources to the edge of the network and reduces the network access latency. Although edge computing has many advantages, due to the increased functionality and real-time requirements of smart devices, the chances of resource failures in edge computing are not low, which can reduce the availability of resources and affect the response latency. In smart car networking, smart manufacturing, driverless and other smart devices, and latency-sensitive applications, delays due to reliability issues must be taken seriously. Few previous works have deeply studied the fault tolerance of edge clouds, and tasks are often resubmitted to available resources after they fail, which may lead to unfulfilled service level agreements (SLA) due to the delays encountered by requests [2]. Fault-tolerant scheduling is also widely used as an effective way to assure reliability. However, the current literature on fault-tolerant task scheduling does not sufficiently consider the varying requirements of different tasks for quality of service (QoS) so that edge resources can be allocated as needed. Therefore, the ability to guarantee the reliable completion of latency-sensitive tasks within time constraints and the effective utilization of edge resources is a key issue.

Due to the long startup time of traditional virtual machines and their tendency to waste resources, many systems and studies have used container technology to build the underlying technology stack of edge computing platforms based on the reliability and bandwidth limitations of edge networks [3]. For example, Google launched "the Anthos for Telecom" [4], which will bring Anthos to the edge network by packaging its applications into containers that can run at the edge, providing an immersive experience to customers.

Flynn [5] and Deis [6] are also container-based PaaS (platform as a service) projects that can automatically build and deploy any application to a Docker container cluster. Edge Cloud Kubernetes Engine (ECK) [7], launched by Tencent Cloud, and "ACK Edge Kubernetes" [8], launched by AliCloud, ensure that the node's services are still available in the case of disconnection. Compared with cloud containers [9], edge containers have the advantages of low latency and scalability and are suitable for real-time or latency-sensitive tasks. Limited resources and low-latency services require more in-depth optimization in the application of task scheduling and resource management in edge containers.

It is critical to guarantee the correctness and real-time performance of results in the required timeframe for latency-sensitive tasks. Fault tolerance can be achieved by scheduling multiple copies of a task on different processors. The primary/backup (PB) model [10] and the triple-mode redundancy (TMR) approach [11] are two basic methods in which all copies of a task must be scheduled and generate correct results before its deadline, even if there are errors. In TMR, multiple copies are usually run, and error checking is implemented by comparing the results after completion. With the PB approach, a backup copy is activated if the result of the primary copy is incorrect. The PB model has the advantage of low hardware resource requirements, but it cannot protect against faults [10].

We use the traditional PB model for edge failures and build an edge computing system based on the characteristics of containers. Unlike the traditional cloud environment, in the edge container context, the effective coordination of efficiency and fault tolerance is a challenge. The main contributions of this paper are the following:

1. For latency-sensitive tasks, we propose a dynamic QoS-aware fault-tolerant task scheduling mechanism, which extends the traditional PB model by introducing different reliability requirements for tasks; i.e., when scheduling task copies, their QoS levels are dynamically adjusted according to their reliability requirements and latency constraints.
2. After the primary succeeds, the resources and time interval occupied by the backup are released. In order to make full use of the released resources, the start time of task copies that are waiting in the queue will be reselected to improve resource utilization at the container level.
3. To reduce the resource pressure exerted by the PB model at the edge, we propose a container resource-adaptive adjustment mechanism for the resource quota of each edge container, which replenishes resources according to their usage by the assigned task copies, as well as the latency constraints and reliable service requirements of unexecuted task copies.

The rest of this paper is organized as follows: Section 2 briefly analyzes the problem and related work, and Section 3 provides a systematic analysis. Section 4 describes the FASDQ solution, the primary/backup scheduling algorithm, and the container resource-adaptive adjustment algorithm. Section 5 reports and analyzes the experimental results, and conclusions are drawn at the end of the paper.

2. Related Work

Among fault-tolerant approaches, the PB model has been extended to remote clouds in several studies, in which each task has a primary copy and backup copy, and fault-tolerant scheduling is performed in two different computing instances [12]. Primary copies are used when tasks are executed normally, and backup copies are executed when the primary copies fail in order to ensure that the task requests are completed within the specified deadline [13]: this is known as a passive backup scheme [14]. However, the passive backup scheme requires the task to have a sufficient time delay to start the backup copy, which is unrealistic for delay-sensitive tasks. In contrast, an active backup scheme [15] can complete tasks with less slack. The authors of [16] proposed a backup overlay in which task copies are overlapped in the same time slot of the same computation instance, and a release technique, which refers to releasing the resources of the corresponding backup copy after the primary copy succeeds. In [17], two backup schemes were analyzed, and the fault-tolerant elastic scheduling algorithm (FESTAL) was designed for real-time tasks in a virtual cloud environment. This method balances fault tolerance and high resource utilization in the cloud.

Various approaches have been proposed and studied to deal with fault tolerance in cloud computing systems. In edge networks, unlike cloud computing environments, the occurrence of failures may cause a large portion of edge-side resources to go offline, resulting in delays and affecting the overall QoS level. To address the failure of edge servers, redundancy-based replication techniques have also been studied for fault tolerance in edge computing. The research work in [18] provided fault tolerance for the Internet of Things (IoT) by modeling the failure behavior of the Internet grid storage area network (SAN) to provide fault tolerance through redundancy to reduce system downtime. In [19], the PB model was extended to the edge cloud, and a fault-tolerant based QoS-aware scheduling algorithm (FTBQA) was proposed to improve the reliability of edge cloud application services. DeCoRAM middleware [20] tolerates processor failures by considering the execution time and failover order of primary and backup copies in order to meet the soft real-time requirements of each task. To achieve fault tolerance, the authors of [21] proposed a reliable and fault-tolerant agent-based hierarchical IoT–cloud architecture by replicating data at the edge of the network, which redirects the backup server when the service fails to apply. The system in [22] adaptively replicates data according to application requirements for data loss and latency. The replication operation is deferred to some extent, and the data loss and latency requirements are satisfied. In [23], the same input sequence was enforced for virtual machines (VM) replicated across hosts, enabling fast multicore scalable VM fault tolerance. NiLiCon [24] provides seamless failover from a failed container to a backup container of another host, providing fault tolerance to protected applications and external clients. Replication-based fault-tolerant scheduling techniques have been extensively studied in edge cloud environments, but they can also place a burden on resource-limited edge servers to provide timely service to task copies.

Optimizing resource utilization on a fault-tolerant basis is a challenge due to the cost of coordinating reliability safeguards, especially since replication-based fault-tolerance techniques suffer from relatively high resource consumption. Currently, edge computing is still in its infancy in terms of resource management. The authors of [25] developed a computing resource management algorithm for multi-user mobile edge computing (MEC) systems. This algorithm minimizes the average power consumption of mobile devices and MEC servers. Another study [26] proposed a node resource reallocation mechanism based on the characteristics of containers; this method calculates the resource quota for each task in the subsequent phases of task execution and reallocates CPU (central processing unit) resources in the node, making it possible to increase the number of tasks processed in a certain time cycle and effectively reducing the task latency. To some extent, these resource management techniques allocate resources to tasks and effectively improve system resource utilization, but none of them involve fault tolerance. Thus, they cannot support fault tolerance when optimizing resource management.

In summary, for resource-limited edge computing, it is crucial to realize the fault-tolerance capability of the system while effectively utilizing the limited resources at the edge. We find that existing research lacks studies in this area. In this paper, we propose a fault-tolerant adaptive scheduling mechanism with dynamic QoS awareness, which extends the PB model by determining the QoS level for task copies according to reliable service and latency requirements. In addition, during scheduling, FASDQ adaptively adjusts resources according to the resource usage to meet the demands of task copies. This mechanism achieves fault tolerance, accelerates the execution time of tasks, and improves the efficiency of resource use.

3. Problem Analysis and System Model

3.1. Problem Analysis

Traditional cloud-based industrial Internet of Things (IIoT) architectures struggle to meet requirements such as low latency and high reliability. For example, traditional telecom networks are unable to meet industry-specific requirements for reliable, predictable, and efficient communications. With low latency, scalability, and reduced bandwidth, edge containers are suited for serving real-time or latency-sensitive applications. In 2019, Edgegap, a disruptive gaming startup, released a gaming platform that reduced latency by 58% using real-time telemetry and edge containers [27]. However, edge containers are deployed close to users, their resources are constrained by the environment, and they have higher uncertainty than traditional cloud containers, so reliability assurance for edge containers is critical.

Fault-tolerant scheduling is an effective method to guarantee reliability and is based on replication techniques. The PB model is widely used for fault-tolerant scheduling. In order to reduce the extra resource consumption of backups, the authors of [16] studied a backup overlapping technique in which backups are able to overlap in the same time slots of the same compute instances. However, since the actual compute instances of the edge containers are containers, it is only the primary copies of tasks that cannot be scheduled in the same edge node, whereas their backup copies can overlap in the same container. Another typical technique of the PB model is the reclaiming of resources for backup copies when corresponding primary copies succeed [10]. If the time slot can be utilized after the backup is released, it greatly reduces the latency.

In addition, to meet the strict QoS requirements imposed by latency-sensitive tasks, the method in this paper improves the reliability of task copies by dynamically assigning them QoS levels when task scheduling is performed. Regardless of whether the fault-tolerant technique is based on replication or the assignment of QoS levels to tasks to guarantee reliability, we find that: (1) while processing task requests, task copies are always gathered in containers with higher processing capability, resulting in an uneven distribution of idle and busy containers (in Figure 1, containers C_{12} and C_{21} have more task copies, while container C_{31} is idle); and (2) the existing resources cannot meet the demand of some task copies. Most previous studies have focused on migrating containers as a whole, which has some effect on resource tuning in edge nodes, but they have ignored the effective utilization of resources within individual containers.

Therefore, our method makes more adaptive adjustments to resources at the container and node levels separately. When the resources of a task's backup are released, the copies waiting in that container are rescheduled so that the container can handle more task copies; if the current resources cannot meet the task demand, nearby edge nodes that are in the shutdown state will be assessed to determine whether they can be turned on. Furthermore, if some resources are idle for a long time during system operation, the resource adjustment mechanism will eliminate the idle resources and thus improve resource utilization.

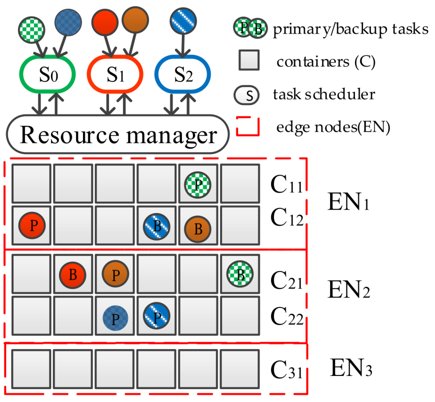


Figure 1. Problem analysis in the process of task copy scheduling.

In the method described in this paper, fault-tolerant adaptive scheduling is performed for latency-sensitive tasks in the edge container, extending the PB model by determining QoS levels for the reliability assurance of task copies. Then, the on-demand adjustment of edge container resources occurs according to resource usage during task scheduling. The approach proposed in this paper not only achieves a tradeoff between the real-time requirement and reliability of tasks, but also avoids reducing resource utilization due to an unbalanced load on edge container resources.

3.2. System Model

In Figure 2, edge users can access the edge cloud through wireless networks and submit requests to it. Each edge server includes two modules: the Task Scheduler and the Resource Manager. The Task Scheduler determines whether the edge container can satisfy the requested latency constraint based on the arrival order of requests, and if the request is accepted, the Task Scheduler transmits it to the Resource Manager, which receives tasks and allocates resources for task requests.

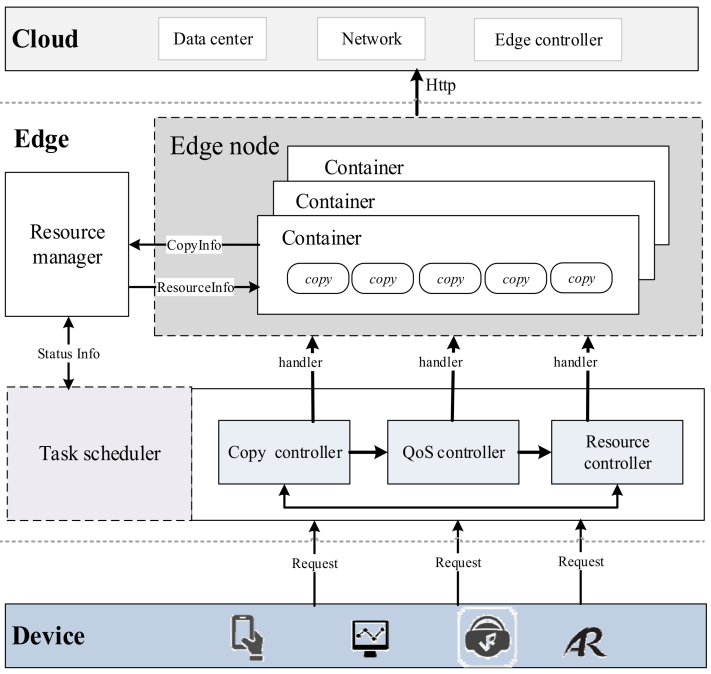


Figure 2. Task scheduling model, where QoS is Quality of Service.

Before introducing relevant concepts, we first define the parameters and symbols that are used in this paper, as shown in Table 1.

Relevant concepts and definitions are given below.

There is a set $EN = \{EN_1, EN_2, \cdots, EN_n\}$ of edge nodes around devices, where each edge node has fixed computing resources, including CPU, RAM (random access memory), bandwidth, etc. The processing capability P_i (MIPS, million instructions per second) is used to determine the type of computing resources of the edge nodes. Each node EN_i includes multiple containers denoted by a set of containers $c_i = \{c_{i1}, c_{i2}, \cdots, c_{im}\}$ with different processing capability P_{ij}.

We use the set $T = \{t_1, t_2, \cdots, t_n\}$ of latency-sensitive tasks in end devices. Each task has three attributes, $t_i = \{a_i, d_i, l_i\}$: arrival time a_i, deadline d_i, and task size l_i, which is measured in millions of instructions (MI).

Each task is replicated into two copies: t_i^P and t_i^B, which are available in different nodes so that the backup can take over execution in another node if the primary fails.

Table 1. Parameters and symbols.

Parameters and Symbols	Description
EN_i, c_{ij}	Edge node i and container j of edge node i.
t_i^*	Primary copy t_i^P and backup copy t_i^B.
$a_i,\ d_i$	Arrival time and deadline of t_i and their units are second.
l_i	Task size (unit: MI, millions of instructions).
s_i^P, f_i^P	Start time and finish time of t_i^P
s_i^B, f_i^B	Start time and finish time of t_i^B
est_i^P, lst_i^B	The earliest start time of t_i^P and the latest start time of t_i^B
$state\left(t_i^B\right)$	State of t_i^B
$q\left(t_i^*\right)$	QoS level of t_i^*
$eq_{kl}\left(t_i^*\right)$	t_i^P
$eq_{kl}\left(t_i^*\right)$	The execution time of t_i^P and tiB and the unit is second.
P_i, P_l^{rem}	The processing capability and the remaining processing capability of EN_i(unit: MIPS).
P_{ij}, P_{kl}^{rem}	The processing capability and the remaining processing capability of c_{ij} (unit: MIPS).
$P_{kl}\left(q\left(t_i^*\right)\right)$	The processing capability provided to t_i^* with t_i^* of c_{kl} (unit: MIPS).
η_{ij}^*	Indicates whether t_i^* succeeds in EN_j: if it succeeds, $\eta_{ij}^* = 1$; otherwise, $\eta_{ij}^* = 0$
AQL	Average QoS level denoting the reliable demand of t_i^*.
z_i^*	Indicates whether t_i^* is assigned to the container successfully.

Definition 1. *Backup states $state\left(t_i^B\right)$: the backup is divided into two standby states according to its start time and the completion time of its primary (active (represented by 1) and passive (represented by 0)), as shown in Equation (1).*

$$state\left(t_i^B\right) = \begin{cases} 1 & f_i^P > s_i^B \\ 0 & otherwise \end{cases} \tag{1}$$

If the completion time of the primary copy is later than the start time of its backup copy, the backup state is set to 1; otherwise, it is set to 0. Active backups do not stop until the primary is completed, while passive backups start only after the primary fails. As shown in Figure 3, t_1^B is passive, but t_2^B is active, and the part that is redundant with the primary is denoted by t_2^{BR}.

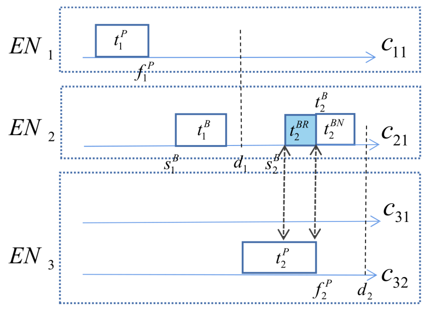


Figure 3. Time point of task copies.

In container c_{kl}, the execution time $e_{kl}\left(t_i^*\right)$ of t_i^* (* denotes two copies of tasks: e.g., t_i^* represents t_i^P and t_i^B) is the ratio of its task size over the processing capability of this container [28,29]:

$$e_{kl}\left(t_i^*\right) = \frac{l_i}{P_{kl}} \tag{2}$$

In this paper, Equation (2) is improved to reduce task latency (see Equations (3) and (5)). The execution time of task copies is changed by assigning different QoS levels that provide different processing capability to different task copies on demand, so the primary copy can be completed as early as possible within the time constraint. In the container c_{kl}, the QoS level of t_i^* is denoted by $q(t_i^*)$; $P_{kl}(q(t_i^*))$ is the processing capability of t_i^* within $q(t_i^*)$. The two copies of a task can be assigned different QoS levels to improve task schedulability. Their execution times and constraint conditions are described below.

Definition 2. *The execution time of t_i^P. This is defined as the ratio of the task copy size over the processing capability of a container with the QoS level $q(t_i^P)$, as shown in Equation (3).*

$$eq_{kl}\left(t_i^P\right) = \frac{l_i}{P_{kl}\left(q\left(t_i^P\right)\right)} \tag{3}$$

In order to meet the delay requirements of primary copies, the following constraints should also be met:

$$\forall t_i \in T, eq_{kl}\left(t_i^P\right) \le d_i - est_i^P \tag{4}$$

Definition 3. *The execution time of t_i^B. According to the backup state of tasks, the execution time of the backup copy depends on whether its primary t_i^P succeeds. Assuming that the backup t_i^B is scheduled in the container c_{st} (not in the edge node of t_i^P), then:*

$$eq_{st}\left(t_i^B\right) = \begin{cases} 0, & \eta_{ij}^P = 1, state\left(t_i^B\right) = 0 \\ f_i^P - s_i^B, & \eta_{ij}^P = 1, state\left(t_i^B\right) = 1 \\ \frac{l_i}{P_{st}\left(q\left(t_i^B\right)\right)}, & \eta_{ij}^P = 0 \end{cases} \tag{5}$$

where η_{ij}^* denotes whether task copies succeed on EN_j. If so, $\eta_{ij}^* = 1$; otherwise, $\eta_{ij}^* = 0$. Equation (5) is explained as follows:

1. If t_i^P succeeds and $state\left(t_i^B\right) = 0$, then t_i^B is not executed;
2. If t_i^P succeeds and $state\left(t_i^B\right) = 1$, then the execution time of t_i^B is $\left[s_i^B, f_i^P\right]$;
3. If t_i^P fails, then the execution time of t_i^B is $eq_{st}\left(t_i^B\right) = \frac{l_i}{P_{st}\left(q\left(t_i^B\right)\right)}$.

In order to meet the delay requirements of tasks and reduce the time redundancy between the active backup and its primary copy, the following conditions should also be met:

$$\forall t_i \in T, eq_{st}\left(t_i^B\right) \le d_i - lst_i^B \tag{6}$$

When a task copy is successfully assigned to an edge container, the remaining processing capability of the container should be calculated first.

Definition 4. *Remaining processing capability. This is defined as the number of instructions processed by the remaining resources of the edge container per second, and the unit is MIPS. Therefore, the remaining processing capability of c_{kl} and EN_l are shown in Equations (7) and (8).*

$$P_{kl}^{rem} = P_{kl} - \sum_{i=1}^{N} P_{kl}(q(t_i)) \tag{7}$$

$$P_l^{rem} = P_l - \sum_{k=1}^{m} \sum_{i=1}^{N} P_{kl}(q(t_i)) \tag{8}$$

Maximizing the average QoS level of task copies in the edge container under fault tolerance is one of the scheduling objectives of this paper.

Definition 5. *The average QoS level. This is obtained as the ratio of all QoS levels of all successful tasks to the number of all successful tasks* [30]*, denoted by AQL. If t_i^* is successfully assigned to a container, then $z_i^* = 1$; otherwise, $z_i^* = 0$.*

$$AQL = max\frac{\sum_{j=1}^{m}\sum_{i=1}^{n} z_i^P q\left(t_i^P\right)\eta_{ij}^P + \sum_{j=1}^{m}\sum_{i=1}^{n} z_i^B q\left(t_i^B\right)\eta_{ij}^B}{\sum_{j=1}^{m}\sum_{i=1}^{n}\left(z_i^P\eta_{ij}^P + z_i^B\eta_{ij}^B\right)} \quad (9)$$

The main goal of task scheduling in this paper is to make the edge container accommodate more task copies while adopting the PB model for fault tolerance. In the next section, we provide the solution according to the system model.

4. Solution: FASDQ Framework and Key Algorithms

We designed the FASDQ framework shown in Figure 4, which consists of two main parts: the primary/backup scheduling mechanism and the resource-adaptive adjustment mechanism. When a task request is received at the edge, the task is first replicated into two copies by the Copy Controller, and the task copies are assigned different QoS levels by the QoS Controller to change their execution times according to different reliability quality requirements to determine the optimal scheduling slot for the copies. The task copies in the queue are rescheduled by the Resource Controller after the resources of the backups are released so that the primary copies of the tasks can start as early as possible.

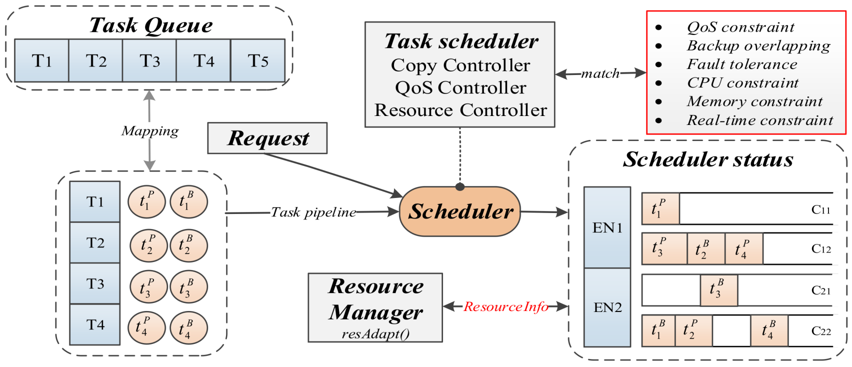


Figure 4. The framework of FASDQ (Fault-tolerant Adaptive Scheduling mechanism with Dynamic QoS-awareness).

In addition to dynamic QoS-aware task copy scheduling, the FASDQ framework also includes a container resource-adaptive adjustment mechanism, which adaptively replenishes resources for tasks with insufficient processing capability during task execution to improve resource utilization in the edge node. When task copies cannot be scheduled at the edge, the tasks are handed over to the cloud for processing.

The following introduces the dynamic QoS-aware primary/backup scheduling mechanism and the container resource-adaptive adjustment mechanism at the edge.

4.1. Dynamic QoS-Aware Primary/Backup Scheduling Mechanism

In this paper, we consider a case in which the traditional PB model application is extended at the edge container side. FASDQ uses a backup overlapping technique to reduce redundancy in the task scheduling process and determines the schedulable interval of tasks in the edge container with dynamic QoS-awareness to find the best time interval for scheduling. This process is represented by the scheduling function.

According to the analysis in [17], when the primary copies of multiple tasks are not scheduled in the same nodes, their backups in the same virtual machine can overlap. In this paper, we also apply this rule to our scheduler mechanism. In order to ensure that no conflicts occur during the execution of task copies and achieve fault tolerance, we allow backup copies to overlap in time and space when scheduling two copies of a task. Task copies in the same container are subject to the following constraints: (1) the primary copy cannot overlap with other copies; (2) the redundant part of the active backup copy cannot overlap with other copies; (3) for the passive backup copy, if $EN\left(t_i^P\right) = EN\left(t_j^P\right)$, then their backups cannot overlap.

The goal of the function scheduler $schedulable\left(t_i^P, c_{jk}\right)$ is to designate schedulable time slots for t_i^P in c_{jk}. The schedulable time slots need to be chosen so that t_i^P starts as early as possible before its deadline in $[a_i, d_i]$; the unschedulable time slots are those that are occupied by other primary/backup copies. Therefore, we traverse the schedulable time slot in $[a_i, d_i]$ from left to right to find the earliest start time $eq_{jk}\left(t_i^P\right)$ of t_i^P on c_{jk} that satisfies Equation (4); otherwise, return $+\infty$.

The goal of the function scheduler $schedulable\left(t_i^B, c_{st}\right)$ is to designate schedulable time slots for t_i^B in c_{st}. If t_i^P is scheduled successfully, then the scheduling of t_i^B will start. The schedulable time slots need to be chosen so that t_i^B starts as late as possible within its deadline in $[a_i, d_i]$, and this slot is not part of $EN\left(t_i^P\right)$; the unschedulable time slots of t_i^B are the time slots occupied by its primary copies, the time slot occupied by the backup copy before f_i^P, and the redundant parts of active backup copies. As shown in Figure 5, t_1^B and t_5^B are passive backups, t_2^B and t_3^B are active backups, and t_4^P is the primary copy. In other words, the unschedulable time slots of t_i^B in c_{21} are $\left[s_1^B, f_1^B\right]$, $\left[s_2^B, f_i^P\right]$, $\left[s_3^B, f_3^P\right]$, and $\left[s_4^B, f_4^P\right]$. Therefore, the schedulable time slot in $[a_i, d_i]$ is traversed from right to left to find the latest start time $eq_{lk}\left(t_i^B\right)$ of t_i^B in c_{lk} that satisfies Equation (6); otherwise, return $-\infty$.

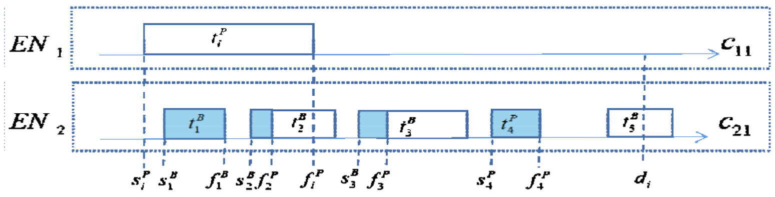


Figure 5. Unschedulable time slot for backup copies.

Algorithm 1 describes the process of dynamic QoS-aware task scheduling for the primary copies in order to assign QoS levels to primary copies on demand to guarantee reliability. The main function of Algorithm 1 is arranging the processing capability of the edge nodes in descending order so that the primary copies are first assigned to the edge container with a higher processing capacity (lines 1–3). Then, it finds the earliest start time of t_i^P by $schedulable\left(t_i^P, c_{jk}\right)$ and a container satisfied time constraint in Equation (4); if t_i^P is successfully scheduled, the label flag is set to be 1, otherwise, it is set to 0 (lines 5–14); if failed, the QoS level will be degraded by 1 until it becomes q_1, and the above operation will be repeated (line 15). If it is still not found, then the function $resAdapt()$ is called (see Algorithm 3) (lines 17–23).

The time complexity of Algorithm 1 depends on the number of tasks K. Suppose initially that there are L active nodes with N containers of each node in the neighborhood of the user, and let the QoS level of the task copies be a random integer in (1, m). m and L are negligible in comparison with N and K. Thus, the time complexity of $schedulable\left(t_i^P, c_{jk}\right)$ is O(NK). Then, Algorithm 1 comes to the function $resAdapt()$. We will analyze its time complexity in Section 4.2.

If t_i^P is successfully scheduled, the scheduling of t_i^B starts. Algorithm 2 describes the process of dynamic QoS-aware task scheduling for the backup. The main function of Algorithm 2 is sorting the edge nodes in descending order of remaining processing capability, except for the nodes in which its primary copy is located (lines 1–3). Then, the latest start time of t_i^B that satisfies the time constraint in Equation (6) is found by $schedulable\left(t_i^B, c_{lk}\right)$ in a container and the backup states are marked. If a container is successfully selected, the label flag is set to 1, otherwise, it is set to 0 (lines 5–18). If failed, the QoS level will be degraded by 1 until it becomes q_1, and the above operation will be repeated (line 19); if it is still not found, then the function $resAdapt()$ is called (see Algorithm 3) (lines 22–28).

Algorithm 1. Primary Scheduling in FASDQ

```
Input: The set of tasks: T; QoS level: $Q = \{q_1, q_2, \cdots, q_m\}$; the set of ENs:
$EN = \{EN_1, EN_2, \cdots EN_n\}$.
Output: $\{c(t_i^P)\}$ // The container that can be scheduled for $t_i^P$ with the earliest start time.
Begin:
1    for each new task $t_i \in T$, do
2        $flag = 0; est^P = +\infty; c(t_i^P) = null; q(t_i^P) = q_m$;
3        Sort $ENs$ in descending order of processing capability;
4        while $q(t_i^P) \neq q_1$, do
         // The QoS level of $t_i^P$ is initialized to the maximum value within time constraints.
5            for each $c_{jk} \in EN_j$, do
6                Set the processing capability $P_{jk}(q(t_i^P))$;
7                $eq_{kl}(t_i^P) = \frac{l_i}{P_{kl}(q(t_i^P))}$;
8                $est_i^P = schedulable(t_i^P, c_{jk})$;
9                  if $est_i^P \leq est^P$ and $eq_{jk}(t_i^P) \leq d_i - est_i^P$, then
10               //Select the container with the earliest completion time of the primary.
11                     $c(t_i^P) = c_{jk}; est^P = est_i^P; flag = 1$;
12                     Calculate the remaining processing capability of $c_{jk}$ and $EN_j$;
13             end if
14           end for
15           if $flag == 0$, then $q_t = q_{m-1}$; // The QoS level is degraded.
16       end while
17       if $flag == 0$, then
         //The existing container resources cannot meet the scheduling requirements.
18           $c_{new} = resAdapt(t_i^P)$;
19           if $c_{new} \neq null$, then
20             $c(t_i^P) = c_{new}$; // Create a new container.
21             return $c(t_i^P)$;
22             end if
23       else Reject $t_i$;
24   end for
END
```

Assuming that the number of the successfully scheduled primary is X, the time complexity of $schedulable(t_i^B, c_{lk})$, in the worst case, is O(NX). Then, Algorithm 2 comes to the function $resAdapt()$. We will analyze its time complexity in Section 4.2.

When the primary copy of the task succeeds and the corresponding backup copy is released, the vacant time slot can be used by other copies in the same container. If the primary can start earlier and the backup can start later within the time constraint, the redundant part of the active backup can be reduced, and the time performance of the primary copy that has not yet been executed is improved. Therefore, a new start time is chosen for the other copies in that edge container. The selection process is shown in Figure 6, where σ_i^B is defined as the time required to release the backup.

Figure 6a shows the arrangement of each task copy in the container before t_1^p succeeds. Figure 6b: After t_1^p succeeds and t_1^B is released, then $schedulable(t_3^P, c_{21})$ determines whether t_3^p can be scheduled in $[f_1^P - \sigma_1^B, s_2^B]$. If so, the new earliest start time $nest_3^P$ of t_3^P is returned, and $state(t_3^B)$ is changed. In this example, $state(t_3^B)$ is changed from 1 to 0, reducing energy consumption. Figure 6c: $schedulable(t_2^B, c_{21})$ determines whether t_2^B can be scheduled in $[s_2^B, f_3^P]$, and then the new latest start time $nlst_3^p$ of t_2^B is updated; this can also change state(t_2^B) from 1 to 0, which can effectively reduce the redundancy.

After the primary succeeds, the resources and time interval occupied by the backup are released. The rescheduling of task copies can result in the earlier completion of the primary copy within the time constraints, and the backup can start later, which effectively reduces the task delay.

Algorithm 2. Backup Scheduling in FASDQ

Input: The set of tasks T_p whose primary copy is successfully assigned to a container, $\{c(t_i^P)\}$; $Q = \{q_1, q_2, \cdots, q_m\}$; the neighboring edge nodes of task t_i: $EN(t_i) \subseteq EN$.
Output: $\{c(t_i^B)\}$ //After the primary is successfully scheduled in a container, output the container for which the backup is scheduled.
Begin:

```
1  for each backup task $t_i^B \in T_p$, do
2      $flag = 0$; $lst^B = 0$; $c(t_i^B) = null$; $q(t_i^B) = q_m$;
3      Sort $\{EN - EN(t_i^P)\}$ in descending order of the remaining processing capability;
4      while $q(t_i^B) \neq q_1$, do //The QoS level of $t_i^B$ is initialized to the maximum value within
       time constraints.
5          for each $EN_l \subseteq EN - EN(t_i^P)$, do
6              for each $c_{lk} \in EN_l$, do
7                  Set the processing capability $P_{lk}(q(t_i^B))$;
8                  Calculate $eq_{lk}(t_i^B)$ using Equation (5);
9                  $lst_i^B = schedulable(t_i^B, c_{lk})$;
10                   if $lst_i^B \geq lst^B$ and $eq_{st}(t_i^B) \leq d_i - lst_i^B$, then
11                 //Select the container for the backup copy with the latest start time.
12                     $c(t_i^B) = c_{lk}$; $lst^B = lst_i^B$; $flag = 1$;
13                     Calculate the remaining processing capability of $c_{lk}$ and $EN_l$;
14                   if $f_i^P \leq lst_i^B$, then $state(t_i^B) = 0$; //Mark the backup state.
15                   else $state(t_i^B) = 1$;
16                 end if
17             end for
18         end for
19       if $flag == 0$, then $q_t = q_{m-1}$; // The QoS level is degraded.
20     end while
21     if $flag == 0$, then $c_{new} = resAdapt(t_i^P)$;
       //The existing container resources cannot meet the scheduling requirements.
22       if $c_{new} \neq null$, then
23         $c(t_i^B) = c_{new}$; // Create a new container.
24         return $c(t_i^B)$;
25       end if
26     else reject $t_i$;
27 end for
END
```

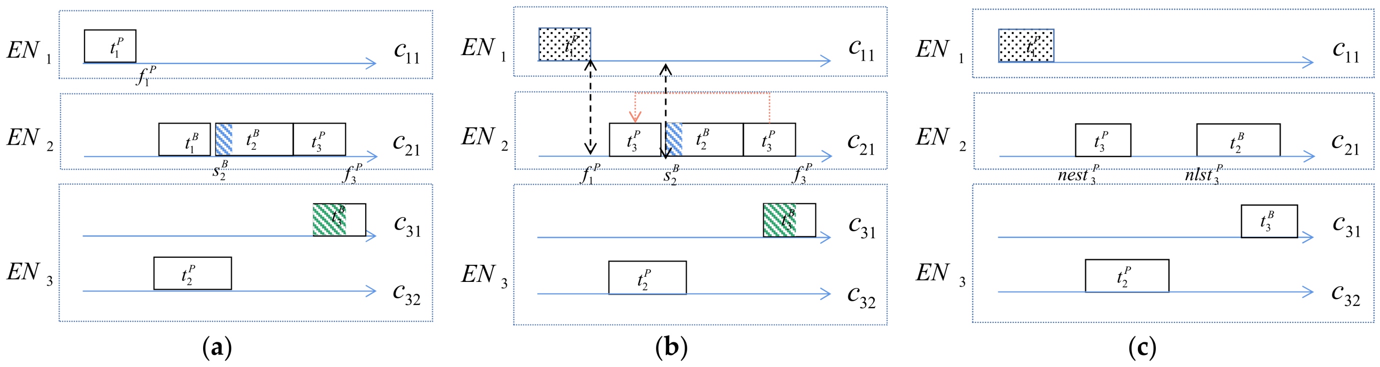


Figure 6. Task rescheduling process, is finished; (**b**) t_1^B is released and t_3^p is performed in advance; (**c**) state(t_2^B) is changed from active to passive.

4.2. Container Resource-Adaptive Adjustment Mechanism

In the above fault-tolerant scheduling process, the copy rescheduling mechanism eases the resource utilization of one container. In this subsection, we present the optimization of resource utilization across edge nodes, which means that edge containers can accept as many tasks as possible. However, if the edge container accepts multiple tasks over a period of time, which may lead to insufficient available resources for some tasks, then it will not be able to meet the latency constraint of task requests. To address the above situation, this paper proposes an elastic adjustment mechanism for container resources: i.e., when the current computational resources cannot meet the demand of task copies, the mechanism can: (1) add containers dynamically, (2) balance their workload with other containers, and (3) migrate containers to another edge node with sufficient processing capability. This process is shown by $resAdapt()$.

Migration constraints. To ensure fault tolerance when migrating containers across edge nodes, the following constraints need to be satisfied during container migration:

(1) $c\left(t_i^B\right)$ cannot be migrated to the edge node of t_i^P, nor can $c\left(t_i^P\right)$ be migrated to the same edge node of t_i^B. Otherwise, two copies of the task are in the same edge node and it cannot be fault-tolerant.

(2) If there is overlap between t_i^B and t_j^B, $c\left(t_i^P\right)$ cannot be migrated to $EN\left(t_j^P\right)$.

The newly created container has a different processing capability and should meet conditions in terms of time:

$$a_i + t(c_{new}) + \theta_{new} + eq_{new}(t_i^*) \le d_i \tag{10}$$

where $t(c_{new})$ denotes the creation time of the new container, and θ_{new} is the ready time of the task in the new container.

resAdapt$()$ is a container resource-adaptive adjustment algorithm, as shown in Algorithm 3.

1. The set $\{EN(t_i)\}$ of running edge nodes near the user is sorted in ascending order of remaining processing capability p_i^{rem}, and these nodes are traversed in turn and judged to be able to create new containers (processing capacity is denoted by $p(c_{new})$) to accommodate the task copy (lines 1–7). If there is no suitable running node, some containers are migrated to a nearby running node to make space to create a new container (lines 8–11).
2. If the creation of containers in the running node still cannot meet the scheduling demand of task copies, an unpowered node near the task copy (processing capacity is indicated by $p(EN_{new})$) starts up, and a new suitable container in this node is created (lines 13–18).

We now evaluate the time complexity of Algorithm 3. Firstly, it takes O(L) to create a new container to an active node, and it takes O(L) to migrate a container to another destination node. Hence, the time complexity of adding a new container by migration (see lines 8–12) is O(L^2). For creating a new container to EN_{new}, the time complexity is O(1). Therefore, the complexity of $resAdapt()$ is O(L + L^2 + 1) = O(L^2).

In conclusion, the time complexity of primary scheduling in FASDQ is O(NK + L^2), and the time complexity of backup scheduling in FASDQ is O(NX + L^2).

Algorithm 3. Container resource-adaptive adjustment algorithm $resAdapt()$

```
Begin:
1   for each $EN_i \subseteq \{EN(t_i)\}$, do
    //Create a new container in the running neighboring node.
2       Sort active nodes $EN_i$ in ascending order by the remaining processing power;
3       Select a new container $c_{new}$ satisfying Equation (10);
4       if $p_i^{rem} > p(c_{new})$, then //Determine whether the remaining processing power of $EN_i$
    can accommodate the new container.
5           Create $c_{new}$ in $EN_k$;
6           return $c_{new}$;
7       else
8               Migrate $c(p_{min})$ in $EN_k$ to other nearby nodes with migration constraints;
    //Migrate the container with the least processing power in the target node to another
    neighboring node to accommodate the new container.
9               if $EN_k$ can accommodate $c_{new}$, then
10                  Create $c_{new}$ on $EN_k$;
11                  return $c_{new}$;
12          end if
13      end if
14  end for
15  for each $EN_{new}$ in nearby nodes, do //Enable a neighboring node in shutdown mode.
16      If $p(EN_{new}) > p(c_{new}) \&\& a_i + \theta_{new} + t(EN_{new}) + t(c_{new}) + eq_{new}(t_i^*) \leq d_i$, then
17          Create $c_{new}$ in $EN_{new}$;
18          return$c_{new}$;
19      else return $null$;
20  end for
END
```

5. Experiments

5.1. Formatting of Mathematical Components

This section describes simulation experiments for latency-sensitive tasks in FASDQ on the EdgeCloudSim [31] platform, which is capable of modeling edge computing environments and provides a modular architecture that can support various key functions. Since the FASDQ proposed in this paper is mainly in an edge environment, the experimental framework was designed at the edge, as shown in Figure 7.

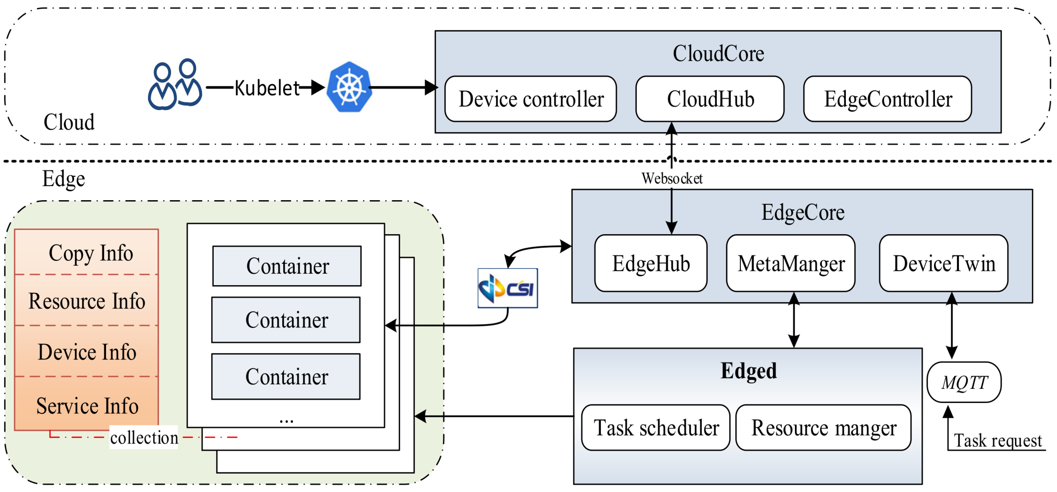


Figure 7. Task rescheduling process, where MQTT is message queuing telemetry transport.

In this study, Kubernetes v1.8.3 was extended and deployed on the Centos 7.0 system. The experiments used fault injection to interrupt the execution of task copies. Four types of edge nodes and four types of containers were set up in EdgeCloudSim, and the processing capability of each edge node was set to 1000, 1500, 2000, and 3000 MIPS. The processing capability of the containers was 250, 500, 700, and 1000 MIPS, respectively, to evaluate the performance of FASDQ.

We want the edge containers to process task requests as quickly as possible. Therefore, we implemented the Task Scheduler and Resource Manager modules in the edge server for edge containers, which enable the containers to run a larger number of tasks in parallel.

In this paper, we compare FASDQ with the following solutions:

(1) Fault-tolerant elastic scheduling algorithm (FESTAL) [17]: it can be added or removed certain virtual machines depending on the load state to optimize resource utilization while supporting fault tolerance in the cloud. In terms of using virtualization technology, the approach in this paper uses the more lightweight container virtualization technology. This experiment applied FESTAL to the edge side for comparative experiments to verify the effectiveness of FASDQ in terms of latency and resource utilization.

(2) Fault-tolerance based QoS-aware scheduling algorithm (FTBQA)[19]: adjusts the QoS level of tasks and has an adjustment mechanism, which is invoked when the backup copy of a task in a compute node is deleted after the corresponding primary copy succeeds.

(3) Non-Resource Adjustment (NONRAD): a variant of FASDQ that does not use the resource-adaptive adjustment mechanism.

(4) Non-Backup Overlapping (NONBOL): a variant of FASDQ that lacks backup overlapping techniques to verify the impact of backup overlapping techniques on system reliability and QoS levels.

To study the performance under a more realistic scenario, we used an open dataset, YFCC100M [32], provided by Yahoo in 2014. YFCC100M contains 100 million media objects. There are approximately 0.8 million videos and 99.2 million images. Each object in YFCC100M contains several metadata, such as media source, owner name, camera, title, tags, and location [33]. We filtered the multimedia contents of 300 videos and 300 photos in the YFCC100M dataset. Multimedia content, namely, the number of tasks, is a random number in $[5000, 400, 000]$. At a given time, some of the multimedia devices are active, and the average number of active devices is 320. The distance of the edge base station was set to 1000 m, and a hexagonal structure was adopted to depict the range of the geographical location [34].

In order to more clearly observe the impact of different parameters on performance, experiments based on random synthetic workloads were conducted. The data traffic generated from each source point is related to total active users and is transmitted to the edge computing system as packets. The packet size is a random number in $[34, 6550]$ and the unit is Byte. The average packet arrival rate is one packet per second per node. The bandwidth capacity between the users and the edge network is 1 Gbps [35].

Assume that the task arrivals at the edge server follow a Poisson distribution and that the average interval of task arrivals is $1/\lambda$, uniformly distributed in $\left[\frac{1}{\lambda}, \frac{1}{\lambda} + 2\right]$ [17]. The task size is uniformly distributed in $\left[1 \times 10^5, 2 \times 10^5\right] MI$ with a deadline of $d_i = a_i + \max\left\{eq\left(t_i^*\right)\right\} + basedeadline$ [36], and $basedeadline$ is the benchmark deadline, which is a random positive real number in $\{100, 140, \cdots, 420\}$ in units of s, indicating the range of task delays. The edge server failure rate is uniformly distributed with a time unit of 10^{-7}/h [14].

In each set of experiments, only one parameter was changed, and the others were left unchanged. Table 2 shows the parameters and their values.

Table 2. Task parameters.

Parameter	(Fixed)-(Min, Max, Step)
Task size ($\times 10^5$ MI)	([1, 2])
Task count ($\times 10^4$)	(1)-(0.5, 4, 0.5)
Interval time ($1/\lambda$)	(4)-(0, 16, 2)
Base deadline ($\times 10^2$ s)	(3)-(1, 4.2, 0.4)

5.2. Analysis of Experimental Results

5.2.1. Evaluation of the Effectiveness of Fault-Tolerant Scheduling

In this experiment, the effectiveness of fault-tolerant scheduling of primary copies was evaluated by the task guarantee rate. The guarantee rate (GR) [17] is defined as the percentage of total tasks that are guaranteed to be completed successfully among all submitted tasks.

(1) The task arrival interval $1/\lambda$ value range is from 0 to 16 with a step of 2 s. The value is inversely proportional to the workload per unit time. Other parameters remain unchanged.
(2) Base deadline is set from 100 to 420 with a step of 40 s. The higher the value of the base deadline, the more relaxed the task deadline constraint. The other parameters remain unchanged.

Figure 8a shows the variation in the guarantee rate with the task arrival rate of the five algorithms. The arrival rate decreases as the task volume increases, but all have higher guarantee rate (GR) values. Because the task arrival rate decreases, the workload decreases and the resources are more sufficient, so the GR value increases as the arrival rate decreases. Compared with FASDQ, NONRAD, and NONBOL, NONRAD has the lowest GR value because it does not use the resource-adaptive adjustment mechanism proposed in this paper, and when the task volume is large, the current resources cannot meet the demand of the task copy. NONRAD cannot expand resources on demand, resulting in tasks that do not start on time, thus missing deadlines. When a large number of task requests occur at the same time, FASDQ has a better effect on GR. This is because FASDQ can confirm the QoS level of task copies to reduce their execution time, and the resource adjustment mechanism can add new resources adaptively. FASDQ is thus able to handle more tasks under the same conditions. This is in line with our expectations.

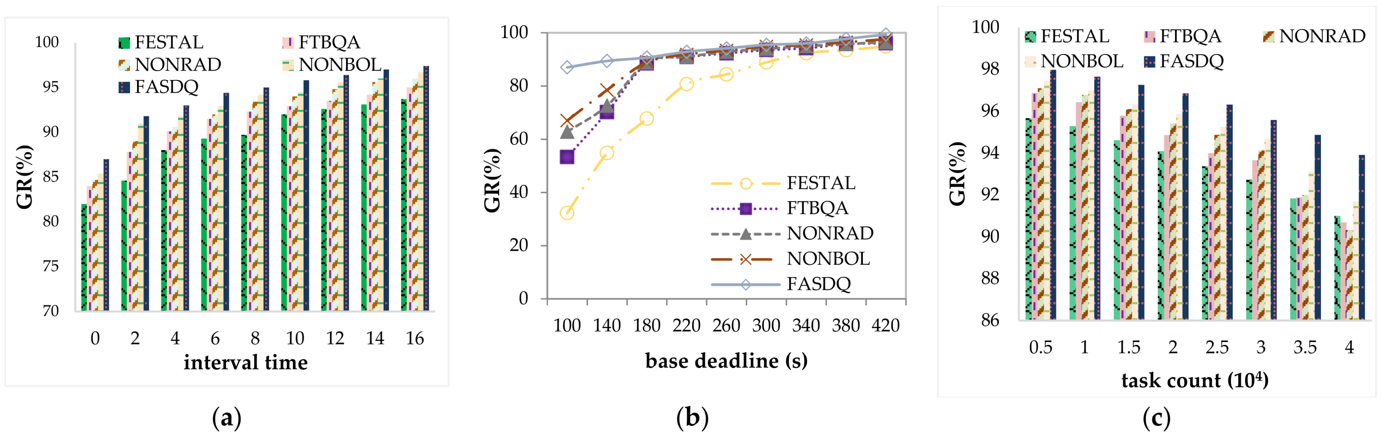


Figure 8. Comparison of guarantee rate (GR). (**a**) Impact of interval time on GR; (**b**) impact of base deadlines on GR; (**c**) impact of task count on GR.

Figure 8b shows the effect of the base deadline on the GR value. The data show that base deadline has a greater effect on the GR value because when the task is urgent, the system does not have time to adjust the available resources; therefore, task copies that arrive later are not able to meet the deadline. For several methods, when the latency is relatively lenient, the variation in GR values is usually stable because they implement fault tolerance and use overlapping technology. In particular, FASDQ can provide different QoS levels to tasks according to their different urgency levels to guarantee reliable service. Moreover, FASDQ extends the fault tolerance mechanism by considering the dynamic resource adjustment technique when resources are insufficient. Therefore, FASDQ performs better in terms of GR value compared with the other methods.

Figure 8c shows that the guarantee rate decreases as the number of tasks increases. Since failures are uniformly distributed, the number of tasks increases, leading to more failures, but all methods maintain a high guarantee rate, except for FTBQA. When the number of tasks increases, the mechanism can handle new requests by adjusting resources; while FTBQA has no resource-adaptive adjustment, it can guarantee a high GR because if there are no resources in the edge cloud to meet the task requirements, the task will be considered for scheduling in a remote cloud data center. With the increase in the number of tasks, FASDQ's GR value remains above 94%, which proves the effectiveness of the proposed method once again.

In the proposed method, the latency is reduced by providing a QoS level to change the execution time of task copies, allowing the system to accommodate more task copies, so the effect of task latency can be better measured by the average reduced time. Figure 9 shows the results for the average reduction in time for the five methods. Both FASDQ and NONBOL, which adaptively adjust resources, significantly reduce the task latency with time constraints between 180 and 420 s, while the reduction in time is not significant between 100 and 180 s. This is due to the resource-adaptive adjustment mechanism, which is optimized to reduce resource consumption and improve resource utilization, resulting in the increased data-processing capability of edge servers and thus reduced task completion time. The difference in the latency results between FESTAL and FASDQ shows that providing QoS levels to tasks also improves their execution time. However, for tasks with short latency constraints, FASDQ cannot significantly reduce the task latency because of insufficient resources for adjusting the urgent-task system.

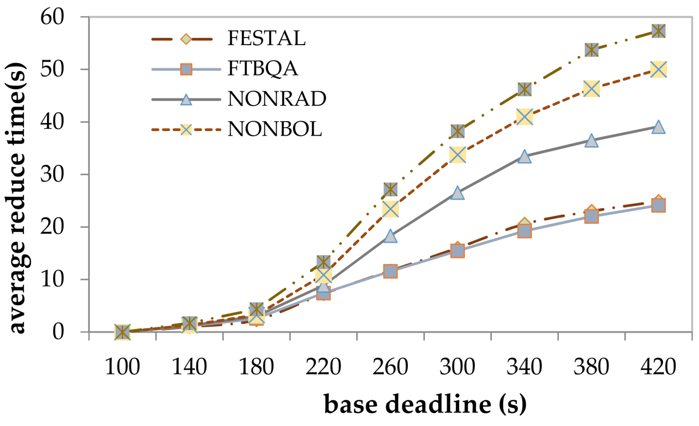


Figure 9. Average time reduction for a task.

Because FASDQ dynamically assigns the QoS level to copies during scheduling, this experiment also evaluated the average QoS level (AQL) on the basis of $1/\lambda$, base deadline, and the number of tasks. AQL is the average QoS level of all accepted tasks, and a higher AQL indicates a higher reliability performance of the framework. Let $q\left(t_i^*\right)$ denote the QoS level of task copies t_i^* with $0 \leq q\left(t_i^*\right) \leq 1$ taking a value of $[0, 0.1, 0.2, \cdots, 0.9, 1]$.

Figure 10a shows the variation in AQL as the task arrival rate decreases. As the task arrival rate decreases, the burden is reduced, the competition for resources decreases, and the task copies can be assigned higher QoS levels. Since FASDQ, NONRAD, NONBOL, and FTBQA can dynamically adjust the QoS level according to the demand of the copies when scheduling tasks, its average QoS level is higher than that of FESTAL. Compared with NONRAD and NONBOL, FASDQ has a higher average QoS level, which indicates that the combined application of the proposed method provides a higher QoS level for task copies, enabling the edge containers to handle more tasks in the same time interval.

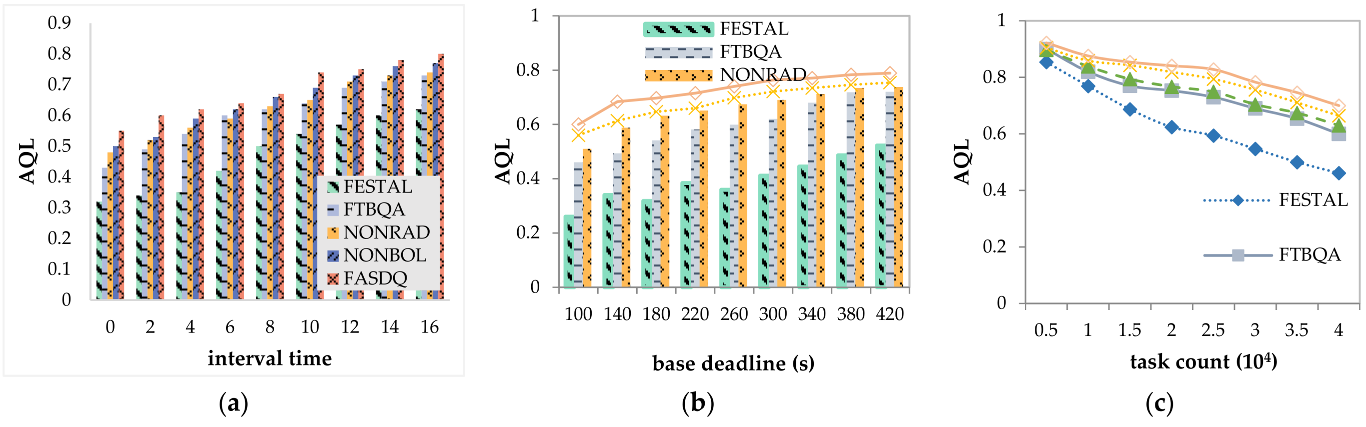


Figure 10. Comparison of average QoS levels (AQL). (**a**) Impact of interval time on AQL; (**b**) impact of base deadline on AQL; (**c**) impact of task count on AQL.

Figure 10b shows that the more relaxed the task deadline, the better the scheduling of task copies, and the task copies can receive more reliable service guarantees. Therefore, the QoS levels are improved in all five methods. The AQL of FESTAL fluctuates more and is smaller than those of the other methods because it schedules without providing QoS levels to the copies on demand. As shown, the AQL of FASDQ can also exceed 0.6 for urgent task deadlines. This is because FASDQ can adaptively assign different QoS levels according to the urgency of the task. The more urgent the task, the higher the QoS level, and the more resources it consumes, resulting in fewer tasks being processed. However, FASDQ uses a resource adjustment mechanism, so its average QoS level can be higher than those in the other methods.

Figure 10c depicts the decreases in AQL as the number of tasks increases. As shown, when the number of tasks is between 5000 and 40,000, FASDQ can obtain a higher AQL than the other methods. The continuous increase in the number of tasks with a fixed number of available resources causes the average computational resources per task to become increasingly scarce, and the task copies obtain a lower QoS level. The results of comparing NONBOL and NONRAD show that the resource-adaptive adjustment mechanism proposed in this paper strongly improves the scheduling effect.

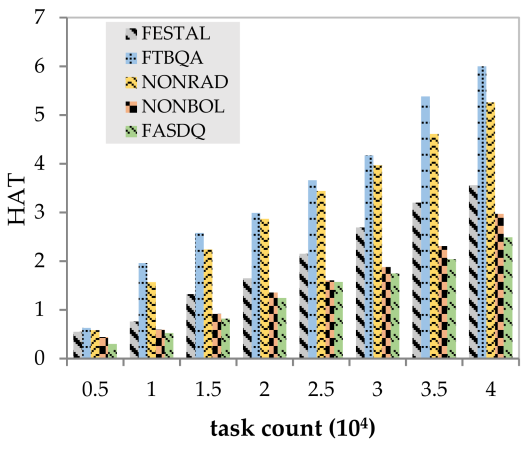
5.2.2. Resource Consumption Evaluation

In this subsection, we verify the effect of the number of tasks on the total active time (HAT) of the edge nodes and the ratio of the service time of tasks to the active time of the edge node (RTH). The number of tasks in this experiment increased from 5000 to 40,000 with a step of 5000. HAT reflects the resource consumption of the system, RTH reflects the resource utilization of the system, and RTH is defined as follows:

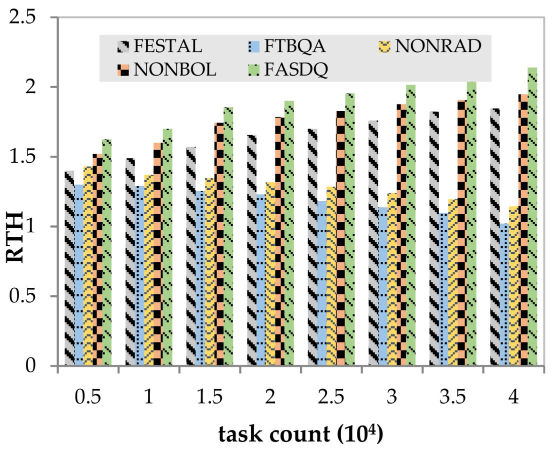
$$RTH = \max_{t_i \in T, EN_k \in EN} \{TET/HAT\} \tag{11}$$

where TET denotes the total execution time of the task.

Figure 11a shows that as the number of tasks increases, the HAT of the five algorithms maintains an increasing trend. The HAT is worse in both FTBQA and NONRAD, but for different reasons. FTBQA does not use container technology, and when there is a lack of computing resources, it needs to increase the edge server resources to meet task requirements, which results in significant resource consumption. The HAT is worse in NONRAD because there is no resource adjustment mechanism to replenish the processing capability for task copies in a timely manner, which also further illustrates the predominant role of resource-adaptive adjustment in fault-tolerant scheduling. When the number of tasks increases, FASDQ can migrate containers with a lower processing capability in the current candidate nodes to free up resources. This allows the candidate nodes to create new containers, thus avoiding the need to add or start new nodes and saving node resources.



(**a**)



(**b**)

Figure 11. Comparison of resource utilization. (**a**) Impact of task count on HAT (the total active time of the edge nodes); (**b**) impact of task count on RTH (the ratio of the task service time to the active time of the edge node).

Figure 11b depicts the variation in RTH performance as the number of tasks increases for the five methods. The RTH performance of NONRAD and FTBQA is poor when the number of tasks increases: when the tasks increase substantially, the system is burdened, and more active nodes must be added to the processing capability, resulting in greater resource consumption. The RTH of NONRAD outperforms that of FTBQA by 8.6%, indicating that the overlapping technique can effectively use the available resources to accommodate more tasks and avoid a significant increase in resources, effectively improving resource utilization. Compared with the other methods, the RTH of FASDQ is higher and relatively stable because FASDQ provides better QoS of reliability for task copies and reduces the execution time of tasks; therefore, FASDQ can always process more task copies in a given time, which effectively improves resource utilization.

In summary, the FASDQ mechanism extends the PB model to achieve fault tolerance by providing QoS levels for task copies, which effectively reduces task latency. From the evaluation of AQL, the FASDQ mechanism clearly provides higher QoS levels for latency-sensitive tasks and performs better in terms of reliability requirements and scheduling. As illustrated by the variation in HAT and RTH values, FASDQ can effectively improve resource utilization in the edge node and container levels by implementing container resource-adaptive adjustment in terms of consumption and saving resources.

6. Conclusions and Future Work

This paper introduces FASDQ, a fault-tolerant adaptive scheduling mechanism with dynamic QoS-awareness that guarantees reliable service for latency-sensitive tasks and improves the resource utilization of edge containers. FASDQ extends the PB model and proposes a dynamic QoS-aware task scheduling mechanism to achieve fault tolerance, which dynamically provides QoS levels to task copies according to the different latency requirements of tasks to change the execution time, so the edge containers can accommodate more task copies with a given processing capability while being fault-tolerant. In addition, FASDQ implements a resource-adaptive adjustment mechanism to address the uneven resource distribution and insufficient processing capability of tasks during execution, which effectively replenishes resources for tasks with insufficient processing capacity and reduces resource consumption at the node and container levels. The experimental results show that the method can significantly reduce the execution time of each task copy by about 12%, effectively reducing the task latency and improving system resource utilization.

In this paper, we do not consider task scheduling enough when the tasks are uploaded to the cloud after the failure of edge container scheduling, and we will further extend the fault-tolerant adaptive scheduling mechanism between cloud edges in the future so that the mechanism can tolerate multiple edge server failures. Furthermore, this paper addresses relatively static user requests, such as monitoring devices that are deployed in fixed locations. In realistic scenarios, such as 5G systems, there is a need to support mobility for mobile users. If the corresponding contents, which are located in edge containers, could be migrated to the place close to mobile users as they move, the QoS of reliability will be improved during the movement of the mobile users and the communication delay will be reduced between mobile users and the nodes. However, we did not delve into the task requests of the mobile users, which are common in realistic environments. This is our research focus for future studies.

Author Contributions: Conceptualization, N.C. and R.W.; methodology, R.W.; software, R.W. and X.Y.; validation, R.W., N.C., X.Y., and L.H.; formal analysis, R.W.; investigation, R.W.; writing—original draft preparation, R.W.; writing—review and editing, N.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by the Natural Science Foundation of China under Grant 61762008, the National Key Research and Development Project of China under Grant 2018YFB1404404, and the Major Special Project of Science and Technology of Guangxi (No. AA18118047-7).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in YFCC100M at 10.1145/2812802, reference number [32].

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
2. Yao, G.; Ding, Y.; Hao, K. Using Imbalance Characteristic for Fault-Tolerant Workflow Scheduling in Cloud Systems. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 3671–3683. [CrossRef]
3. Awada, U.; Zhang, J. Edge Federation: A Dependency-Aware Multi-Task Dispatching and Co-location in Federated Edge Container-Instances. In Proceedings of the 2020 IEEE International Conference on Edge Computing (EDGE), Honolulu, HI, USA, 18–20 September 2020; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2020; pp. 91–98.
4. An In-Depth Analysis of Google's 5G Mobile Edge Cloud Strategy. Available online: http://www.qianjia.com/html/2020-03/10_362269.html (accessed on 10 March 2020).
5. Flynn. Available online: https://flynn.io/ (accessed on 31 January 2018).
6. Deis. Available online: https://github.com/deis/deis (accessed on 20 May 2017).
7. Edge Container Service. Available online: https://cloud.tencent.com/document/product/457/42876 (accessed on 20 November 2020).
8. Introduction to Edge Containers (ACK Edge Kubernetes). Available online: https://developer.aliyun.com/article/703032 (accessed on 20 May 2019).
9. Aguilera, X.M.; Otero, C.; Ridley, M.; Elliott, D. Managed Containers: A Framework for Resilient Containerized Mission Critical Systems. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018; Institute of Electrical and Electronics Engineers (IEEE): Piscatawy, NJ, USA, 2018; pp. 946–949.
10. Ghosh, S.; Melhem, R.; Mosse, D. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.* **1997**, *8*, 272–284. [CrossRef]
11. Ramamurthy, C.; Chellappa, S.; Clark, L.T.; Chandarasekaran, R.; Srivatsan, C. Physical Design Methodologies for Soft Error Mitigation Using Redundancy. In Proceedings of the 2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS), Moscow, Russia, 14–18 September 2015; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2015; pp. 1–5.

12. Zhu, X.; Wang, J.; Qin, X.; Wang, J.; Liu, Z.; Demeulemeester, E. Fault-Tolerant Scheduling for Real-Time Tasks on Multiple Earth-Observation Satellites. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *26*, 3012–3026. [CrossRef]
13. Ding, W.; Guo, R.; Qin, C. A Fault-Tolerant Scheduling Algorithm with Software Fault Tolerance in Hard Real-Time Systems. *Comput. Res. Dev.* **2011**, *48*, 691–698.
14. Liu, J.; Wei, M.; Hu, W.; Xu, X.; Ouyang, A. Task scheduling with fault-tolerance in real-time heterogeneous systems. *J. Syst. Arch.* **2018**, *90*, 23–33. [CrossRef]
15. Guo, Y.; Zhu, D.; Aydin, H.; Han, J.-J.; Yang, L.T. Exploiting primary/backup mechanism for energy efficiency in dependable real-time systems. *J. Syst. Arch.* **2017**, *78*, 68–80. [CrossRef]
16. Al-Omari, R.; Somani, A.K.; Manimaran, G. Efficient overloading techniques for primary-backup scheduling in real-time systems. *J. Parallel Distrib. Comput.* **2004**, *64*, 629–648. [CrossRef]
17. Wang, J.; Bao, W.; Zhu, X.; Yang, L.T.; Xiang, Y. FESTAL: Fault-Tolerant Elastic Scheduling Algorithm for Real-Time Tasks in Virtualized Clouds. *IEEE Trans. Comput.* **2014**, *64*, 2545–2558. [CrossRef]
18. Xing, L.; Tannous, M.; Vokkarane, V.M.; Wang, H.; Guo, J. Reliability Modeling of Mesh Storage Area Networks for Internet of Things. *IEEE Internet Things J.* **2017**, *4*, 2047–2057. [CrossRef]
19. Sun, H.; Yu, H.; Fan, G.; Chen, L. QoS-Aware Task Placement with Fault-Tolerance in the Edge-Cloud. *IEEE Access* **2020**, *8*, 77987–78003. [CrossRef]
20. Balasubramanian, J.; Gokhale, A.; Dubey, A.; Wolf, F.; Lu, C.; Gill, C.; Schmidt, D. Middleware for Resource-Aware Deployment and Configuration of Fault-Tolerant Real-time Systems. In Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, Stockholm, Sweden, 12–15 April 2010; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2010; pp. 69–78.
21. Grover, J.; Garimella, R.M. Reliable and Fault-Tolerant IoT-Edge Architecture. In Proceedings of the 2018 IEEE Sensors Applications Symposium (SAS), Seoul, Korea, 12–14 March 2018; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2018; pp. 1–4.
22. Wang, C.; Gill, C.; Lu, C. Adaptive Data Replication in Real-Time Reliable Edge Computing for Internet of Things. In Proceedings of the 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), Montreal, QC, Canada, 21–24 April 2020; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2020; pp. 128–134.
23. Wang, C.; Chen, X.; Jia, W. PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18), Renton, WA, USA, 9–11 April 2018.
24. Zhou, D.; Tamir, Y. Fault-Tolerant Containers Using NiLiCon. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, 18–22 May 2020; Institute of Electrical and Electronics Engineers (IEEE): Piscataway, NJ, USA, 2020; pp. 1082–1091.
25. Mao, Y.; Zhang, J.; Song, S.H.; Letaief, K.B. Stochastic Joint Radio and Computational Resource Management for Multi-User Mobile-Edge Computing Systems. *IEEE Trans. Wirel. Commun.* **2017**, *16*, 5994–6009. [CrossRef]
26. Yin, L.; Luo, J.; Luo, H. Tasks Scheduling and Resource Allocation in Fog Computing Based on Containers for Smart Manufacturing. *IEEE Trans. Ind. Inform.* **2018**, *14*, 4712–4721. [CrossRef]
27. Are You Still Figuring out Cloud-Native? Edge Containers Are Already Here. Available online: https://new.qq.com/omn/20200927/20200927A04VLZ00.html (accessed on 27 September 2019).
28. Tsai, M.-Y.; Chiang, P.-F.; Chang, Y.-J.; Wang, W.-J. Heuristic Scheduling Strategies for Linear-Dependent and Independent Jobs on Heterogeneous Grids. In *Programmieren für Ingenieure und Naturwissenschaftler*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2011; Volume 261, pp. 496–505.
29. Wang, W.-J.; Chang, Y.-S.; Lo, W.-T.; Lee, Y.-K. Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments. *J. Supercomput.* **2013**, *66*, 783–811. [CrossRef]
30. Guo, P.; Liu, M.; Wu, J.; Xue, Z.; He, X. Energy-Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks in Cloud-Based 5G Networks. *IEEE Access* **2018**, *6*, 53671–53683. [CrossRef]
31. Sonmez, C.; Ozgovde, A.; Ersoy, C. EdgeCloudSim: An environment for performance evaluation of Edge Computing systems. In Proceedings of the 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), New York, NY, USA, 2–4 October 2020; IEEE: Piscataway, NJ, USA, 2017; pp. 39–44.
32. Thomee, B.; Shamma, D.A.; Friedland, G.; Elizalde, B.; Ni, K.; Poland, D.; Borth, D.; Li, L.-J. YFCC100M. *Commun. ACM* **2016**, *59*, 64–73. [CrossRef]
33. Zhou, P.; Wang, K.; Xu, J.; Wu, D.; Wang, K. Differentially-Private and Trustworthy Online Social Multimedia Big Data Retrieval in Edge Computing. *IEEE Trans. Multimed.* **2018**, *21*, 539–554. [CrossRef]
34. Urgaonkar, R.; Wang, S.; He, T.; Zafer, M.; Chan, K.; Leung, K.K. Dynamic service migration and workload scheduling in edge-clouds. *Perform. Eval.* **2015**, *91*, 205–228. [CrossRef]
35. Sarkar, S.; Chatterjee, S.; Misra, S. Assessment of the Suitability of Fog Computing in the Context of Internet of Things. *IEEE Trans. Cloud Comput.* **2018**, *6*, 46–59. [CrossRef]
36. Zhang, Q.; Lin, M.; Yang, L.T.; Chen, Z.; Li, P. Energy-Efficient Scheduling for Real-Time Systems Based on Deep Q-Learning Model. *IEEE Trans. Sustain. Comput.* **2017**, *4*, 132–141. [CrossRef]