DAG Scheduling in Mobile Edge Computing

GUOPENG LI, University of Science and Technology of China, China
HAISHENG TAN, University of Science and Technology of China, China and USTC-Deqing Alpha Innovation Research Institute, China
LIUYAN LIU, University of Science and Technology of China, China
HAO ZHOU, University of Science and Technology of China, China and USTC-Deqing Alpha Innovation Research Institute, China
SHAOFENG H.-C. JIANG, Center on Frontiers of Computing Studies, Peking University, China
ZHENHUA HAN, Microsoft Research Asia, China
XIANG-YANG LI and GUOLIANG CHEN, University of Science and Technology of China, China and USTC-Deqing Alpha Innovation Research Institute, China

12

In Mobile Edge Computing, edge servers have limited storage and computing resources that can only support a small number of functions. Meanwhile, mobile applications are becoming more complex, consisting of multiple dependent tasks, modeled as a Directed Acyclic Graph (DAG). When a request arrives, typically in an online manner with a deadline specified, we need to configure the servers and assign the dependent tasks for efficient processing. This work jointly considers the problem of dependent task placement and scheduling with on-demand function configuration on edge servers, aiming to meet as many deadlines as possible. For a single request, when the configuration on each edge server is fixed, we derive `FixDoc` to find the optimal task placement and scheduling. When the on-demand function configuration is allowed, we propose `GenDoc`, a novel approximation algorithm, and analyze its additive error from the optimal theoretically. For multiple requests, we derive `OnDoc`, an online algorithm easy to deploy in practice. Our extensive experiments show that `GenDoc` outperforms state-of-the-art baselines in processing 86.14% of these unique applications, and reduces their average completion time by at least 24%. The number of deadlines that `OnDoc` can satisfy is at least 1.9× that of the baselines.

CCS Concepts: • **Networks → Cloud computing**; • **Theory of computation → Scheduling algorithms;**

A preliminary version with part of this work titled “Dependent task placement and scheduling with function configuration in edge computing” was published in Proceedings of the 27th International Symposium on Quality of Service (IWQoS’19) [26], and titled “Online DAG scheduling with on-demand function configuration in edge computing” in Proceedings of the 14th International Conference on Wireless Algorithms, Systems, and Applications (WASA’19) [25].
This work was supported in part by the National Key R&D Program of China under Grant No. 2021ZD0110400, NSFC under Grant No. 62132009, and the Fundamental Research Funds for the Central Universities at China.
Authors’ addresses: G. Li and L. Liu, University of Science and Technology of China, No. 96 Jinzhai Road, Hefei, Anhui 230026, China; emails: {guopengli, liuliuy}@mail.ustc.edu.cn; H. Tan (corresponding author), H. Zhou, X.-Y. Li, and G. Chen, University of Science and Technology of China, No. 96 Jinzhai Road, Hefei, Anhui 230026, China and USTC-Deqing Alpha Innovation Research Institute, Huzhou, Zhejiang, China; emails: {hstan, kitewind, xiangyangli, glchen}@ustc.edu.cn; S. H.-C. Jiang, Center on Frontiers of Computing Studies, Peking University, No. 5 Yiheyuan Road, Beijing 100871, China; email: shaofeng.jiang@pku.edu.cn; Z. Han, Microsoft Research Asia, No. 701 Yunjin Road, Shanghai 200232, China; email: Zhenhua.Han@microsoft.com.

1550-4859/2023/10-ART12 $15.00
https://doi.org/10.1145/3616374

Additional Key Words and Phrases: DAG scheduling, function configuration, edge computing, online algorithm

ACM Reference format:
Guopeng Li, Haisheng Tan, Liuyan Liu, Hao Zhou, Shaofeng H.-C. Jiang, Zhenhua Han, Xiang-Yang Li, and Guoliang Chen. 2023. DAG Scheduling in Mobile Edge Computing. *ACM Trans. Sensor Netw.* 20, 1, Article 12 (October 2023), 25 pages.
https://doi.org/10.1145/3616374

1 INTRODUCTION

With the rapid development of cloud computing, many applications are offloaded from mobile devices to remote cloud data centers. However, the long propagation delay, limited Internet bandwidth, and unstable networking environment make it hard to meet the **Quality of Service (QoS)** requirements of some latency-sensitive applications, such as autonomous driving and augmented reality [30]. To mitigate the latency, **mobile edge computing (MEC)** is proposed to deploy relatively small-scale servers, called *edge servers*, at the edge of the Internet (e.g., wireless access points) so that the resource-limited devices can leverage the computation resource nearby with low latency [1, 28, 30]. The serverless computing [2, 16] architecture has been advocated by major cloud providers as the service provision model in MEC, such as Alibaba EdgeRoutine [10] and Lambda@Edge [20], which allows users to execute functions on the edge without managing the edge servers [50]. Serverless computing has been proven more scalable, elastic, user-friendly, and cost-efficient than the traditional **IaaS (Infrastructure as a Service)** architecture when supporting MEC [33–35, 46]. However, serverless architecture in MEC introduces several new challenges for resource management.

Limited Resources in Edge Servers: Edge servers are expected to serve a broad range of applications. However, due to the limited space and high operation cost, the edge servers are typically not densely deployed, and each server is relatively constrained in computation and storage compared with the remote cloud. Only a subset of the functions can be configured in each edge server. A common practice is to use the *on-demand* configuration that allows different functions to share edge servers in a fine-grained manner. When a task is dispatched to an edge server, it will first configure the function to serve the task by fetching it from the cloud and preparing the environment locally. If there are not sufficient resources to fetch the function from the cloud, a replacement will need to be executed at the edge. If a task is dispatched to the remote cloud, the configuration overhead could be avoided (or greatly reduced as no fetching time is involved) at the cost of data transmission from mobile devices to the remote cloud. Therefore, one key challenge to adopting the serverless architecture in MEC is to play the tradeoff between the function configuration overhead on edge servers and the data transmission cost and delay to the remote cloud.

Complex Inter-task Dependency: Modern mobile applications usually consist of multiple dependent tasks (aka, computation modules), which can be modeled as a **Directed Acyclic Graph (DAG)**. For example, more than 75% of the total of 4 million jobs (applications) in the Alibaba data trace are involved with dependent tasks [4]. Figure 1 demonstrates a video processing application from Facebook [15] where multiple dependent tasks together complete the video classification computation. Specifically, the tasks of an application could be dependent due to various precedence constraints, i.e., a task cannot be started before the completion of all its predecessors. Moreover, the cross-server data transmission will typically occur when dependent tasks are placed on different servers, resulting in communication overhead. The complex inter-task dependency and communication make resource management more challenging in

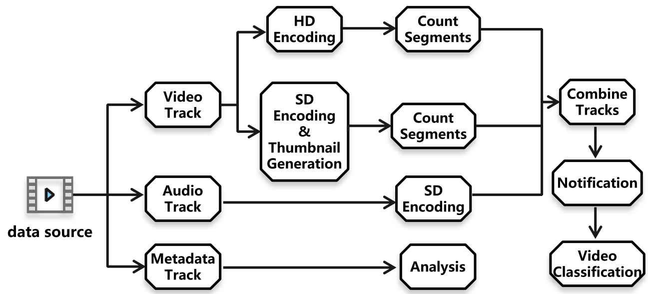


Fig. 1. The DAG of a video processing application.

MEC. Placing the parallel tasks onto different servers could increase the execution parallelism for faster computation, but it introduces higher communication overhead. Furthermore, a task may have a different processing time on different edge servers (e.g., the video processing tasks run much faster on servers with GPUs). There are tradeoffs among DAG parallelism, heterogeneous processing time, and the cross-server communication overhead.

In this article, we consider on-demand function configuration and DAG scheduling jointly in edge computing. We consider the MEC environment with heterogeneous edge servers and a remote cloud. Multiple requests to various applications, with specific deadlines, arrive online in arbitrary time and order. Our objective is to place and schedule the application tasks onto edge servers and the remote cloud so as to satisfy as many request deadlines as possible. Our main contributions can be summarized as follows:

— For the special case when there is only a single request, we first prove its NP-hardness. If the edge server configuration is fixed and given, we propose an efficient algorithm, `FixDoc`, which solves the problem optimally. Then, based on `FixDoc`, we design an approximation algorithm named `GenDoc`, for the problem with on-demand function configuration. `GenDoc` exploits the task dependency and configures the functions to leverage the application parallelism with low configuration and communication overhead. Its additive error bound from the optimal is proved (Theorem 2).
— For the case when multiple requests arrive online in arbitrary time and order for various applications, we derive a novel online algorithm, called `OnDoc`. To the best of our knowledge, this is the first work that studies function configuration and DAG scheduling jointly in an online manner in edge computing. `OnDoc` maintains multiple task scheduling lists to dramatically reduce the idle time of edge servers. In addition, `OnDoc` is easy to implement and does not introduce large scheduling overhead.
— We conduct extensive simulations on the trace from Alibaba consisting of 3 million applications. Experiment results show that `GenDoc` outperforms state-of-the-art baselines in processing 86.14% of these unique applications, and reduces their average completion time by at least 24% (and up to 54%). For multiple requests, `OnDoc` can adapt well to different network environments and performs consistently better than the heuristic baselines on various experiment settings, e.g., the number of requests satisfying their deadline by `OnDoc` can be at least 1.9× that of the baselines.

The rest of this article is structured as follows: Section 2 presents related work. Section 3 defines the system model and formulates the problem. Our algorithms and theoretical analysis are presented in Sections 4 and 5. In Section 6, we present simulation results. Finally, we conclude this work in Section 8.

Table 1. Comparison of Related Work

Related Work	Task Scheduling	Task Granularity	Function Configuration
SDTO [7], LODCO [31], Dedas [32], OnDisc [41], [8, 37, 43, 56, 60], and so forth.	✓	Individual	✗
Hermes [18], [19, 57, 58], and so forth.	✓	A subset of DAG	✗
TDCA [13], ITAGS [40], HEFT [44], [3, 12, 29, 38, 47, 53, 59], and so forth.	✓	**DAG**	✗
SEEN [6], RL policy [14], [49], and so forth.	✗	Individual	✓
PMW [5], Hermod [17], OnMuLa [22], Camul [42], OREO [51], CaLa [55], and so forth.	✓	Individual	✓
`FixDoc`, `GenDoc` and `OnDoc` [this work]	✓	**DAG**	✓

2 RELATED WORK

In this section, we will introduce the progress of dependent task scheduling in different areas, including traditional distributed systems, cloud computing, and edge computing. Task granularity can be classified into two types: independent tasks and dependent tasks. Independent tasks are those that do not depend on the others and have no priority constraints. Dependent tasks are those that have priority constraints within the same application. For applications composed of dependent tasks, the scheduling process must take into account these constraints. Also, we study the latest research on sever configuration in edge computing. From the summary table of related work (Table 1), we can discover that none of these works jointly consider on-demand function configuration and DAG scheduling so far. In addition, although DAG scheduling has been extensively studied in different fields, there is still much room for improvement. For example, there are few studies on DAG scheduling with a performance guarantee, and the work in the online algorithm does not entirely cover the actual scenarios. In this work, we study the dependent task scheduling problem. We provide a performance-guaranteed offload strategy for single-DAG scheduling configured on demand under edge computing and design a reliable online algorithm for online DAG scheduling problems.

2.1 Scheduling of Dependent Tasks

Computation offloading and task scheduling in edge computing have been extensively studied in recent years. Most works only consider independent task scheduling [8, 31, 37, 41, 56, 60]. As mobile applications become increasingly complicated, a mobile application can consist of several dependent tasks modeled as DAGs. A large number of heuristic algorithms have been proposed to solve the task scheduling problem of static single application requests on multiple heterogeneous processors in offline situations. The goal of the problem is usually to minimize the application completion time [13, 44, 59]. Topcuoglu et al. [44] proposed the well-known heuristic algorithm, called HEFT, to minimize the earliest finish time of the tasks in the application with an insertion-based approach. The algorithm He et al. [13] proposed is based on task replication, which uses the computing time of the tasks in the application in exchange for communication time; the same task can be duplicated and run on different processors. Zhao and Sakellariou [59] handle the scheduling of multiple DAGs simultaneously in an offline environment and come up with multiple heuristic algorithms to achieve the fairness of DAG scheduling and reduce the completion time of all DAGs.

Also, the related problem that appears in the data center network is the problem of placing the **network function virtualization (NFV)** chains [19, 57]. As the development of NFV, a linear application diagram is placed between the fixed source and destination physical nodes to perform a series of operations on packets sent from the source to the target. Zhang et al. [57] model the request scheduling problem based on the key concepts from an open Jackson network and propose an algorithm to improve resource utilization and a heuristic algorithm to reduce response latency.

Cloud-oriented applications need to be partitioned for computing when they are unloaded, which means they need to decide which part of the tasks in the application should be uploaded to the cloud [9, 12, 18, 40, 58]. Kao et al. [18] place tasks in applications on multiple embedded devices in order to minimize application delay while meeting specified resource consumption. They propose a novel, **fully polynomial-time approximation scheme (FPTAS)**. However, they do not consider the impact of resource competition on the operation of the task. Sundar and Liang [40] considered the execution and communication cost jointly to minimize the total cost subject to the application deadline. By appropriately allocating the application deadline among individual tasks, the tasks were scheduled in a greedy manner.

The application offload strategy in the cloud computing environment may not be extended to three-layer or multi-layer edge computing systems. Due to the limited number and performance of edge servers, the edge server cannot simultaneously support the calculation of tasks beyond its upper limit. Therefore, many scholars research how to perform reasonable application partitioning on terminal devices, edge servers, and clouds under different scenarios [24, 47, 54]. Yang et al. [54] jointly considered the two-dimensional resource allocation of computing offload and computing, and network bandwidth, and proposed an online computing partition strategy. This strategy can effectively reduce the average completion time of unloading multiple chain structure applications. In order to optimize the reliability of calculation unloading, Liu and Zhang [24] designed heuristic strategies to deal with the code partitioning of individual applications. This strategy reduces the probability of failure under the constraint of meeting task offload delay.

2.2 Sever Configuration in Edge Computing

Besides optimizing the performance with fixed server configuration, some works focused on reconfiguration in edge computing. Hou et al. [14] proposed an online algorithm with rigorous competitive analysis for edge server reconfiguration. From the perspective of application service providers who need to rent CPU/storage resources on edge servers, Chen et al. [6] derived a learning approach to maximize the benefit under a limited budget. Yang et al. [52] first studied the joint optimization of service placement and load dispatching in the mobile cloud systems. While in MEC, Xu et al. [51] proposed an efficient online algorithm for service placement and task scheduling, which can reduce the computation latency significantly. Unlike the above online settings, some works relied on an assumption of the arrival patterns of mobile applications, e.g., Amble et al. [5] assumed that request arrival is an independent and identically distributed process. Wang et al. [49] considered a Markov process.

3 MODEL AND PROBLEM DEFINITION

We provide the system model and problem formulation in this section. Important notations are listed in Table 2.

Table 2. List of Notations

Notation	Description
$\mathcal{S}$	The set of $m-1$ heterogeneous edge servers and a remote cloud.
C_i	The capacity of server s_i.
$d_{i,j}$	The data rate between servers s_i and s_j.
V	The set of all tasks.
F	The set of all functions.
$f_j = map(v_j)$	The map between the task and the function.
r_k	The i-th task of request $\mathcal{R} = \{r_1, r_2, \ldots\}$.
L_k	The deadline of r_k.
s_{a_k}	The initial server of r_k.
$G(\mathcal{V}^k, \mathcal{E}^k, \mathcal{W}^k)$	The task DAG of the application that the request r_k calls.
v_i^k	The i-th task in G^k.
$w_{i,j}^k$	The amounts of data transferred from task v_i^k to v_j^k.
$p_{i,j}^k$	The processing time for v_i^k at server s_j.
FT_k	The completion time of request r_k.
v_{entry}^k and v_{exit}^k	The pseudo entry task and the pseudo exit task of G^k.
$EST_{i,j}^k$ and $EFT_{i,j}^k$	The earliest start time and the earliest finish time of task v_i^k on server s_j.

3.1 System Model

Networking model: The network consists of heterogeneous edge servers and a remote cloud, denoted as $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$. Each edge server s_i has a limited capacity C_i, which means that the multi-dimension resources (e.g., CPU, I/O, and storage) available in s_i can maximally configure a number of C_i functions (i.e., deploying functions and serving the corresponding tasks) simultaneously. Note that a special server s_m is to represent the remote cloud, where we assume there are enough resources to configure all functions. The data rate between servers s_i and s_j is denoted as $d_{i,j}$. We set $d_{i,j} = d_{j,i}$ and $d_{i,j} = +\infty$ if $i = j$.

Application model: There are multiple applications in the edge computing system, each of which is modeled by a DAG, which is represented by $G(\mathcal{V}, \mathcal{E}, \mathcal{W})$. Here, $\mathcal{V}$ is the set of nodes denoting the tasks, $\mathcal{E}$ is the set of directed links defining the task dependence, and the set $\mathcal{W}$ denotes the amount of required data transferred from the predecessor task to the successor on each link. For instance, a link (v_i, v_j) with weight $w_{i,j}$ specifies that there is $w_{i,j}$ amounts of data transferred from task v_i to v_j. Hence, v_j cannot start before the data transfer is finished. The computation and communication of one task cannot overlap. If two tasks are placed at different servers s_x and s_y, the communication delay, i.e., $w_{i,j}/d_{x,y}$, needs to be taken into consideration.

Request model: Each request with some initial data arrives online at one of the edge servers termed *the initial server*, which will call for an application denoted by a DAG with a deadline. Given the DAG of a request, a task without any predecessor tasks is called *the entry task* and a task without any successor is called *the exit task*. For ease of presentation, we let the exit (entry) tasks all connect to a *pseudo* exit (entry) task, which does not take any processing time or resources, so that there will be exactly one pseudo exit (entry) task in each DAG. The weight of links adjacent to the pseudo exit task is the amount of the output data produced by the exit tasks. For the pseudo entry task, the weight of the additional links is the amount of initial data received by each entry task. The pseudo entry and exit tasks must be processed in the initial server of the request, which means that initial data must be transferred from and the result must be sent back to the initial server.

Application configuration model: Without loss of generality, we assume that an application is composed of one or more tasks, and each type of task exactly maps to a function. We define a mapping $map : V \rightarrow F$, where V is the set of all tasks and F is the set of all functions. For any $v \in V$ and $f \in F$, $map(v) = f$ means that task v is to be processed by function f. To process task v_j on server s_i, s_i must have configured the corresponding function $f_j = map(v_j)$ locally. Specifically, when task v_j is assigned to edge server s_i without function f_j, s_i has to suffer a configuration time, denoted as C_{i,f_j}, to download the function from the remote cloud and deploy it. Each deployed function on an edge server can process one task at one moment, which means that the queuing delay is considered. Recall that we assume the remote cloud has configured all functions. An edge server can configure a new function directly as long as it has enough capacity. Otherwise, a replacement will incur to release a configured function for the new one.

3.2 Problem Formulation

Here, we consider a series of requests arriving *online* in arbitrary time and order, denoted as $\mathcal{R} = \{r_1, r_2, \ldots\}$. Each request r_k calls for one application in the edge system with initial data and is submitted to the edge system from one edge server termed *the initial server* s_{a_k}, which will call for an application denoted by a DAG with a deadline L_k. Except for the parameters of the network and the DAGs of all applications, we cannot know any information of a request before its arrival, e.g., the application it calls for, the amount of initial data, the initial edge server, and its deadline. Let v_i^k denote the i-th task of request r_k. The processing time for v_i^k at server s_j is $p_{i,j}^k$, which can be known at the request's arrival. The assumption is practical since we can estimate the processing time well based on the previous record. Note that since the initial data of each request might be different, the processing time of the same task from different requests of the same application might be different. When r_k arrives at server s_{a_k} in time $t_k^{\uparrow}$, we decide where to process each task (called *task assignment*). Here, if task v_i^k is assigned to server s_j, we should first configure the corresponding function $map(v_i^k)$ if s_j does not hold it. If a task is assigned to the remote cloud, we can process it as soon as it arrives at the cloud. As the granularity of tasks is relatively small, we do not consider task preemption to avoid extra processing overhead. That is to say, once one task is started, it will be continuously processed until its completion. The completion of the exit task indicates the completion of the request, which should be before the deadline. Under the aforementioned model, our goal is to satisfy as many request deadlines as possible.

A simple example of our model is illustrated in Figure 2. Specifically, there is an edge-cloud system containing three edge servers and a remote cloud. The capacity of all edge servers is set to 2, while the cloud holds all functions. Two requests, r_1 and r_2, call for the same application with their own initial data arriving at server s_1 and s_2, respectively. We take r_1 as an example, whose tasks 1 and 2 are assigned to edge server s_1, and tasks 3 and 4 to s_3. s_1 then needs to download function f_2 from the remote cloud and choose to drop the existing function f_4. In addition, task 2 can be processed only if f_2 has been deployed on s_1 and its predecessor node (i.e., task 1) has been finished.

3.3 Problem Analysis

As you can see, there are two kinds of constraints in this problem. One is the precedence constraint, and the other is the capacity constraint. Precedence constraint means that tasks

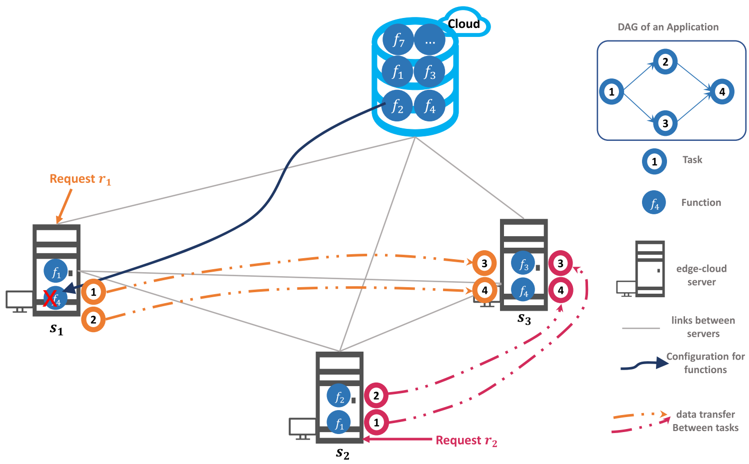


Fig. 2. An illustration of our model. The capacity of edge servers is set to 2. Requests r_1 and r_2 both request the same application. Tasks 1 and 2 of r_1 are assigned to edge server s_1. Since s_1 is already at its full capacity, in order to download function f_2 from the remote cloud, a decision is made to drop the existing function f_4.

cannot start execution before the data of its precursors are transferred to server s_j. Capacity constraint means that tasks have to wait until server s_j has spare capacity. Let us first analyze the precedence constraint. The capacity constraint will be discussed in Section 5.2.

We define $EFT_{i,j}^k$ as the earliest finish time of task v_i^k on server s_j and $EST_{i,j}^k$ as the earliest corresponding start time. As illustrated above, $EFT_{i,j}^k$ and $EST_{i,j}^k$ are two most important attributes. Ideally, we wish the completion time of request r_k, denoted as FT_k, is the earliest completion time of its pseudo exit task v_{exit}^k. Here, the pseudo exit task v_{exit}^k should be finished on the initial server s_{a_k}. For (pseudo) entry task v_{entry}, we have

$$EST_{v_{entry},a_k}^k = EFT_{v_{entry},a_k}^k = t_k^{\uparrow}, s_{a_k} \in S. \tag{1}$$

The *precedence constraint* is denoted as

$$EST_{i,j}^k \geqslant \max_{i' \in pre(i,k)} \min_{1 \le l \le m} \left\{ EFT_{i',l}^k + \frac{w_{i',i}^k}{d_{l,j}} \right\}. \tag{2}$$

Here, $pre(i,k)$ represents the set of predecessor tasks of v_i^k. Also, the communication delay is included in the inequation (2) and $w_{i',i}^k$ denotes the amount of data transfer from $v_{i'}^k$ to v_i^k. Task v_i^k will not start on server s_j until all its precursor tasks $v_{i'}^k$ are finished on server s_l and transfer data from server s_l to server s_j.

Since we do not consider task preemption, the earliest completion time of a task only needs to consider its earliest start time and processing time. We have

$$EFT_{i,j}^k = EST_{i,j}^k + p_{i,j}^k. \tag{3}$$

In addition, the completion time of a request can be considered as the finish time of the exit task. So we have

$$FT_k \geq EFT_{v_{exit},S_{a_k}}^k. \tag{4}$$

4 ALGORITHM FOR SINGLE REQUEST

In this case, there is only one request. Our goal can be converted to minimize the application completion time so that the request can be satisfied before its deadline. DAG scheduling for

ALGORITHM 1: `FixDoc`

Input: $G(\mathcal{V}, \mathcal{E}, w)$, $\mathcal{S}$ with preloaded functions, start/end server s_a

1 Assume $v_0, v_1, \ldots, v_{J+1}$ are listed in topological order.
2 Define $EFT_j^k := \infty$ for $0 \leq j \leq J+1$ and $1 \leq k \leq K$.
3 Let $EFT_0^a := 0$.
4 **for** $j = 1$ *to* J **do**
5   **for** $k = 1$ *to* K **do**
6   **if** *server* s_k *without function* v_j **then**
7             $p_{j,k} := \infty$.
8 **for** $j = 1$ *to* $J+1$ **do**
9   **for** $k = 1$ *to* K **do**
10         $EFT_j^k := \max_{i:(v_i, v_j) \in \mathcal{E}} \{\min_{1 \leq l \leq K} \{EFT_i^l + \frac{w_{i,j}}{d_{l,k}} + p_{j,k}\}\}$.
11 EFT_{J+1}^a is the optimal value, and the solution can be reconstructed from EFT_{J+1}^a.

heterogeneous systems has been proved NP-hard [13], which is a special case of our problem that the capacity of all edge servers is set uniform and the function configuration time is negligible. Therefore, this problem is also NP-hard.

4.1 Scheduling With Fixed Configuration

In this case, we can set the configuration time as $+\infty$. Therefore, we can only make use of the preloaded functions on each server. We next propose our algorithm `FixDoc` to solve the problem optimally.

To achieve the earliest finish time of the exit dummy task, which is equivalent to the minimum completion time of the request G, we have to figure out the earliest finish time of its predecessor tasks first. Thus, we design a dynamic programming (DP) method (defined in Algorithm 1). Specifically, in a given edge system, a request G is initialized on edge server s_a, where the two dummy tasks (v_0 and v_{J+1}) will be placed and executed. As we defined in Section 3.3, we use EFT_j^k to denote the earliest finish time of task v_j on server s_k. As the processing time of task v_0 is 0, we have $EFT_0^a = 0$ (line 3). Note that task v_j can be executed on the server s_k only when the needed function on s_k is configured. We here change the value of $p_{j,k}$ to $+\infty$ if there is no function v_j configured on server s_k (lines 4–7). With no function configuration considered, the task execution only needs to meet the task dependency constraints. That is to say, for each direct predecessor task v_i of task v_j, the value of EFT_j^k is at least equal to the minimum sum of three parts: the finish time of v_i, the communication time between v_i and v_j, and the processing time of v_j on s_k. Following the topological order of tasks,[1] each EFT_j^k only needs to be updated once and we can quickly get the minimal completion time EFT_{J+1}^a (line 11). The correctness is immediate from the optimality of the DP, and we conclude the following theorem.

Theorem 1. `FixDoc` *solves the special case FIX optimally in* $O(J^2K^2)$ *time, where* J *and* K *are the number of tasks in* $\mathcal{V}$ *and servers in* $\mathcal{S}$*, respectively.*

It is notable to emphasize that in `FixDoc`, one task might be placed and executed on multiple servers repeatedly with bounded times, which is a key characteristic to design our algorithm for the general cases with on-demand configuration.

[1] The topological order of a directed graph is a linear ordering of its vertices such that for every directed edge (v_i, v_j) from vertex v_i to vertex v_j, v_i comes before v_j in the ordering.

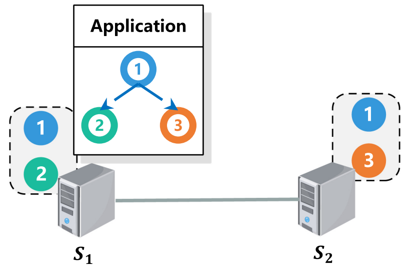


Fig. 3. An example of necessary repeated execution.

Repeated task execution is necessary. We indicate the need for repeated execution of some tasks through an example illustrated in Figure 3. Here, we consider two servers and three tasks, which form a DAG as shown in the figure. Based on the initial fixed configuration, tasks $1, 2$ can be executed on s_1, and $1, 3$ can be executed on s_2. Therefore, we have two alternative choices: (1) execute task 1 on one server and transfer its output to the other, and (2) execute task 1 repeatedly on each server. It is straightforward that when the communication overhead for the output of task 1 is larger than its execution (i.e., the output contains a large amount of data or the link data rate among s_1 and s_2 is very small), the better choice is to execute task 1 repeatedly on each server. Actually, redundant execution is a common method to reduce application response time [11, 13, 36].

Although repeated execution of tasks is necessary, we cannot afford the overhead of excessive redundant task execution. In particular, we need an upper bound for the total number of task executions among all servers, as shown in the following lemma, which we shall see later is of great importance in the analysis of the general cases in Section 4.2.

Lemma 1. *In* `FixDoc`*, the total number of task executions among all servers is bounded by* γ*, where* $\gamma := \min\{\frac{(J-1)(J-2)}{2}, (J-2)\cdot K\}$.

Proof. Impose a topological order of vertices of the DAG. Observe that to execute the i-th task, at most $i-1$ previous tasks need to be (repeatedly) executed. This concludes $\gamma \le \sum_{i=0}^{J-1} i = \frac{(J-1)(J-2)}{2}$. On the other hand, a task does not execute multiple times on the same server, so $\gamma \le (J-2)\cdot K$. This finishes the proof. □

4.2 Scheduling With On-demand Configuration

We next study the general case of the problem where each server can configure the functions on demand and derive our strategy called `GenDoc` (Algorithm 2).

Generally speaking, `GenDoc` takes `FixDoc` as a subroutine by feeding it a set of functions preloaded greedily onto edge servers. Then, we translate the output of `FixDoc` to a feasible solution of DAG scheduling with the on-demand configuration problem under the initial server configurations, where necessary on-demand function configuration is involved. Note that function preloading in `FixDoc` is conducted virtually to guide the task placement and scheduling, while actual function configuration is performed on demand. Next, we describe `GenDoc` line by line in detail.

To minimize the application completion time, we greedily execute each task v_j on the server s_k that has the minimum processing time $p_{j,k}$. So we design function preloading to ensure that s_k have been configured with the function v_k. Recall that the task and its corresponding function share the same notation. We first ignore the actual capacity constraint C_k, but define a virtual capacity constraint C_k^{vir} for each edge server s_k $(1 \le k < K)$, calculated as follows.

ALGORITHM 2: `GenDoc`

Input: $G(\mathcal{V}, \mathcal{E}, w)$, $\mathcal{S}$, start/end server s_a

/* The initial configuration of each edge server in $\mathcal{S}$ can be arbitrary, while the remote cloud s_K has configured all functions. */

1 Let $N_k := \{v_j \mid k = \arg\min_{k'} p_{j,k'}\}$ for $1 \le k < K$.

2 Let C'_k be the capacity required for server s_k to configure simultaneously the corresponding functions for tasks in N_k, and set $C' := \max_k C'_k$.

3 $C_k^{\text{vir}} := \max\{C_k, C'\}$.

4 Let $\mathcal{S}' := \mathcal{S} \backslash \{s_K\}$.

5 **for** $j = 1$ *to* J **do**

6   Let $\mathcal{S}_j := \mathcal{S}'$.

7 **while** $\mathcal{S}' \neq \emptyset$ **do**

8   **for** $j = 1$ *to* J **do**

9         $s_k := \arg\min_{s_k \in \mathcal{S}_j} \{p_{j,k}\}$.

10         $\mathcal{S}_j := \mathcal{S}_j \backslash \{s_k\}$.

11   **if** $s_k \in \mathcal{S}'$ **then**

12   Preload the function for task v_j on s_k.

13   **if** s_k *reaches capacity* C_k^{vir} **then**

14             $\mathcal{S}' := \mathcal{S}' \backslash \{s_k\}$.

15 Run `FixDoc` with G, s_a, $\mathcal{S}$ with the virtual server capacities C^{vir} and the greedy configuration functions.

16 Execute tasks under the initial server configuration according to the order and server placement in the solution returned by `FixDoc`. If this introduces waiting tasks or exceeds capacity on a server, conduct the on-demand configuration and execute the tasks whenever the server capacity is released and available.

Based on the greedy preloading policy, we can obtain the set of functions N_k that each edge server s_k needs to configure (line 1). We set C_k^{vir} as the virtual capacity of s_k so that it can configure simultaneously all the functions in N_k. Then, we set $C' := \max_k C'_k$ as the maximum virtual capacity among all edge servers, and further $C_k^{\text{vir}} := \max\{C_k, C'\}$ for each edge server (lines 2–3).

Take C_k^{vir} as the capacity of each edge server s_k. In each iteration of the loop (lines 7–14), we preload all the functions one by one to edge servers with enough capacity, so that each function $map(v_j)$ is configured onto the server that has the minimum processing time for the corresponding task among all the servers without $map(v_j)$ configured. The iteration terminates when no edge server has enough available capacity to configure any function. Note that here one function might be configured on multiple servers and hence the corresponding task can be executed on several candidate edge servers. Example 1 illustrates the whole process of function preloading.

Example 1 (Illustration of the Function Preloading in `GenDoc`*).* Figure 4 describes how to preload functions on three edge servers $\{s_1, s_2, s_3\}$ with actual available capacity as $C_1 = 1$, $C_2 = 2$, and $C_3 = 4$, respectively. The application arrived at contains nine tasks, where tasks v_0 and v_8 are dummy tasks. Table A presents the processing time of each task (except dummy tasks) on each edge server. According to line 1 in Algorithm 2, functions v_2 and v_5 needed to be configured on servers s_1, v_3, and v_6 are on s_2, and the remaining tasks are on s_3 as shown in Table B. Thus, we have $C' = max\{C'_1, C'_2, C'_3\} = 3$. Hence, the virtual capacity C_k^{vir} of server s_k is shown in Table C. Then, we configure functions on each edge server until not enough virtual capacity is available. Specifically, Table D records the result of the functions configured in each step. In the second iteration, s_2 reaches its virtual capacity after configuring function v_1, so S' only contains servers s_1

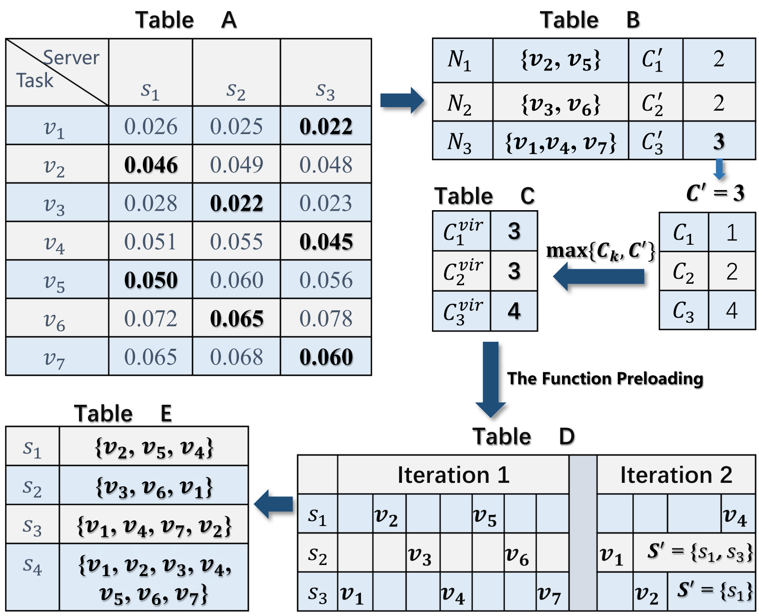


Fig. 4. An example of the function preloading in GenDoc.

and s_3. Likewise, the configuration of function v_3 on s_3 is failed due to not enough virtual capacity. The final result of functions preloaded on each server is concluded in Table E. Note that the remote cloud s_4 has configured all functions.

Next, we call FixDoc with the inputs of the DAG G, the initial edge server s_a, the server sets $\mathcal{S}$ with the virtual server capacities C^{vir}, and the greedy configuration functions (line 15). Since it is under the initial server configuration and possible that $C_k^{\text{vir}} > C_k$, on-demand configuration and waiting are needed in translating the result of FixDoc to a solution of DAG scheduling with the on-demand configuration problem (line 16). That is to say, when the actual server capacity is not enough to configure a function, the corresponding task should wait at the server until some other tasks are completed and release enough capacity.

We next analyze the performance gap between GenDoc and the optimal theoretically when the edge servers are all empty configured in the initial server configuration.

Theorem 2. *Let $C_{max} := \max_{1 \le k \le K} C_k$ be the maximum number of functions that an edge server can configure. Let $R_{max} := \max_{1 \le j \le J, 1 \le k \le K} r_{j,k}$ be the maximum on-demand configuration time for any function on any server. Define ρ_1 as the ratio between the maximum transferring time (for all tasks between any server) over R_{max}, and ρ_2 as the ratio between the maximum processing time (for all tasks on any server) over R_{max}. Let ALG be the completion time of the solution given by* GenDoc, *and OPT be the optimal completion time. We have*

$$ALG \le OPT + \gamma(\rho_1(C_{max} + 1) + 1 + \rho_2) \cdot R_{max},$$

where $\gamma := \min\{\frac{(J-1)(J-2)}{2}, (J-2) \cdot K\}$ as defined in Lemma 1.

Proof. Please refer to Appendix A.1. □

Before presenting the proof, we discuss how the parameters behave in practice. Typically, the available capacity of edge servers is not large, so $C_{\max} = O(1)$. Moreover, since the on-demand configuration is typically much more expensive than the transferring time or processing time in edge computing, it is usually the case that $0 < \rho_1, \rho_2 < 1$ and are small.

4.3 Analysis of GenDoc in Practical Cases

GenDoc would perform badly when all tasks are placed and executed on one server due to its greedy function preloading,[2] which would lead to a huge configuration and waiting time. However, this extreme case will not happen frequently in practice. Our experiments on real data traces validate that GenDoc consistently performs well, and in particular the all-task-to-one-server scenario never happens.

We show that under some reasonable assumption of input data, our greedy preloading of functions is balanced over all servers, and this justifies why our algorithm does not suffer excessive waiting and configuration time.

A simple assumption. We only need one simple assumption on the execution time of tasks: for each task v_j, its execution time on each server is **independent and identically distributed (i.i.d.)**. That is, the execution time for task v_j in each server is sampled independently from a distribution $\mathcal{D}_{v_j}$. Note that the distribution of different tasks can be different. The following fact is immediate from this assumption. Recall that there are J actual tasks (excluding the two dummy tasks v_0 and v_{J+1}) and K servers.

Fact 1. *For any task v_j and server s_k, the probability that s_k takes the minimum execution time of v_j among all servers is $\frac{1}{K}$.*

Claim 1. *With probability at least $1 - \delta$, each server has at most $2 \cdot \frac{J}{K} + 3 \ln \frac{1}{\delta}$ functions preloaded in the greedy procedure in* GenDoc.

Proof. Please refer to Appendix A.2. □

Conclusion. By union bound and Claim 1, the probability that no server is assigned more than $\left(2\frac{J}{K} + 6 \ln J\right)$ functions is at least $1 - \frac{1}{K}$. Therefore,

- when the total capacity is roughly no less than J, we may expect $\text{ALG} \le O(1) \cdot \text{OPT} + O(1) \cdot R_{max}$;
- otherwise, due to the capacity constraint, even the optimal solution suffers too much configuration time, so our algorithm performs not so much worse than OPT as well.

5 ALGORITHM FOR MULITPLE REQUESTS

We have come up with an algorithm called GenDoc to solve the DAG scheduling problem in the case of a single request. However, GenDoc is not efficient in the general practical cases where multiple requests arrive online in arbitrary time and order. The reason is as follows. Firstly, for a single request, there is little need to duplicate the configuration of a function. So GenDoc increases the probability that the task be placed on the server that requires the least processing time and ignores the possibility of duplicating the configuration of a function. However, in the situation in which there are multiple requests, GenDoc may spend much time in configuring functions. For example, assume that there are two servers, "A" and "B," and each server can configure only one function. Assume that the first request needs functions "a," "b," and "c." According to the processing time, we should configure function "a" at server "A," configure function "b" at server "B," and configure function "c" at server "A." After we configure functions "a" and "b," due to

[2]This extreme case might happen when all tasks (excluding the dummy tasks) are run the fastest on exactly the same edge server, while the edge server capacity is too small to configure multiple functions simultaneously.

ALGORITHM 3: `OnDoc`

set $Q \leftarrow \emptyset$;
`/* Thread for maintaining` Q `*/`
if *new request r arrives* **then**
  $l \leftarrow$ scheduling list of r;
  $Q \leftarrow Q \cup \{l\}$;
`/* Thread for assigning tasks and configuring functions */`
while $Q \neq \emptyset$ **do**
  `/*` H `consists of head of each scheduling list in` Q `*/`
  Construct set H ;
  We assign task v^* to server $tar(v^*)$
  Configure $tar(v^*)$ with corresponding function before v^* executed;
  if $\exists l \in Q$ *and* $l = \emptyset$ **then**
    delete l from Q;

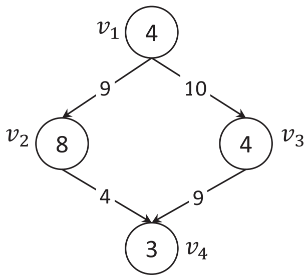
the capacity constraint, we have to release function "a" and configure function "c" at server "A." Unfortunately, the second request needs function "a" again, so we have to configure a function once more, which will cost time. Furthermore, `GenDoc` will not deal with the next request until the earlier one is finished, which can bring much queuing time.

In this case, we consider there are a series of requests arriving online in arbitrary time and order, as we defined in Section 3. Moreover, we present an online algorithm named `OnDoc` for it. We describe the details of `OnDoc` (Algorithm 3) in the rest of this section. `OnDoc` is a variant list scheduling scheme, which remains the simplest and most efficient of many prevalent list scheduling schemes of DAG scheduling. The main idea of list scheduling is to define priorities of tasks and assign tasks to the server in priority order. In additon, we have to modify the function configuration of each edge server simultaneously due to the limited capacity. Hence, `OnDoc` is composed of three parts: priority-calculating strategy, task-assigning strategy, and function-configuring strategy. We next present these strategies in detail.

5.1 Priority-Calculating Strategy

When addressing the DAG scheduling problem with list scheduling strategies, a useful method that can prioritize the tasks efficiently and simply is significant. Shin et al. [39] classifies task priorities applied by most list scheduling heuristics into three types: *S-level, B-level, T-level. S-level*, called static level, is the longest path from the task to the exit task with computation cost taken into consideration only. *B-level* is also calculated from bottom (exit task) to top (entry task). The difference between *S-level* and *B-level* is that communication cost is taken into consideration by *B-level* as well. *T-level*, namely, is the sum of computation cost and communication cost of the longest path from the entry task to the concerned task. After computing task priorities, we can prioritize these tasks corresponding to the decreasing (increasing) order of *S-level, B-level* (*T-level*). However, all these task priorities are static and can only prioritize tasks from one request. The challenge remaining is that we have to schedule tasks from multiple requests concurrently in most cases and we cannot employ these task priorities to determine the priority of tasks from different requests.

Figure 5 shows a DAG with four tasks, i.e., v_1, v_2, v_3, v_4. The number on each node indicates the computation cost required to complete the task. The number on the edge indicates the communication cost. We calculate the values of each task under three priority metrics: S-level, B-level, and T-level.



Priority	S-level	B-level	T-level
v_1	15	30	0
v_2	11	15	13
v_3	7	16	14
v_4	3	3	27

Fig. 5. An example of calculating priorities.

To tackle the aforementioned challenge, we maintain multiple task scheduling lists $Q = \{l_1, l_2, \ldots\}$ rather than follow the idea of list scheduling methodology to merge tasks from all requests to construct one prioritized list. Q is the set of prioritized lists of requests. Let Q be empty initially. When $t = t_k^{\uparrow}$, request r_k arrives at initial server s_j with deadline L_k; we employ the *B-level* as task priorities to get a scheduling list l_k of r_k and insert l_k into Q instantly (lines 1–4 in Algorithm 3). Every time we are scheduling, we only take the head of all the scheduling lists in Q into consideration. To choose an appropriate task to assign, we first determine the target server based on the task-assigning strategy for all candidate tasks. Then led by the idea to reduce the idle time of edge servers, we choose the task that can start execution at the earliest time in its target server among all the candidate tasks to assign (lines 6–7). Each scheduling process ends with the chosen task already starting execution in its target server, then the chosen task is deleted from its scheduling list and the next scheduling begins. If the scheduling list of request r_k is empty or the request exceeds the corresponding deadline, we delete it from Q (lines 9–10).

5.2 Task-Assigning Strategy

Except for the pseudo entry task, we assign each task v_i^k to the server that can finish it the earliest time. And we call the target server of $v_{i'}^k$ as $tar(v_{i'}^k)$. For ease of formula, we record the status of edge servers all the time. For instance, we maintain a set $A_j = \{(v_1', r_1'), (v_2', r_2'), \ldots, (v_n', r_n')\}$ (without pseudo entry task) for server s_j; when task v_i^k is assigned to server s_j, we insert a tuple (i, k) to A_j. Without loss of generality, we assume that

$$EFT_{v_1',j}^{r_1'} \geqslant EFT_{v_2',j}^{r_2'} \geqslant \cdots \geqslant EFT_{v_n',j}^{r_n'}$$

and denote $A_j(m) = \{(v_1', r_1'), (v_2', r_2'), \ldots, (v_m', r_m')\}$. For convenience, if $m \geq |A_j|$, we fill the set up with $(m - |A_j|)$ virtual tuples (v_{entry}, r_k).

Then we define an indicator binary variable as follows:

$$x_{i,j} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if otherwise.} \end{cases} \tag{5}$$

Recalling the precedence constraint in Section 3.3, we can describe it in detail as

$$EST_{i,j}^k \geqslant \max_{i' \in pre(i,k)} EFT_{i',tar(v_{i'}^k)}^k + \frac{w_{i',i}^k}{d_{tar(v_{i'}^k),j}}. \tag{6}$$

Here, $pre(i, k)$ represents the set of predecessor tasks of v_i^k. Also, the communication delay is included in the inequation (6) and $w_{i',i}^k$ denotes the amount of data transfer from $v_{i'}^k$ to v_i^k.

The capacity constraint is

$$EST^k_{i,j} \geqslant \min_{i',r' \in A_j(C_j)} EFT^{r'}_{i',j} + C_{j,map(v^k_i)} * x_{map(v^k_i)map(v^{r'}_{i'})}. \tag{7}$$

As a result, $EST^k_{i,j}$ can be computed as below:

$$\begin{aligned} EST^k_{i,j} = \max \Bigg\{ & \max_{i' \in pre(i,k)} EFT^k_{i',tar(v^k_{i'})} + \frac{w^k_{i',i}}{d_{tar(v^k_{i'}),j}}, \\ & \min_{i',r' \in A_j(C_j)} EFT^{r'}_{i',j} + C_{j,map(v^k_i)} * x_{map(v^k_i),map(v^{r'}_{i'})} \Bigg\}. \end{aligned} \tag{8}$$

Then we tentatively enumerate tasks' EFT in each server to determine their target server. For any task v^k_i (i is not entry or exit), we have $tar(v^k_i) = \min_{s_j \in \mathcal{S}} EFT^k_{i,j}$.

5.3 Function-Configuring Strategy

In Equation (8), when $x_{map(v^k_i),map(v^{r'}_{i'})}$ equals to 1, it means that we have to configure the corresponding function to allow the task to begin executing. First, we have to check whether there is enough capacity for configuration. If so, we just take up the spare space simply; if not, we need to decide which function will be replaced. In `OnDoc`, to allow the configuration to start as soon as possible, we always choose the function that can be replaced at the earliest time to drop. If there are many candidate functions that can be replaced at the same time, we choose one uniformly random (line 8).

6 SIMULATION

6.1 Simulation Setup

Parameter configuration. We conduct simulations in an edge-cloud cluster with five edge servers and a remote cloud data center. Each edge server has a limited available capacity, which by default, we set as three, and we conduct experiments to study the influence when the available capacity changes. The data transmission to the cloud suffers a long latency, which by default, set as 20 times that between two edge servers. We also study the parameter sensitivity of this ratio [27, 45]. The configuration time for each function on the edge server is set to 500 ms [48], the cloud data center has all functions configured, and thus the configuration time can be saved there. If not specified explicitly, the overhead of offloading a task to the remote cloud is set $[100, 2000]$ ms [21, 23]. The processing time in the remote cloud server is set as $0.75\times$ (in average value) that in edge servers, as the remote cloud typically provides more storage space and better intra-network communication than edge servers [41].

Data trace. We conduct the simulations based on Alibaba's trace of data analytics, which contains 3 million production jobs (called applications in this work) with the DAG dependency information [4]. We filter out the duplicated jobs with the same DAG information and have 20,365 unique applications, each of which has 2–198 tasks. Specifically, more than 98% of the DAGs contain less than 50 tasks. In addition, we scale data transmission time among edge servers and the processing time of each task to $[5, 100]$ ms and $[10, 200]$ ms, respectively, so as to make it more consistent with the characteristics of low latency in mobile edge computing.

6.2 Baselines

We compare `GenDoc` and `OnDoc` primarily with the following approaches.

(1) *Local Heuristic (Local):* To verify that an application should be scheduled in the task-level granularity instead of being treated as an indivisible task, we implement this baseline that always places all the tasks within an edge server that can minimize the completion time. The tasks are executed in the topological order of its DAG.
(2) ***Heterogeneous Earliest-Finish-Time (HEFT)*** [44]: HEFT is a widely adopted algorithm for DAG scheduling. It contains two phases: in the *task prioritizing phase*, HEFT computes the priority of the tasks based on their computation and communication cost; and then in the *processor selection phase*, HEFT schedules the tasks according to their priority and places each task to the server with the earliest completion time.
(3) ***Greedy Heuristic (Greedy)*****:** This heuristic balances the tradeoff between communication time and task parallelism. For each task v_j, the algorithm always schedules a task to the "nearest" server with the available function capacity to minimize the communication time from v_j's predecessor tasks.
(4) ***First-Come-First-Serve (FCFS)****:* FCFS is a popular scheduling policy that is commonly used by the methods based on queuing theory [41]. We implement it by converting multiple task scheduling lists to a single list with respect to the releasing order and always assign the task in the head of the list to its target server.

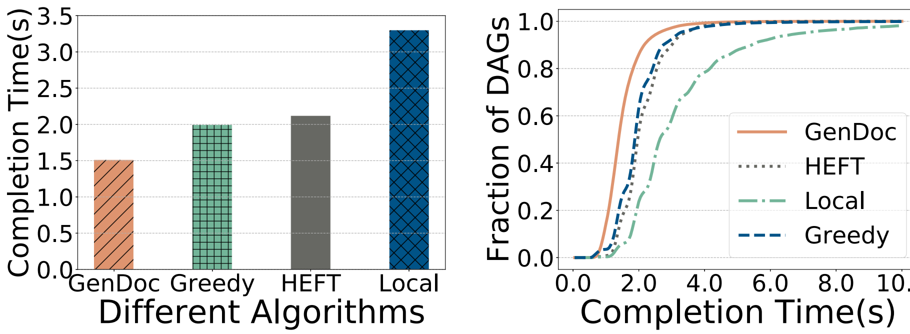
Our main results can be summarized as follows:

— GenDoc can reduce 24%, 29%, and 54% of the completion time on average compared to *Greedy*, *HEFT*, and *Local*, respectively.
— For all jobs, the maximum completion time required by GenDoc is less than 10s, while 24.43s, 15.89s, and 40.25s are needed under *Greedy*, *HEFT*, and *Local*.
— For job completion times, GenDoc is superior to all baselines on $\sim$86.41% of the DAGs.
— Under the default setting, the number of requests that satisfy their deadline in OnDoc is 1.9×, 51.6× that of *Local* and *FCFS*.
— The makespan of OnDoc is minimal compared to *Local* and *FCFS*.

6.3 Results for One Request

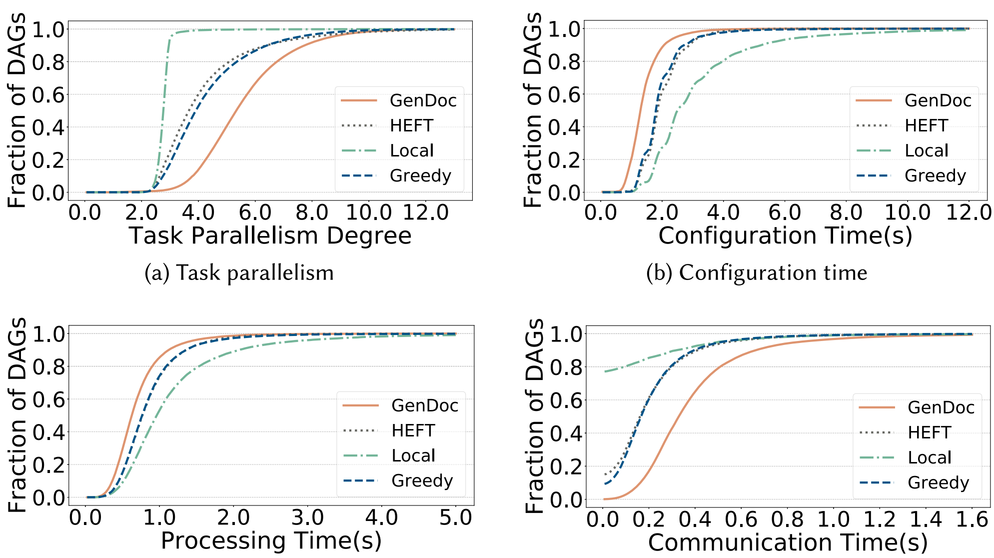
In this part, we present the experimental results on Alibaba's production trace and dissect the source of improvement of using GenDoc. We also conduct extensive sensitivity experiments using three specific DAGs to study the impact of communication-computation ratio, the function configuration time, and the transmission overhead to remote clouds. In all cases, our algorithm GenDoc outperforms the baselines significantly.

6.3.1 The Overall Performance. Figure 6 illustrates the overall performance of the four algorithms under Alibaba's cluster trace. Figure 6(a) shows the average job completion time of the 20,365 DAGs of the four algorithms. GenDoc can reduce 24%, 29%, and 54% of the completion time on average compared to *Greedy*, *HEFT*, and *Local*, respectively. *Greedy* essentially optimizes the earliest start time of each task, while *HEFT* focuses on the earliest finish time of the task. For most jobs, the processing times on different servers slightly vary, thus *HEFT* and *Greedy* have similar performance. For all jobs, the maximum completion time required by GenDoc is less than 10 s, while 24.43 s, 15.89 s, and 40.25 s are needed under *Greedy*, *HEFT*, and *Local*, respectively. There is a small set (about 24.4% in Alibaba's trace) of jobs with many more tasks (at least 18 tasks in one DAG) than the rest that makes it difficult for *Local* to utilize the parallelism of task execution and function configuration under the limited server capacity. Figure 6(b) shows the distributions of job completion times. GenDoc is superior to all baselines on $\sim$ 86.41% of the DAGs.



(a) Average completion time (b) CDF of completion time

Fig. 6. Completion time under Alibaba data.

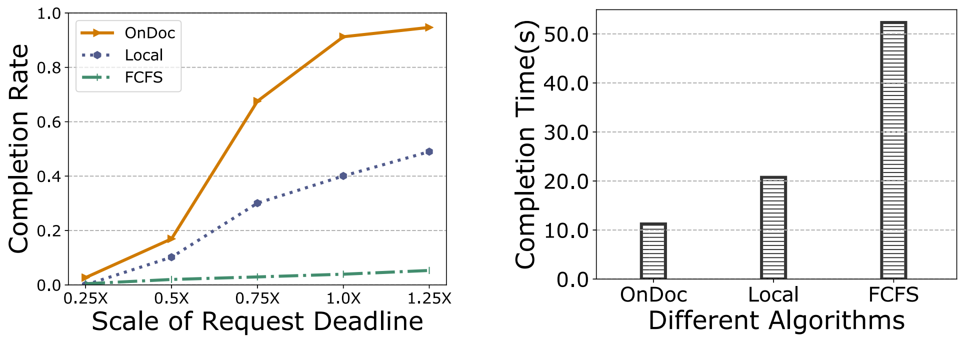


(a) Task parallelism (b) Configuration time

(c) Task processing time (d) Communication time

Fig. 7. CDF of (a) task parallelism, (b) configuration time, (c) task processing time, and (d) communication time.

6.3.2 Sources of Improvements. To understand why GenDoc achieves better performance than the three baselines, we further conduct a detailed analysis for dissecting the source of GenDoc's performance gain. Figure 7(a) shows the average task parallelism degree during the execution of the four algorithms, i.e., the average number of running tasks over job execution time. GenDoc has much higher task parallelism degree than the three baselines, which runs at least four tasks for more than 86% of the jobs. By actively offloading tasks to the remote cloud when the gain can outweigh the transmission and configuration overhead, GenDoc can leverage the higher resource utilization to reduce the job completion time. Note that all algorithms have the task parallelism degree of at least 2.0. This is because the execution of a task includes the on-demand configuration time, the task processing time, and the cross-task communication. Even for a job with a DAG of a chain, the predecessor task's processing and communication can overlap with the configuration time of successor tasks.



(a) Proportion of satisfied deadlines

(b) Makespan

Fig. 8. Overall performance of different algorithms.

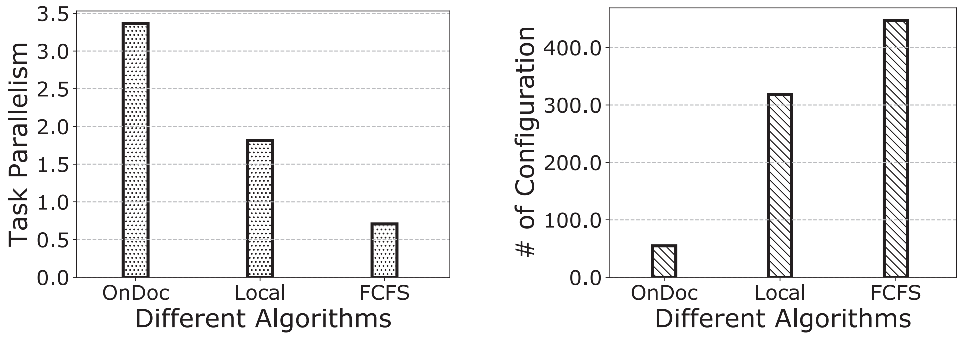
To understand how the algorithms play the tradeoffs among function configuration, task processing, and inter-server communication, we decompose the job completion time into three parts, respectively. Figures 7(b), (c), and (d) show the distribution of the three parts over the 20, 365 DAGs. GenDoc incurs a higher communication time than the baselines, while spending less time on function configuration and task processing. Since the communication overhead in the workloads is generally smaller than the task processing time, GenDoc leverages the heterogeneous processing time and higher task parallelism to compensate communication overhead. Moreover, since the functions can be configured in parallel, the higher task parallelism can also help to reduce the function configuration time. GenDoc achieves the best performance because it optimizes the scheduling decisions in a unified framework by considering the tradeoffs among the function configuration, heterogeneous processing time, task parallelism, inter-server communication, and offloading overhead to the remote cloud.

6.4 Results for Multiple Requests

In this part, we first illustrate the overall performance. The result shows that OnDoc outperforms other baselines dramatically. The number of deadlines that are satisfied is at least 1.9× that of the baselines under default setting. Furthermore, we conduct multi-group experiments to study the influences of different settings of various parameters (i.e., the offload overhead to the cloud and the capacity of the edge servers).

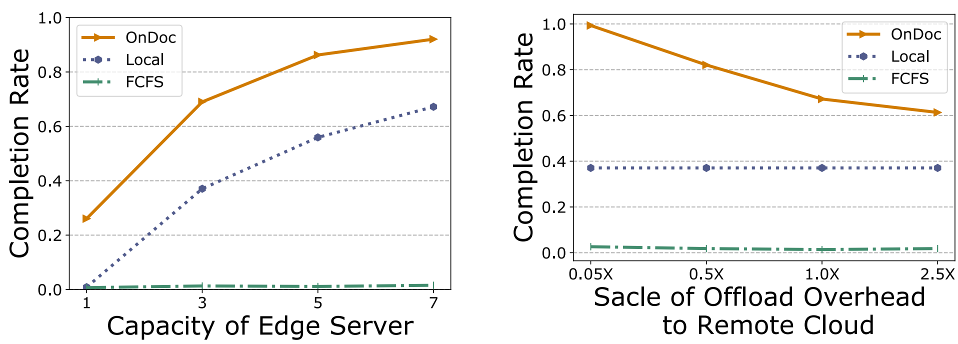
6.4.1 The Overall Performance. Figure 8 demonstrates the performance of all algorithms on the workloads from Alibaba. We scale the deadline of each request from 0.25× to 1.25× of the original value in the default setting with other parameters remaining as the default value. Figure 8(a) shows that the performance of all algorithms gets better with the deadlines increasing. Meanwhile, OnDoc outperforms the baselines dramatically. Under the default setting, the number of requests that satisfy their deadline in OnDoc is 1.9×, 51.6× that of *Local* and *FCFS*, respectively. Furthermore, the makespan, the gap between the release time of the first request and the completing time of the last completed request, of our scheduling is minimum, which is a surprising by-product shown by Figure 8(b).

6.4.2 Sources of Improvements. To understand why OnDoc outperforms the baselines, we conduct some further analysis. (1) From the perspective of the parallelism between tasks from different or even the same request, Figure 9(a) demonstrates that OnDoc can exploit the parallelism well. Task parallelism is defined as the average number of running tasks at every moment. Figure 9(a)



(a) Task parallelism (b) # of function configuration

Fig. 9. Task parallelism and # of function configuration of different algorithms.



(a) Impact of the capacity of edge servers (b) Impact of communication delay to cloud

Fig. 10. The proportion of requests satisfying deadlines with different settings.

depicts that task parallelism of OnDoc is the highest. It means that OnDoc utilizes the resources of edge servers to process more tasks concurrently than the baselines. Hence, OnDoc makes use of computing resources more sufficiently. Meanwhile, it exploits the parallelism between tasks better. (2) Configuration time is relatively larger compared with processing time and communication cost of data transfer between edge servers. The communication time from edge servers to the remote cloud is the same order of magnitude as configuration time if the amount of transferring data is large. Thus, an appropriate tradeoff between the long distance communication and configuration plays a significant role. Figure 9(b) shows that the number of function configurations by OnDoc is dramatically less than the baselines. Further analysis, which depicts 39.5% of all tasks are assigned to the remote cloud, illustrates that OnDoc utilizes the remote cloud to decrease the configuration cost without inducing notable communication cost. Almost all the tasks assigned to remote cloud request for a relatively small amount of data, which means that OnDoc assigns tasks with a large amount of input data to edge servers to avoid notable communication delay. Meanwhile, OnDoc employs cloud to mitigate edge servers' pressure to avoid repeating configurations.

6.4.3 Sensitivity Analysis. We also conduct abundant experiments to investigate the impact of different settings. Figure 10(a) demonstrates the performance of all algorithms under different capacity settings. OnDoc and *Local* are affected significantly because more capacity means more computing resources. The number of function replacing decreases due to the increase in capacity. On the one hand, some repeating configurations are avoided. On the other hand, more capacity can support more task execution simultaneously, which can exploit the parallelism between tasks better. To study the impact of communication time to the remote cloud, we scale it from 0.05× to 2.5× the default value. Figure 10(b) illustrates when communication time is 0.05×, `OnDoc` can finish all requests before their deadline. Since the communication time to cloud equals to that between edge servers, the cloud can provide powerful computing resources (i.e., infinite capacity, no function configuration time, faster task processing) without inducing more communication cost than edge servers. It is easily understood that *Local* is not affected by the overhead for it only uses edge servers. *FCFS* performs badly and is not sensitive to the above parameters since its task assignment policy is the main bottleneck constraining performance. The requests needing large processing time will block the execution of all requests that released after it.

7 DISCUSSION

This work jointly addresses the problem of dependent task placement and scheduling with on-demand function configuration on edge servers, aiming to meet as many deadlines as possible. Another practical challenge in serverless computing is how to mitigate cold starts. This challenge becomes more complex when the application involves multiple dependent tasks, especially in serverless edge computing with multiple servers. While we can treat the configuration time as the cold start latency in serverless computing, it is fixed at 500 ms, which does not accurately represent the real-world cold start scenario. In serverless edge computing, the cold start time may vary significantly due to the different computing capabilities of each edge server and the different programming languages of each function. We intend to explore the container scheduling problem in serverless edge computing, specifically focusing on the variation in cold start latency. This will be the focus of our future work.

8 CONCLUSION

In this article, we jointly consider the problem of dependent task placement and scheduling with on-demand function configuration on servers. Our objective is to meet as many request deadlines as possible. We first consider the situation where there is only one request. Then, we derive a novel approximation algorithm, called `GenDoc`, and prove its additive error from the optimal solution. Based on the real data trace from Alibaba, extensive simulations show that our algorithm can run efficiently and significantly reduce the completion time under various scenarios compared with state-of-the-art heuristic baselines. Then we propose an efficient heuristic algorithm, named `OnDoc`, to solve the problem where requests for some application with specific deadlines and initial data arrive online in arbitrary time and order. Specifically, `OnDoc` is based on list scheduling schemes and can be implemented easily in practice. In addition, a significant amount of simulations validate that `OnDoc` has a stable and superior performance compared with the baselines.

A APPENDIX

A.1 Proof Of Theorem 2

Proof. Let ALG_1 be the cost given by `FixDoc`, and let ALG_2 be the cost of converting the solution given by `FixDoc` to the feasible solution (line 14). Then $\text{ALG} \leq \text{ALG}_1 + \text{ALG}_2$.

We start with bounding at ALG_1. Let P be the input instance of the problem DAG scheduling with on-demand configuration, and let P' be the `FIX` instance generated by `FixDoc`. Let OPT' be the optimal solution returned by `FixDoc` (the optimality is proved in Theorem 1), on the instance P'. Then $\text{ALG}_1 = \text{OPT}'$.

Next, we give an upper bound of OPT′ with respect to OPT. To do so, we convert an optimal solution Q for P to a feasible solution Q' for P', such that $f(Q') \leq f(Q) + \gamma\rho_1(C_{\max} + 1) \cdot R_{max}$. By the optimality of OPT′, we conclude $\mathrm{OPT}' \leq \mathrm{OPT} + \gamma\rho_1(C_{\max} + 1) \cdot R_{max}$.

We construct Q' by first ignoring the configuration operations in Q. Then we simulate other operations of Q in Q'. Observe that operations other than the configuration in Q, together with the DAG, define a mapping from each task to a subset of servers that the task is to be executed. For each task v_j that is to be executed on server s_k in Q, if s_k is the cloud, we also let v_j execute on s_k. This does not introduce additional cost for such execution on Q' compared with Q. Otherwise, let s' be the server that the processing time of v_j is minimized, and in Q' we first execute v_j on s' and then transfer the outcome to s_k. Suppose in Q, there are x predecessors of v_j that are executed on servers other than s_k, and there are y on s_k. Then in Q' we suffer at most $(y + 1)$ additional transferring costs of tasks, which is bounded by $(y + 1) \cdot \rho_1 \cdot R_{max}$. Observe that $y \leq C_{\max}$, so $\rho_1(y + 1) \cdot R_{max} \leq \rho_1 \cdot J \cdot R_{max}$.

Therefore, by Lemma 1, we have that $\mathrm{ALG}_1 = \mathrm{OPT}' \leq \mathrm{OPT} + \gamma\rho_1(C_{\max} + 1) \cdot R_{max}$.

Then we analyze ALG_2. By Lemma 1, there are at most γ on-demand configuration operations needed for all tasks. Also, a task that is executed without waiting in `FixDoc` may need to wait in the solution of DAG scheduling with on-demand configuration problem because the virtual capacity may be bigger than the actual capacity. However, since there are γ tasks that can execute, the completion time suffers at most γ times the maximum processing time, which is $\gamma\rho_2 \cdot R_{max}$. Therefore, $\mathrm{ALG}_2 \leq \gamma(1 + \rho_2) \cdot R_{max}$.

In conclusion, we have $\mathrm{ALG} \leq \mathrm{OPT} + \gamma(\rho_1(C_{\max} + 1) + 1 + \rho_2) \cdot R_{max}$. This completes the proof. □

A.2 PROOF OF CLAIM 1

Proof. By Fact 1, for any task v_j, the probability that a fixed server s_k has the minimum execution time for task v_j is $\frac{1}{K}$. Therefore, for each server, the expected number of functions assigned to it is $\frac{J}{K}$.

Fix a server s_k. Let X_i be the random variable that indicates whether the i-th task is assigned to s_k. That is, $X_i = 1$ if the i-th task is assigned to s_k, and $X_i = 0$ otherwise. Observe that X_i's are independent, and $\Pr[X_i = 1] = \frac{1}{K}$. Let $X := \sum_{1 \leq i \leq J} X_i$. Then $\mathbb{E}[X] = \frac{J}{K}$. We do a case analysis with respect to $\mathbb{E}[X]$, which is $\frac{J}{K}$.

— $\mathbb{E}[X] \leq 3 \ln \frac{1}{\delta}$. By Chernoff bound, for $\lambda \geq 1$,

$$\Pr[X \geq (1 + \lambda) \cdot \mathbb{E}[X]] \leq \exp\left(-\frac{\lambda \cdot \mathbb{E}[X]}{3}\right).$$

Taking $\lambda := 3 \ln \frac{1}{\delta} \cdot \frac{J}{K} \geq 1$, we get $\Pr[X \geq \frac{J}{K} + 3 \ln \frac{1}{\delta}] \leq \delta$.

— $\mathbb{E}[X] > 3 \ln \frac{1}{\delta}$. By Chernoff bound, for $0 \leq \lambda \leq 1$,

$$\Pr[X \geq (1 + \lambda) \cdot \mathbb{E}[X]] \leq \exp\left(-\frac{\lambda^2 \cdot \mathbb{E}[X]}{3}\right).$$

Taking $\lambda := \sqrt{3 \ln \frac{1}{\delta} \cdot \frac{J}{K}} \leq 1$, we have $\Pr[X \geq 2 \cdot \frac{J}{K}] \leq \delta$.

Combining the two cases concludes the claim. □

REFERENCES

[1] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. 2017. Mobile edge computing: A survey. *IEEE Internet of Things Journal* 5, 1 (2017), 450–465.

[2] Mania Abdi, Samuel Ginzburg, Charles Lin, José M. Faleiro, Íñigo Goiri, Gohar Chaudhry, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. 2023. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*.

[3] Shareef Ahmed and James H. Anderson. 2022. Exact response-time bounds of periodic DAG tasks under server-based global scheduling. In *Proceedings of the 2022 IEEE Real-Time Systems Symposium (RTSS'22)*. IEEE, 447–459.

[4] Alibaba trace 2023. *Alibaba: Clusterdata*. Retrieved March 8, 2023 from https://github.com/alibaba/clusterdata

[5] Meghana M. Amble, Parimal Parag, Srinivas Shakkottai, and Lei Ying. 2012. *Content-aware Caching and Traffic Management in Content Distribution Networks*. IEEE INFOCOM.

[6] Lixing Chen, Jie Xu, Shaolei Ren, and Pan Zhou. 2018. Spatio–temporal edge service placement: A bandit learning approach. *IEEE Transactions on Wireless Communications* 17, 12 (2018), 8388–8401.

[7] Min Chen and Yixue Hao. 2018. Task offloading for mobile edge computing in software defined ultra-dense network. *IEEE Journal on Selected Areas in Communications* 36, 3 (2018), 587–597.

[8] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. 2018. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal* 6, 3 (2018), 4005–4018.

[9] Delong Cui, Wende Ke, Zhiping Peng, and Jinglong Zuo. 2016. Multiple DAGs workflow scheduling algorithm based on reinforcement learning in cloud computing. In *Computational Intelligence and Intelligent Systems: 7th International Symposium (ISICA'15), Revised Selected Papers 7*. Springer, 305–311.

[10] EdgeRoutine 2023. *CDN EdgeRoutine*. Retrieved March 8, 2023 from https://www.aliyun.com/activity/cdn/edgeroutine

[11] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 347–360.

[12] Songtao Guo, Jiadi Liu, Yuanyuan Yang, Bin Xiao, and Zhetao Li. 2018. Energy-efficient dynamic computation offloading and cooperative task scheduling in mobile cloud computing. *IEEE Transactions on Mobile Computing* 18, 2 (2018), 319–333.

[13] Kun He, Xiaozhu Meng, Zhizhou Pan, Ling Yuan, and Pan Zhou. 2018. A novel task-duplication based clustering algorithm for heterogeneous computing environments. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2018), 2–14.

[14] I.-Hong Hou, Tao Zhao, Shiqiang Wang, and Kevin Chan. 2016. Asymptotically optimal algorithm for online reconfiguration of edge-clouds. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. 291–300.

[15] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. 2017. SVE: Distributed video processing at Facebook scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 87–103.

[16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud programming simplified: A Berkeley view on serverless computing. arXiv:1902.03383. Retrieved from https://arxiv.org/abs/1902.03383

[17] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*. 289–305.

[18] Yi-Hsuan Kao, Bhaskar Krishnamachari, Moo-Ryong Ra, and Fan Bai. 2017. Hermes: Latency optimal task assignment for resource-constrained mobile computing. *IEEE Transactions on Mobile Computing* 16, 11 (2017), 3056–3069.

[19] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. 2018. Deploying chains of virtual network functions: On the relation between link and server usage. *IEEE/ACM Transactions on Networking* 26, 4 (2018), 1562–1576.

[20] Lambda@Edge 2023. *Lambda@Edge*. Retrieved March 8, 2023 from https://aws.amazon.com/lambda/edge/

[21] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. 1–14.

[22] Guopeng Li, Chi Zhang, Hongqiu Ni, and Haisheng Tan. 2023. Online file caching on multiple caches in latency-sensitive systems. In *Proceedings of the 11th International Conference on Computational Data and Social Networks (CSoNet'22)*. Springer, 292–304.

[23] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. 2019. Computation offloading toward edge computing. *Proceedings of the IEEE* 107, 8 (2019), 1584–1607.

[24] Jianhui Liu and Qi Zhang. 2019. Reliability and latency aware code-partitioning offloading in mobile edge computing. In *Proceedings of the 2019 IEEE Wireless Communications and Networking Conference (WCNC'19)*. IEEE, 1–7.

[25] Liuyan Liu, Haoqiang Huang, Haisheng Tan, Wanli Cao, Panlong Yang, and Xiang-Yang Li. 2019. Online DAG scheduling with on-demand function configuration in edge computing. In *Proceedings of the 14th International Conference on Wireless Algorithms, Systems, and Applications (WASA'19)*. Springer, 213–224.

[26] Liuyan Liu, Haisheng Tan, Shaofeng H.-C. Jiang, Zhenhua Han, Xiang-Yang Li, and Hong Huang. 2019. Dependent task placement and scheduling with function configuration in edge computing. In *Proceedings of the IEEE/ACM IWQoS*.

[27] Shiyao Ma, Jingjie Jiang, Bo Li, and Baochun Li. 2016. Custody: Towards data-aware resource sharing in cloud-based big data processing. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER'16)*. IEEE, 451–460.

[28] Pavel Mach and Zdenek Becvar. 2017. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1628–1656.

[29] S. Eman Mahmoodi, R. N. Uma, and K. P. Subbalakshmi. 2016. Optimal joint scheduling and cloud offloading for mobile applications. *IEEE Transactions on Cloud Computing* 7, 2 (2016), 301–313.

[30] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358.

[31] Yuyi Mao, Jun Zhang, and Khaled B. Letaief. 2016. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications* 34, 12 (2016), 3590–3605.

[32] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. 2019. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2287–2295.

[33] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2021. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*. 168–181.

[34] Liam Patterson, David Pigorovsky, Brian Dempsey, Nikita Lazarev, Aditya Shah, Clara Steinhoff, Ariana Bruno, Justin Hu, and Christina Delimitrou. 2022. HiveMind: A hardware-software system stack for serverless edge swarms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 800–816.

[35] István Pelle, Francesco Paolucci, Balázs Sonkoly, and Filippo Cugini. 2021. Latency-sensitive edge/cloud serverless dynamic deployment over telemetry-based packet-optical network. *IEEE Journal on Selected Areas in Communications* 39, 9 (2021), 2849–2863.

[36] Jesse Pool, Ian Sin Kwok Wong, and David Lie. 2007. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of HotOS*.

[37] Xiaoyu Qiu, Luobin Liu, Wuhui Chen, Zicong Hong, and Zibin Zheng. 2019. Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. *IEEE Transactions on Vehicular Technology* 68, 8 (2019), 8050–8062.

[38] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2021. Defuse: A dependency-guided function scheduler to mitigate cold starts on FaaS platforms. In *Proceedings of the 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS'21)*. IEEE, 194–204.

[39] KwangSik Shin, MyongJin Cha, MunSuck Jang, JinHa Jung, WanOh Yoon, and SangBang Choi. 2008. Task scheduling algorithm using minimized duplications in homogeneous systems. *Journal of Parallel and Distributed Computing* 68, 8 (2008), 1146–1156.

[40] Sowndarya Sundar and Ben Liang. 2018. Offloading dependent tasks with communication delay and deadline constraint. In *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOM'18)*. IEEE, 37–45.

[41] Haisheng Tan, Zhenhua Han, Xiang-Yang Li, and Francis C. M. Lau. 2017. Online job dispatching and scheduling in edge-clouds. In *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOM'17)*. IEEE, 1–9.

[42] Haisheng Tan, Shaofeng H.-C. Jiang, Zhenhua Han, Liuyan Liu, Kai Han, and Qinglin Zhao. 2019. CAMul: Online caching on multiple caches with relaying and bypassing. In *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOM'19)*. IEEE, 244–252.

[43] Liang Tong, Yong Li, and Wei Gao. 2016. A hierarchical edge cloud architecture for mobile computing. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (IEEE INFOCOM'16)*. IEEE, 1–9.

[44] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274.

[45] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 301–316.

[46] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. 2021. Lass: Running latency sensitive serverless computations at the edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 239–251.

[47] Jin Wang, Jia Hu, Geyong Min, Wenhan Zhan, Qiang Ni, and Nektarios Georgalas. 2019. Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning. *IEEE Communications Magazine* 57, 5 (2019), 64–69.

[48] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18).* 133–146.
[49] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K. Leung. 2015. Dynamic service migration in mobile edge-clouds. In *Proceedings of the 2015 IFIP Networking Conference (IFIP Networking'15).* IEEE, 1–9.
[50] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–61.
[51] Jie Xu, Lixing Chen, and Pan Zhou. 2018. Joint service caching and task offloading for mobile edge computing in dense networks. In *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOM'18).* IEEE, 207–215.
[52] Lei Yang, Jiannong Cao, Guanqing Liang, and Xu Han. 2015. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers* 65, 5 (2015), 1440–1452.
[53] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. 2013. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review* 40, 4 (2013), 23–32.
[54] Lei Yang, Bo Liu, Jiannong Cao, Yuvraj Sahni, and Zhenyu Wang. 2019. Joint computation partitioning and resource allocation for latency sensitive applications in mobile edge clouds. *IEEE Transactions on Services Computing* 14, 5 (2019), 1439–1452.
[55] Chi Zhang, Haisheng Tan, Guopeng Li, Zhenhua Han, Shaofeng H.-C. Jiang, and Xiang-Yang Li. 2022. Online file caching in latency-sensitive systems with delayed hits and bypassing. In *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOM'22).* IEEE, 1059–1068.
[56] Jiao Zhang, Xiping Hu, Zhaolong Ning, Edith C.-H. Ngai, Li Zhou, Jibo Wei, Jun Cheng, and Bin Hu. 2017. Energy-latency tradeoff for energy-aware offloading in mobile edge computing networks. *IEEE Internet of Things Journal* 5, 4 (2017), 2633–2645.
[57] Qixia Zhang, Yikai Xiao, Fangming Liu, John C. S. Lui, Jian Guo, and Tao Wang. 2017. Joint optimization of chain placement and request scheduling for network function virtualization. In *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17).* IEEE, 731–741.
[58] Weiwen Zhang, Yonggang Wen, and Dapeng Oliver Wu. 2013. Energy-efficient scheduling policy for collaborative execution in mobile cloud computing. In *Proceedings of 2013 IEEE INFOCOM.* IEEE, 190–194.
[59] Henan Zhao and Rizos Sakellariou. 2006. Scheduling multiple DAGs onto heterogeneous systems. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium.* IEEE.
[60] Tongxin Zhu, Tuo Shi, Jianzhong Li, Zhipeng Cai, and Xun Zhou. 2018. Task scheduling in deadline-aware mobile edge computing systems. *IEEE Internet of Things Journal* 6, 3 (2018), 4854–4866.

Received 12 April 2023; revised 27 June 2023; accepted 7 August 2023