Efficient and Fault Tolerant Data Stream Processing With Uncertain Data Rates in Serverless Edge Computing

Zichuan Xu, *Member, IEEE*, Peichen Liu, *Graduate Student Member, IEEE*, Qiufen Xia, *Member, IEEE*, Weifa Liang, *Fellow, IEEE*, Guangyuan Xu, Wenzheng Xu, *Member, IEEE*, Pan Zhou, *Senior Member, IEEE*, and Hao Li

***Abstract*—Data stream processing is a functionality of various AI applications to obtain continuous insights fromdata streams. Serverless edge computing (SEC) is a key solution for implementing data stream processing requests by deploying serverless functions into cloudlets. However, existing data stream processing methods focus more on processing delay, ignoring fault tolerance and complex dependencies among functions, resulting in critical events being missed in the event of any fault and processing inefficiency. Besides, due to the uncertainty of data streams, existing function deployment methods may not be suitable for their newly changed data rates, causing resource waste or shortages. To address these problems, we first propose an optimization framework to enable efficient and fault tolerant function deployment, such that the delay of data stream processing is minimized while meeting its fault tolerant requirements and resource capacity constraints of cloudlets in an SEC network. We then design an online learning algorithm that predicts data rate changes through a multi-timescale machine learning method and proactively adjusts instance locations and numbers to absorb data rate uncertainty. Experimental results in a real test-bed show that our proposed algorithms outperform their counterparts by 13.5% on the average delay and 26.3% on the average fault tolerance.**

Received 22 December 2024; revised 16 October 2025; accepted 14 December 2025. Date of publication 23 December 2025; date of current version 5 February 2026. The work of Zichuan Xu and Qiufen Xia was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62172068 Grant 62172071, in part by Shandong Provincial Natural Science Foundation under Grant ZR2023LZH008, Grant ZR2023LZH013, and Grant ZR2023LZH016, in part by the joint research project with China Coal Research Institute under Grant 2022-3-KJHZ003, and in part by the CCF-Ant Research Fund. *(Corresponding author: Wenzheng Xu.)*

Zichuan Xu and Guangyuan Xu are with the School of Software, Dalian University of Technology, China (e-mail: z.xu@dlut.edu.cn; 32017112@mail.dlut.edu.cn).

Peichen Liu and Qiufen Xia are with the International School of Information Science and Engineering, Dalian University of Technology, China, and also with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, Dalian 116024, China (e-mail: liupeichen@mail.dlut.edu.cn; qiufenxia@dlut.edu.cn).

Weifa Liang is with the Department of Computer Science, City University of Hong Kong, China (e-mail: weifa.liang@cityu.edu.hk).

Wenzheng Xu is with the Department of Computer Science, Sichuan University, Chengdu 610000, China (e-mail: wenzheng.xu@scu.edu.cn).

Pan Zhou is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: panzhou@hust.edu.cn).

Hao Li is with the Institute of Software, China Coal Research Institute, Beijing 100013, China (e-mail: hateif@163.com).

This article has supplementary downloadable material available at https://doi.org/10.1109/TSC.2025.3647054, provided by the authors.

Digital Object Identifier 10.1109/TSC.2025.3647054

***Index Terms*—Serverless edge computing, data stream processing, uncertain data rate, online learning.**

I. Introduction

DRIVEN by the widespread adoption of Internet of Things (IoT) services [1] and the increasing reliance on AI services [2], data stream processing applications have gained prominence in domains of finance, telecommunications, healthcare, logistics, and etc. These applications require extremely-low delays to guarantee timely processing and response to data stream processing requests for effective decision-making. For instance, in finance, real-time analysis of market data streams is crucial for making split-second investment decisions. However, in data stream processing applications, there are faulty events, including semantic anomalies or rule conflicts. Thus, the tolerability to faulty events of such stream processing applications is another vital requirement for users. Fault tolerance requirements refer to maintaining the data stream processing continuity with faulty events, and ensuring the recoverability of the processing. Failing to meet the fault tolerance requirement may interrupt the continuously processing of data streams, thereby missing critical events hidden in data streams [3]. For example, in industrial equipment monitoring, if fault tolerance can not be met, critical data may be missed, causing production interruptions [4].

Mobile edge computing (MEC) empowered by the technique of serverless is emerging as the key technology for data stream processing [5]. Based on the Function-as-a-Service (FaaS), serverless edge computing (SEC) abstracts away the complexity of resource and allows applications to be provided in code-level functions, enabling efficient and fault tolerant data stream processing [6]. Specifically, the cloud service provider of an SEC network usually deploys agile and lightweight serverless functions in cloudlets within the proximity of users to implement their data stream requests. As such, the delay experienced by users is reduced by implementing data stream processing in an SEC network. Besides, through the active standby failover mechanism, the SEC network can deploy lightweight standby instances to meet fault tolerance requirements, as the active functions can quickly switch to standby instances when a fault occurs.

The current function deployment technique of MEC or serverless computing may not solely guarantee the efficiency and fault tolerant requirements of data stream processing requests. Serverless computing in data centers may easily provide enough standby instances using abundant computing resources [7].

However, conventional MEC networks may not ensure fault tolerant data stream processing, due to their capacitated resources and the fact that an MEC network may not provide enough standby instances for all requests. Thus, directly application of such methods to an SEC network may easily cause resource violations [8]. Once the failure of a single function, the whole application will malfunction if there is no standby instance, resulting in higher recovery delays and inefficient function processing. Besides, the data rate of data stream processing applications is uncertain. The uncertain data streams may lead to changes in resource requirements, and pre-placed serverless functions may not be able to adapt to the new resource requirements. Therefore, we need to proactively adjust the number and locations of standby instances, thereby improving the efficiency and fault tolerance of serverless functions with uncertain data rates.

Specifically, enabling efficient and fault tolerant data stream processing in SEC networks poses fundamental challenges. First, the serverless functions of each data stream processing request have complexly dependencies. For example, an autonomous driving service may need an AI pipeline consisting of sensing data from radar, image, vehicle status, and inferences [9]. Due to complex dependencies of code-level functions may incur the high delay during data stream processing if the locations of the functions are not carefully deployed. Therefore, considering the dependencies of functions, it is challenging to implement efficient data stream processing via careful function placements. Second, the fault tolerance is a key factor to guarantee the availability of data stream processing services. For example, in healthcare services on heart diseases or respiratory diseases, any failure or delay in the processing of their stream data may lead to a late diagnosis of a critical disease. The highly distributed and constrained resources of the SEC network further complicate the recovery from faulty functions, because the number of standby instances may not be enough to satisfy fault tolerant requirements of requests due to resource availability [10]. Therefore, given the distributed and limited resources of an SEC network, it is challenging to select an appropriate number of standby instances for each function. Third, the data rate of each data stream processing request may have uncertainty and fluctuate over time. The pre-allocated functions may not be able to meet changing data rates. Specifically, it is necessary to design a suitable data rate prediction algorithm to dynamically predict changes in the data rate and relocate standby instances to avoid resource shortage or waste. However, due to the periodic nature of data rate changes, it is challenging to accurately predict data using traditional prediction methods, which poses a challenge for the relocation of proactive functions.

However, most existing studies either ignored dependencies among serverless functions [11], or did not consider fault tolerant requirements of each data steam requests [12], [13]. Also, there are rarely studies that take data rate uncertainty of data stream processing into consideration in SEC networks [14], [15]. To address the above-mentioned challenges, we consider the problem of data stream processing with uncertain data rates in an SEC network, by minimizing the delay while meeting the fault tolerant requirement of users. We first investigate the problem of the efficient and fault tolerant function placement for data stream processing requests in an SEC network, by considering both fault tolerant requirements of requests and resource capacitated constraints of cloudlets. We then study the problem of proactive data stream processing request adjustment by re-locating serverless functions with different resources.

The main contributions of this paper are as follows.

- We first design an optimization framework for implementing data stream processing requests with given data rates to determine the appropriate number of standby instances and find locations of serverless function and their standby instances for each function in the SEC network, such that the delay is minimized while satisfying fault tolerant requirements of each request and resource constraints of cloudlets in an SEC network.
- Consider that the data rates of data stream requests are dynamically changing and uncertain, we propose a multi-timescale machine learning-based online algorithm to predict changes in data rates and proactively adjust the number and locations of existing standby instances of serverless functions, such that the changes of uncertain data rates can be absorbed.
- We evaluate the performance of the proposed algorithms based on a real test-bed and real-world datasets and results show that our algorithms outperform existing works by 13.5% on the average delay and 26.3% on the average fault tolerance.

The rest of the paper is arranged as follows. Section II reviews related studies. Section III introduces the system model and problem definitions. Section IV proposes an optimization framework for the problem of fault tolerant function placement with given data rates. Section V devises an online learning algorithm for the proactive data stream request adjustment problem with uncertain data rates. Section VI evaluates the performance of the proposed algorithms experimentally with a real test-bed and real-world datasets. Section VII concludes the paper.

II. Related Work

Data stream processing is a fundamental functionality of various stream processing applications. Various systems, architectures and algorithms in serverless computing are designed to enable efficient data stream processing [16], [17], [18]. For example, Song et al. [17] designed a new stream processing serverless system that enables fast reactive scaling of stream queries. These studies however focus on enabling the efficiency of data stream processing in data centers with abundant resources. Direct implementation of such methods into MEC networks may not guarantee the performance of data stream processing, because cloudlets may not have enough resources to process data streams.

Enabling data stream processing in MEC networks has attracted attention in the past few years [8], [12], [19], [20], [21]. For example, Liu et al. [8] proposed a scalable and adaptive stream processing method in an MEC network, to handle concurrent IoT queries in the network. Fu et al. [21] proposed a new framework for edge stream processing with a congestion-aware scheduler. However, the above-mentioned works did not consider an SEC network that implements data stream processing via provisioning a set of serverless functions.

SEC can effectively provide highly-available edge resources for data stream processing applications in code-level functions [10], [18]. Existing studies on SEC mainly focused on generic applications, by reducing the cold start delay and improving the resource utilization of edge nodes [11], [22], [23], [24]. However, none of them focus on data stream processing in SEC networks. For instance, Roy et al. [22] proposed a technique to reduce time costs by strategically warming up a serverless function based on its time-varying probability of the

TABLE I
COMPARISON OF EXISTING WORKS AND OUR WORK

Ref	Network	Request type	Data stream processing	Fault tolerance mechanism	Type of delay				Uncertain data rates
					Processing	Cold-start	Recovering	Transmission	
[16]	Cloud	DAG	✓	✓	✓	×	×	✓	×
[17]	Cloud	DAG	✓	×	✓	×	×	✓	×
[18]	Cloud	DAG	✓	×	✓	×	×	✓	×
[19]	Edge	DAG	✓	×	✓	×	×	✓	×
[12]	Edge	DAG	✓	×	✓	✓	×	✓	×
[8]	Edge	DAG	✓	✓	✓	×	✓	✓	×
[20]	Edge	DAG	✓	×	✓	×	×	✓	×
[21]	Edge	DAG	×	×	✓	×	×	✓	×
[11]	Cloud	Function	×	×	×	✓	×	×	×
[23]	Cloud	Function	×	×	✓	✓	×	×	×
[24]	Edge	Function	✓	×	✓	✓	×	×	×
[25]	Edge	DAG	✓	×	✓	×	×	✓	×
[27]	Cloud	Function	×	✓	✓	×	×	✓	×
[28]	Cloud	Function	×	✓	✓	×	×	✓	×
[7]	Cloud	DAG	×	✓	×	×	×	×	×
[29]	Cloud	Function	×	✓	×	×	×	×	×
Our work	SEC	DAG	✓	✓	✓	✓	✓	✓	✓

[1] Network refers to the scenario studied in this work, which mainly include edge or cloud.
[2] Request type refers to the request type considered for the job, including DAG and single function.
[3] Data stream processing, fault tolerance mechanism, and uncertain data rates refer to whether the work has considered these problem.
[4] Type of delay refers to which delay is considered in the work.

TABLE II
NOTATIONS AND DEFINITIONS

Notations	Definition
$G = (V, E)$	the SEC network operated by a service provider, where V is a set of cloudlets and E is a set of links
$\mathcal{CL}$, cl_j, $C(cl_j)$, and M_j	the set of cloudlets, jth cloudlet, its available memory capacity, and its allocated memory
R, r_m, s_m and t_m	the set of data stream processing requests, and mth data stream processing request and the source function and terminates of r_m
$G'_m = (V'_m, E'_m)$	the DAG of r_m, where V'_m is the set of functions and E'_m is the data flows among the functions
τ, f_l, and $\rho_{m,\tau}$	the time slot in T, the lth function in V'_m, and the data rate of data stream processing request r_m in τ
γ	the unit of memory resource allocated to process a unit data rate
n_m, p_l, and δ_m	the fault tolerance of each function of r_m, the failure probability of f_l, and the fault tolerant requirement of r_m
$Pr(f_l)$	the probability of at least one instance of f_l is available, where $Pr(f_l) = 1 - (p_l)^{n_m+1}$
$d^o_{l,j}$, $d^{proc}_{m,l,j,\tau}$ and d_e	the cold-start delay, the processing delay, and the transmission delay of a unit data rate of path e
$x_{m,l,j,\tau}$	the binary variable that determines whether a function f_l of r_m in cloudlet CL_j in τ
$x'_{m,l,j,\tau}$	the binary variable that determines whether the function f_l of r_m is newly-instantiated in CL_j in τ
α_j	the influence factor that captures cloudlet CL_j's impact of memory allocation on the processing delay
$\mathcal{P}'$, p', $\mathcal{P}$, and p	the set of execution paths from s_m to t_m, and a path in $\mathcal{P}'$ and the set of paths in G to which $\mathcal{P}'$ in G'_m is mapped, and a path in $\mathcal{P}$
$z_{p',p,\tau}$	the indicator variable that shows whether $p' \in \mathcal{P}'$ is mapped to $p \in \mathcal{P}$ in time slot τ
$d^{pt}_{m,\tau}$, $d^{rec}_{m,\tau}$	the sum of processing and transmission delay as well as the recovering delay of r_m in τ
$\phi(p_s)$, $\psi(p_s)$, and $\mathcal{P}'_l$	the starting point and the ending point of sth path p_s, and the paths in DAG G'_m that involves function f_l
κ_m	the state buffer for each function of data stream processing request r_m
$y_{l,\tau}$	the binary variable whether f_l is switched to a standby instance in a new cloudlet in τ
K	the maximum number of standby instances, where $K \leq \lvert V \rvert$
$\mathcal{S}_m$ and p_q	the partition sequence of the request r_m, and the partition of the partition sequence, where $p_q \in \mathcal{S}_m$
$cl^1_{j,q}$, $cl^2_{j,q}$, and $cl^3_{j,q}$	the virtual cloudlets for cloudlet cl_j
G''_m, cl'_j, and $cl_{j''}$	the auxiliary graph of the modified network and the dummy cloudlet for cloudlet cl_j in G'_m and G''_m
ξ_m	the adjustment ratio for the data stream processing request r_m
g_m and g^{max}_m	the number of groups needed and the maximum number of groups for the prediction of data rates of r_m
$\bar{\rho}_m$, $\hat{\rho}_{m,\tau+1}$, and h_g	the average and predicted data rate of r_m in $\tau + 1$, and the size of hidden states for each group

next invocation. Kaffeis et al. [24] devised an efficient algorithm to schedule serverless functions in server clusters. Besides, none of these works considered the fault tolerance problem incurred by serverless function failures.

Fault tolerance is an important method to ensure high availability of services [25]. Existing works have initially addressed the problem of fault tolerance in serverless architecture by proposing algorithms or systems [7], [26], [27], [28]. Most of such investigations are not performed in the context of SEC networks. Besides, most of these proposed methods are not particularly designed for data stream processing. For example, Qi et al. [27] proposed a serverless system for fault-tolerant stateful serverless computing, by leveraging asymmetric logging as a foundation. Zhang et al. [7] proposed a system for fault tolerant stateful serverless workflows. Barrak et al. [28] introduced a peer-to-peer serverless architecture with fault tolerance mechanism for AI model training.

To sum up, although existing studies have proposed various methods to separately optimize data stream processing, function effectiveness, and fault tolerance, there are no works that can address efficient and fault-tolerant data stream processing in SEC networks. Besides, existing works do not consider uncertain data rates. In this work, we consider the problem of data stream processing with uncertain data rates in an SEC network, by adopting active-standby failover mechanisms and minimizing the delay while meeting the fault tolerant requirement. The comparison of existing works and our work is shown in Table I.

III. PRELIMINARIES

In this section, we introduce the system model, notions and notations, and define the optimization problems precisely. All the symbols used in this paper are listed in Table II.

A. System Model

We consider an SEC network $G = (V, E)$ operated by a cloud-native service provider, with a set V of cloudlets providing

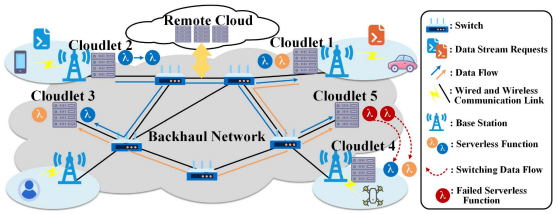


Fig. 1. An example of implementing data stream processing in an SEC network.

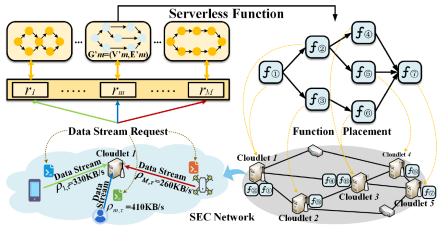


Fig. 2. An example of the placement of a data stream processing request into an SEC network.

services within the proximity of users, and a set E of backhaul links/paths interconnect the cloudlets. Fig. 1 shows an example of the SEC network. Each cloudlet $cl_j \in \mathcal{CL}$ usually consists of several servers or accelerators, and is either attached to an access point or located in the backhaul of the SEC network. Due to the space limitation of each edge location, a cloudlet usually has a certain amount of resources to process data streams. We consider that the resources of each cloudlet are virtualized using cloud-native technologies, and offered to users in terms of *serverless functions*. According to the popular serverless platforms [29], [30], we assume that the memory resource is the key resource type demanded by serverless functions. Specifically, each cloudlet cl_j has memory capacity for executing tasks in serverless functions, which is denoted by $C(cl_j)$.

B. Data Stream Processing in an SEC Network

Nowadays, many AI applications, such as autonomous driving, traffic monitoring, virtual reality, and augmented reality, require continuous data stream processing. For instance, in an autonomous driving system, the traffic events captured by vehicles need to be processed timely such that the system can make real-time decisions [9], [31]. Further, in an AR application, the video stream captured by an AR headset needs to be processed timely, such that virtual objects can be augmented into the stream in no time [32], [33]. Considering the wide existence of data stream processing in many applications, many production-level data analytic systems, such as Apache Kafka [34] or Flink [35], etc., support data stream processing. In such systems, users issue requests to process data streams of their applications. Let r_m be a data stream processing request in an SEC network, which can be implemented in serverless functions that are submitted directly to the SEC network.

Serverless-enabled requests are implemented in code-level functions that are independent on each other. Therefore, a data stream processing request r_m is represented by a Directed Acyclic Graph (DAG) [24], [36] with nodes representing serverless functions and edges denoting the data flows among the serverless functions. Let $G'_m = (V'_m, E'_m)$ be such a DAG of serverless functions, where V'_m is the set of serverless functions of r_m and E'_m is the set of data flows among the serverless functions of r_m. There is a source function and a sink function, and the data stream originates from the source function and terminates at the sink function, which are denoted by s_m and t_m, respectively. Denote by f_l a function in V'_m, which needs to process the data streams that are passed on by its predecessor in DAG G'_m.

C. Uncertain Data Rates of Data Stream Processing Requests

The data stream of a data stream processing request r_m is time-varying and depends on many factors, such as the stability of data sources, the environment, and the network bandwidth [37]. For example, in a medical monitoring system, human digital twins continuously collect the data of patients and update their data stream of the health situation timely. We thus define the rate of the data stream arrivals as *data rate*. Assuming that time is equally divided into equal slots, the data rate of r_m is defined as the volume of data arriving into the system per time unit in time slot τ. Let $\rho_{m,\tau}$ be the data rate of the request r_m in τ. Although the data rate of a data stream processing request can be obtained on its arrival, the future data rates in subsequent time slots may be uncertain. Therefore, the placement of the functions of each data stream processing request needs to be proactive, such that the data stream processing request does not need to be migrated frequently due to changes in data rate. Without loss of generality, we assume that the memory resource required to implement a data stream processing request r_m is proportional to its data rate, i.e., $\gamma \cdot \rho_{m,\tau}$, where γ is the unit of memory resource allocated to process a unit data rate. Fig. 2 shows an example of a data stream processing request and its implementation in an SEC network.

D. Fault Tolerant Requirements of Data Stream Processing Requests

Faults can occur anywhere and at anytime in a network due to natural disasters at the locations of cloudlets, software malfunctions in the system, hardware failures, and etc. In SEC networks, the fault tolerance is crucial to guarantee the timely and continuous processing of data streams, considering that serverless functions usually execute a small piece of code with complex dependencies. Active-standby failover mechanisms [38] are usually adopted to avoid the failures of function execution. In an active-standby failover, an *active function* of data stream processing request r_m is placed into a cloudlet, and a few *standby instances* of the function are placed into other cloudlets [39]. Note that the number of standby instances of each function guarantees the fault-tolerant requirements and the efficiency of the system. Too many standby instances may lead to a high maintenance cost, while too few standby instances may not be able to respond to failures proactively.

Each data stream processing request r_m needs $1 + n_m$ instances for each function of its DAG G'_m to guarantee its fault tolerant requirement, with one active instance and n_m standby instances. We assume that $1 + n_m \leq |V|$, that is, the number of standby instances of each function is no greater than the number of cloudlets in the SEC network. The rationale behind this is that there is at most an active function or a standby instance for each function in each cloudlet. The value of n_m determines the fault tolerance of each function of data stream processing request r_m. Denote by p_l failure probability of function f_l. Notice that p_l can be predicted by machine learning algorithms [40]. Let $Pr(f_l)$ be the probability of at least one instance of function f_l is fault tolerant, then, $Pr(f_l) = 1 - (p_l)^{n_m+1}$. To ensure the quality of

data stream processing, we consider each request r_m has a fault tolerant requirement δ_m given a priori in [38], we have

$$\Pi_{f_l \in V'_m} Pr(f_l) \geq \delta_m. \tag{1}$$

Note that a large number of standby instances determined by n_m may lead to higher resource consumption and consequent resource violations. Thus, the value of n_m needs to make a trade-off between fault tolerant requirements and resource constraints.

E. Delay Model

The delays experienced by a data stream processing request r_m consist of the processing delay by functions of r_m, transmission delays among the functions, and the recovering delays of from a failed function to a standby instance.

The *processing delay* of data stream processing request r_m mainly depends on both its data rate and the memory resource allocated to itself [41]. The reason is that with more memory resource allocated to r_m, more data can be loaded to the memory at the same time, and thus reducing the overhead due to memory management overhead. Denote by M_j the amount of memory allocated by cloudlet CL_j to process a unit data rate. The influence factor that captures cloudlet CL_j's impact on memory allocation on the processing delay is α_j. Besides, the processing delay includes cold start delay, which is related to whether the current cloudlet has idle function instances. We use a binary variable $x_{m,l,j,\tau}$ to represent whether function f_l of r_m is assigned to cloudlet CL_j in time slot τ. On one hand, if there are no idle function instances, a function of r_m has to been instantiated. Consequently, a cold-start delay has to be included in the processing delay. Let $d^o_{l,j}$ be the cold-start delay incurred by instantiating a function f_l in CL_j. Note that the cold-start delay can be given a priori constant by periodic time series prediction methods [42]. On the other hand, there may be some idle function instances. We thus use another binary variable $x'_{m,l,j,\tau}$ to show whether function f_l of r_m is newly-instantiated in CL_j in time slot τ ($x'_{m,l,j,\tau} = 1$) or is assigned to an existing function instance ($x'_{m,l,j,\tau} = 0$). Let $d^{proc}_{m,l,j,\tau}$ be the processing delay by a function f_l of r_m in CL_j at τ, then,

$$d^{proc}_{m,l,j,\tau} = x_{m,l,j,\tau} \cdot \alpha_j \cdot M_j \cdot \rho_{m,\tau} + x'_{m,l,j,\tau} \cdot d^o_{l,j}. \tag{2}$$

The *transmission delay* of the data stream processing request r_m is incurred by the execution path of functions in its DAG. There exist multiple execution paths in DAG G'_m of r_m from its source function s_m to sink function t_m. Let $\mathcal{P}'$ be the execution paths from source function s_m to the sink function t_m in G'_m, and p' be one of such paths. Denote by p and $\mathcal{P}$ a path in the SEC network to which the execution path p' in G'_m is mapped and the set of such paths, respectively. Let p_s be a segment of p, and the number of functions in the execution path p' of G_f is $|p'|$. We use a indicator variable $z_{p',p,\tau}$ to indicate whether $p' \in \mathcal{P}'$ is mapped to $p \in \mathcal{P}$ in time slot τ. The sum of processing and transmission delays of r_m in τ is the maximum delay of the execution paths in $\mathcal{P}'$ incurred by the processing of functions and transmission of data streams in paths of $\mathcal{P}$, which is denoted by $d^{pt}_{m,\tau}$, then,

$$d^{pt}_{m,\tau} = \max_{p' \in \mathcal{P}'} \sum_{p \in \mathcal{P}} z_{p',p,\tau} \cdot \left(\sum_{f_l \in p'} \sum_{CL_j \in p} d^{proc}_{m,l,j,\tau} + \sum_{l=1}^{|p'|-1} \sum_{p_s \subseteq p} x_{m,l,\phi(p_s),\tau} \cdot x_{m,l+1,\psi(p_s),\tau} \sum_{e \in p_s} \rho_{m,\tau} \cdot d_e \right), \tag{3}$$

where $\phi(p_s)$ is the starting point of path p_s, and $\psi(p_s)$ is the ending point of p_s.

When an active function fails, it needs to be switched to a standby instance. The delay during such a process is the *recovering delay* [43]. Let $\mathcal{P}'_l$ be the paths in DAG G'_m that involves function f_l. If f_l is switched to its standby instance, all the execution paths in $\mathcal{P}'_l$ will be affected. That is, given a path $p' \in \mathcal{P}'_l$, the predecessor of f_l in p' needs to forward its processed data stream to the new cloudlet of f_l's standby instance. Besides, since we consider stateful serverless functions, such as processed states, connection states, etc., it needs to forward the states generated to the new destination for later use. We assume that each serverless function maintain a state buffer for the states generated by each function. Let κ_m be the size of the state buffer for each function of data stream processing request r_m. Denote by $p_{l,\tau}$ the switching path from time slot $\tau - 1$ to time slot τ. The recovering delay $d^{rec}_{m,\tau}$ if a failed function is switched to a standby instance in a different cloudlet at τ, we have

$$d^{rec}_{m,\tau} = \sum_{f_l \in V'_m} y_{l,\tau} \cdot \kappa_m \sum_{e \in p_{l,\tau}} d_e, \tag{4}$$

where $y_{l,\tau}$ indicates that whether f_l is switched to a standby instance in a new cloudlet in time slot τ. Therefore, the delay experienced by a data stream processing request r_m is

$$d_{m,\tau} = d^{pt}_{m,\tau} + d^{rec}_{m,\tau}. \tag{5}$$

F. Problem Definitions and NP-Hardness

Given an SEC network $G = (V, E)$, a set R of requests for processing data streams, with each data stream processing request r_m represented by a DAG $G'_m = (V'_m, E'_m)$, assuming that the available resources of the SEC network are enough to admit each single data stream processing request r_m, we consider the following optimization problems.

On the arrival of each data stream processing request r_m, assuming that the data rate of r_m is given as a priori, the *fault tolerant function placement problem with given data rates* in G is aim to find the locations of the serverless functions in V'_m for r_m, determine the number of standby instances, and place the standby instances for each function of r_m into the cloudlets, such that the average delay experienced among all data stream processing requests in R is minimized, subject to the memory resource capacity on each cloudlet and the fault tolerant requirement of r_m.

Although the data rates of data stream processing requests are known in their arrival time slots, the data rates in future time slots are dynamically changing and uncertain. Given a finite time slot with T, the *proactive data stream processing request adjustment problem* in G is to predict the data rates of data stream processing requests in different time scales, proactively adjusting the number n_m of standby instances by re-locating, and assigning them to serverless functions with different memory resources. As such, minimizing the average delay incurred by all data stream processing requests in R while meeting the memory resource capacity of each cloudlet and the fault tolerant

requirement of r_m by proactively adjusting the number and locations of standby instances.

We now analyze the NP-hardness of defined problems.

Theorem 1: Given an SEC network $G = (V, E)$, a request r_m needs to process by functions in a DAG $G'_m = (V'_m, E'_m)$, and a fault tolerant requirement δ_m of r_m, the fault tolerant function placement problem is NP-hard.

Please see Appendix A, available in the supplemental file.

IV. An Efficient Algorithm for the Fault Tolerant Function Placement Problem With Given Data Rates

We now propose an efficient algorithm for the fault tolerant function placement problem, assuming that the data rate of a data stream processing request r_m is given a priori.

A. An Optimization Framework to Find an Appropriate Number of Standby Instances

Due to capacitated resources in cloudlets, the number n_m of standby instances has to be chosen carefully to find a fine tradeoff between the fault tolerance and delay experienced by each data stream processing request. Specifically, if a larger value of n_m, the standby instances may consume too much memory resource, making the active instances being placed into far locations in the SEC network and leading to long delays. However, a smaller value of n_m may lead to the violation of fault tolerant requirement of the data stream processing request. Motivated by this, we design an optimization framework with a customized binary search method to find a proper value of n_m, such that the fault tolerant requirement is met, and the delay is minimized.

We set the minimum and maximum values of n_m to 1 and K, respectively, meaning that each function should have at least one and at most K standby instances. Considering that in each time slot τ, there are already some data stream processing requests in the SEC network, we could keep track of their number of standby instances and their experienced delays in time slot $\tau - 1$. We thus associate each value of n_m with the average delays experienced by data stream processing requests which have n_m standby instances. Let $< n, d >$ be such an associated tuple.

The binary search process proceeds as follows. Firstly, let n_{min} and n_{max} be the minimum and maximum values of the current search round, respectively, where $n_{min} = 1$ and $n_{max} = K$. The value of n_m is set to $\lfloor n_{min} + \frac{n_{max} - n_{min}}{2} \rfloor$. We then check whether the fault tolerant requirement in inequality (1) is met. If not, we set n_{min} to $\lfloor n_{min} + \frac{n_{max} - n_{min}}{2} \rfloor$ and perform the next round of searching; otherwise, we check the delay of n in its corresponding tuple by two cases:

Case 1, if no data stream processing requests adopted this value so far, its delay is NIL. We then implement data stream processing request r_m by invoking the function placement algorithm in the next subsection. The obtained delay is updated to the tuple, and the next round of searching is continued with $n_{min} = n$;

Case 2, otherwise, we obtain the average delay directly from the tuple. If the delay is smaller than the delay obtained in the previous round, we set $n_{min} = n$ and continue for the next round of searching; besides, we set $n_{max} = n$ and continue for the next round. The above procedure continues until $n_{max} \leq n_{min}$ and n_m is set to the current value of n. After obtaining a proper value of n_m for data stream processing request r_m, we then proceed to place the functions of the data stream processing request r_m in the next subsection.

The proposed framework is shown in **Algorithm** 1, referred to as `Fwk`.

Algorithm 1: `Fwk`.

Input: An SEC network $G = (V, E)$, a set R of data stream processing requests for processing data streams, and their fault tolerant requirements.
Output: The placement locations for the active and standby instances of functions of each r_m.

Create a tuple $\langle n, d \rangle$ for each r_m and $1 \leq n \leq K$, where $d \leftarrow NIL$.
for $\tau \leftarrow 1 \cdots T$ **do**
  Let R_τ be the set of data stream processing requests that arrive in time slot τ;
  for each data stream processing request $r_m \in R_\tau$ **do**
    $n_{min} \leftarrow 1$; $n_{max} \leftarrow K$;
    $n_m \leftarrow -1$; /*the number of standby instances for data stream processing request r_m*/
    while $n_{min} \leq n_{max}$ **do**
      $n' \leftarrow \lfloor n_{min} + \frac{n_{max} - n_{min}}{2} \rfloor$;
      if the fault tolerant requirement is met **then**
        Get d from tuple $\langle n_m, d \rangle$;
        if d is NIL **then**
          Invoke **Algorithm** `Heu` in the next subsection to obtain a value for d;
          $n_{min} \leftarrow n_m$ and **continue** the next round of searching;
        if d is smaller than that in the previous round **then**
          $n_{min} \leftarrow n'$ and **continue** the next round of searching;
        else
          $n_{max} \leftarrow n'$ and **continue** the next round of searching;
      else
        $n_{min} \leftarrow \lfloor n_{min} + \frac{n_{max} - n_{min}}{2} \rfloor$;
      $n_m \leftarrow n'$;
    Invoke **Algorithm** `Heu` to place the functions of request r_m with given the selected number n_m of standby instances;

B. Function Placement With a Given Number of Standby Instances

Given a value n_m for each function of the data stream processing request r_m, we now devise an efficient heuristic to find locations for its serverless functions and standby instances. Since there is a recovering delay among active functions and standby instances, they need to be located in close locations. Separately placing a number of copies of DAG G'_m however may increase the recovering delays. To avoid such high recovering delays, we propose a method that jointly places each function in G'_m and its n_m standby instances into the SEC network G. The proposed heuristic consists of three stages: (1) partitioning DAG G'_m into a number of partitions, (2) placing the active functions of r_m into the SEC network, and (3) placing standby instances of r_m.

Stage 1. DAG partitioning: To reduce the transmission delays, we first partition DAG G'_m into a number of sequential partitions, by applying topological sorting on the DAG G'_m of r_m [44]. In

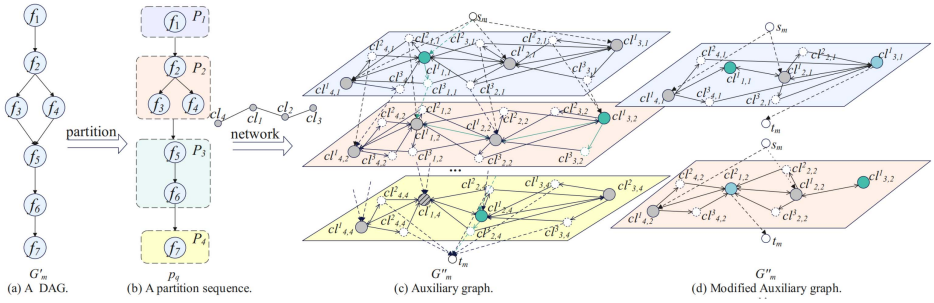


Fig. 3. An example of the constructed auxiliary graph G''_m.

this case, we can avoid massive data stream transmissions among different cloudlets when the functions are scattered in the SEC network. We first perform a depth-first search on G'_m to obtain the dependency among the functions in V'_m. According to the dependencies between functions, we can obtain a topological sort of the functions. We then divide the G'_m into a set of partitions based on the results of topological sorting, each partition consisting of a set of functions that can be instantiated to the same cloudlet. Assuming that the computing resources of each server can handle at least one partition. Therefore, each partition is a minimal schedule unit. Besides, all the partitions form a sequence chain, which represents the execution sequences of the functions in the partitions and referred to as *partition sequence*. An example of the found partition sequence is shown in Fig. 3(b). Let $\mathcal{S}_m$ be the partition sequence of data stream processing request r_m. Each partition $p_q \in \mathcal{S}_m$ has several functions, and the partitions have disjoint sets of functions of DAG G'_m.

Stage 2. Active function placement: We then place the partition sequence $\mathcal{S}_m$ of each data stream processing request r_m to the SEC network G, where the functions of the placed partition sequence will serve as active functions. To this end, we transfer the problem into the problem of finding a shortest path in an auxiliary graph G''_m.

For each partition p_q, we first modify the SEC network G by eliminating the cloudlets that do not have enough memory resources to execute the functions in p_q. Instead of removing the cloudlets with insufficient resources directly, we treat such cloudlets as *dummy cloudlets* with zero processing delay. Let cl'_j be a dummy cloudlet for cloudlet cl_j. The rationale behind this is that it will not execute the functions in p_q.

For the rest cloudlets with enough computing resources, we create three *virtual cloudlets*, denoted by $cl^1_{j,q}$, $cl^2_{j,q}$, and $cl^3_{j,q}$. There is an edge from $cl^1_{j,q}$ to $cl^2_{j,q}$ and the delay of this edge is set to $d(\langle cl^1_{j,q}, cl^2_{j,q}\rangle) = \alpha_j \cdot M_j$. There is also an edge from $cl^1_{j,q}$ to $cl^3_{j,q}$, and its delay is $d(\langle cl^1_{j,q}, cl^3_{j,q}\rangle) = \alpha_j \cdot M_j + \frac{\max_{f_l \in p_q} d^o_{l,j}}{\rho_{m,\tau}}$. In other words, edge $d(\langle cl^1_{j,q}, cl^2_{j,q}\rangle)$ means that executing functions in p_q using existing serverless functions; while $d(\langle cl^1_{j,q}, cl^3_{j,q}\rangle)$ means that using newly-instantiated serverless functions to execute the functions. In addition, if there is an edge from cl_j to $cl_{j'}$ in G, there are two edges $\langle cl^2_{j,q}, cl^1_{j',q}\rangle$ and $\langle cl^3_{j,q}, cl^1_{j',q}\rangle$ in the auxiliary graph. Besides, the transmission delays of such edges are the same as link $\langle cl_j, cl_{j'}\rangle$ in the SEC network G. Besides, there is a bi-directional edge between the first virtual cloudlets of different cloudlets in the layer, i.e., $\langle cl^1_{j,q}, cl^1_{j',q}\rangle$ and $\langle cl^1_{j',q}, cl^1_{j,q}\rangle$. The delays of such edges are set to the transmission delay between cl_j and $cl_{j'}$ in the SEC network.

We then add the nodes and edges of the modified network into the auxiliary graph G''_m. Each of such modified networks within G''_m is referred to as one of its layers. In total, we will have $\mathcal{S}_m$ layers in G''_m.

We proceed by connecting different layers of the auxiliary graph. The overall principle is that there is a directed edge from a virtual cloudlets $cl^2_{j,q}$ and $cl^3_{j,q}$ in partition p_q to its corresponding cloudlet $cl^1_{j,q+1}$ of partition p_{q+1}. The transmission delay of such an edge is set to zero. We then add a common source node and sink node to the built auxiliary graph. There is a directed edge from the common source node to each virtual cloudlet $cl^1_{j,1}$ of the first layer, and the delay of this edge is set to the minimum delay from the source function s_m to cloudlet cl_j in the SEC network G. Similarly, there is an edge from each virtual cloudlet $cl^2_{j,|\mathcal{S}_m|}$ or $cl^3_{j,|\mathcal{S}_m|}$ of the last layer to the common sink node, and the delay of this edge is set to the minimum delay from the cloudlet to the sink function t_m of data stream processing request r_m in G. We then find the shortest path from the source node to the sink node of the auxiliary graph G''_m. The cloudlets in the found shortest path execute the partitions in $\mathcal{S}_m$. Fig. 3 shows an example of the auxiliary graph.

Stage 3. Standby instance placement: Given the placed active functions of G'_m, we now place n_m standby instances for each active function. To this end, we modify the auxiliary graph G''_m and place the standby instances. Suppose the partition p_q is placed into cloudlet cl_j and its consecutive partition p_{q+1} is located in cloudlet $cl_{j'}$. Clearly, the n_m standby instances partition of p_q should be close to both cl_j and $cl_{j'}$. First, the virtual cloudlets whose cloudlet does not have enough computing resources to host a standby instance partition of p_q will be converted to a dummy cloudlet. Second, we re-connect the common source node in G''_m with the virtual cloudlet $cl^1_{j'',1}$ of the rest cloudlets with enough computing resources for an instance of each function in p_q except cl_j. The delay of the edge is set to the delay of transmitting the state buffer of r_m from cl_j to $cl_{j''}$. Also, virtual cloudlets $cl^2_{j'',1}$ and $cl^3_{j'',1}$ are connected to the virtual cloudlet $cl^1_{j',1}$ of $cl_{j'}$, and their delays are set to the delays of transmitting the state buffer from $cl_{j''}$ to $cl_{j'}$. Third, $cl^1_{j',1}$ is connected to the common virtual sink, and its delay is set to zero. After modifying the auxiliary graph G''_m, we then find the shortest path from the common source to the common sink of G''_m, delete the nodes in the path, and repeat until we find n_m disjoint shortest paths. The cloudlet in each of the path hosts a standby instance for each function in partition p_q. The detailed steps are described in **Algorithm** `Heu`.

Algorithm 2: Heu.

Input: An SEC network $G = (V, E)$, a data stream processing request r_m with a number n_m of standby instances of its functions within its DAG G'_m.
Output: The placement locations for the active and standby instances of functions of request r_m.
1: **/* Stage 1: DAG participating*/**
2: Perform a topological sort on the DAG G'_m of r_m;
3: Let $\mathcal{S}_m$ be the partition sequence of r_m;
4: **/* Stage 2: Active function placement*/**
5: Given partitions in $\mathcal{S}_m$, construct auxiliary graph G''_m for r_m;
6: Find a shortest path in auxiliary graph G''_m;
7: The $q+1$ node of the shortest path is the location for functions in p_q;
8: **/* Stage 3: Standby instances placement*/**
9: Modify the auxiliary graph by converting virtual cloudlets into dummy cloudlets if they do not have enough resources to host standby instances for functions in p_q;
10: Let cl_j be the location hosting the active instance of p_q in **stage 2**;
11: Reconnect the common source node to $cl^1_{j'',1}$ and set its delay to the delay from cl_j to $cl_{j''}$;
12: Reconnect $cl^2_{j'',1}$ and $cl^3_{j'',1}$ to $cl^1_{j',1}$ similarly;
13: Find a disjoint shortest path from the common source node to the common sink of G''_m, and place the standby instance in each cloudlet of the disjoint path for each function in p_q;

C. Algorithm Analysis

We now analyze the correctness and performance of the proposed algorithm.

Lemma 1: Algorithm Heu delivers a feasible solution to the fault tolerant function placement problem with a given value of n_m of a data stream processing request r_m, which violates the memory resource constraint by a factor of $O(|V'_m|)$.

Please see Appendix B, available in the supplemental file.

Theorem 2: Given an SEC network $G = (V, E)$, and a data stream processing request r_m needs to process its data streams by functions in a DAG $G'_m = (V'_m, E'_m)$, and a fault tolerant requirement δ_m of each request r_m, the algorithm Fwk delivers a feasible solution to the fault tolerant function placement problem in time $O(|R|\log|V|(|V'_m||V|\log(|V'_m||V|) + |V'_m|(|E| + |V|)))$.

Please see Appendix C, available in the supplemental file.

V. Online Algorithms for Proactive Data Stream Processing Request Adjustment Problem

A. Overview

The data rate of a data stream processing request is uncertain in future time slots. To handle such changes, we will require more memory resources for the active functions of r_m when its data rate increases dramatically; otherwise, we need to release idle resources timely to avoid resource wastage. We observe that standby instances of the functions can be released or re-instantiated if they are not in use. This will not affect the timeliness of data stream processing if no fault happens. Based on this observation, we dynamically adjust the number n_m of standby instances by re-locating them to functions with different memory resources.

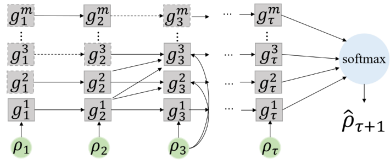


Fig. 4. An example of the MT-LSTM.

B. Online Learning Algorithm

As standby instances are already placed into the SEC network, they are not expected to be re-located too much. Otherwise, the recovering delay may be high if faults happen. We define an *adjustment ratio* as the portion of the n_m standby instances to be re-located. Let ξ_m be the adjustment ratio for r_m. If the data rate of data stream processing requests increases dramatically, a large value of ξ_m is needed, since we rely more on standby instances to absorb the changes of uncertain data rates. As such, the value of ξ_m is highly correlated with the data rate of data stream processing requests. The algorithm has two stages: (1) predicting the data rate of each request and the value of ξ_m of each request, (2) re-locating the standby instances of each request and switching between the active function and standby instance.

Stage 1. Data rate and adjustment ratio predictions: We observe that the data rate $\rho_{m,\tau}$ of data stream processing request r_m in time slot τ depends on the most recent events or some events that happened in different time scales. For example, the environment where the data streams of r_m originate may experience an unexpected event, leading to a surge of the data stream in a short time. Considering that such events may happen on different time scales, the prediction method should predict across different time scales. The multi-timescale long short-term memory neural network (MT-LSTM) is proposed to learn time series with patterns spanning different time scales [45]. An MT-LSTM learns when to forget the historical information of the memory unit, and when to update the memory unit with new information. Fig 4 shows an example of the MT-LSTM.

For each data stream processing request r_m, to learn its data rates across different time scales based on MT-LSTM, it partitions the gates, memory cells, and hidden states into g_m groups with each group being activated at a given length of time scale [45]. Let h_g be the size of hidden states for each group. Each data stream processing request r_m may have multiple lengths of time scales, within each its data rates change. We then determine the number g_m of groups needed for the MT-LSTM of data stream processing request r_m, where each group corresponds to the length of a time scale. The groups can further be categorized from the slowest to the fastest ones. The slowest group corresponds to the longest time scale. However, the MT-LSTM is designed for natural language processing applications, which may not be apply to the prediction of the data rate of each data stream processing request. To make the MT-LSTM suitable for the scenario of data stream processing, we determine the number g_m of time scales for each MT-LSTM of request r_m. Specifically, we observe that a longer time scale usually leads to larger changes in the data rate. Therefore, we use the average

Algorithm 3: `Adj`.

Input: An SEC network $G = (V, E)$, a set R of data stream processing requests, the fault tolerant requirement δ_m of each request r_m, and the number of standby instances of its functions.

Output: Re-locate a part of standby instances of functions in the DAG of each data stream processing request in R.

1: **/*Stage 1: Data rate and adjustment ratio predictions for each** $r_m \in R$***/**
2: **for** $\tau \leftarrow 1 \cdots T$ **do**
3: **for** each data stream processing request $r_m \in R$ **do**
4: Calculate the average data rate of last ζ_m time slots to obtain $\tilde{\rho}_m$.
5: $g_m \leftarrow \lceil \log_2 \tilde{\rho}_m \rceil - 1$;
6: Train the MT-LSTM to obtain the predicted rate value $\hat{\rho}_{m,\tau+1}$ for the next time slot.
7: $\xi_m \leftarrow \frac{\hat{\rho}_{m,\tau+1}}{\rho_{m,\tau}+\hat{\rho}_{m,\tau+1}}$; $n_m \leftarrow \lceil n_m \cdot \xi_m \rceil$;
8: **/*Stage 2: Re-locate the standby instances and switching*/**
9: Invoke stage 3 of **Algorithm** `Heu` to re-locate the standby instances given the adjusted number n_m of standby instances.

data rate per time slot so far to estimate the number of groups needed. Let $\tilde{\rho}_m$ be the average data rate of r_m so far. According to [45], the maximum number of groups for the prediction of data rate of r_m then is

$$g_i^{max} = \lceil \log_2 \tilde{\rho}_m \rceil - 1. \quad (6)$$

This means that the fastest group in the MT-LSTM of data stream processing request r_m has the most frequent changes in data rates, and the slowest group has a stable data rate.

We then train the MT-LSTM to predict the data rate of each request in the next time slot. Specifically, each input of the MT-LSTM of request r_m includes the data rates of its last ζ_m time slots. Let $\hat{\rho}_{m,\tau+1}$ be the predicted data rate of r_m in the next time slot $\tau + 1$. We then determine the adjustment ratio ξ_m of request r_m by

$$\xi_m = \hat{\rho}_{m,\tau+1}/(\rho_{m,\tau} + \hat{\rho}_{m,\tau+1}). \quad (7)$$

The rationale behind is that the more changes of the data rate more standby instances need to be re-located or re-assigned.

Stage 2. Relocating standby instance and switching: Having obtained the values of $\hat{\rho}_{m,\tau+1}$ and ξ_m, we re-locate the standby instances and switch from active instances to standby instances of the functions of r_m by invoking the third stage of **Algorithm** `Heu` with the data rate of $\hat{\rho}_{m,\tau+1}$, by selecting a number of $\lceil n_m \cdot \xi_m \rceil$ standby instances of each function for re-location. The detailed algorithm is given in Algorithm 3, which is referred to as `Adj`.

C. Algorithm Analysis

We now analyze the performance and correctness of the proposed algorithm in the following theorem.

Theorem 3: Given an SEC network $G = (V, E)$, a set R of data stream processing requests, each request $r_m \in R$ needs to process its data stream by functions in a DAG $G'_m = (V'_m, E'_m)$, and a fault tolerant requirement δ_m of each request r_m, algorithm `Adj` delivers a feasible solution to the proactive adjustment problem.

Please see Appendix D, available in the supplemental file.

Theorem 4: The time complexity of the proposed algorithm `Adj` is $O(T|R|(\sum_{g_i^{max}} \frac{|\rho_{m,\tau}|}{g_m} h_g^2 + \log |V|(|V'_m| |V| \log(|V'_m||V|) + |V'_m|(|E| + |V|))))$.

Please see Appendix E, available in the supplemental file.

Theorem 5: The performance bound of the delay for our proposed algorithm `Adj` in current time slot τ is $d_\tau^{adj} \leq \frac{\alpha_j^{max} M_{j,\tau}^{max}}{\alpha_j^{min} M_{j,\tau}^{min}} d_\tau^{opt} + \sum_{p \in p_\tau^{adj}} d_{e,\tau}^{max}$ $(\sum_{p_s \in p} \sum_{e \in p_s} \rho_{m,\tau} + \sum_{f_l \in p} \sum_{e \in p_{l,\tau}} \kappa)$, where $M_{j,\tau}^{max}$ and $M_{j,\tau}^{min}$ are maximum and minimum of amount memory allocated by cloudlet CL_j to process a unit of data rate at t, and $d_{e,\tau}^{max}$ represents the maximum of transmission delay of all paths at t. Also, the fault tolerance gap between our proposed algorithm `Adj` and the optimal solution at time slot τ is $Pr^{adj} \leq 1 - (1 - Pr^{opt})(p_l)^{\hat{\epsilon} \cdot n_m}$, where Pr^{adj} and Pr^{opt} are the probability of fault tolerance at τ obtained by `Adj` and optimal solution, respectively.

Please see Appendix F, available in the supplemental file.

VI. Experiments

In this section, we evaluate the performance of the proposed algorithms against their counterparts based on real datasets.

A. Experimental Settings and Test-Bed Implements

Parameter settings: We consider an SEC network consisting of 50 to 200 cloudlets. The memory capacity of each cloudlet varies from 128 GB to 512 GB [46]. The data rate of each data stream processing request varies from [0.01, 100] MB/s [47]. Based on analyzing the performance metrics of 1,250 virtual machines in Bitbrains' distributed data center [48], we set the amount of memory required for each cloudlet to process a unit data rate to 85 MB, and the impact of each cloudlet memory allocation on processing delay is set within [0.05, 0.09] [41]. Referring to the real serverless computing platform [49], we adjust the memory allocated to the function to [128, 10240] MB, with an increment of 128 MB. The transmission delay for transmitting a unit volume of data along link e is set to [0.1, 0.5] ms [50]. The probabilities of failure for each function instance are in the range of [0.001, 0.003] [51]. The fault tolerant requirement is set at 99.1% to 99.9% [38]. The maximum number of standby instances is set to 3. The results in each figure are based on the average of 10 runs of the proposed algorithms and their benchmarks.

Test-bed: We built a test-bed of the SEC network with both real hardware devices and a real serverless platform, as shown in Fig. 5. The hardware includes a server equipped with an i9-13900HX CPU, 64 G RAM, a PC equipped with an i7-13700 CPU and 32 G RAM, and an Nvidia Jetson nano. The server is used to implement the experimental platform, the PC is used to verify the algorithm `Fwk`, `Adj` and their benchmarks, and the Jetson is used to simulate edge devices and run the MT-LSTM algorithm. We divide the server into four virtual machines with equal computing resources, where consisting of a master node and three worker nodes. All virtual machines use the Ubuntu 24.04 operating system and deploy Kubernetes v1.31[1] with

[1]https://github.com/kubernetes/kubernetes

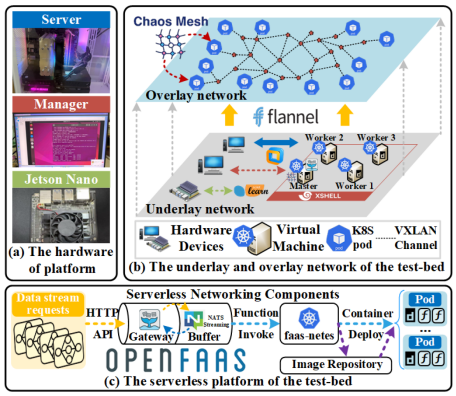


Fig. 5. A test-bed of a serverless platform in an SEC network.

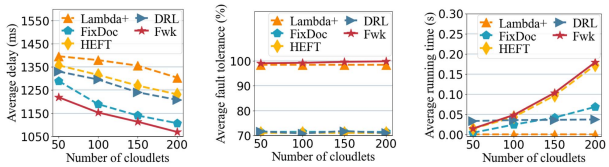
containerd v1.7,[2] consisting of one master node and three worker nodes. The set of pods is activated as cloudlets in the Kubernetes cluster. We use Flannel to build an overlay network of the test-bed, and use K8s-netsim to simulate pod-pod connectivity. We further deployed OpenFaaS [52] as a serverless platform for the test-bed on the master node. We use the functions that measure the impact of load size on processing time when triggering serverless functions in [53] as functions in each data stream processing request. We finally implemented MT-LSTM on Jetson Nano with sci-kit learn [54]. All the experiments are developed by Python 3.9 with multiple libraries, such as request, kuberneter, and dragonk8s, etc.

Dataset: We now introduce the dataset used in our experiment. Data stream processing requests are obtained from the DAG of batch workloads of Alibaba's cluster-trace-v2018 [55], which is widely adopted in the study of serverless function scheduling. We construct DAGs for the data stream processing requests using the dependencies of each task node in the dataset as dependencies among functions. Besides, according to (2), data rate and memory usage show a positive correlation. Therefore, we use the percentage of memory usage in this dataset as the percentage of the data rate. We then obtain the actual data rate value within its range [0.01, 100] MB/s according to the ratio. We also use Huawei Public Cloud Trace 2025 as the value of cold start delay [42]. The cold start delay $d^o_{l,j}$ is randomly selected by the value of Region 1 cold starts in this dataset.

Benchmarks: We compare the proposed algorithms with the following four benchmarks:

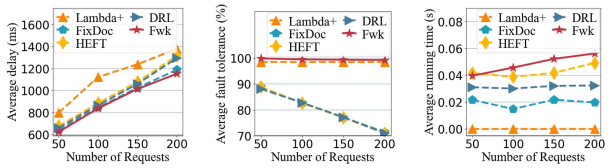
- `FixDoc`: A function placement and DAG scheduling algorithm for serverless applications [14]. It takes into account the dependence of the serverless functions and leverages a dynamic programming-based approach to efficiently place functions, ultimately reducing the processing and transmission delays of the data stream processing requests.
- `Lambda+`: A AWS Lambda's scheduling strategy to improve resource utilization [49], which designs a greedy-based method to place the functions of the same DAG onto the same cloudlet until its capacity is reached. We expanded this work by adding an active-standby mechanism.
- `HEFT`: A heterogeneous-earliest-finish-time algorithm [56]. HEFT first ranks the sequential partitions for a DAG based on the processing and transmission delays of each function and then places the functions into the cloudlet according to their ranks to minimize the processing and transmission delays.
- `DRL`: A deep reinforcement learning-based function placement method [13], where the negative of the delay for each request is set as a reward, and functions are placed into cloudlets to minimize the delay.

[2]https://github.com/containerd/containerd



(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 6. The impact of the different number of cloudlets on the performance of algorithms `Fwk`, `Lambda+`, `FixDoc`, `HEFT`, and `DRL`.



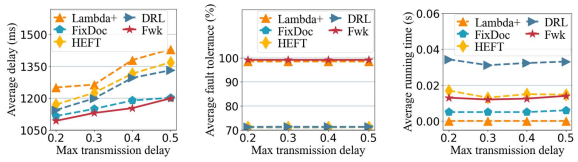
(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 7. The impact of different number of requests on the performance of algorithms `Fwk`, `Lambda+`, `FixDoc`, `HEFT`, and `DRL`.

B. Performance Evaluation

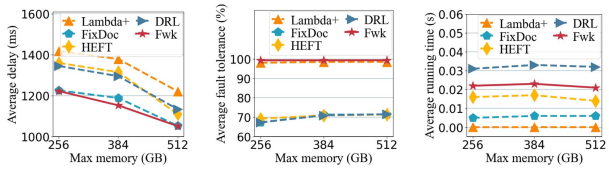
Impact of the number of cloudlets: We first evaluate the performance of algorithms in terms of the average delay, average fault tolerance, and average running time in Fig. 6, by varying the number of cloudlets from 50 to 200 while fixing the number of requests at 200. From Fig. 6(a), we can see that the average delay of `Fwk` is 21.9%, 3.5%, 15.7%, and 12.9% lower than those of `Lambda+`, `FixDoc`, `HEFT`, and DRL respectively, when the network size is 200. The rationale behind this is that `Fwk` finds the locations for functions by constructing auxiliary graphs, minimizing transmission delays between functions as much as possible. Further, we can see that the average delay obtained by the algorithms is decreasing with the growth of the number of cloudlets. The reason is that as the number of cloudlets grows, the functions can be placed in locations with lower processing delays with high probability. In Fig. 6(b), we can see that `Fwk` and `Lambda+` have much higher fault tolerance than their counterparts. Furthermore, `Fwk` can achieve better fault tolerance compared with `Lambda+`, `FixDoc`, `HEFT`, and DRL, due to the binary search mechanism, `Fwk` carefully finds the trade-off between the fault tolerance and the average delay. In Fig. 6(c), we can see that the average running times of `Fwk`, `FixDoc`, and `HEFT` increase with the growth of the number of cloudlets, and `Fwk` achieves the highest average running time. The reason is that the time to find the shortest path of the auxiliary graph becomes longer as the nodes of the auxiliary graph increases exponentially with the growth of the number of cloudlets.

Impact of the number of requests: We then investigated the impact of the number of requests on the performance of algorithms in Fig. 7, by varying the number of requests from



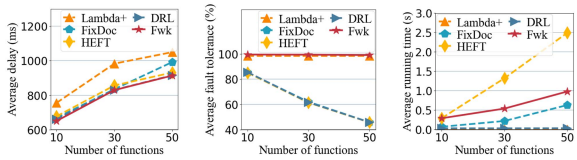
(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 8. The impact of different maximum transmission delay on the performance of algorithms Fwk, Lambda+, FixDoc, HEFT, and DRL.



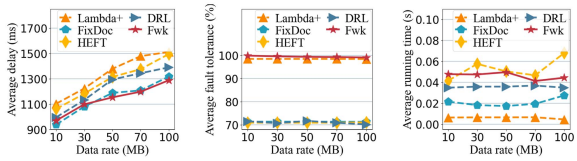
(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 9. The impact of different maximum memory capacity on the performance of algorithms Fwk, Lambda+, FixDoc, HEFT, and DRL.



(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 10. The impact of different number of functions on the performance of algorithms Fwk, Lambda+, FixDoc, HEFT, and DRL.



(a) Average delay (b) Average fault tolerance (c) Average running time

Fig. 11. The impact of different data rates on the performance of algorithms Fwk, Lambda+, FixDoc, HEFT, and DRL.

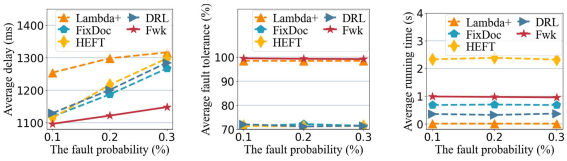
50 to 200 while fixing the number of cloudlets at 100. From Fig. 7(a), we can see that the average delay of all algorithms keeps increasing as the number of requests grows. The reason is that more functions need to be placed when the number of requests increases, which results in higher average delays due to the lack of resources. Specifically, the average delay of Fwk is 19.6%, 3.1%, 14.1%, and 11.3% lower than those of Lambda+, FixDoc, HEFT, and DRL, respectively, when the number of requests is 200. From Fig. 7(b), we can see that the average fault tolerance of Fwk and Lambda+ has remained stable, and other algorithms continue to decrease as the number of requests grows. In particular, the average fault tolerance of Fwk is 27.1% higher than its counterparts when the number of requests is 200. The reason is that Fwk carefully selects the right number and locations for standby instances of a data stream processing request to ensure its fault tolerant requirements. The average running times of algorithms are shown in Fig. 7(c), from which the runtime of Fwk increases slightly as the number of requests increases compared with its counterparts.

Impact of different maximum transmission delay: We further evaluate the performance of algorithms in Fig. 8 by varying the maximum transmission delay for transmitting a unit volume of data along each link from 0.2 to 0.5 ms, while fixing the number of cloudlets at 100 and the number of requests at 200. We first can see that the average delay increases with the growth of the maximum transmission delay of a unit of data in Fig. 8(a), due to the increased transmission delay. Specifically, Fwk delivers the lowest average transmission delay, where outperforms Lambda+, FixDoc, HEFT, and DRL 19.2%, 1.2%, 14.2% and 11.1%, respectively, when the maximum transmission delay is 0.5. The reason is that Fwk considers placing DAG partitioning as close as possible to adjacent cloudlets, which reduces the effect of the maximum transmission delay on the average delay. Further, we can see from the Fig. 8(b) and (c) that all the algorithms show a steady trend in terms of average fault tolerance and average running time, since the maximum transmission delay between two cloudlets has little effect on the fault tolerance of the requests and the average running time of the algorithms.

Impact of different maximum memory capacity: We then investigated the impact of the maximum memory capacity of each cloudlet on the performance of algorithms in Fig. 9 by varying the maximum memory capacity of each cloudlet from 256 to 512 GB, while fixing the network size at 100 and the number of requests at 200. From Fig. 9(a), we can see that the average delay obtained by all algorithms decrease with the growth of the maximum memory capacity of cloudlets, where the average delay of Fwk is 16.2%, 0.9%, 5.6%, and 7.9% lower than those of Lambda+, FixDoc, HEFT, and DRL, respectively, when the max memory capacity is 512. The reason is that with the growth of the memory capacity of each cloudlet, the functions in each request can be placed into cloudlets with lower processing and transmission delays. In Fig. 9(b), we can see that the average fault tolerance of cloudlets remains stable, because the maximum memory capacity does not impact the fault tolerance of each cloudlet directly. In Fig. 9(c), we can see that the memory capacity has essentially no effect on the running time of the algorithms. The reason is that the average running time is related to the number of requests, cloudlets, and functions, instead of memory capacity, as shown in Theorem 2.

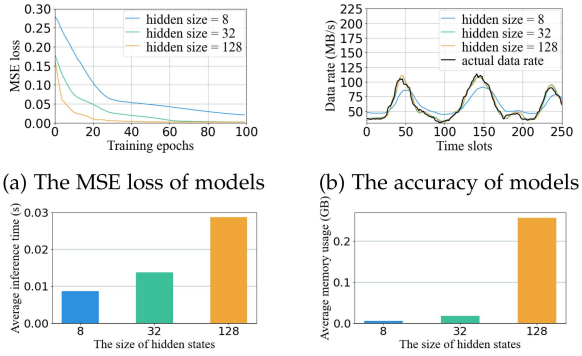
Impact of the number of functions: We now investigated the impact of the number of functions of each request on the performance of algorithms in Fig. 10, by varying the number of functions of each request from 10 to 50 while fixing the number of cloudlets at 100 and the number of requests at 200. From Fig. 10(a) and (c), we can see that the average delays and average running times increase when the number of functions grows from 10 to 50. Specifically, Fwk achieves the lowest average delay when the number of functions is 50. The rationale behind this is that as the number of functions of each request increases, the memory resource in a cloudlet becomes more strained, potentially leading to more dispersed function placement for each request. We can also see from Fig. 10(b) that the fault tolerance of FixDoc, HEFT, and DRL has decreased significantly, while that of Fwk remains stable. The reason is that the active-standby failover mechanism ensures the fault tolerant requirement for each request.

Impact of different data rates: We also evaluate the impact of different data rates of requests on the performance of algorithms in Fig. 11, by varying the maximum data rate of a request from



(a) Average delay (b) Average fault tolerance (c) Average running time

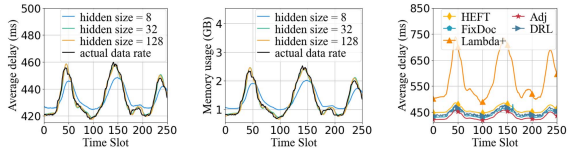
Fig. 12. The impact of different fault probabilities on the performance of algorithms Fwk, Lambda+, FixDoc, HEFT, and DRL.



(a) The MSE loss of models (b) The accuracy of models

(c) Average inference time (d) Average memory usage

Fig. 13. The performance of MT-LSTM with different sizes of hidden states deployed on Jetson Nano and scikit-learn.



(a) Average delay of algorithm Adj (b) The memory usage of algorithm Adj (c) Average delay with $h_g = 32$

Fig. 14. The impact of the uncertain data rate on the performance of algorithms Adj, Lambda+, FixDoc, HEFT, and DRL.

10 Mb/s to 100 Mb/s while fixing the network size at 100 and the number of requests at 200. From Fig. 11(a), we can see that the average delays of algorithms Fwk and its counterparts increase with the growth of the maximum data rate, because transmitting more data leads to higher delay. Specifically, we can observed that our proposed algorithm Fwk outperforms Lambda+, FixDoc, HEFT, and DRL 17.4%, 2.3%, 16.1%, and 7.9%, receptively, when maximum data rate is 100 MB/s. The average fault tolerance of requests and average running time by the algorithms are shown in Fig. 11(b) and (c). We can see that Fwk achieves the highest average fault tolerance when the max data rate is 100. Besides, we can see that the running time of Fwk is stable. The reasons are similar to the analysis in Fig. 9.

Impact of different fault probabilities: We then study the impact of the different fault probabilities of each request on the performance of algorithms in Fig. 12, by varying the fault probabilities of each function from 0.1% to 0.3% while fixing the number of cloudlets at 100 and the number of requests at 200. It can be observed from Fig. 12(a) that the average delays of all algorithms keep increasing as the fault probability grows. This is because more faults lead to higher recovery delays. Meanwhile, we can also see that the growth trends of Fwk and Lambda+ are significantly slower than those of HEFT, FixDoc, and DRL. This is because Fwk and Lambda+ adopt an active-standby failover mechanism, resulting in lower recovery delay. In Fig. 12(b) and (c), we can see that the average fault tolerance and running time of algorithms remain stable. The rationale behind this is similar to the reason in Fig. 11, we do not repeat it here for the sake of space.

The performance of MT-LSTM: We now evaluate the performance of MT-LSTM with different sizes of hidden states in terms of the MSE loss, accuracy, average inference time, and average memory usage in Fig. 13, by varying the size of hidden states from 8 to 128. Notice that the experiments on the performance of MT-LSTM were implemented using scikit-learn and run on Jetson nano. From Fig. 13(a), we can see that as the number of training epochs increases, the MSE loss decreases significantly. Meanwhile, the larger the size of hidden states, the greater the decrease in MSE loss. Besides, due to the difference in MSE loss, we can see from Fig. 13(b) that the loss is converging, and the higher the accuracy of the models. The reason is that the size of the hidden states means that the more extensive the hidden state information of the model, resulting in the better the performance. Fig. 13(c) and (d) show the inference time and memory usage of the models with different hidden sizes. We can see that as the size of hidden states increases, inference time and memory usage increase significantly. The reason is that the time complexity of MT-LSTM is related to the size of hidden states, and the theoretical analysis is shown in Theorem 4.

Impact of the uncertain data rate: We finally investigate the performance of the algorithm within a finite time horizon with 250 time slots in Fig. 14 with the network size fixed at 50, the number of requests sets at 100, and the data rate sets according to the data rate in MT-LSTM with different sizes of hidden state and actual data rate in the dataset [55]. Fig. 14(a) and (b) show that the average delay and memory usage for process data rates of algorithm Adj. We can see that the average delay and memory usage deviate significantly from the actual data rate when the size of hidden states is 8. Therefore, we typically select MT-LSTM models with a hidden state size of 32 or 128 in practice. Besides, we evaluated the performance of algorithms Adj, Lambda+, FixDoc, HEFT, and DRL when the size of hidden states is set as 32. It can be seen from Fig. 14(c) that the trend of average delay is similar to the trend of data rate, and algorithm Adj achieves the best results. The reason is that Adj can predict the data rate accurately and re-place the standby instances in a more appropriate location.

VII. Conclusion

In this paper, we studied the problems of efficient and fault tolerant data stream processing with uncertain data rates in an SEC network. We first proposed an optimization framework for the problem via a customized binary search process and an auxiliary graph construction method. If the data rates of data stream processing requests are dynamically changing and uncertain, we studied the proactive adjustment problem. We then designed a novel online learning algorithm to predict the data rates based on MT-LSTM and dynamically adjust the number and locations of standby instances to absorb uncertain data rates. We also conducted experiments based on a real test-bed with real world datasets. Experimental results showed that the performance of each data stream processing request outperforms its counterparts by 13.5% on the average delay and 26.3% on the average fault tolerance.

REFERENCES

[1] B. B. Sinha and R. Dhanalakshmi, "Recent advancements and challenges of Internet of Things in smart agriculture: A survey," *Future Gener. Comput. Syst.*, vol. 126, pp. 169–184, 2022.

[2] Y. Yang, K. Peng, S. Wang, X. Xu, P. Xiao, and V. C. Leung, "Fairness-aware incentive mechanism for multi-server federated learning in edge-enabled wireless networks with differential privacy," *IEEE Trans. Mobile Comput.*, vol. 24, no. 10, pp. 9919–9933, Oct. 2025.

[3] K. Peng, B. Zhao, M. Bilal, and X. Xu, "Reliability-aware computation offloading for delay-sensitive applications in MEC-enabled aerial computing," *IEEE Trans. Green Commun. Netw.*, vol. 6, no. 3, pp. 1511–1519, Sep. 2022.

[4] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques—Part I: Fault diagnosis with model-based and signal-based approaches," *IEEE Trans. Ind. Electron.*, vol. 62, no. 6, pp. 3757–3767, Jun. 2015.

[5] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Trans. Serv. Comput.*, vol. 16, no. 2, pp. 1522–1539, Mar./Apr. 2023.

[6] F. Tütüncüoğlu, S. Jošilo, and G. Dán, "Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing," *IEEE/ACM Trans. Netw.*, vol. 31, no. 2, pp. 695–709, Apr. 2023.

[7] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proc. OSDI '20*, 2020, pp. 1187–1204.

[8] P. Liu, D. D. Silva, and L. Hu, "DART: A scalable and adaptive edge stream processing engine," in *Proc. USENIX ATC '21*, 2021, pp. 239–252.

[9] H. Dui, S. Zhang, M. Liu, X. Dong, and G. Bai, "IoT-enabled real-time traffic monitoring and control management for intelligent transportation systems," *IEEE Internet Things J.*, vol. 11, no. 9, pp. 15842–15854, May 2024.

[10] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, pp. 1–61, 2023.

[11] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," *IEEE Internet Things J.*, vol. 10, no. 5, pp. 3917–3927, Mar. 2023.

[12] J. Xu, B. Palanisamy, Q. Wang, H. Ludwig, and S. Gopisetty, "Amnis: Optimized stream processing for edge computing," *J. Parallel Distrib. Comput.*, vol. 160, pp. 49–64, 2022.

[13] Y. Yang et al., "Multi-objective deep reinforcement learning for function offloading in serverless edge computing," *IEEE Trans. Serv. Comput.*, vol. 18, no. 1, pp. 288–301, Jan./Feb. 2025.

[14] L. Liu et al., "Dependent task placement and scheduling with function configuration in edge computing," in *Proc. ACM IWQoS '19*, 2019, pp. 1–10.

[15] S. Deng et al., "Dependent function embedding for distributed serverless edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2346–2357, Oct. 2022.

[16] Z. Cai, Z. Chen, X. Chen, R. Ma, H. Guan, and R. Buyya, "SPSC: Stream processing framework atop serverless computing for industrial Big Data," *IEEE Trans. Cybern.*, vol. 54, no. 11, pp. 6509–6517, Nov. 2024.

[17] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in *Proc. USENIX ATC '23*, 2023, pp. 301–314.

[18] S. R. Poojara, C. K. Dehury, P. Jakovits, and S.N. Srirama, "Serverless data pipeline approaches for iot data in fog and cloud computing," *Future Gener. Comput. Syst.*, vol. 130, pp. 91–105, 2022.

[19] G. Amarasinghe, M. D. D. Assunção, A. Harwood, and S. Karunasekera, "A data stream processing optimisation framework for edge computing applications," in *Proc. IEEE 21st Int. Symp. Real-Time Distrib. Comput.*, 2018, pp. 91–98.

[20] B. Ramprasad et al., "Shepherd: Seamless stream processing on the edge," in *Proc. IEEE/ACM SEC '* 22, 2022, pp. 40–53.

[21] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "EdgeWise: A better stream processing engine for the edge," in *Proc. USENIX ATC '19*, 2019, pp. 929–946.

[22] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc. ACM ASPLOS '22*, 2022, pp. 753–767.

[23] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proc. ACM HPDC '21*, 2021, pp. 239–251.

[24] S. Khare et al., "Linearize, predict and place: Minimizing the makespan for edge-based stream processing of directed acyclic graphs," in *Proc. IEEE/ACM SEC '* 19, 2019, pp. 1–14.

[25] A. Sari and M. Akkaya, "Fault tolerance mechanisms in distributed systems," *Int. J. Communications, Netw. System Sci.*, vol. 8, no. 12, pp. 471–482, 2015.

[26] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proc. ACM EuroSys '20*, 2020.

[27] S. Qi, X. Liu, and X. Jin, "Halfmoon: Log-optimal fault-tolerant stateful serverless computing," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 314–330.

[28] A. Barrak, M. Jaziri, R. Trabelsi, F. Jaafar, and F. Petrillo, "Spirt: A fault-tolerant and reliable peer-to-peer serverless ml training architecture," in *Proc. IEEE 23rd Int. Conf. Softw. Quality, Reliability, Secur.*, 2023, pp. 650–661.

[29] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.

[30] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX ATC '18*, 2018, pp. 133–146.

[31] J. Yuan, M. A. Abdel-Aty, L. Yue, and Q. Cai, "Modeling real-time cycle-level crash risk at signalized intersections based on high-resolution event-based data," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 11, pp. 6700–6715, Nov. 2021.

[32] R. Viola, A. Martıń, M. Zorrilla, J. Montalbán, P. Angueira, and G.-M. Muntean, "A survey on virtual network functions for media streaming: Solutions and future challenges," *ACM Comput. Surv.*, vol. 55, no. 11, pp. 1–37, 2023.

[33] M. Xu et al., "A full dive into realizing the edge-enabled metaverse: Visions, enabling technologies, and challenges," *IEEE Commun. Surveys Tuts.*, vol. 25, no. 1, pp. 656–700, Firstquarter 2023.

[34] "Apache kafka," 2024. [Online]. Available: https://github.com/apache/kafka

[35] "Apache flink," 2024. [Online]. Available: https://github.com/apache/flink

[36] B. Zhao, K. Peng, K. Zhang, H. Sun, Z. Tu, and D. Chu, "Serflow: A multi-stage service-enhanced mechanism for workflow applications in cpss with end-edge-cloud collaboration," *IEEE Internet Things J.*, vol. 12, no. 16, pp. 33792–33813, Aug. 2025.

[37] Z. Xu et al., "Learning-driven algorithms for responsive AR offloading with non-deterministic rewards in metaverse-enabled MEC," *IEEE/ACM Trans. Netw.*, vol. 32, no. 2, pp. 1556–1572, Apr. 2024.

[38] "Aws lambda sla," 2022. [Online]. Available: https://aws.amazon.com/cn/lambda/sla/

[39] Y. Bouizem, N. Parlavantzas, D. Dib, and C. Morin, "Active-standby for high-availability in faas," in *Proc. 6th Int. Workshop Serverless Comput.*, 2020, pp. 31–36.

[40] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting performance and cost of serverless computing functions with saaf," in *Proc. 2020 IEEE DASC/PiCom/CBDCom/CyberSciTech*. IEEE, 2020, pp. 640–649.

[41] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX ATC '20*, 2020, pp. 205–218.

[42] A. Joosen et al., "Serverless cold starts and where to find them," in *Proc. EuroSys*, 2025, pp. 938–953.

[43] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 64–78.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2022.

[45] P. Liu, X. Qiu, X. Chen, S. Wu, and X.-J. Huang, "Multi-timescale long short-term memory neural network for modelling sentences and documents," in *Proc. 2015 Conf. Empirical Methods Natural Lang. Process.*, 2015, pp. 2326–2335.

[46] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput.*, 2020, pp. 41–50.

[47] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An IoT benchmark for distributed stream processing systems," *Concurrency Computation: Pract. Experience*, vol. 29, no. 21, pp. 1–22, 2017.

[48] "Gwa–t–12 bitbrainsm," 2023. [Online]. Available: http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains/

[49] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand container loading in AWS lambda," in *Proc. USENIX ATC '23*, 2023, pp. 315–328.

[50] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.

[51] "Service status," 2023. [Online]. Available: https://cloudharmony.com/status-of-dns-and-compute-and-storage
[52] D.-N. Le, S. Pal, and P. K. Pattnaik, "Openfaas," in *Cloud Computing Solutions: Architecture, Data Storage, Implementation and Security*. Hoboken, NJ, USA: Wiley, 2022 .
[53] F. Carpio, M. Michalke, and A. Jukan, "Benchfaas: Benchmarking serverless functions in an edge computing network testbed," *IEEE Netw.*, vol. 37, no. 5, pp. 81–88, Sep. 2023.
[54] F. Pedregosa et al., "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[55] Q. Weng et al., "Beware of fragmentation: Scheduling GPU-Sharing workloads with fragmentation gradient descent," in *Proc. USENIX ATC '23*, 2023, pp. 995–1008.
[56] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

Zichuan Xu (Member, IEEE) received the BSc and ME degrees in computer science from the Dalian University of Technology in China in 2008 and 2011, respectively, and the PhD degree in computer science from the Australian National University in 2016. From 2016 to 2017, he was a research associate with the Department of Electronic and Electrical Engineering, University College London, U.K.. He is currently a professor in School of Software with the Dalian University of Technology. His research interests include mobile edge computing, serverless computing, network function virtualization, Internet of Things, and algorithm design. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems.

Peichen Liu (Graduate Student Member, IEEE) is currently working toward the PhD degree with the School of Software, Dalian University of Technology, Dalian, China. His research interests include mobile edge computing, reinforcement learning, approximation algorithm design, and AI infrastructure/system.

Qiufen Xia (Member, IEEE) received the BSc and ME degrees in computer science from the Dalian University of Technology in China, in 2009 and 2012, respectively, and the PhD degree in computer science from the Australian National University in 2017. She is currently an associate professor with the Dalian University of Technology. Her research interests include mobile cloud computing, query evaluation, Big Data analytics, Big Data management in distributed clouds, and cloud computing.

Weifa Liang (Fellow Member, IEEE) received the BSc degree computer science from Wuhan University, China in 1984, the ME degree computer science from the University of Science and Technology of China in 1989, and the PhD degree computer science from the Australian National University in 1998. He was a professor in the Australian National University. He is currently a professor in the Department of Computer Science with the City University of Hong Kong. His research interests include design and analysis of energy efficient routing protocols for Internet of Things, mobile edge computing (MEC), network function virtualization (NFV), software-defined networking (SDN), approximation algorithms, combinatorial optimization, and graph theory. He is currently an associate editor in the Editorial Board of *IEEE Transactions on Communications*.

Guangyuan Xu is currently working toward the PhD degree with the School of Software, Dalian University of Technology, Dalian, China. His current research interests include mobile edge computing, reliability computing, satellite edge computing, and resource allocation.

Wenzheng Xu (Member, IEEE) received the BSc, ME and PhD degrees in computer science from Sun Yat-Sen University, Guangzhou, China, in 2008, 2010, and 2015, respectively. He currently is an associate professor with the Sichuan University and was a visitor with the Australian National University. His research interests include wireless ad hoc and sensor networks, mobile computing, approximation algorithms, combinatorial optimization, online social networks, and graph theory.

Pan Zhou (Senior Member, IEEE) received the BS degree in the Advanced Class of the Huazhong University of Science and Technology, the MS degree in the Department of Electronics and Information Engineering from HUST, Wuhan, China, in 2006 and 2008, respectively, and the PhD degree in the School of Electrical and Computer Engineering with the Georgia Institute of Technology (Georgia Tech), in 2011, Atlanta, USA. He is currently a full professor and the PhD advisor with Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, HUST.. He is also an associate editor for *IEEE Transactions on Network Science and Engineering*. His current research interest includes: security and privacy, Big Data analytics, machine learning, and information networks.

Hao Li is currently an associate researcher in the Institute of Software at China Coal Research Institute. Hao's research interests include coal mine intellectualization, automation of fully mechanized mining face, edge computing, AI and Internet of Things.