

FaaS Deliver: Cost-Efficient and QoS-Aware Function Delivery in Computing Continuum

Guangba Yu , *Student Member, IEEE*, Pengfei Chen , Zibin Zheng , *Fellow, IEEE*, Jingrun Zhang , Xiaoyun Li , and Zilong He , *Graduate Student Member, IEEE*

Abstract—Serverless Function-as-a-Service (FaaS) is a rapidly growing computing paradigm in the cloud era. To provide rapid service response and save network bandwidth, traditional cloud-based FaaS platforms have been extended to the edge. However, launching functions in a heterogeneous computing continuum (HCC) that includes the cloud, fog, and the edge brings new challenges: determining where functions should be delivered and how many resources should be allocated. To optimize the cost of running functions in the HCC, we propose an adaptive and efficient function delivery engine, named *FaaS Deliver*, which automatically unearths a cost-efficient function delivery policy (FDP) for each function, including the FaaS platform selection and resource allocation. Real system implementation and evaluations in a practical HCC demonstrate that *FaaS Deliver* can unearth the most cost-efficient FDPs from among 180,200 FDPs after a few trials. *FaaS Deliver* reduces the average cost of function execution from 38% to 78% compared to some state-of-the-art approaches.

Index Terms—Computing continuum, function as a service, online learning, serverless computing.

I. INTRODUCTION

SERVERLESS Function-as-a-Service (FaaS) is a rapidly evolving cloud computing paradigm for deploying cloud applications. It promises to provide a simplified programming model that facilitates the creation of cloud applications in a more accessible and cost-effective way. Many public cloud platforms have released their FaaS services, such as AWS Lambda [1], Azure Functions [2], and Google Cloud Functions [3]. In the FaaS paradigm, most operational concerns in FaaS platforms, such as provisioning and managing servers, are delegated to

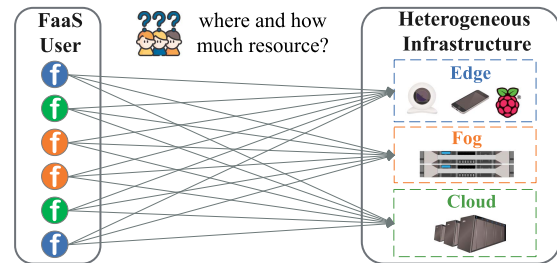





Fig. 1. Functions delivery problem in the heterogeneous computing continuum (HCC). , ,  are FaaS functions suitable for the platforms of edge, fog, and cloud, respectively. FaaS users need to determine where functions should be delivered and how many resources should be allocated.

FaaS platform providers [4], [5]. Instead, FaaS users can focus on the logic programming of functions, which provides generalized expressions of computational tasks in heterogeneous environments.

Advances in FaaS frameworks make it easier to integrate resources from cloud computing, away from users, to fog and edge computing, which are closer to users [6], [7], [8]. Nowadays, FaaS functions can be deployed on a heterogeneous computing continuum (HCC) that contains a multitude of heterogeneous resources, ranging from the cheap and constrained devices of edge platforms to the modestly priced and mid-range device of fog platforms, to the cloud platforms with extensive resources [9], [10]. Such a feature provides more robust adaptability for FaaS to handle different types of workload [11]. For example, a simple time-sensitive task, such as generating a QRCode, can be delivered to an edge device to reduce the time of network transmission and provide a better user experience.

Although HCC offers a promising future of more generic FaaS, an intuitive question from FaaS users is *where functions should be delivered and how many resources should be allocated*. We refer to this problem as the *Function Delivery* problem, as shown in Fig. 1. We identified three fundamental factors affecting the cost of FaaS functions, namely function placement, CPU allocation, and memory allocation. In this study, we refer to the combination of these three factors as the *Function Delivery Policy (FDP)*, which is denoted as *(platform, CPU, memory)*. We aim to determine the optimal FDP that resolves the function delivery problem while minimizing users' financial costs. We define the optimal FDP as the policy with the lowest cost that satisfies the Service Level Objectives (SLOs) of performance.

Manuscript received 23 December 2022; revised 10 April 2023; accepted 7 May 2023. Date of publication 10 May 2023; date of current version 8 October 2023. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B010165002, in part by the National Natural Science Foundation of China under Grant 62272495, in part by the Basic and Applied Basic Research of Guangzhou under Grant 20202030328, in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054, in part by CCF-Lenovo Blue Ocean Research Fund, the Project of Core Blockchain of Ministry of Education of China under Grant 2020KJ010801, and in part by the Fundamental Research Funds for the Central Universities, Sun Yat-sen University under Grant 22qntd1004. Recommended for acceptance by E. Damiani. (Corresponding author: Pengfei Chen.)

Guangba Yu, Pengfei Chen, Jingrun Zhang, Xiaoyun Li, and Zilong He are with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510275, China (e-mail: yugb5@mail2.sysu.edu.cn; chenpf7@mail.sysu.edu.cn; zhangjr35@mail2.sysu.edu.cn; lixy223@mail2.sysu.edu.cn; hezlong@mail2.sysu.edu.cn).

Zibin Zheng is with the School of Software Engineering, Sun Yat-sen University, Guangzhou 510275, China (e-mail: zhizbin@mail.sysu.edu.cn).

Digital Object Identifier 10.1109/TSC.2023.3274769

As we will elaborate in Section II-B, launching functions with insufficient resources or inappropriate platforms can lead to SLO violations or even failures. On the other hand, allocating more resources to a function than it requires incurs unnecessary costs. Therefore, choosing the appropriate FDP for each function is crucial to make the fullest use of the HCC's capabilities and minimize cost. As the majority of FaaS functions are recurring tasks, the cost savings from a proper FDP are even more significant. However, for FaaS users, delivering functions to HCC with the cheapest policy while satisfying SLO is at its core an intractable NP-hard problem [12]. A straightforward approach to find the optimal FDP is to explore all FDPs in the HCC like Costless [13]. However, it is not applicable in practice since exploring all FDPs for each function is prohibitively expensive. Existent function delivery engines like COSE [14] try to deliver functions to heterogeneous FaaS platforms, but they do not take into account the heterogeneity of FaaS platforms.

FaaS Deliver Framework: In order to optimize the cost of launching functions in the HCC, we propose an adaptive and efficient function delivery engine, named *FaaS Deliver*,¹ which automatically unearths a cost-efficient FDP to guide FaaS users in configuring functions. The key idea of *FaaS Deliver* is to build a performance model based on Tree-structured Parzen Estimator (TPE) [15] that allows us to distinguish optimal or near-optimal FDPs from the rest in the tree-structured search space (details in Section III-C). To integrate TPE into *FaaS Deliver* for the FaaS paradigm, we perform several customizations (details in Section III-C2): (i) dynamic resource space management; (ii) employing multivariate TPE; (iii) downward-closure based memory pruning; (iv) SLO violations penalization. For each function invocation, *FaaS Deliver* employs the customized TPE to generate an FDP that is expected to be less costly in launching the function. The execution results are then fed back to the TPE to update the online learning model incrementally. In addition, *FaaS Deliver* applies a heuristic transfer learning approach to narrow the search space when updating functions.

Overall, the key contributions of our work are as follows:

- We provide in-depth insights into running FaaS functions in HCC. These insights provide some guidance for designing an efficient and effective function delivery engine in the HCC (Section II-B).
- Building on the guidelines, we propose a lightweight function delivery engine, named *FaaS Deliver*, that automatically unearths the most cost-efficient FDP with adaptation to different functions and computing devices based on the customized adaptive TPE approach (Section III).
- We design and implement the *FaaS Deliver* prototype. Real system implementation and evaluations in a practical HCC demonstrate that *FaaS Deliver* can unearth the most cost-efficient FDPs from among 180,200 FDPs after a few trials. *FaaS Deliver* reduces the average cost of function execution from 38% to 78% compared to some state-of-the-art approaches (Section IV-B).

The rest of this study is organized as follows. The background and motivation are introduced in Section II. Section III shows

the basic idea and detailed design of *FaaS Deliver*. Section IV delineates the experimental setup and presents the results obtained. Some discussions on the generality and limitations of *FaaS Deliver* is conducted in Section V. The related work is examined in Section VI, followed by the conclusion of this study in Section VII.

II. BACKGROUND AND MOTIVATION

A. Background

Heterogeneous Computing Continuum: As the pillar of modern IT systems, a large number of heterogeneous computing devices interconnected via network interaction with each other. This infrastructure is known as the heterogeneous computing continuum (HCC) including the edge platforms [7], [16], the fog platforms [8], [17], the cloud platforms [18], [19] and other devices. There are some differences between devices in HCC. The cloud achieves the best scalability due to its massive scale, while the fog and edge usually possess limited computational resources. However, the edge has a lower inter-cluster transmission latency (less than 1 millisecond (ms)) than the fog (less than 10 ms) and the cloud (more than 10 ms) because the edge is closer to end users [20]. Therefore, running applications in HCC is quite attractive since it supports diverse types of applications by leveraging the advantages of different computing devices.

Function Delivery Network: Most existing FaaS functions run only on the homogeneous cloud [4], [21] or only on the edge [22], [23]. Attracted by the advantages of HCC, Function Delivery Network (FDN) is proposed to extend FaaS to HCC to support functions with varying computational requirements. FDN is a network of distributed heterogeneous platforms analogous to Content Delivery Networks [9]. A target platform in the FDN is a cluster of heterogeneous devices and a FaaS platform on top of it. In the FDN, delivering a given function can be regarded as launching the function on a target platform with specifically configured resources to handle a request. In contrast to current heterogeneous platforms, FDN needs to consider the different performance of heterogeneous platforms as well as the network delay between platforms and end users.

An Example of FDN Use Cases: Cognitive mobile personal assistants monitor health data via biosensors and can predict and raise alerts for urgent situations of a patient, like low blood sugar levels [24]. Critical concerns in this case are prediction accuracy and inference latency, and the latter one can be reduced by launching functions in HCC [22]. The process of model training is split into two phases. The service provider first trains a basic model on representative samples of the entire population. This is a resource-intensive process that should be scheduled to cloud devices. After that, accounting for patient-specific patterns, the basic model is transmitted to fog devices and refined using transfer learning and patient-specific data. This refined model is then served on edge devices (i.e., the patient's device), thereby enabling a low-latency inference. Further, since the model training and inference are not continuous, the event-driven FaaS paradigm that allocates resources to a function on demand is better than a long-running application.

¹<https://github.com/yuxiaoba/FaaSDeliver>

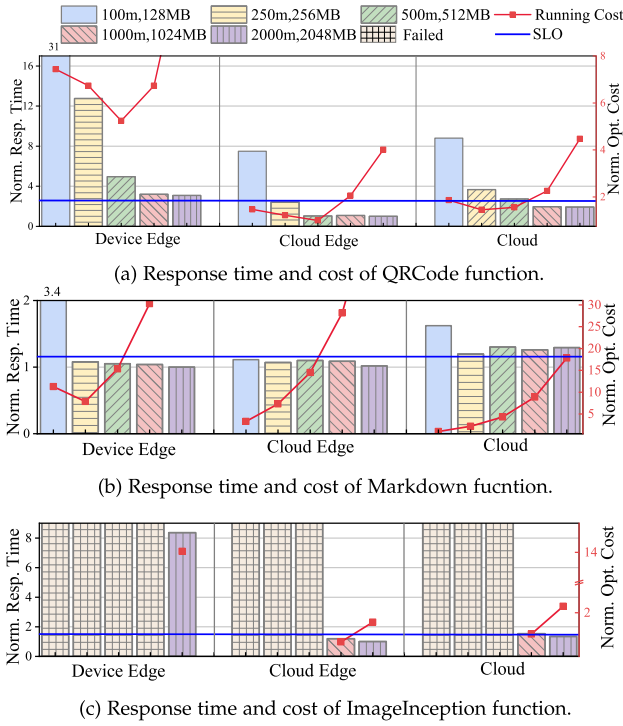


Fig. 2. The response time and execution cost of the QRCode, Markdown, and ImageInception function under various FDPs. The response time is normalized into the minimal value and the execution cost is normalized into the lowest cost of each function. The policy “100 m, 128 MB” denotes launching functions with 100 millicores CPU and 128 MB memory. In cases where functions cannot be launched with the policy, we mark it with a “Failed” label.

B. Motivation

As mentioned in the introduction, an FDP consists of three factors, namely FaaS platform, CPU allocation, and memory allocation. We do not take the IO and network allocation into account as COSE does, because neither public FaaS platforms (e.g., AWS Lambda and Azure Function) nor open-sourced FaaS platforms (e.g., Knative [25] and OpenFaaS [26]) support IO and network configuration. However, the IO and network delay are actually part of the FaaS platform factor. In other words, if the edge platform is chosen, the policy is equipped with a low network delay.

In this section, we present the characterization results for three FaaS functions, i.e., Markdown, ImageInception, and PageRank (details will be shown in Sec. IV-A), in a real HCC environment in Table II. Fig. 2 shows the response time and execution cost of the above three functions under various FDPs. Studying the characterizations in Fig. 2 offers insights into designing a function delivery framework.

Motivation ①. FaaS Deliver should consider the heterogeneity of the FaaS platforms: As shown in Fig. 2, the performance of functions on different platforms exhibit significant variations when the same amount of resource allocation is provided. For instance, when allocated with 100 millicores (m) CPU and 128 MB memory, the Markdown function satisfies its Service Level Objective (SLO) at the edge, but fails to meet the SLO when delivered to both the fog and the cloud. Such discrepancies

stem from the underlying platform heterogeneity, which encompasses the computational capacity, IO bandwidth, and network delay. These factors are platform-dependent and cannot be adequately captured by the amount of CPU and memory alone. Schedulers that ignore the platform property, e.g., COSE [14] or KNative [25], may lead to inefficient executions of functions or even failed launches.

Motivation ②. FaaS Deliver should consider both cost and SLO in the search process: While minimizing the execution cost is a primary objective for FaaS users, it is important to note that the FDP with the lowest cost may not always satisfy the strict SLO requirements. This is evidenced by the Markdown function illustrated in Fig. 2(b), where the FDP with the lowest execution cost is (Cloud, 100 m, 128 MB), but the response time violates the strict SLO due to excessive network transmission time, which is not acceptable. Therefore, the primary goal of *FaaS Deliver* is to locate the FDP with the lowest cost for a given function while ensuring that the SLO of that function is satisfied.

Motivation ③. FaaS Deliver should skip the locally optimal FDP of each platform to find the globally optimal FDP: Fig. 2(a) shows that the QRCode function has at least three locally optimal FDPs: (Edge, 500 m, 512 MB), (Fog, 500 m, 512 MB), and (Cloud, 250 m, 256 MB). Consequently, FaaS providers may become confined to one locally optimal FDP, such as (Cloud, 250 m, 256 MB) in QRCode, and fail to identify the globally optimal FDP, such as (Edge, 500 m, 512 MB) in QRCode, leading to unnecessary expenses. Hence, *FaaS Deliver* must possess the ability to transcend the limitations of multiple locally optimal FDPs.

Motivation ④. FaaS Deliver can benefit from pruning search space of memory: Each function necessitates a minimum amount of memory, denoted as M , to initiate the function, which is typically unknown beforehand. Assigning less memory than M to a function will result in the function failing to initiate. As shown in Fig. 2(c), allocating less than 1000 MB of memory for the ImageInception function on edge will cause it to fail. Consequently, we can curtail the lowest allocatable memory configuration for the search space of ImageInception on edge to 1000 MB. This step will decrease the search space, expedite the search process, and reduce the number of function launch failures.

III. APPROACH

A. Problem Formulation

In our proposed approach, we transform the function delivery task into a cost optimization problem. Suppose launching a FaaS function f with an FDP $\mathbf{p} = (\text{platform}, \text{CPU}, \text{memory})$, the execution cost is given by

$$C^f(\mathbf{p}) = t^f(\mathbf{p}) \times \text{Pr}(\mathbf{p}),$$

where $t^f(\mathbf{p})$ denotes the execution time of f under \mathbf{p} , and $\text{Pr}(\mathbf{p})$ is the price (cost per unit time) for the FDP \mathbf{p} . As an example, a function that runs on cloud for 1 seconds with a 100 m CPU and 512 MB memory (i.e., $\mathbf{p} = (\text{Cloud}, 100, 512)$) would incur

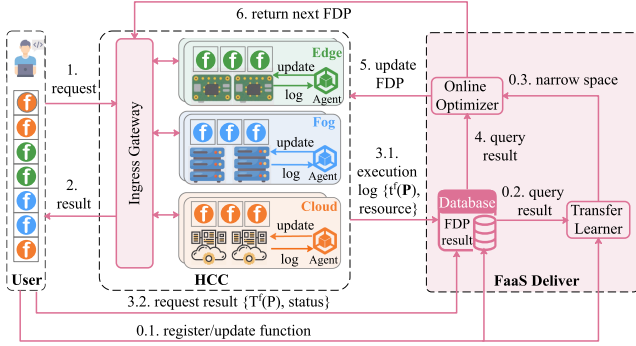


Fig. 3. An overview of *FaaSDeliver* framework. Step 1-6 execute every iteration during the optimization, while step 0.1 is run when a function is registered or updated, and step 0.2-0.3 are executed only when a function is updated.

the following cost

$$1 \times (100 \times 0.000016\$ + 512 \times 0.000005\$) = 0.00416\$,$$

where 0.000016\$ is the cloud-specific CPU price per millicore-second, and 0.000005\$ is the cloud-specific memory price per MB-second.

To account for the additional delays associated with different FaaS platforms in the HCC, our target is to select the FDP expected to minimize execution cost while its response time satisfying the SLO. The response time to run a function (i.e., end-to-end latency) is given by

$$T^f(\mathbf{p}) = t^f(\mathbf{p}) + d(\mathbf{p}), \quad (1)$$

where $d(\mathbf{p})$ is the duration other than the execution time of a function (e.g., cold start delay and network delay). As measured in study [27], cold start time of a function is relatively constant on the same platform, but varies from platform to platform. If a FaaS platform has a longer cold start time, it manifests itself as the larger $d(\mathbf{p})$, which may lead to an SLO violation.

According to the motivation ②, we formulate the function delivery problem of FaaS function f as follows:

$$\begin{aligned} & \underset{\mathbf{p}}{\text{minimize}} && C^f(\mathbf{p}) = t^f(\mathbf{p}) \times \text{Pr}(\mathbf{p}) \\ & \text{subject to} && T^f(\mathbf{p}) \leq \text{SLO}, \\ & && \mathbf{p} \in \mathcal{P} \end{aligned}, \quad (2)$$

where SLO represents the upper boundary of acceptable latency of functions specified by FaaS users, and \mathcal{P} is the set of all FDPs. Knowing $t^f(\mathbf{p})$ and $T^f(\mathbf{p})$ under all FDPs would make it straightforward to solve (2), but it is prohibitively expensive to search all FDPs. To solve this problem cost-efficiently, we propose *FaaSDeliver*, which can automatically and rapidly select the optimal FDP for a given function.

B. FaaSDeliver Overview

The overall architecture of *FaaSDeliver* is presented in Fig. 3. *FaaSDeliver* is designed to identify the most cost-effective FDP in order to guide FaaS users in configuring functions. The system is composed of two primary components: (i) an *Online Optimizer* component, which is responsible for learning the function's performance model, i.e., the relationship between cost/runtime

and FDP for the FaaS function, and determining the optimal FDP that minimizes cost while adhering to the SLOs; (ii) a *Transfer Learner* component, which is responsible for narrowing the search space of FDPs to expedite the optimization process when the function is updated. In this study, *FaaSDeliver* is incorporated into an HCC environment, but it could also be directly leveraged by FaaS users.

Before invoking functions, FaaS users need to register or update their functions within *FaaSDeliver* using standard Application APIs (step 0.1 in Fig. 3). Fig. 3 highlights the interactions among users, the HCC environment, and our *FaaSDeliver* framework during the search process for each invocation. The process is as follows:

- *Step 1*: The user client issues a function request to the ingress gateway of the HCC. The gateway then routes the request to the FaaS platform defined in the last FDP (the first request is selected randomly) to invoke the FaaS function.
- *Step 2*: Upon receiving the request, the FaaS platform launches the function to handle the request, and returns the results to the client.
- *Step 3*: The monitor agent within the FaaS platform reports the execution log (i.e., $t^f(\mathbf{p})$ and allocatable resource), and the client reports the request result (i.e., $T^f(\mathbf{p})$ and request status) to *FaaSDeliver*.
- *Step 4*: The *Online Optimizer* module of *FaaSDeliver* queries the FDP results from the FDP database and incrementally updates the existent online learning performance model to generate the new promising FDP for exploration.
- *Step 5*: *FaaSDeliver* updates the function deployment to the specific FaaS platform in the HCC based on the new promising FDP. The corresponding FaaS platform scheduler takes over the scheduling for FaaS functions.
- *Step 6*: *FaaSDeliver* returns the new FDP to the gateway for next request.

FaaSDeliver repeats step 1-6 above until the stop condition is met and outputs the final optimal FDP.

If FaaS users update existent functions at step 0.1, *FaaSDeliver* needs to build a new performance model for the modified function. Considering that FaaS users typically update functions by making minor changes to the historical code rather than writing an entirely new function, repetitive learning from scratch is costly and unnecessary for these subtle updates. Consequently, when updating existent functions, *FaaSDeliver* incorporates a *Transfer Learner* module to narrow the search space of FDPs, thereby accelerating the optimization process. *Transfer Learner* queries the FDP results of previous versions of the updated function at step (0.2) and produces a compressed search space for *Online Optimizer*. Specifically, step 0.1 is executed when a function is registered or updated, while step 0.2-0.3 are performed when a function is updated.

C. Online Optimizer

Online Optimizer is the core component of *FaaSDeliver*. *Online Optimizer* is designed to determine the subsequent FDP to explore and converge to the optimal FDP finally. Considering that the variation of functions and the heterogeneity of HCC, it

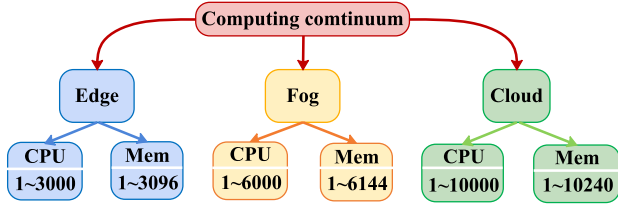


Fig. 4. An example of a tree-structured hybrid search space \mathcal{P} in HCC. The search space is organized as a tree structure in the sense that some leaf variables are only possible when launching the function on a specific platform (e.g., only the cloud can allocate 8000 m CPU for a function).

is challenging to pre-define a performance model to solve the problem in (2). In this study, we use an online learning technique, which can start without a pre-defined performance model to find the most cost-efficient FDP with recurring invocations of the same function. Specifically, *FaaS Deliver* employs Tree-structured Parzen Estimator (TPE) [15] to tackle the function delivery problem.

Why TPE? Acquiring insights from motivation ① and ③, this study intends to address the function delivery problem by considering both the inherent heterogeneity of the underlying platform and the need to effectively escape local optima in order to identify the optimal FDP. Existing performance modeling-based approaches [28], [29] and collaborative filtering-based techniques [30], [31] are hindered by the high costs associated with offline profiles. Many state-of-the-art approaches (e.g., Cherrypick [32] and COSE [14]) employ Gaussian Process Bayesian Optimization (GPBO) to solve optimization problems analogous to (2). However, GPBO encounters difficulties in managing the tree-structured hybrid characteristics of the FDP search space (e.g., Fig. 4), which encompasses platforms (categorical variables), CPUs (discrete variables), and memory (discrete variables). Consequently, it falters in effectively addressing the heterogeneity of platforms. The limitations of GPBO in processing categorical variables can be attributed to its utilization of Kriging as a surrogate [33]. Reinforcement Learning (RL) is well suited for learning FDPs in the tree-structured space, but it requires extensive training iterations and more computing resources to converge to the optimal FDP [23], [34].

In contrast, the tree-based TPE can naturally deal with tree-structured inputs and scales linearly with the number of observations due to it uses non-parametric Parzen kernel density estimators to model the distribution of good and bad configurations [15]. The non-parametric nature of TPE makes it practical even when we have no knowledge about function-specific logic and platform heterogeneity. Meanwhile, TPE provides a tight feedback loop for exploring FDP space and generating optimal FDPs. It allows direct learning from function invocations to understand the impact on the performance of functions after tuning FDPs.

1) Leveraging TPE: In the context of *FaaS Deliver*, all possible FDPs comprise the search space \mathcal{P} and are referred to as sample points. The objective function $C^f(\mathbf{p})$ in (2) is unknown beforehand but can be observed through experiments. Every time

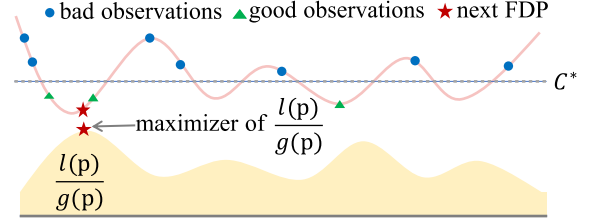


Fig. 5. An illustration of the TPE search procedure. The top part depicts how TPE splits evaluated observations based on $\gamma = 0.3$. The bottom part shows how TPE selects the next FDP to be explored.

the function is invoked, the TPE iterates once. At each iteration, TPE collects a new observation, and at the end of the iteration the algorithm decides which FDP should be explored next time.

To identify the global optimum of the objective function, TPE intelligently explores \mathcal{P} by evaluating the objective function with different observations $\mathcal{O} = \{(\mathbf{p}_1, C_1), \dots, (\mathbf{p}_k, C_k)\}$. *FaaS Deliver* first splits the observations into good FDPs group \mathcal{O}_l and bad group \mathcal{O}_g , defining C^* as the splitting value for these two groups (as shown in the top part of Fig. 5). The value C^* is selected to be a quantile γ of the observed C values satisfying

$$P(C^* > C) = \gamma.$$

We discuss how γ affects *FaaS Deliver* in Section IV-B4. Then TPE uses a kernel density estimator to model the densities of good FDPs $l(\mathbf{p})$ and bad FDPs $g(\mathbf{p})$

$$\begin{aligned} l(\mathbf{p}) &= p(y < C^* | \mathbf{p}, \mathcal{O}) \\ g(\mathbf{p}) &= p(y > C^* | \mathbf{p}, \mathcal{O}) \end{aligned} \quad (3)$$

over the input configuration space instead of modeling the objective function f directly by $p(C^f | \mathcal{O})$. Briefly, $l(\mathbf{p})$ models the density of better observations, and $g(\mathbf{p})$ models the density of poor observations. Due to the nature of kernel density estimators, TPE easily supports mixed continuous and discrete spaces [35]. Intuitively, the next expected FDP is the one that is the most likely to be good and the least likely to be bad. In other words, we prefer the FDP \mathbf{p} with a high probability under $l(\mathbf{p})$ and a low probability under $g(\mathbf{p})$, i.e., maximum ratio of $l(\mathbf{p})/g(\mathbf{p})$. TPE employs the following expected improvement (EI) function as the acquisition function to maximize improvement [36]

$$\begin{aligned} \text{EI}_{C^*}(\mathbf{p}) &= \int_{-\infty}^{\infty} \max(C^* - C, 0) p(C | \mathbf{p}) dC \\ &= \int_{-\infty}^{C^*} (C^* - C) p(C | \mathbf{p}) dC \\ &\propto \left(\gamma + (1 - \gamma) \frac{g(\mathbf{p})}{l(\mathbf{p})} \right)^{-1}. \end{aligned} \quad (4)$$

We chose EI function because it has been shown to outperform others and it does not require parameter tuning [14].

TPE at work: To provide an estimate about the shape of the cost model for TPE, an initial set of FDPs is generated by randomly selecting N_r FDPs ($N_r = 3$ by default) from each

platform within the HCC (line 1 in Algorithm 1). The multidimensional space exploration performed by TPE is an iterative procedure, guided by the acquisition function, and is informed by the historical data of previously assessed FDPs. Throughout each iteration, the following steps are executed:

- 1) TPE generates many candidates by sampling according to $l(\mathbf{p})$ and evaluates them based on the ratio $l(\mathbf{p})/g(\mathbf{p})$.
- 2) The TPE algorithm returns the candidate \mathbf{p}^* with the greatest EI as the next FDP (as shown in the bottom part of Fig. 5).
- 3) *FaaS Deliver* updates the function deployment to the specific FaaS platform based on the new promising FDP.
- 4) Once the function is launched, TEP observes the objective function results for the FDP and updates the good and bad densities $l(\mathbf{p})$ and $g(\mathbf{p})$ accordingly.
- 5) The newly evaluated FDPs are incorporated into the densities, which in turn influence the sampling of future points.

The search process of TPE continues iteratively until the maximum number of iterations, N_t ($N_t = 300$ by default), is reached. We discuss the influence of N_t and N_r on *FaaS Deliver* in Section IV-B4. Ultimately, TPE identifies the FDP with the lowest execution cost as the optimal FDP.

2) *Adaptive TPE for FaaS Functions*: In order to enhance the effectiveness and efficiency of TPE within the context of FaaS scenarios, we have introduced a series of adaptive modifications to the traditional TPE approach. These alterations have been carefully tailored to address the unique requirements and challenges posed by FaaS deployments. A comprehensive representation of the adapted TPE algorithm can be found in Algorithm 1, which offers a clear and concise pseudo-code breakdown of the method employed.

(i) *Dynamic resource space management*: The conventional TPE methodology presupposes that the search space remains invariant throughout the optimization process. Nevertheless, practical applications often deviate from this assumption. In situations where a substantial number of requests, pertaining to either the same function or distinct functions, are initiated in rapid succession, the maximum allocatable resources inevitably decline due to consumption. To illustrate, the maximum allocatable CPU resources for an edge device may decrease from 3000 m to 1000 m when an excessive number of functions are executed on the edge. It is imperative for *FaaS Deliver* to acknowledge and adapt to these fluctuations, ensuring the selection of feasible FDPs that maintain the stipulated SLOs within the constraints of the available resources. In light of these considerations, we have introduced dynamic adjustments to the TPE search space prior to each selection, taking into account the prevailing resource allocation conditions (line 5 of Algorithm 1).

(ii) *Employing multivariate TPE*: The traditional TPE approach is characterized by its independence, which inadvertently neglects the interdependencies that may exist between parameters. For instance, within the context of an independent TPE, given an FDP $\hat{\mathbf{p}} = (\text{Cloud}, 100, 128)$, the likelihood density of favorable FDPs is estimated through the product of univariate Parzen estimators, as follows:

$$l(\hat{\mathbf{p}}) = l_{\text{plat}}(\text{cloud}) * l_{\text{cpu}}(100) * l_{\text{mem}}(128).$$

Algorithm 1: Adaptive TPE for FaaS Function f .

Input: search space S , stop iteration N_t , random number N_r , observations \mathcal{O} , penal factor ξ , default quantile γ_d

```

1  $\mathcal{O} \leftarrow \text{RandomSample}(N_r);$ 
2 for  $iteration = 1$  to  $N_t - N_r$  do
3    $\gamma \leftarrow \min(\lceil iteration * 0.1 \rceil, \gamma_d);$ 
4    $\mathcal{O}_l, \mathcal{O}_g \leftarrow \text{SplitObservations}(\mathcal{O}, \gamma);$ 
5    $S \leftarrow \text{GetCurrentResource}();$ 
6    $l(\mathbf{p}), g(\mathbf{p}) \leftarrow \text{ProbabilityDensity}(\mathcal{O}_l, \mathcal{O}_g, S);$ 
   /* Sample candidates from  $l(\mathbf{p})$  */
7    $\mathbf{c} \leftarrow \{\mathbf{p}^{(i,j)} \sim l(\mathbf{p})\};$ 
   /* Select best candidate by EI */
8    $\mathbf{p}^* \leftarrow \text{argmax}_{\mathbf{p} \in \mathbf{c}} \text{EI}_{C^*}(\mathbf{p});$ 
9    $\text{success}, t^f(\mathbf{p}^*), T^f(\mathbf{p}^*) \leftarrow \text{LaunchFunc}(\mathbf{p}^*);$ 
10  if  $\text{!success}$  then
   /* Downward memory prune */
11  |  $\mathbf{p}^*.platform.\text{low\_memory} \leftarrow \mathbf{p}^*.memory;$ 
12  end
13  if  $T^f(\mathbf{p}^*) > SLO$  then
   /* Penalize SLO violations */
14  |  $C^f(\mathbf{p}^*) \leftarrow C^f(\mathbf{p}^*) * (T^f(\mathbf{p}^*)/SLO) * \xi;$ 
15  end
16   $\mathcal{O} \leftarrow \mathcal{O} \cup \{(\mathbf{p}^*, C^f(\mathbf{p}^*))\};$ 
17 end
18 return  $\mathbf{p}$  with the minimum  $C^f$  value in  $\mathcal{O}$ 
```

In this work, we have implemented a multivariate TPE approach (line 6 in Algorithm 1), which directly models $l(\mathbf{p})$ and $g(\mathbf{p})$ utilizing a singular multivariate Parzen estimator predicated on Kernel Density Estimations (KDEs) [37]. In other words, the likelihood density of $\hat{\mathbf{p}}$ can be calculated as

$$l(\hat{\mathbf{p}}) = l(\text{cloud}, 100, 128).$$

Through this methodology, the multivariate TPE is capable of capturing the underlying dependencies between parameters, thereby offering a more comprehensive and robust optimization process.

(iii) *Downward-closure based memory pruning*: As discussed in the motivation ④, each function necessitates the minimum memory threshold, denoted as M , for a successful initiation. The precise value of M for each function remains unknown in advance, due to the inherent opacity of FaaS functions. In a scenario where function f requires 256 MB of memory (M) but TPE samples an FDP $\hat{\mathbf{p}} = (\text{Cloud}, 100, 128)$, the function will inevitably fail to launch due to insufficient memory (i.e., $128 < 256$). From this observation, it can be deduced that allocating less than 128 MB of memory would similarly result in the failure of f . Consequently, *FaaS Deliver* can effectively adjust the lower bound of the memory space from 1 to 128, as exploring a portion of the search space incapable of launching the function would yield negligible insights into the optimal FDP. To this end, we have incorporated a pruning mechanism for the downward memory allocation following a failed launch

(lines 10-12 in Algorithm 1), thereby minimizing the likelihood of subsequent failures.

(iv) *SLO violations penalization*: Gaining insights from the motivation ④, we proactively increase the execution cost in instances where the selected FDP fails to comply with its SLO, thereby discouraging the selection of proximate FDPs in subsequent iterations. In this work, if the response time $T^f(\mathbf{p})$ of function f exceeds its SLO, we incorporate a penalty factor ξ to $C^f(\mathbf{p})$ (line 13-15 in Algorithm 1) to increase cost as follows:

$$C^f(\mathbf{p}) = C^f(\mathbf{p})_{\text{cost}} * (T^f(\mathbf{p})/\text{SLO}) * \xi, \quad (5)$$

where $C^f(\mathbf{p})_{\text{cost}}$ is the actual execution cost, $T^f(\mathbf{p})/\text{SLO}$ indicates the degree of anomaly under \mathbf{p} , and ξ ($\xi = 100$ default) represents the penalty factor. We will discuss how ξ affects *FaaS Deliver* in Section IV-B2.

D. Transfer Learner

In the majority of instances, FaaS developers tend to modify an existing function by implementing minor alterations to its historical code, as opposed to developing an entirely new function. Consequently, both the pre-update and post-update functions may exhibit analogous execution logic and resource demands. As a result, redundant learning from scratch for functions with subtle updates is both costly and unwarranted. Therefore, it is a logical approach to transfer knowledge from historical versions to speed up the online learning of updated functions.

Transfer Learner module calculates the performance similarity between historical and updated versions of a function f . When the versions exhibit similar performance under certain well-chosen FDPs, it can be inferred that they possess analogous resource requirements. As a result, it is justifiable for the *Transfer Learner* module to refine the search space for the updated version, transitioning from an extensive space to a more restricted one based on the identified similarities. This approach accelerates the optimization process and reduces the computational overhead. Conversely, if no discernible similarity is found, *FaaS Deliver* proceeds to establish a performance model for the updated function from scratch. This ensures that the performance characteristics of the updated function are accurately modeled, even if it does not closely resemble its historical version.

Similarity calculation: A function may go through multiple updates in its history. We consider the latest update as the updated version and the historical updates as historical versions. In this study, we solely focus on the latest N_{his} ($N_{\text{his}} = 5$ default) historical versions because the latest five historical versions can help capture the most recent trends in function development and changes, ensuring that *Transfer Learner* module is working with the most up-to-date information. Older versions may not reflect the current development practices, dependencies, or performance characteristics, therefore, may not be as beneficial for the knowledge transfer process.

When function f is updated, we mark its updated version as U and one of its historical versions as H . Given the optimal FDP \mathbf{p} of H and its corresponding execution time t_H^f , *Transfer Learner* launches U under the same \mathbf{p} and gets the execution time t_U^f . If t_U^f is proximate to t_H^f , this implies that they behave similarly

under the same FDP. Consequently, U has similar resource requirements to H . *FaaS Deliver* computes the similarity score s between H and U as follows:

$$s = 1 - \frac{|t_U^f - t_H^f|}{(t_U^f + t_H^f)/2}. \quad (6)$$

The right part of (6) is the symmetric mean absolute percentage error (SMAPE), which is an accuracy measure based on relative errors [38]. If the similarity score surpasses the similarity threshold score_{\min} ($\text{score}_{\min} = 0.8$ default), we determine the performance of U and H is similar. Subsequently, we incorporate the similarity score and optimal FDP of H into the similar group G . By selecting a threshold of 0.8, we ensure that only those historical versions that exhibit a strong resemblance in performance with the updated version are considered for further analysis. This threshold helps avoid including dissimilar historical versions, which would not contribute valuable information for optimizing the updated version's performance. It is important to note that the threshold value can be adjusted based on the specific requirements of a given application or the desired level of similarity. For the remaining historical versions, we iteratively execute the aforementioned calculation process. Ultimately, the group $G = \{(\text{score}_1, \mathbf{p}_1^*), \dots, (\text{score}_n, \mathbf{p}_n^*)\}$ is produced to constrict the search space of U .

Narrow search space: We introduce a straightforward and generic method for devising a compact search space tailored to the updated function. The core idea of *Transfer Learner* is to ascertain the most compact search space $\hat{\mathcal{P}}$ that encompassing all the optimal FDPs within the similar group G .

Transfer Learner first constructs an initial tree-structured search space $\mathcal{P} = \{\mathcal{P}_{\text{edge}}, \mathcal{P}_{\text{fog}}, \mathcal{P}_{\text{cloud}}\}$ for updated functions. We represent each subspace $\mathcal{P}_{\text{platform}}$ as two bounding boxes, i.e., $\mathcal{P}_{\text{platform}} = \{[\hat{l}_{\text{cpu}}, \hat{u}_{\text{cpu}}], [\hat{l}_{\text{mem}}, \hat{u}_{\text{mem}}]\}$. Subsequently, *Transfer Learner* divides FDPs in G into three subgroups $\{G_{\text{edge}}, G_{\text{fog}}, G_{\text{cloud}}\}$ based on the platform of FDPs. The $\mathcal{P}_{\text{platform}}$ is excluded from \mathcal{P} if the corresponding G_{platform} is empty. Finally, motivated by [39], the bounding boxes of each platform have a simple closed-form solution $\hat{\mathcal{P}}_{\text{platform}} = \{[\hat{l}_{\text{cpu}}, \hat{u}_{\text{cpu}}], [\hat{l}_{\text{mem}}, \hat{u}_{\text{mem}}]\}$, which can be defined as

$$\hat{\mathcal{P}}_{\text{platform}} = \begin{cases} \hat{l}_{\text{cpu}} = \min\{[\mathbf{p}_i^*.\text{cpu} * \text{score}_i]\}_{i=1}^n \\ \hat{u}_{\text{cpu}} = \max\{[\mathbf{p}_i^*.\text{cpu} * (1 + (1 - \text{score}_i))]\}_{i=1}^n \\ \hat{l}_{\text{mem}} = \min\{[\mathbf{p}_i^*.\text{mem} * \text{score}_i]\}_{i=1}^n \\ \hat{u}_{\text{mem}} = \max\{[\mathbf{p}_i^*.\text{mem} * (1 + (1 - \text{score}_i))]\}_{i=1}^n \end{cases} \quad (7)$$

where n denotes the number of FDPs in the subgroups of the platform. These solutions are both simple and intuitive. Although the initial lower and upper bounds might delineate excessively broad ranges, the newly defined ranges represent the smallest intervals encapsulating all the FDPs in G . Upon evaluating each platform, the *Transfer Learner* obtains the novel search space $\hat{\mathcal{P}}$, which comprises a tightly constrained search space tailored to the updated function.

Example: We take the search space \mathcal{P} in Fig. 4 and the QRCode (QR) function as an example. Assume QR_1 , QR_2 and QR_3 are three historic versions of QRCode. QR_1 , QR_2

TABLE I
THE LIST OF FAAS FUNCTIONS USED IN EXPERIMENTS

Function	Description	Runtime
QRCode (QR)	Generate QRCode of given text	Golang
Markdown (MD)	Render Markdown text to HTML	Python
SentimentAnalysis (SA)	Sentiment analysis of given text	Python
ResizeImage (RI)	Reset the size of given images	C
ImageInception (II)	Classify given image through DNN	Python
PageRank (PR)	Calculate PageRank of given graph	C

and QR_3 have reached their optimal FDPs (Edge, 225, 32), (Edge, 425, 32), and (Edge, 500,32), respectively. Now, the QR function is updated and represented as QR_4 . *Similarity Determiner* computes the similarity scores between QR_4 and the other three historical versions as 0.3, 0.85, and 0.9, respectively. *Transfer Learner* then obtains the similar group $\mathcal{G} = \{(0.85, (\text{Edge}, 425, 32)), (0.9, (\text{Edge}, 500, 32))\}$. Neither fog nor cloud are in the optimal FDPs, *Transfer Learner* excludes these two platforms and narrows down the search space \mathcal{P} to $\hat{\mathcal{P}}_{\text{Edge}}$, where

$$\begin{aligned}\hat{\mathcal{P}}_{\text{Edge}} &= \{[\hat{\ell}_{\text{cpu}}, \hat{u}_{\text{cpu}}], [\hat{\ell}_{\text{mem}}, \hat{u}_{\text{mem}}]\} \\ &= \{[425 * 0.85, 500 * 1.1], [32 * 0.85, 32 * 1.15]\} \\ &= \{[360, 550], [27, 37]\}.\end{aligned}$$

Since $\hat{\mathcal{P}}$ is more compact than \mathcal{P} , TPE method converges to the optimal FDP of QR_4 without much effort.

IV. EXPERIMENTAL EVALUATIONS

In the section, we evaluate the following questions:

- *RQ1*: How effective is *FaaSDeliver* in finding optimal FDP?
- *RQ2*: What is the contribution of each design in adaptive TPE of *FaaSDeliver*?
- *RQ3*: What is the contribution of *Transfer Learner* when updating functions?
- *RQ4*: How sensitive is *FaaSDeliver* to parameters and environments?

A. Experimental Setup

Representative FaaS functions: We develop or refactor 6 FaaS functions² in Table I, which can be launched on servers with X86 and ARM CPU architecture in heterogeneous computing continuum (HCC). Some of these functions have been widely used in other studies [9], [40]. This group of functions represents different types of computation and SLO requirements that FaaS platforms typically perform: (i) *simple functions with strict SLO*: lightweight but time-sensitive functions (e.g., Markdown function) should be launched at the edge platform to reduce the time of network transmission; (ii) *medium functions with strict SLO*: these functions require more resources and should be launched at the fog cluster due to the strict SLO (e.g., ImageInception function); (iii) *complex functions*: these are resource-intensive

functions that should be launched at the cloud (e.g., PageRank function).

Heterogeneous computing continuum: We evaluate the proposed function benchmarks on a real HCC including three heterogeneous FaaS platforms ranging from edge, fog to cloud. Table II shows the resource quota of each platform, the FaaS platform used, the distance to users, and the number of nodes. Cgroups [41] is enabled for fine-grained resource isolation. We purchased 3 Raspberry Pi and deployed them near to users (less than 100 meters) to construct the edge platform. For the fog platform, we purchased 5 Dell Desktops and deployed them in the community of users. We rented 5 VMs from a public cloud provider to build the cloud platform. We do not use AWS Lambda or Azure Functions directly because they can only configure the memory of the function and not the CPU. We would like to test *FaaSDeliver* in adjusting both memory and CPU, encouraging public FaaS providers to unbind the correspondence between CPU and memory.

Taking the actual cost of purchasing devices and renting cloud VMs into account [42], the price of per CPU and memory for edge, fog and cloud are set according to the relation 1 : 4 : 6 in our experiments. Considering the overhead of FaaS platforms, we set the search space \mathcal{P} of the HCC to be the same as Fig. 4. Note that *FaaSDeliver* can be used for finer-grained resource allocation. To better present experimental results, we set an offset of 25 m for CPU and 32 MB for memory when allocating resources. In total, 180,200 FDPs are available on our testbed.

Implementation Details: We implemented *FaaSDeliver* based on Python 3.8. We adapt the TPE algorithm provided by Optuna [43] to implement the adaptive TPE for FaaS functions. In the *Online Optimizer* module, we set the quantile γ and penalty factor ξ to 25 and 100 by default. During the search process, we set N_r and N_t to 3 and 300. When the number of iterations corresponding to a function is greater than 300, the search process for that function will be terminated and the optimal result will be generated. Our *FaaSDeliver* is compatible with two widely-used open-source FaaS Frameworks, KNative [25] and OpenFaaS [26]. In addition, *FaaSDeliver* is easy to integrate with public FaaS providers such as AWS Lambda. *FaaSDeliver* is deployed in the cloud platform.

Baselines: We compare *FaaSDeliver* with several baselines:

- *Random search (RS)* is the approach that randomly chooses FDPs from search space. This earlier study [44] shows empirically and theoretically that RS are more efficient for hyper-parameter optimization than grid search. We treat RS as the basic approaches without any model. RS are implemented based on Optuna [43].
- *COSE* is a framework that uses GPBO to find the optimal configuration for FaaS functions in the homogeneous platforms [14]. We compare COSE to evaluate whether the GPBO-based approach can handle tree-structured search space. COSE is coded based on BayesOpt [45].
- *Independent COSE (ICOSE)* runs COSE on each homogeneous platform independently and selects the best result as the final FDP. We compare ICOSE to evaluate whether running the GPBO-based approach independently on each platform can handle tree-based spaces.

²<https://github.com/yuxiaoba/Serverless-Benchmark>

TABLE II
PRACTICAL HETEROGENEOUS COMPUTING CONTINUUM USED FOR EVALUATING FUNCTIONS IN TABLE I

Name	PaaS Platform	FaaS Platform	CPU Processor	Resource Quota	Number	Distance to User	Source of Node
Edge	K3S	OpenFaaS	ARMv7 rev 3 (v7l) @ 1.2 GHz	4 Core CPU, 4 GB Memory	3	<100 meters	Purchase Raspberry Pi
Fog	Kubernetes	Knative	Intel(R) Silver 4116 @ 2.10 GHz	8 Core CPU, 8 GB Memory	5	<1 kilometers	Purchase Dell Desktops
Cloud	Kubernetes	Knative	GenuineIntel @ 2.20GHz	12 Core CPU, 12 GB Memory	5	>100 kilometers	Rent from Public Cloud

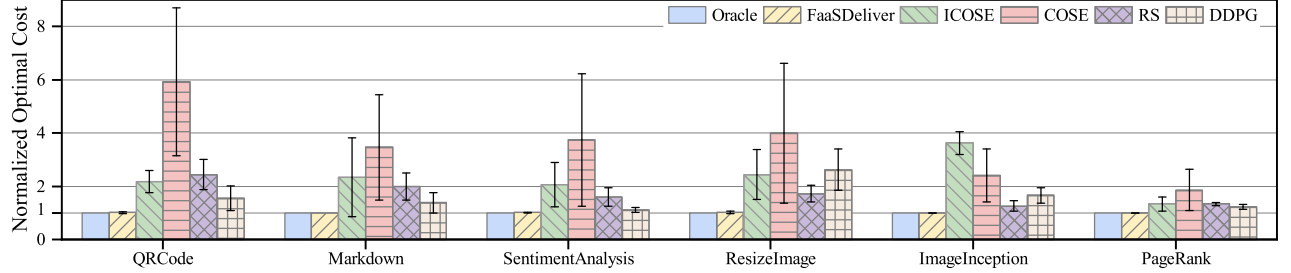


Fig. 6. The cost of optimal FDPs found by different approaches after 300 iterations. The results are normalized by the cost of the oracle FDP. The FDPs found by ICOSE, COSE, RS and DDPG are generally 2×, 4.8×, 2×, 1.5× more expensive than *FaaS Deliver*'s.

- *Deep deterministic policy gradient (DDPG)* is a reinforcement learning technique that combines both Q-learning and Policy gradients [34]. We choose DDPG, because it does not have additional components such as Double Q-Learning, which may complicate the analysis of this comparison.

B. Experiments and Results Analysis

1) *Effectiveness Validation (RQ1)*: We first validate the optimal FDP found by *FaaS Deliver*. In our case, the optimal FDP of a function represents the cheapest execution cost found while satisfying its SLOs. Considering the search cost, we stop the search process after 300 iterations. For ICOSE, COSE is performed separately for 100 iterations on each platform and outputs the best result. For each experiment, we repeat it 25 times and calculate the average result. All results are normalized to the cost of the oracle optimal FDP, which is the ground truth obtained by exhaustive offline search.

FaaS Deliver unearths better FDPs with more stability than other approaches under the same number of iterations: Fig. 6 presents the normalized optimal cost of FDPs suggested by each approach for 6 functions. Overall, *FaaS Deliver* can find the optimal or sub-optimal FDPs near to the oracle FDPs from 180,200 FDPs. Across functions, the cost of *FaaS Deliver*'s optimal FDPs is only 0-11% higher than the cost of oracle FDPs. For each function, *FaaS Deliver* outperforms the other four approaches. Compared with ICOSE, COSE, RS, DDPG approaches, *FaaS Deliver* can cut the execution cost of functions by 48%, 78%, 52% and 36% on average, respectively. The DDPG approach is the best alternative approach because the RL algorithm can handle tree-structure spaces better than other three approaches. In addition, the tail of the FDPs suggested by ICOSE, COSE, RS, and DDPG approaches could be further from oracle FDPs than *FaaS Deliver*'s FDPs.

FaaS Deliver unearths the sub-optimal FDPs in a shorter time than other approaches: To better understand the search process of *FaaS Deliver* and baseline approaches, we present the results of different iterations in Fig. 7 to show how approaches converge to their best FDP for these 6 functions. Because the invocations of functions are not continuous, we measure the convergence speed of approaches by the number of iterations rather than the total time. Overall, the results in Fig. 7 show that *FaaS Deliver* achieves the fastest convergence speed for all functions. We observe that *FaaS Deliver* needs only 30 iterations to find an acceptable FDP (2× optimal cost), while the other approaches usually take at least 100 iterations. Further, *FaaS Deliver* converges to the optimal or sub-optimal FDP in around 120 iterations. It is counter-intuitive to observe that COSE approach is not better than RS, which is consistent with previous work [46]. This deterioration is caused by the high uncertainty in fitting a regression model due to the tree-structured search space [47].

Why does FaaS Deliver perform well? All the above optimization approaches have been proved to be able to find the optimal (or sub-optimal) solution in a large search space in previous work [14], [15], [34], [44]. The main reason why *FaaS Deliver* performs better than other baselines is that *FaaS Deliver* can recognize the target platform fast and prune the sub search spaces of other platforms. However, GPBO-based COSE have difficulty in handling the categorical parameters that indicate what the conditional groups are. Though DDPG is able to handle tree-structured spaces but it needs more resources to calculate the next FDP and more iterations to converge to the optimal FDP. Therefore, *FaaS Deliver* is more suitable for the discrete tree-structured search spaces of HCC (e.g., Fig. 4) than other approaches.

Table III shows the choices of platforms for 6 functions during the search process, which allows us to dissect results in a greater depth. From Table III, we find that over 80% of *FaaS Deliver*'s searches are effectively spent on target platforms

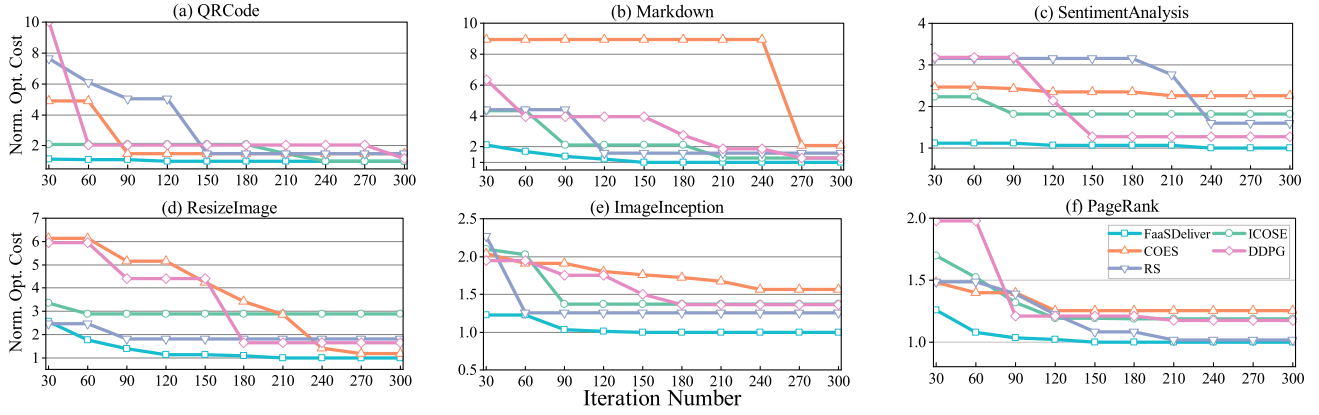


Fig. 7. Progression of the search processes for target functions. Not only does *FaaSDeiver* find better FDPs compared to other baselines, but it also converges to optimal FDPs faster.

TABLE III
THE DISTRIBUTION OF THE COUNT OF SEARCHES ON DIFFERENT PLATFORMS FOR DIFFERENT APPROACHES

Function	Approach	Search Count			Optimal Platform
		Edge	Fog	Cloud	
QRCode	RS	96	104	100	Fog
	COSE	131	87	165	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	18	265	17	
	DDPG	128	88	84	
Markdown	RS	98	100	102	Edge
	COSE	48	87	165	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	262	18	20	
	DDPG	120	98	82	
SentimentAnalysis	RS	99	103	98	Fog
	COSE	265	27	8	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	16	268	16	
	DDPG	118	83	99	
ResizeImage	RS	106	94	100	Fog
	COSE	273	6	21	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	18	264	18	
	DDPG	123	97	80	
ImageInception	RS	81	111	108	Fog
	COSE	64	44	192	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	19	262	19	
	DDPG	123	90	87	
PageRank	RS	100	90	110	Cloud
	COSE	48	87	195	
	ICOSE	100	100	100	
	<i>FaaSDeiver</i>	18	33	249	
	DDPG	121	81	95	

(e.g., the optimal FDP of Markdown is (Edge, 225 m, 32 MB)), and *FaaSDeiver* selects the edge platform (262 times) more than fog (18 times) and cloud (20 times). Compared with other approaches, *FaaSDeiver* can recognize the target platform earlier and search for more local FDPs on the target platform. COSE cannot recognize the target platform because the tree-structured space generates a high uncertainty in fitting performance models. DDPG selects each platform more evenly, which is similar to RS and ICOSE. Therefore, *FaaSDeiver* has a higher chance of finding the optimal FDP with fewer iterations.

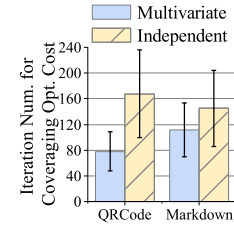


Fig. 8. Iterations required for converging to the optimal FDP.

2) *Adaptability Analysis (RQ2)*: In this subsection, we validate the effectiveness of the adaptive TPE introduced in Section III-C2.

Contribution of multivariate TPE: To show the effectiveness of multivariate TPE, we compare the number of iterations required to find the optimal FDP for *FaaSDeiver* with multivariate TPE and independent TPE, respectively. From Fig. 8, *FaaSDeiver* with multivariate TPE reaches the optimal FDP faster than independent TPE. For QRCode, *FaaSDeiver* with multivariate TPE converges to the optimal FDP with only half of the number of iterations required for independent TPE. Moreover, in other functions not shown, we observe similar results that the multivariate TPE has a better performance.

The reason why multivariate TPE performs better than independent TPE is that the multivariate TPE can capture dependencies among platforms, CPU, and memory, whereas the independent TPE cannot. For the independent TPE, the densities of good/bad parameters are estimated as the product of univariate Parzen estimators. Multivariate TPE, on the other hand, uses a single multivariate Parzen estimator. We depict the importance of FDP parameters assessed by multivariate and independent TPE in Fig. 9. We find that multivariate TPE gives a higher score for CPU and a lower score for memory than independent TPE. Actually, both QRCode and Markdown are CPU-sensitive, requiring more than 200 m CPU, and memory-insensitive, requiring 32 MB memory. Compared with independent TPE, multivariate TPE tends to tune the CPU more often, resulting in faster convergence.

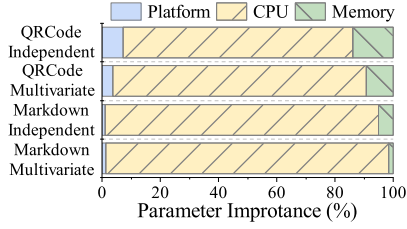


Fig. 9. Parameter importance assessed by multivariate and independent TPE. Multivariate TPE gives higher scores than independent TPE for CPU.

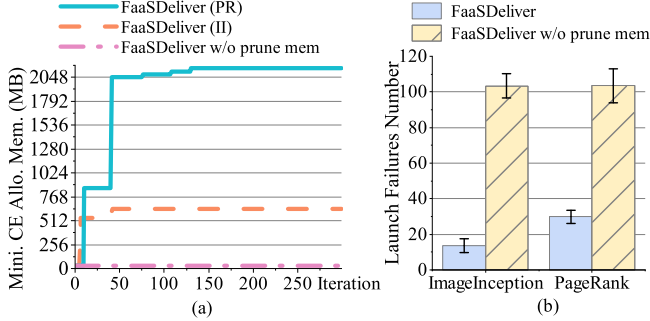


Fig. 10. (a) The change curves of the minimum allocatable memory during search processes. (b) Comparison of the number of failed launches during search processes.

Contribution of downward-closure based memory pruning: A function invocation may result in failure if initiated with inadequate memory resources. To mitigate the occurrence of failed invocations, *FaaS Deliver* incorporates a downward-closure based memory pruning technique. We subsequently investigate the impact of the downward-closure based memory pruning by comparing two variants of *FaaS Deliver*, namely one with memory pruning and the other without memory pruning for the ImageInception and PageRank functions. The selection of ImageInception and PageRank functions is predicated on their relatively substantial memory requirements, necessitating 640 MB and 2176 MB of memory, respectively.

Fig. 10(a) illustrates the procedure through which *FaaS Deliver* modifies the minimum allocatable memory at the fog platform for the ImageInception and PageRank functions. This adjustment aims to eliminate the memory space that is inadequate for initiating the functions, thereby reducing the frequency of failed starts. Fig. 10(b) shows the number of failed invocations for the ImageInception and PageRank functions over the course of 300 iterations. As evidenced in Fig. 10(b), *FaaS Deliver* without memory pruning exhibits a significantly higher rate of launch failures (4-6 \times) in comparison to *FaaS Deliver* with memory pruning. This disparity in performance underscores the critical role of memory pruning in diminishing launch failures and, ultimately, improving the user experience.

Contribution of penalizing SLO violations: In (5), we actively increase the execution cost of failed FDPs to avoid choosing an FDP near failed FDPs. In Fig. 11, we investigate the influence of penalty factor ξ on the final results across different functions. Fig. 11 shows *FaaS Deliver* has a poor performance when not

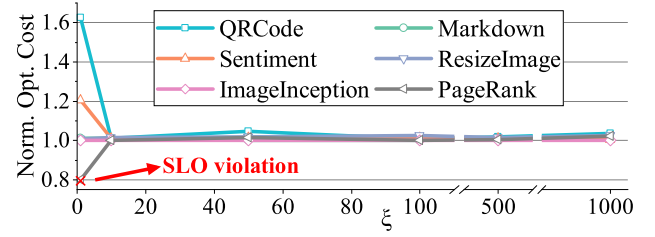


Fig. 11. The cost of optimal FDP determined by *FaaS Deliver* under different penalty factor ξ . $\xi = 1$ implies no active penalty is added.

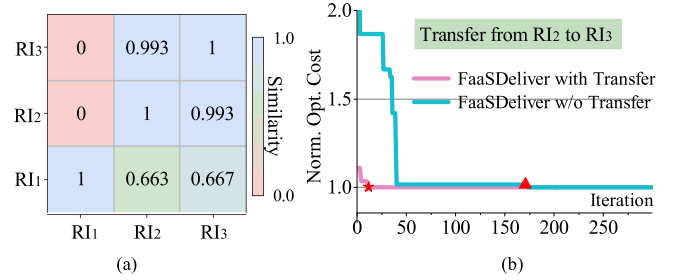


Fig. 12. (a) Similarity scores between 3 versions of the ResizeImage function. (b) Comparison of the search progress with and without *Transfer Learner* when updating the ResizeImage function. The symbols \star and \blacktriangle represent the number of iterations required for *FaaS Deliver* to converge to the optimal FDP with and without the Transfer Learner module, respectively.

actively increasing execution cost (i.e., $\xi = 1$), which demonstrates the importance of penalizing SLO violations. It is noticed that the cost of the optimal FDP suggested by *FaaS Deliver* is less than the optimal cost in PageRank when ξ is equal to 1. However, when we check the FDP, we find that this FDP cannot meet the SLO of PageRank. This is because some minor SLO violations (e.g., 10 ms) may be hidden by a large amount of resource allocation (e.g., 1000 MB memory). We recommend that the value of ξ should be set greater than 100 in *FaaS Deliver*.

3) Effectiveness of Transfer Learning (RQ3): In this part, we show transfer learning can significantly reduce the overhead of online learning when updating the ResizeImage function. We leverage the knowledge from historical versions (RI₁ and RI₂) to help find the optimal FDP of the updated version (RI₃) of ResizeImage. Fig. 12(a) shows the similarity result between 3 versions of ResizeImage. In this experiment, RI₁ and RI₂ was profiled 300 times. Fig. 12(a) depicts the similarity score among 3 versions of ResizeImage function. RI₂ and RI₃ failed to launch under the optimal FDP of RI₁. Thus, the similarity score between RI₁ and RI₂ and the score between RI₁ and RI₃ are both equal to 0. As shown in Fig. 12(a), RI₁ has significant performance differences from RI₂ and RI₃ (i.e., their similarity scores $s < \text{score}_{\min}$, where $\text{score}_{\min} = 0.8$). Owing to the high similarity score (i.e., 0.993) between RI₂ and RI₃, which surpasses the predefined threshold score_{\min} , we deduce that the performance of RI₂ and RI₃ is similar. As a result, we designate RI₂ as the transfer source and RI₃ as the transfer target.

Now we show the search processes of RI₃ with transfer learning from RI₂ and without transfer learning in Fig. 12(b). From

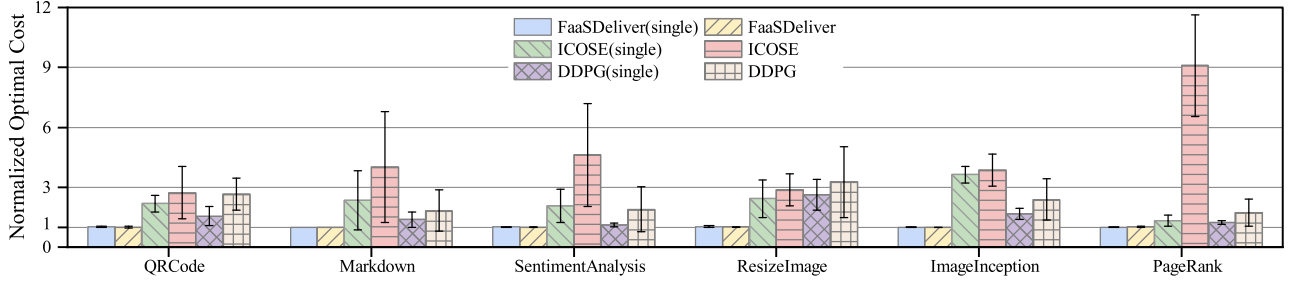


Fig. 13. The cost of optimal FDPs found by different approaches when running six functions together in HCC by replaying the real workload of Azure Functions. *FaaS Deliver*(single) means the results of *FaaS Deliver* without co-running functions.

Fig. 12(b), *FaaS Deliver* without *Transfer Learner* needs more than $30\times$ iterations to find the optimal FDP than *FaaS Deliver* with *Transfer Learner*. *Transfer Learner* considerably boosts TPE by narrowing down the search space, thus accelerating the process of optimization. In our experiments, *Transfer Learner* reduce search space of RI_3 based on RI_2 from the complete search space to $\hat{P} = \{\text{Fog}, [900, 950], [64, 128]\}$. These results imply that *FaaS Deliver* can capture common characteristics between historical versions and the updated version of the same function. Consequently, *Transfer Learner* introduces a positive contribution in reducing the overhead of *FaaS Deliver*.

4) *Sensitivity Analysis (RQ4)*: In this part, we extend our experiments with real workload of Azure Functions and different configurations to investigate the *FaaS Deliver*'s sensitivity to co-running functions and configurations.

Sensitivity to co-running functions under the real workload: To evaluate the performance of *FaaS Deliver* in real scenarios that with co-running functions in HCC, we replay the real workload of functions invoked by HTTP in the Azure Function dataset [48]. To construct a dataset of invocation rates (i.e., RPS), we first analyzed the per-minute function invocation count in the HTTP traces. For the time intervals with no invocations, we treat them as zero invocation rates. Considering the limited resources of our testbed, we excluded functions with more than 100 invocations per minute. Subsequently, we randomly sample six invocation rates and utilize them to trigger the functions under evaluations. We repeat the evaluation 25 times and calculate the average result. Fig. 13 shows the results of average cost normalized to the optimal result when functions are invoked 300 times in the real workload. From Fig. 13, co-running functions have little effect on the effectiveness of *FaaS Deliver* on average. For all functions, the most significant difference in the cost with and without co-running functions is only 0.02. While ICOSE and DDPG are more sensitive to the co-running functions than *FaaS Deliver*. For example, at PageRank function, ICOSE with co-running functions cost $9\times$ more than ICOSE without co-running functions.

Sensitivity to configurations: Fig. 7 shows how N_t influences the results of optimal FDP for 6 functions. From Fig. 7, we observed that *FaaS Deliver* requires at least 120 iterations (i.e., $N_t = 120$) to converge to the optimal or sub-optimal FDP. Once *FaaS Deliver* has converged to the optimal FDP, adding more iterations does not improve the result. In Figs. 14 and

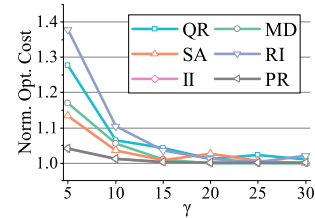


Fig. 14. Sensitivity to γ .

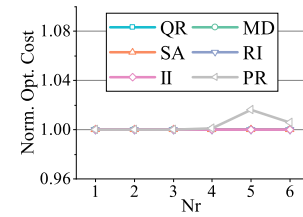


Fig. 15. Sensitivity to N_r .

15, the x-axis presents various γ values and N_r values, while the y-axis presents the average cost normalized to the optimal FDP. From the Fig. 14, we can find that *FaaS Deliver* has a poor performance when γ is less than 10. This can be attributed to the role of γ in distinguishing between favorable and unfavorable observations, as delineated in (3). When a smaller γ is employed, the likelihood of mis-classification from negative observations to positive ones increases, leading to suboptimal performance. As shown in Fig. 15, *FaaS Deliver* is insensitive to N_r . *FaaS Deliver* can converge to the optimal FDP under different values of N_r .

5) *Scalability Analysis*: In this part, we validate the scalability of *FaaS Deliver* when expanding HCC from 3 to 12 platforms. Fig. 16 shows the comparison results of *FaaS Deliver* over 300 iterations under the different number of FaaS platforms. We consider 1 edge, 1 fog, and 1 cloud platform as one HCC set (3 platforms). We scale platforms to two-fold HCC sets (i.e., 6 platforms), three-fold HCC sets (i.e., 9 platforms) and four-fold HCC sets (i.e., 12 platforms) to evaluate the scalability of *FaaS Deliver*. The new adding platforms have lower performance (i.e., longer execution time) than the platforms in one-fold HCC. Thus, the optimal FDP of the scaled platform remains constant with 3 platforms. From Fig. 16, *FaaS Deliver*

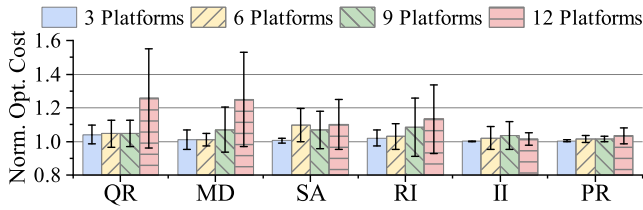


Fig. 16. The cost of optimal FDPs found by *FaaS Deliver* when expanding HCC from 3 to 12 platforms.

has 2%, 3%, and 11% cost increase on average from 3 to 6, 9, and 12 heterogeneous platforms, respectively. Although the average cost does not show a significant rise, some considerable tail costs are non-negligible as the scale of FaaS platforms grows. For example, the worst result of QRCode is $1.5\times$ than its optimal result. *FaaS Deliver* can alleviate this performance degradation by iterating more times.

6) *Overhead Analysis.*: To demonstrate the efficiency of *FaaS Deliver*, we evaluate its runtime overhead. On average, *FaaS Deliver* takes 0.01 s to calculate the next FDP in each iteration, excluding the time required for function execution and network transmission. A recent study reported that the invocation interval for more than 99% of functions in production serverless traces by Azure is greater than one second [21]. Consequently, the calculation of the next FDP is completed before the next function invocation, ensuring that it does not impact the subsequent invocations. If the invocation interval between two requests is less than 0.01 seconds, *FaaS Deliver* reuses the last FDP to handle the requests.

The resource requirements for *FaaS Deliver* are modest, with 128 MB of memory and 0.1 CPU core. These requirements correspond to 0.1% of the 128 GB memory and 3% of the 32-core CPU of the running server equipped with the Intel Xeon Gold 6242 CPU. This demonstrates that *FaaS Deliver* is an efficient solution with minimal overhead, making it well-suited for optimizing FaaS performance in HCC environments.

V. DISCUSSION

Performance SLOs: None of the commercial cloud provider offers SLOs in terms of performance (availability only) [49], which hinders the adoption of latency-critical functions on HCC. In this study, we infer latency SLOs through profiling functions based on FaaSProfiler [40], and assume that all requests sent to the same function correspond to the same SLO. Our SLO-aware FaaS delivery framework could potentially enable the FaaS providers to offer SLO guarantees and change pricing models to be SLO-aware. In cases where SLO violations are unacceptable, we can initiate two function instances for each request: one to ensure that user requests are processed promptly, and the other for FDP exploration. Once the TPE converges to the optimal FDP, only a single function instance is required to handle the request. This approach ensures that the performance SLOs are met while still allowing for the exploration and optimization of FDPs.

Function Chain: As demonstrated in a previous study [50], over 30% of FaaS applications are comprised of a single function. *FaaS Deliver* is good at delivering these applications effectively to HCC environments. However, for the remaining applications that consist of multiple functions, *FaaS Deliver* does not explicitly account for the dependencies among functions within the same chain. Rather than directly addressing these dependencies, *FaaS Deliver* indirectly tackles them through reactive performance measurements. This approach enables the system to adapt to the performance characteristics of individual functions within a multi-function application, optimizing each function's performance without explicitly considering the interdependencies among them. While this method may not fully exploit the potential benefits of considering function dependencies, it still provides a practical and efficient solution for optimizing the performance of FaaS applications in HCC environments.

Function Input: We acknowledge that the same function with different inputs may have different resource requirements. Fortunately, from Azure study [21], we find that over a third of the functions are periodic tasks, which typically deal with similar inputs. The requirements varies very slightly due to different inputs for these tasks. Furthermore, in the private HCC, we can classify these inputs based on their characteristics like OFC [51] to alleviate the impact of input variations. For example, for ImageInception function, we can classify images into different buckets according to their size, i.e., [(0MB, 5MB), (5MB–10MB), ...]. *FaaS Deliver* then maintains a performance model for each bucket-function pair to mitigate the impact of different images.

Function Cold Start: To alleviate the impact of startup process of functions, *FaaS Deliver* calculates the execution cost in (2) based on the execution time rather than response time. Thus, the cold or warm start time would not affect the execution cost. The response time is limited to less than the SLO in (2) to prevent an overly long start time. If launching a function on a platform with a long cold start time results in an SLO violation, *FaaS Deliver* will not deliver the function to that platform or allocate more resources to reduce execution time to meet SLO.

VI. RELATED WORK

Delivering FaaS functions to cloud: In recent years, many public cloud providers have introduced their serverless computing solutions, including AWS Lambda, Google Cloud Functions, and Azure Functions. Firecracker [52], USETL [53], and Sock [54] focus on solving the cold start problem of functions in FaaS platforms. However, these studies do not provide insight into how the delivery of functions to FaaS platforms can be optimized. Additionally, several studies have been conducted to describe and understand the architecture, resource allocation, and performance variations of different commercial serverless platforms (e.g., AWS Lambda, Azure Functions, and Google Cloud Functions) [27], [40], [55]. Nevertheless, the major focus of their work is on performance profiling of various FaaS platforms and not on delivering FaaS Functions.

Moreover, there have been some research works aimed at delivering FaaS Functions to cloud platforms [5], [18], [56], [57],

[58], [59], [60]. AWS Compute Optimizer [56] recommends optimal AWS resources for lambda-based functions to reduce costs and improve performance by using machine learning to analyze historical utilization metrics. One of the major limitations of AWS Compute Optimizer is that non-AWS users cannot benefit from the solution as it is a proprietary tool of AWS. Virtual serverless providers (VSPs) [18] aggregate public serverless offerings on cloud to allow developers to get rid of vendor lock-in problems and exploit pricing and performance variation across providers. Lin et al. [59] use the Probability Refined Critical Path Greedy algorithm to model the performance and cost of FaaS functions in the AWS Lambda. MBS [60] is a framework that optimizes the batching of ML inference serving requests on FaaS platforms to minimize their monetary cost while meeting their SLOs. SLAM [57] detects the relationship of multi-function serverless applications through distributed tracing and models the execution time and memory to identify the optimal memory configurations for them. However, it should be noted that these approaches based on FaaS platforms in the cloud cannot be directly applied to HCC.

Delivering FaaS functions to edge: Amazon and Microsoft have extended their FaaS platforms closer to the edge of the network with AWS Greengrass³ and Azure Functions on IoT Edge,⁴ which promise seamless integration into the respective cloud ecosystems. Glikson et al. [61] and Aske et al. [62] propose an extension of FaaS to the edge, enabling IoT and Edge devices to be seamlessly integrated as application execution infrastructure. Sledge [63] presents a novel and efficient WebAssembly-based serverless framework for the edge to support high density multi-tenancy, low startup time, bursty client request rates, and short-lived computations of serverless workload.

In regard to delivering FaaS functions to edge platforms, Thomas et al. [22] and Anshul et al. [64] construct FaaS platforms for operating edge AI applications in edge clouds, but the target devices need to be configured by function developers. Autoscale [23] employs an intelligent execution scaling engine based on reinforcement learning that selects the optimal execution target of edge inference. LaSS [65] uses principled queuing-based methods to deliver functions to edge platforms with an appropriate allocation. However, Autoscale and LaSS only focus on edge platforms but ignore the benefits of the cloud. EdgeFaaS [66] provides virtual function interfaces for consistent function management and optimizes the scheduling of functions and placement of data according to their performance. On the other hand, our work does a much more comprehensive assessment of resource allocation.

Delivering FaaS functions to computing continuum: Stefan et al. [67] propose a unified cloud and edge data analysis FaaS platform, which extends the notion of FaaS to the edge and facilitates managing data analysis. Subsequently, research on delivering functions to the optimal device in HCC has gained increased attention [9], [68], [69], [70], [71], [72], [73]. Sebasti et al. [70] introduce an open-source platform to support serverless computing for scientific data-processing workflow-based

applications across the cloud continuum. FogFlow [71] is an open-source FaaS platform supporting the deployment and orchestration of functions, so-called fog functions, on Cloud and Edge infrastructures. FogFlow adopts a data-centric programming model instead of the more classical topic-based approach. SERVERLESS4IOT [72] presents Model-Driven Engineering techniques for the design, deployment, and maintenance of hybrid applications involving FaaS functions and other software components over the Cloud-Edge-IoT continuum. A3-E [73] provides a unified model for managing where to execute a certain function based on the specific context and user requirements. However, the major focus of their work is on FaaS orchestration and not function delivery.

In the context of delivering functions to computing continuum, Glikson et al. [74] present a novel heterogeneous FaaS platform that deduces function resource specification using Machine Learning (ML) methods, performs smart function placement on Edge/Cloud based on a user-specified QoS requirement, and exploits data locality for function executions. Costless [13] constructs a cost graph for function workflow and formulates the FDN problem as a constrained shortest path problem to find the best delivery policy. However, Costless needs to profile all possible FDPs in the total search space. Anirban et al. [75] present a function placement framework that enables users to specify cost and latency requirements for each function and determines whether to execute the function on the edge device or in the cloud. The framework also identifies the resource needed to meet the performance goals. A similar work to ours is COSE [14], which uses BO-based performance models to select the optimal policy across cloud and cloud edge. However, COSE assumes that the cloud and cloud edge have homogeneous resources and cannot handle the tree-structured search space in HCC (e.g., Fig. 4).

VII. CONCLUSION

In order to deliver functions to the heterogeneous computing continuum efficiently, we propose *FaaSDeliver*, a lightweight function delivery engine to automatically find a cost-efficient FDP for each function. *FaaSDeliver* continuously learns the most cost-efficient FDP for FaaS functions through an adaptive TPE performance model and a heuristic transfer learning on the fly. Real system implementation and experimental evaluations demonstrate *FaaSDeliver* has a high chance to find the optimal (or sub-optimal) FDPs for different types of functions via fewer trials than some state-of-the-art approaches. We hope this paper will inspire more research into delivering FaaS functions to HCC so that the FaaS computing paradigm can be used in more scenarios.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments that improved the paper.

REFERENCES

- [1] A. Lambda, "AWS lambda," 2022, Accessed: Apr. 06, 2022. [Online]. Available: <https://aws.amazon.com/lambda>

³<https://aws.amazon.com/greengrass>

⁴<https://docs.microsoft.com/azure/iot-edge>

- [2] Azure, "Azure cloud functions," 2022, Accessed: Jan. 06, 2023. [Online]. Available: <https://azure.microsoft.com/services/functions>
- [3] Google, "Google cloud functions," 2022, Accessed: Apr. 06, 2022. [Online]. Available: <https://cloud.google.com/functions>
- [4] 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.
- [5] S. Ristov, S. Pedratscher, and T. Fahringer, "xAFCL: Run scalable function choreographies across multiple FaaS systems," *IEEE Trans. Serv. Comput.*, vol. 16, no. 1, pp. 711–723, Jan./Feb. 2023.
- [6] Z. Hu et al., "An efficient online computation offloading approach for large-scale mobile edge computing via deep reinforcement learning," *IEEE Trans. Serv. Comput.*, vol. 15, no. 2, pp. 669–683, Mar./Apr. 2022.
- [7] S. Tian, C. Chi, S. Long, S. Oh, Z. Li, and J. Long, "User preference-based hierarchical offloading for collaborative cloud-edge computing," *IEEE Trans. Serv. Comput.*, vol. 16, no. 1, pp. 684–697, Jan./Feb. 2023.
- [8] K. Fizza, N. Auluck, and A. Azim, "Improving the schedulability of real-time tasks using fog computing," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 372–385, Jan./Feb. 2022.
- [9] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Softw.: Pract. Experience*, vol. 51, no. 9, pp. 1936–1963, 2021.
- [10] K. M. Sim, "Intelligent resource management in intercloud, fog, and edge: Tutorial and new directions," *IEEE Trans. Serv. Comput.*, vol. 15, no. 2, pp. 1157–1174, Mar./Apr. 2022.
- [11] Z. Mann, A. Metzger, J. Prade, R. Seidl, and K. Pohl, "Cost-optimized, data-protection-aware offloading between an edge data center and the cloud," *IEEE Trans. Serv. Comput.*, vol. 16, no. 1, pp. 206–220, Jan./Feb. 2022.
- [12] M. Mastrolilli and O. Svensson, "(Acyclic) job shops are hard to approximate," in *Proc. IEEE Symp. Found. Comput. Sci.*, 2008, pp. 583–592.
- [13] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *Proc. IEEE/ACM Symp. Edge Comput.*, 2018, pp. 300–312.
- [14] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring serverless functions using statistical learning," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 129–138.
- [15] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyperparameter optimization," in *Proc. Neural Inf. Process. Syst. Found.*, 2011, pp. 2546–2554.
- [16] Y. Ci, X. Zhao, Y. Li, and Z. Zheng, "Virtdev: Towards providing edge services," *IEEE Trans. Serv. Comput.*, vol. 15, no. 5, pp. 3089–3100, Sep./Oct. 2022.
- [17] H. Sami, A. Mourad, H. Otok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Trans. Serv. Comput.*, vol. 15, no. 5, pp. 2671–2684, Sep./Oct. 2022.
- [18] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, "On merits and viability of multi-cloud serverless," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 600–608.
- [19] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Serv.*, 2019, pp. 68–75.
- [20] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *Proc. IEEE Glob. Commun. Conf.*, 2020, pp. 1–6.
- [21] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 205–218.
- [22] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge AI," in *Proc. 2nd USENIX Workshop Hot Topics Edge Comput.*, 2019, pp. 1–10.
- [23] Y. G. Kim and C. Wu, "AutoScale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchitecture*, 2020, pp. 1082–1096.
- [24] P. Kumar, "Medtronic makes diabetes management easier with real-time insights from IBM streams," 2022, Accessed: Jan. 06, 2023. [Online]. Available: <https://www.ibm.com/cloud/blog/medtronic-makes-diabetes-management-easier-real-time-insights-ibm-streams>
- [25] Knative, "Enterprise-grade serverless on your own terms," 2022, accessed Jan. 6, 2023. [Online]. Available: <https://knative.dev>
- [26] Openfaas, "Serverless functions, made simple," 2022, accessed: Jan. 6, 2023. [Online]. Available: <https://openfaas.com>
- [27] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 133–146.
- [28] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1994–2004.
- [29] D. Jiang, G. Pierre, and C. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *Proc. Conf. Web Appl. Develop.*, 2011.
- [30] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. 11th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2013, pp. 77–88.
- [31] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 127–144.
- [32] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for Big Data analytics," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 469–482.
- [33] Y. Ozaki, Y. Tanigaki, S. Watanabe, and M. Onishi, "Multiobjective tree-structured parzen estimator for computationally expensive optimization problems," in *Proc. Genet. Evol. Comput. Conf.*, 2020, pp. 533–541.
- [34] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–14.
- [35] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proc. 30th Int. Conf. Mach. Learn.*, 2013, pp. 115–123.
- [36] D. R. Jones, "A taxonomy of global optimization methods based on response surfaces," *J. Glob. Optim.*, vol. 21, no. 4, pp. 345–383, 2001.
- [37] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1436–1445.
- [38] P. Goodwin and R. Lawton, "On the asymmetry of the symmetric MAPE," *Int. J. Forecasting*, vol. 15, no. 4, pp. 405–408, 1999.
- [39] V. Perrone and H. Shen, "Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 12751–12761.
- [40] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. IEEE/ACM 52nd Annu. Int. Symp. Microarchitecture*, 2019, pp. 1063–1075.
- [41] P. Menage, "Control groups," 2022, Accessed: Jan. 06, 2023. [Online]. Available: <https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html>
- [42] Azureprice.net, "Azure VM comparison," [EB/OL], 2018, Accessed: Apr. 2022. [Online]. Available: <https://azureprice.net/>
- [43] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proc. ACM SIGKDD Int. Conf. Knowl. Data Mining*, 2019, pp. 2623–2631.
- [44] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012.
- [45] R. Martinez-Cantin, "Bayesopt: A Bayesian optimization library for non-linear optimization, experimental design and bandits," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3735–3739, 2014.
- [46] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. de Freitas, "Bayesian optimization in high dimensions via random embeddings," in *Proc. Int. Joint Conf. Artif. Intell.*, 2013, pp. 1778–1784.
- [47] X. Ma and M. B. Blaschko, "Additive tree-structured conditional parameter spaces in Bayesian optimization: A novel covariance function and a fast implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 9, pp. 3024–3036, Sep. 2021.
- [48] Azure, "Azure cloud functions trace 2019 v2 (invocations per function md anon d01)," 2022, Accessed: Jan. 06, 2023. [Online]. Available: <https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md>
- [49] H. Qiu et al., "SIMPPPO: A scalable and incremental online learning framework for serverless resource management," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 306–322.
- [50] S. Eismann et al., "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Trans. Softw. Eng.*, vol. 48, no. 10, pp. 4152–4166, Oct. 2022.
- [51] D. Mvondo et al., "OFC: An opportunistic caching system for FaaS platforms," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 228–244.
- [52] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 419–434.
- [53] H. Fingler, A. Akshintala, and C. J. Rossbach, "USETL: Unikernels for serverless extract transform and load why should you settle for less?," in *Proc. 10th ACM SIGOPS Asia-Pacific Workshop Syst.*, 2019, pp. 23–30.

- [54] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 57–70.
- [55] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2018, pp. 159–169.
- [56] AWS, "AWS compute optimizer," 2023, Accessed: Jan. 06, 2023. [Online]. Available: <https://aws.amazon.com/compute-optimizer/>
- [57] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "SLAM: SLO-aware memory optimization for serverless applications," in *Proc. IEEE 15th Int. Conf. Cloud Comput.*, 2022, pp. 30–39.
- [58] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-aware dynamic resource configuration for serverless function workflows," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1868–1877.
- [59] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 615–632, Mar. 2021.
- [60] A. Ali, R. Pincirol, F. Yan, and E. Smirni, "Optimizing inference serving on serverless platforms," *Proc. VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.
- [61] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," in *Proc. Int. Syst. Storage Conf.*, 2017, Art. no. 28:1.
- [62] A. Aske and X. Zhao, "Supporting multi-provider serverless computing on the edge," in *Proc. Workshop Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 20:1–20:6.
- [63] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 265–279.
- [64] A. Ahuja, G. Gupta, and S. Kundu, "A serverless approach to federated learning infrastructure oriented for IoT/edge data sources (student abstract)," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 15747–15748.
- [65] B. Wang, A. Ali-Eldin, and P. J. Shenoy, "LaSS: Running latency sensitive serverless computations at the edge," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2021, pp. 239–251.
- [66] R. Jin and Q. Yang, "EdgeFaaS: A function-based framework for edge computing," *CoRR*, vol. abs/2210.01410, 2022. [Online]. Available: <https://arxiv.org/abs/2210.01410>
- [67] S. Nastic et al., "A serverless real-time data analytics platform for edge computing," *IEEE Internet Comput.*, vol. 21, no. 4, pp. 64–71, Jul. 2017.
- [68] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in *Proc. IEEE 13th Int. Conf. Cloud Comput.*, 2020, pp. 609–618.
- [69] R. Kumar et al., "Coding the computing continuum: Fluid function execution in heterogeneous computing environments," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2021, pp. 66–75.
- [70] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, "Serverless workflows for containerised applications in the cloud continuum," *J. Grid Comput.*, vol. 19, no. 3, 2021, Art. no. 30.
- [71] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "FogFlow: Easy programming of IoT services over cloud and edges for smart cities," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 696–707, Apr. 2018.
- [72] N. Ferry, R. Dautov, and H. Song, "Towards a model-based serverless platform for the cloud-edge-IoT continuum," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput.*, 2022, pp. 851–858.
- [73] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, "A unified model for the mobile-edge-cloud continuum," *Trans. Internet Techn.*, vol. 19, no. 2, pp. 29:1–29:21, 2019.
- [74] S. K. R. and J. Lakshmi, "QoS aware FaaS for heterogeneous edge-cloud continuum," in *Proc. IEEE 15th Int. Conf. Cloud Comput.*, 2022, pp. 70–80.
- [75] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. IEEE Int. Symp. Cluster, Cloud Internet Comput.*, 2020, pp. 41–50.



Guangba Yu (Student Member, IEEE) received the master's degree from Sun Yat-Sen University, China, in 2020. He is currently working toward the PhD degree with the School of Computer Science and Engineering, Sun Yat-Sen University, China. His current research areas include distributed system, cloud computing, and AI driven operations.



Zilong He (Graduate Student Member) received the BE and MS degrees from Sun Yat-sen University, China, in 2019 and 2021, respectively. He is currently working toward the PhD degree with the School of Computer Science and Engineering, Sun Yat-sen University, China. His current research areas include anomaly detection algorithms, AI driven operations.



Pengfei Chen received PhD degree from the Department of Computer Science, Xi'an Jiaotong University. He is currently an associate professor with the School of Computer Science and Engineering of Sun Yat-sen University. Meanwhile, he is a PhD advisor. He is interested in distributed systems, AIOps, cloud computing, Microservice and Blockchain. Especially, he has strong skills in cloud computing. So far, he has published more than 50 papers in some international conferences including *International Conference on Automated Software Engineering*, *ACM/IEEE ICSE*, *IEEE INFOCOM*, *WWW* and journals including *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Neural Networks and Learning Systems*, *IEEE Transactions on Reliability*, *IEEE Transactions on Services Computing*. He serves as of program committee member of multiple conferences and reviewers of some internal journals such as *IEEE Transactions on Cybernetics*, *Information Science*, and *Neurocomputing*.



Zibin Zheng (Fellow, IEEE) received the PhD degree in computer science and engineering from the Chinese University of Hong Kong. He is currently a professor and the deputy dean with the School of Software Engineering, Sun Yat-sen University, Guangzhou, China. He authored or coauthored more than 200 international journal and conference papers, including one ESI hot paper and six ESI highly cited papers. His research interests include blockchain, software engineering, and services computing. He is a fellow of the IET. He was the recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, the Best Student Paper Award at ICWS2010.



Jingrun Zhang received the BE degree from Sun Yat-sen University, in 2022. He is currently working toward the MS degree with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. His current research areas include high performance computing and reinforcement learning-driven operations.



Xiaoyun Li received the BE degree from Sun Yat-sen University, in 2019. She is currently working toward the PhD degree with the School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China. Her current research areas include log analysis and AI-driven operations.