HyFaaS

Accelerating Serverless Workflows by Unleashing Hybrid Resource Elasticity

Yue, Xiaofei; Yang, Song; Li, Fan; Zhu, Liehuang; Wang, Xu; Feng, Zhen; Kuipers, Fernando A.

DOI
10.1109/TPDS.2025.3632089

Publication date
2025

Document Version
Final published version

Published in
IEEE Transactions on Parallel and Distributed Systems

HyFaaS: Accelerating Serverless Workflows by Unleashing Hybrid Resource Elasticity

Xiaofei Yue, Song Yang, *Senior Member, IEEE*, Fan Li, *Member, IEEE*, Liehuang Zhu, *Senior Member, IEEE*, Xu Wang, Zhen Feng, and Fernando A. Kuipers, *Senior Member, IEEE*

***Abstract*—Serverless computing promises fine-grained resource elasticity and billing, making it an attractive way to build complex applications as multi-stage workflows. Nonetheless, existing workflow orchestration ignores the heterogeneous demands of the computation and communication parts within a stage, potentially resulting in resource inefficiency on either side. In this paper, we advocate for *computation-communication-separated orchestration* to unleash hybrid resource (i.e., compute and network) elasticity. We present HyFaaS, a serverless workflow orchestrator that improves performance while ensuring cost efficiency. It seamlessly decouples computation and communication as a series of hybrid stages re-expressed within HyDAG, a novel workflow abstraction. HyFaaS uses a gray-box profiling model to identify their Pareto-optimal saturated configurations, and then deploys the saturated workflow to juggle communication and scaling overheads through two-level HyDAG partitioning. Along with event-driven runtime fine-tuning, HyFaaS further scales down the non-critical stages to reduce cost via branch-aware coordination. Experimental results show that HyFaaS surpasses existing solutions by 32.7%–50.4% on end-to-end latency, while lowering cost by up to 1.37×.**

***Index Terms*—Serverless computing, hybrid resource elasticity, workflow orchestration, computation and communication.**

I. Introduction

SERVERLESS computing [1], [2] is an emerging paradigm that reshapes the development of intricate applications in the cloud-native era. Cloud vendors provide serverless services in the form of Function-as-a-Service (FaaS) for developers [3]. Their business logic is packaged as fine-grained, lightweight functions, each running within a container and connected into *workflows*. The underlying resource provisioning of functions is borne by serverless platforms, allowing users to focus only on programming. Coupled with the agile scalability and "*pay-per-use*" billing, serverless workflows have spread in latency-sensitive applications, such as interactive query [4], [5], video processing [6], [7], and machine learning [8], [9].

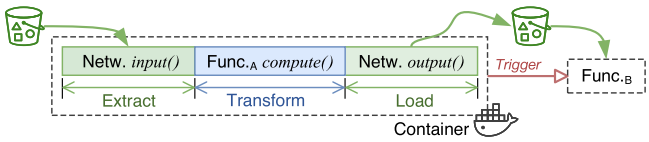


Fig. 1. The ETL pattern of serverless functions.

The orchestration of serverless workflows is eased by commercial clouds (e.g., AWS Step Function [3]) and open-source systems (e.g., OpenWhisk Composer [10]). The workflows are structured into stages, each consisting of parallel instances of the corresponding function [11]. Due to the stateless nature of functions, communication between the stages requires network I/O relying on external storage, with the Extract-Transform-Load (ETL) pattern [12] shown in Fig. 1. With the fine-grained resource elasticity of serverless computing, recent works [11], [13], [14], [15] focus on orchestrating workflows by sensibly matching per-stage requirements (e.g., the number and size of instances [16]) to optimize the *user-centric metrics* [17], such as latency and cost. However, a stage is typically characterized as either I/O- or compute-intensive [13], [18]. Such monolithic resource provisioning ignores the *varying resource sensitivity of computation and communication operations within a stage*, resulting in resource inefficiency. For instance, scaling more resources for an I/O-intensive stage can improve communication efficiency, but incurs an additional compute cost due to low utilization. The reason is that each instance is isolated for fine-grained network bandwidth that increases as it scales up (e.g., with more CPUs or memory) [16], [19]. Conversely, this also applies to compute-intensive stages. Hence, it is essential to rethink the orchestration of serverless workflows.

In our view, hybrid resource elasticity (in compute and network) should be unleashed with *computation-communication-separated orchestration*. The first priority is how to decouple these two operations in each stage, without violating the ETL

Received 10 January 2025; revised 17 June 2025; accepted 6 November 2025. Date of publication 12 November 2025; date of current version 4 December 2025. The work of Song Yang was supported partially by the National Natural Science Foundation of China (NSFC) under Grant 62472028 and Grant 62172038, in part by the Beijing Natural Science Foundation under Grant 4232033, and in part by the National Key Research and Development Program of China under Grant 2023YFB3107300. The work of Fan Li was supported in part by the NSFC under Grant 62372045. The work of Liehuang Zhu was supported in part by the Yunnan Provincial Major Science and Technology Special Plan Projects under Grant 202302AD080003. Recommended for acceptance by S. Al-Kiswany. *(Corresponding author: Song Yang.)*

Xiaofei Yue, Song Yang, and Fan Li are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: xfyue@bit.edu.cn; S.Yang@bit.edu.cn; fli@bit.edu.cn).

Liehuang Zhu is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: liehuangz@bit.edu.cn).

Xu Wang is with the State Key Laboratory of Public Big Data, Guizhou University, Guiyang 550025, China (e-mail: xuwang@gzu.edu.cn).

Zhen Feng is with the Jinan Inspur Data Technology Company Ltd., Jinan 250101, China (e-mail: fengzh@inspur.com).

Fernando A. Kuipers is with the Networked Systems Group, Department of Software Technology, Delft University of Technology, 2628 XE Delft, Netherlands (e-mail: F.A.Kuipers@tudelft.nl).

Digital Object Identifier 10.1109/TPDS.2025.3632089

pattern. By reusing shared memory that enables fast function communication [20], [21], the communication operation can be seamlessly extracted from each function, thus facilitating its decoupling with the computation operation for a stage. In addition, we still need to tackle three fundamental challenges to separately orchestrate them (e.g., with specific deployment and configuration). *First*, a user-facing workflow produces diverse stages with dynamic input sizes and contents, due to varying user-input across invocations [17]. Given the privacy limits of platforms [1], [3], transparently and accurately estimating their performance is critical to optimize workflow latency and cost. *Second*, co-locating adjacent stages facilitates communication by enhancing memory efficiency [13], [22], but with increased scaling overhead (i.e., the time interval between the start of the first parallel instance and last one) [23]. For example, 48% of workflows have a latency skew of at least 2× among instances caused by over-scaling [24]. *Third*, stages on the non-critical paths of workflow DAGs imply potential cost efficiency, without influencing overall performance. To achieve this efficiency throughout the workflow, the non-critical stages should also be considered under complex inter-stage correlations.

To meet the challenges, we propose HyFaaS, a computation-communication-separated orchestrator that offers high performance and cost efficiency for serverless workflows. By seamlessly decoupling computation and communication operations as compute and I/O stages, we clarify the manner and goal of the separated orchestration. The workflow composed of these hybrid stages is expressed using a novel abstraction, HyDAG. It can offer fine-grained information for the co-scheduling of computing and network resources. We identify Pareto-optimal saturated configurations for each hybrid stage with a gray-box profiling model, which merges step-wise white-box modeling and probabilistic black-box prediction. To fully reduce end-to-end latency by juggling communication and scaling overheads, HyFaaS adopts a two-Level HyDAG Partitioning (LHP) algorithm to deploy the saturated workflow across servers. Finally, HyFaaS determines cost-efficient configuration for non-critical stages by assigning loose latency bounds for the corresponding branches. This NP-hard problem is amenable to independent branch-level coordination. Coupled with runtime fine-tuning, we effectively gain the implied cost efficiency.

To sum up, the contributions are summarized below:

- We design HyFaaS, a novel computation-communication-separated orchestrator that optimizes latency and cost efficiency for serverless workflows with HyDAG abstraction.
- We construct a gray-box model to identify Pareto-optimal saturated configurations of the hybrid stages, and deploy each of them across servers using an LHP algorithm for workflow performance improvement.
- We propose a branch-aware workflow coordination algorithm, which gains the implied cost efficiency by deciding cost-efficient configurations for non-critical stages.
- We implement a prototype of HyFaaS, which outperforms state-of-the-art solutions by 32.7%–50.4% on end-to-end latency, while yielding cost savings of up to 1.37×.

The rest of this paper is arranged as follows. In Section II, we provide an outline of serverless workflows and explain the motivation behind our insight. Section III depicts the system architecture and workflow abstraction of HyFaaS. The designs of its performance profiling and orchestration optimization are presented in Sections IV and V, respectively. Section VI shows the implementation of HyFaaS. Section VII demonstrates the experimental results. The related work is reviewed in Section VIII, and the paper concludes in Section IX.

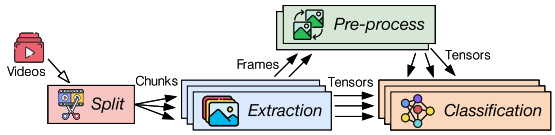


Fig. 2. An example ML inference workflow for video analytics.

II. Background and Motivation

In this section, we provide a brief background on serverless workflows, and the motivation and challenges of computation-communication-separated orchestration.

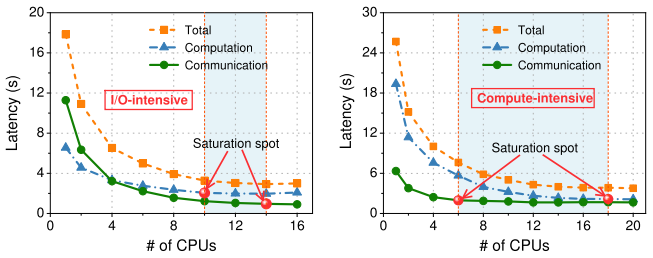
A. Serverless Workflows

In the cloud-native era, many popular applications [6], [8], [13] have shifted to serverless architectures due to the ease of management and fine-grained billing. Another engaging aspect is the ability to scale compute resources flexibly and rapidly, especially for latency-sensitive large applications [4], [8], [23]. These applications usually involve intricate logic, decomposed and organized in the form of multi-stage workflows.

Taking OpenWhisk [10] as an example, the workflow logic is stated in the form of a Directed Acyclic Graph (DAG) via the composer. Fig. 2 demonstrates the DAG of an ML serving workflow designed to classify frames in given video sets [11]. Each node represents a stage composed of parallel instances with identical logic, and edges denote data flow dependencies between stages. With an event-driven architecture, OpenWhisk supports distributed function deployment and serving at scale. The controller receives workflow invocations and initiates the conductor, which monitors the status of each stage and triggers it once the completion of dependent ones. Then, this stage is parsed to spawn a specific number of requests to the parallel instances. Based on a load-balancing deployment policy, they are routed with a specific configuration using a message queue. The invoker on each server pulls requests from the subscribed topic, and serves each of them in a short-lived container. They execute via three phases in the ETL pattern [12]. As illustrated in Fig. 1, each function first (E) extracts data from external storage, then (T) transforms it by executing the function code, and finally (L) loads the result back.

B. Resource Inefficiency in Current Workflows

Easily ignored, serverless computing can offer fine-grained resource isolation and provisioning in terms of network, not just in compute, which we refer to as *hybrid resource elasticity*. In fact, the in-container network bandwidth is limited for production fairness [24]. It increases as the function instance



(a) Frame extraction. (b) Classification.

Fig. 3. The varying resource sensitivities of I/O- and compute-intensive stages in the video analytics workflow.

TABLE I
CHARACTERIZATION OF WORKFLOW APPLICATIONS

Application	I/O-Intensive	Compute-Intensive
Data analytics	Map, groupby	Reduce, join
ML training	Data pre-processing	Training
Image processing	Image retrieval	Watermark, blur
Video analytics	Frame extraction	Object recognition

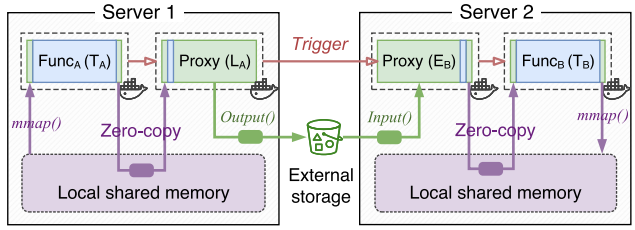
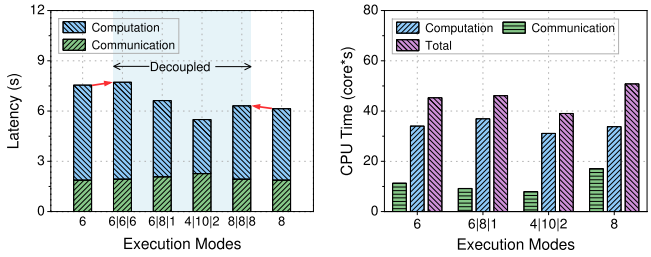


Fig. 4. The computation-communication decoupling for serverless functions. The communication operations of functions A and B (see Fig. 1) are *logically hidden* by the local memory, and *physically extracted* into the proxy functions.



(a) Latency breakdown. (b) Resource usage.

Fig. 5. The function latency breakdown and resource usage of the decoupled and coupled groups. *The decoupled group "$i|j|k$" represents allocating i, k and j CPUs to the* E, L*phases (i.e., communication operation) and* T*phase (i.e., computation operation), respectively.*

scales up vertically [16], [19]. Obeying the ETL pattern, each function includes two main operations executed alternatively: communication (E&L phases) and computation (T phase). So, a larger function size automatically yields better performance in both the computation and communication operations. With the workflow plotted in Fig. 2 as an example, we explore the resource demands and sensitivities of the two operations.

Since excess memory does not lower function latency, we enable a CPU-oriented configuration, in line with prior works [8], [16]. Here we scale up stages by increasing the number of instances (i.e., fan-out degree), after a single CPU has saturated the bandwidth [16], [24]. Fig. 3 shows the average latency in computation and communication operations for all instances. For the *extraction* stage, increasing CPUs can notably decrease the video chunk downloading time with higher bandwidth. On the other hand, the latency reduction in computation tends to plateau quickly, owing to the relatively low logic complexity of the frame filtering and decoding tasks. The *classification* stage is the opposite. The reason lies in the involved extensive tensor handling and matrix calculations, making it compute-intensive. Overall, the computation and communication operations within a stage exhibit varying resource *saturation spots* and sensitivities. For example, one of these operations in the blue areas of Fig. 3 will exhibit resource inefficiency.

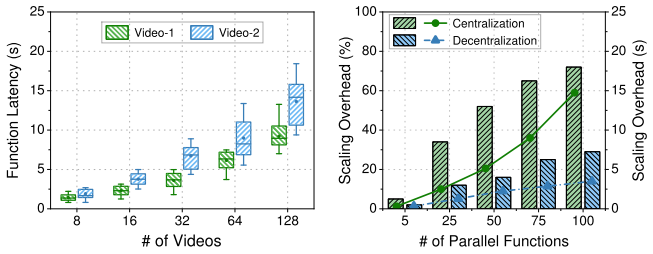
In practice, most workflow applications consist of both I/O and compute-intensive stages, as depicted in Table I. Existing workflow orchestrators [11], [13], [14], [16], [17] focus at best on conducting the above monolithic resource provisioning on demand. None of them takes into account the varying resource sensitivities of computation and communication operations in a stage, without unleashing the hybrid resource elasticity.

C. Computation-Communication-Separated Orchestration

With the ETL pattern, both computation and communication operations are integral to a function, making it a difficult task to orchestrate them separately. A naive method is to adjust its configuration at runtime (e.g., using *docker update* command [25]) before executing each ETL phase. When applied to the workflows, however, it involves numerous container updates, which results in higher management complexity and resource fragmentation on servers. Further, the benefits of this method may be offset by the irreversible performance drop for short-lived functions, due to the ensuing management overhead.

Fortunately, recent works [20], [21], [22], [26] have altered the communication dilemma in ETL. They facilitate zero-copy interaction for functions on the same server using local shared memory, avoiding network and storage access overheads. Our insight is that this fast communication scheme can be reused to hide the local I/O of functions, as illustrated in Fig. 4. By introducing generic and paired *proxy* functions for in-memory data movement between servers, functions A, B only need to seamlessly read from and write to local memory for execution. As a result, the communication operation (i.e., E&L phases) of each function can be logically offloaded to physical proxy functions, leaving it only with the computation operation (i.e., T phase). Note that its E and L phases correspond, respectively, to a receiver proxy and a sender proxy in different pairs. Now, each ETL phase belongs to a distinct function (also i.e., hybrid stage, as described in Section III-B), achieving the decoupling of the computation and communication operations throughout a workflow. They can be separately orchestrated, as usual, to match the varying resource sensitivities and saturations.

Fig. 5 displays that the above decoupling method achieves up to 25.7% and 23.6% reductions in latency and CPU usage for the *classification* stage (see "4| 10 |2" mode), respectively. Also, we simply need to pay the cost of negligible overheads



(a) Latency variation. (b) Scaling overhead.

Fig. 6. The (a) latency variation with varying input sizes and content, as well as (b) scaling overhead and its share with various parallelism.

(see *red arrows*), which primarily arises from the fast memory interactions, regardless of data size [13], [21].

D. Challenges

To improve workflow performance through the above cost-efficient method, we need to tackle three basic challenges:

1) Transparent resource estimation with dynamic input sizes and content: Serverless workflows are invoked by users with varying input data. To explore the impact of such inputs on resource demands behind fluctuating performance, we run the *video analytics* workflow with two types of video sets (see Section VII). Fig. 6(a) displays the latency variations of all its function instances. When the number of videos exceeds 32, the function latency grows rapidly, peaking at 2.6× caused by under-provisioning. The increased range of latency variations means diverse resource demands across stages. Note that the variation in Video-1 is smaller by virtue of its lower absolute value. Overall, input size dominates performance, but the type of input content also plays a vital role in function complexity. Existing studies fail to consider both factors, either profiling each workflow invocation independently [15] or ignoring the potential influence of content [16], [17]. Coupled with privacy limits, it is non-trivial to transparently and accurately estimate resource demands for each stage in workflows.

2) Tough trade-off between communication and scaling overheads: Co-locating successive stages can reduce communication overhead via memory interactions. That is, all of their instances are co-located to avoid lag effects caused by slower storage [13]. Note that they tend to be highly parallel due to limited function resources [24]. As shown in Fig. 6(b), the scaling overhead of Centralization grows with the number of instances and dominates the total latency by up to 72% (bar). This is because it deploys the entire stage to a single server. Then, the locality gains may be offset by the ensuing scaling pressure. In contrast, decentralizing its parallel instances for load-balancing keeps the scaling overhead below 4.8 s (line). The rise in the share is due to the stage being expedited by more instances. Given the advances in container startup time to sub-milliseconds [27], this also motivates us to enable fast scaling to prevent introducing new bottlenecks. At the same time, it should be balanced with the communication overhead carefully for overall performance improvement.

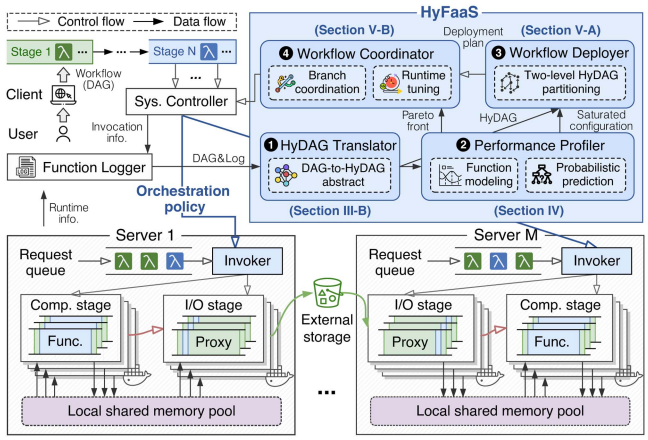


Fig. 7. The system overview of HyFaaS.

3) Potential cost efficiency implied in workflow DAGs: A general workflow is composed of multiple stages that exhibit cascading correlations in its DAG [13], [16]. The correlations imply the opportunity to gain cost efficiency from fine-grained billing (see Fig. 5(a)). Concretely, stages on non-critical paths can be properly scaled down to reduce cost, without degrading the entire workflow performance. However, the diversity and complexity of DAG structures make it difficult to co-consider the configuration of these non-critical stages, particularly for computation-communication-separated workflows.

III. The HyFaaS Framework

This section introduces HyFaaS, which unleashes the hybrid resource elasticity with computation-communication-separated orchestration for serverless workflows.

A. Overview

HyFaaS is a computation-communication-separated orchestrator for serverless workflows. According to our key insight in Section II, HyFaaS strives to improve workflow performance while ensuring cost efficiency. This aligns with the fact that latency-sensitive workflows have greater needs for agile elasticity. Hosting them on serverless platforms is an industry best practice [28]. The system overview is shown in Fig. 7. HyFaaS drives normal functions with specific logic to seamlessly input local in-memory data (with negligible overhead, as stated in Section II-B) for execution, then output back. Data movement across servers is undertaken by proxy processes, wrapped as dedicated functions, in the system. The proxy functions are in pairs and run in sequence, with the receiver being triggered after the sender uploads data to external storage. To achieve its goal by unleashing the hybrid resource elasticity, HyFaaS orchestrates each hybrid stage comprising parallel instances of the normal or proxy functions in the workflows.

B. Workflow Abstraction of HyFaaS

The conventional server-oriented workflow DAG (see Fig. 8 (left)) is unable to explicitly express network-side information,

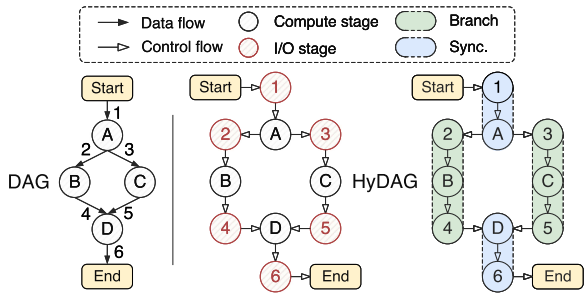


Fig. 8. An example DAG (left) and the corresponding HyDAG (middle) and branch-wise HyDAG (right).

instead wrapping it in compute nodes. This abstraction, which lacks support for the co-scheduling of computing and network resources [13], may result in sub-optimal orchestration. Thus, we design HyDAG, a novel workflow abstraction.

HyDAG: As illustrated in Fig. 8, a serverless workflow is re-expressed as a HyDAG $\mathcal{G}(\mathcal{V},\mathcal{E})$ composed of hybrid stage nodes $\mathcal{V}=\{v_S,v_1,\ldots,v_E\}$, where v_S and v_E are the dummy start node and end node, respectively. Each of others $v_i\in\mathcal{V}$ is either a compute stage with parallel instances of normal functions $\{f_{i,1},f_{i,2},\ldots\}$ executed on CPUs (T phase), or an I/O stage with paired proxy function instances $\{\langle f^{snd}_{i,1},f^{rcv}_{i,1}\rangle,\ldots\}$ between servers (L&E phases). Also, the edges $\mathcal{E}$ are transformed into inter-stage control flow dependencies that fit well with current control-flow-based serverless platforms [2], [19]. For example, edge $e_{i,j}(1\le i\neq j\le|\mathcal{V}|)\in\mathcal{E}$ constrains that hybrid stage v_j cannot start before v_i ends. Fig. 8 depicts the translation of a DAG-based workflow statement into HyDAG format. For any pair of nodes (i.e., compute stages) connected by an edge in the DAG, we insert a new node (i.e., I/O stage) between them for cross-server communication.

C. Running Process of HyFaaS

As illustrated in Fig. 7, HyFaaS is located with the system controller, which consists of three modules: HyDAG translator, performance profiler, and optimizer engine.

The ❶ HyDAG translator and ❷ performance profiler run in the background. After a workflow is registered, the HyDAG translator transforms the user-submitted DAG statement into a HyDAG format (Section III-B). Then, the performance profiler periodically retrieves its recent invocation logs. Utilizing this information, it identifies the saturated configurations for both compute and I/O stages based on the *gray-box* profiling model (Section IV), which combines step-wise white-box modeling and probabilistic black-box prediction (**Challenge 1**).

Once receiving a user's workflow invocation, HyFaaS uses the optimizer engine to make an optimal orchestration policy. First, the ❸ workflow deployer partitions the HyDAG of the saturated workflow at the instance granularity through an LHP algorithm (Section V-A). Each sub-HyDAG is deployed on a distinct server to juggle scaling and communication overheads, so as to minimize end-to-end latency (**Challenge 2**). Then, the ❹ workflow coordinator assigns loose latency bounds to non-critical branches to individually determine their cost-efficient configurations (Section V-B). Along with event-driven runtime fine-tuning, the implied cost efficiency is effectively harvested, without sacrificing the end-to-end latency (**Challenge 3**).

TABLE II
THE NOTATIONS AND THEIR DEFINITIONS

Notation	Definition
$\mathcal{G}(\mathcal{V},\mathcal{E})$	A serverless workflow HyDAG.
$v_i\in\mathcal{V}$	The i-th (compute or I/O) stage in the workflow.
$d_i\in\mathcal{D}$	The number of function instances for stage v_i.
$n_i\in\mathcal{N}$	The number of CPUs per function instance for stage v_i.
p_i^m	The number of instances for stage v_i on the m-th server.
$t^{i,j}_{init},t^{i,j}_{comp}$	The initialization latency and compute latency of the j-th instance for compute stage v_i, respectively.
$t^{i,j}_{comm}$	The transfer latency of the j-th instance for I/O stage v_i.
S_i	The total input size of stage v_i.
γ_i	The share of data to be transferred for I/O stage v_i.
t_i,T	The latency of stage v_i and the workflow, respectively.
c_i,C	The cost of stage v_i and the workflow, respectively.
t_i^Φ,c_i^Φ	The saturated latency and cost of stage v_i, respectively.
Θ_i	The orchestration domain of stage v_i.
$x_i^\theta\in X$	A binary variable $x_i^\theta=1$ if stage v_i is orchestrated with $\theta\in\Theta_i$ at runtime; and 0 otherwise.
$\mathcal{G}'(\mathcal{V}',\mathcal{E}')$	A branch-wise serverless workflow HyDAG.
$B_i\in\mathcal{V}'$	The i-th branch in the branch-wise workflow.
$T_\Phi^{B_i}$	The saturated latency of branch B_i.
$T_\Psi^{B_i}$	The assigned latency bound for branch B_i.

IV. PERFORMANCE PROFILING

HyFaaS periodically retrieves workflow invocation logs to profile each stage (i.e., node in the HyDAG) offline, estimating their saturated resource demands. Instead of directly exploring configuration-latency mappings, we develop a *gray-box* model that integrates step-wise white-box modeling with probabilistic black-box prediction, which mitigates the performance drawbacks in either. The vital notations are listed in Table II. We define $[n]\triangleq\{1,2,\ldots,n\}$ in this paper.

A. Function and Stage Modeling

The compute stage and I/O stage are functionally heterogeneous, we separately profile them into fine-grained steps.

1) Compute Stage: Due to the I/O hiding mechanism, each parallel function instance in compute stage $v_i\in\mathcal{V}$ is composed of two steps: initialization step and compute step.

Initialization step: The initialization step consists of function serving and container startup. At each server, the invoker accepts and serves concurrently the corresponding invocations by checking the local shared container pool [10]. Next, each invocation is either routed to an available instance (i.e., warm-start) or triggers the creation of a new container from scratch (i.e., cold-start). For function instance $f_{i,j}$ deployed on the m-th server, which hosts a total of p_i^m instances of stage v_i, its initialization latency is represented as:

$$t^{i,j}_{init}=o^{i,j}_m\cdot\overline{T^{i,m}_{scale}}+T^i_{cold},\tag{1}$$

where $\overline{T^{i,m}_{scale}}$ is the average scaling latency caused by serving p_i^m invocations, which is platform-specific and can be measured as in Fig. 6(b). Also, $o^{i,j}_m$ is the serving order of $f_{i,j}$ in the m-th

server, and T^i_{cold} is the cold-start overhead. If an idle instance is available, then $T^i_{cold} = 0$.

Compute step: Given the number of function instances and CPUs per instance be $d_i \in \mathcal{D}$ and $n_i \in \mathcal{N}$, respectively, for stage v_i. In the compute step, each instance evenly processes S_i/d_i-sized data with n_i CPUs through *multi-core processing* [16], [29], where S_i refers to the total input size of stage v_i. Then, the compute latency $t^{i,j}_{comp}$ of function instance $f_{i,j}$ is the maximum latency of its parallel sub-tasks, given by:

$$t^{i,j}_{comp} = \max_{k\in[\lceil n_i \rceil]} \left(t^{i,j,k}_{comp}\right), \tag{2}$$

$$t^{i,j,k}_{comp} = (k-1)\cdot T^i_{block} + T^i_{start} + t^{i,j,k}_{exec}, \tag{3}$$

where $t^{i,j,k}_{comp}$ denotes the latency of the k-th sub-task. Although modern operating systems can rapidly dispatch processes (or threads) onto CPUs to run these sub-tasks, their creation covers non-parallelizable actions (e.g., system calls and kernel object initialization). Each sub-task waits (i.e., block overhead T^i_{block}) until the previous one completes these actions, making them start sequentially [30]. Also, T^i_{start} is the startup overhead and $t^{i,j,k}_{exec}$ is the latency for a CPU to process $\frac{S_i}{d_i \cdot n_i}$-sized data.

2) I/O Stage: Each pair of proxy function instances in I/O stage $v_i \in \mathcal{V}$ performs only data transfer step. This is because proxy functions, by nature, serve for generic data movement between network and memory. Their instances can be cached and reused frequently by any I/O stage in the system, making the initialization latency negligible [2].

Transfer step: This step involves a double transfer process. The sender proxy instances of v_i upload in-memory data from its upstream (i.e., precursor in the HyDAG) to external storage. Then, the receiver proxy instances download the data to local memory. Let the size of the data to be moved by I/O stage v_i across servers be $S'_i = \gamma_i S_i$, where γ_i refers to the share. The γ_i-value directly influences the deployment of v_i and depends on the relative position across the connected pair of compute stages. For efficiency, we assign equal bandwidth (i.e., d_i and n_i) to the senders and receivers. Thus, the data transfer latency $t^{i,j}_{comm}$ of paired proxy instances $\langle f^{snd}_{i,j}, f^{rcv}_{i,j}\rangle$ is:

$$t^{i,j}_{comm} = 2\left(\frac{S'_i}{d_i \cdot \min(n_i \cdot W, B)} + T_{req}\right), \tag{4}$$

where W is the network bandwidth bundled to a CPU, B is the bandwidth cap of an instance, and T_{req} is the access overhead for external storage during an I/O request.

3) Hybrid Stage Model: The step-wise function modeling shows that the deployment of stages also affects their performance (in the initialization and data transfer). Recall that p^m_i denotes the number of instances for v_i on the m-th server. We therefore define its deployment (i.e., instance distribution) be $p_i = \{p^1_i, \ldots, p^m_i, \ldots\} \in \mathcal{P}_i$, where $\sum_m p^m_i = d_i$. The total latency t_i of hybrid stage v_i is categorically defined as:

$$t_i = \begin{cases} \max_{j\in[d_i]} \left(t^{i,j}_{init}(p_i) + t^{i,j}_{comp}(d_i, n_i)\right), & \text{Compute stage} \\ \max_{j\in[d_i]} \left(t^{i,j}_{comm}(d_i, n_i, p_i)\right). & \text{I/O stage} \end{cases} \tag{5}$$

Given the sub-linear scaling latency in $t^{i,j}_{init}$, the total latency of a compute stage is dominated by the straggler server with the most instances. Also, the cost c_i of hybrid stage v_i is the aggregated cost of all parallel instances, given by:

$$c_i = \sum_{j\in[d_i]} \mu \cdot n_i \left(t^{i,j}_{comm} + t^{i,j}_{comp}\right) + \sigma, \tag{6}$$

where μ and σ denote the unit price per CPU and invocation request, respectively. Here, $t^{i,j}_{comp}$ and $t^{i,j}_{comm}$ are set to zero for the I/O and compute stages, respectively.

B. Probabilistic Prediction

The latency of a compute stage cannot be directly quantified as explicitly as that of an I/O stage. Motivated by (3), however, it is possible to simulate the execution of the compute stage by replaying the multicore processing. That is, we only need to capture the function's *solo-run* behavior (i.e., $t^{i,j,k}_{exec}$) across single CPU periods, and then extrapolate it to represent the performance of the entire stage.

While most of the function complexity can be approximated by Lagrange interpolating polynomials [31], we find that $t^{i,j,k}_{exec}$ depends not only on input size but also content. Hence, we treat the performance profiling of the solo-run behavior as a black-box ML regression task. Unlike traditional deterministic models [11], [28], we use *Probabilistic Random Forests* (Prob. RF) to quantify inherent variability in serverless environments [15], [16]. Concretely, each decision tree in Prob. RF does not predict a single execution latency (as in traditional RF), but a *probability distribution* over the possible latency values at the falling leaf node. For the solo-run performance of each stage, the distribution is modeled by a Gaussian mixture model with K components, which is expressed as:

$$t^{i,j,k}_{exec} \sim \sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(\mu_k, \sigma_k) \tag{7}$$

where π_k refers to the mixture coefficient (i.e., weight) of the k-th component, and each component is a normal distribution.

Besides input size, we extract content-specific features (e.g., resolution and channel count for *images*, with frame rate for *videos*). They can be got from metadata without violating the privacy [32]. Note that the input sizes of stages are available at runtime, which hinders the offline prediction. Recent works [5], [31] have shown that the size relation between user-input and intermediate data is deterministic. Nonetheless, splitting a high frame rate video spawns fewer frames due to the shorter duration. Thus, we reuse the same features as Prob. RF of the user-input to build once-off polynomial regression models, so as to estimate each S_i. Finally, the input-unrelated parameters (e.g., T_{block}) are estimated using recent statistics [2], [30].

Saturation spot: Given the deployment of compute stages and γ of I/O stages, the latency and cost under all candidate configurations (i.e., $\mathcal{D}$ and $\mathcal{N}$) are estimated by plugging the parameters in (1)–(6). While more CPUs and instances result in better computing and communication power, each of them still has a resource saturation spot (see Fig. 3). Based on (3) and (4), these spots derive from the non-parallelizable storage access and task block overheads. Explicitly, the over-saturated

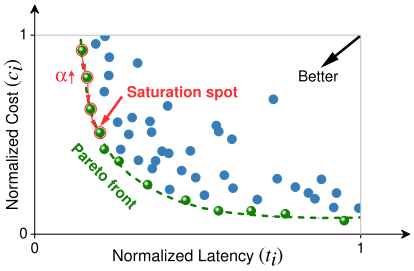


Fig. 9. The selection of a Pareto-optimal resource saturation spot. The green dashed line is the Pareto front with normalized latency and cost components.

configurations are inefficient, as they cannot further speed up the compute or transfer step, and rather increase the cost.

As depicted in Fig. 9, we create a Pareto front [33] for each hybrid stage to quickly identify the saturation spot by pruning sub-optimal configurations. This is because any configuration surrounded by the Pareto front is inferior in either latency or cost, without offering a compensatory advantage in the other. Further, we observe that a small (only 9%) sacrifice on latency results in a substantial (up to 1.7×) reduction in cost, which facilitates marginal returns and is preferable in cases of cost deficiency. Given this, we search along the Pareto front for the configuration (see *red arrows*) that maximizes $t_i + \alpha \cdot c_i$ as the saturation spot[1] of stage v_i, where $\alpha > 0$ is a tunable knob, and the larger it is the higher the priority of cost saving.

V. Workflow Orchestration Optimization

Taking the hybrid stage model, we define the *computation-communication-separated orchestration problem* over a given serverless workflow with HyDAG format $\mathcal{G}(\mathcal{V}, \mathcal{E})$. To unleash hybrid resource elasticity, we co-optimize the aforementioned deployment $p_i \in \mathcal{P}_i$ and resource configuration $d_i \in \mathcal{D}_i, n_i \in \mathcal{N}_i$ for each compute and I/O stage $v_i \in \mathcal{V}$, where $\{\mathcal{D}_i, \mathcal{N}_i\}$ is the Pareto front used to reduce decision space. Recall that the objective is to improve $\mathcal{G}$'s performance while ensuring cost efficiency. As formulated in (8)–(13), the problem generalizes the *hierarchical multi-objective optimization* for single functions [33] to entire workflows, in which end-to-end latency T acts as the main objective and cost C as the secondary.

$$\min_{X} \; T + \alpha C. \tag{8}$$

$$\text{s.t.} \quad T = \max_{L \in \mathcal{L}} \sum_{v_i \in L} \sum_{\theta \in \Theta_i} x_i^\theta \cdot t_i(\theta), \tag{9}$$

$$C = \sum_{v_i \in \mathcal{V}} \sum_{\theta \in \Theta_i} x_i^\theta \cdot c_i(\theta), \tag{10}$$

$$\sum_{\theta \in \Theta_i} x_i^\theta \cdot t_i(\theta) \geq t_i^\Phi, \quad \forall v_i \in \mathcal{V} \tag{11}$$

$$\sum_{\theta \in \Theta_i} x_i^\theta = 1, \quad \forall v_i \in \mathcal{V} \tag{12}$$

$$x_i^\theta \in \{0, 1\}, \quad \forall v_i \in \mathcal{V}, \forall \theta \in \Theta_i \tag{13}$$

[1]The latency and cost at the saturation spot of stage v_i are referred to as its saturated latency t_i^Φ and cost c_i^Φ, respectively.

Algorithm 1: Two-Level HyDAG Partitioning (LHP).

Input : Workflow HyDAG, $\mathcal{G}(\mathcal{V}, \mathcal{E})$; Number of servers, M;
Output: Deployment plan, $\mathcal{P}$;

$\mathcal{G} \leftarrow$ `Initialization`$(\mathcal{V}, M)$;
$N \leftarrow$ The max parallelism of workflow $\mathcal{G}$;
for $n \in \{1, 2, ..., N\}$ **do**
  $\mathcal{G}' \leftarrow \mathcal{G}$;
  if $n < M$ **then**
    $\mathcal{G}' \leftarrow$ `Initialization`$(\mathcal{V}, n)$;
  for compute stage v_i *in* $\mathcal{V}$ **do** // *In topological order*
    // *Init. the partition of* n *function groups*
    $\mathcal{F}_i \leftarrow \{\{f_{i,1}, f_{i,n+1}, ...\}, \{f_{i,2}, ...\}, ..., \{f_{i,n}, ...\}\}$;
    for function set $A \in \mathcal{F}_i$, $B \in \mathcal{F}_i$ **do**
      `Fiduc_Matt`(A, B); ▷ **First level**
    $\mathcal{G}' \leftarrow$`UpdateDAG`$(\mathcal{G}', \mathcal{F}_i)$;
  $\mathcal{F}' = \{\bigcup_{k=1}^{n} \mathcal{F}_i[k] | v_i \in \mathcal{V}\}$;
  // *Init. the indexes of* n *sub-DAGs in each server*
  $\mathcal{I}' \leftarrow \{\{1, M+1, ...\}, \{2, ...\}, ..., \{M, ...\}\}$;
  for index set $\alpha \in \mathcal{I}'$, $\beta \in \mathcal{I}'$ **do**
    `Fiduc_Matt`$(\mathcal{F}'[\alpha], \mathcal{F}'[\beta])$; ▷ **Second level**
  $T_{workflow} \leftarrow$ `E2ELatency`$($`UpdateDAG`$(\mathcal{G}', \mathcal{F}'))$;
  if $T_{workflow} \leq T_{min}$ **then**
    $T_{min}, \mathcal{I}, \mathcal{F} \leftarrow T_{workflow}, \mathcal{I}', \mathcal{F}'$;
  else
    return $\mathcal{P} = \{\mathcal{I}, \mathcal{F}\}$; // *Optimal partitioning result*

where α is the aforementioned knob (i.e., weight) of the cost priority, and path $L \in \mathcal{L}$ represents a series of stages from the start node to the end node of the HyDAG. For simplicity, the whole orchestration domain of stage v_i is co-expressed as $\Theta_i = \{(p_i, d_i, n_i) \mid p_i \in \mathcal{P}_i, d_i \in \mathcal{D}_i, n_i \in \mathcal{N}_i\}$. We also use a variable $x_i^\theta \in X$ to indicate whether stage v_i is orchestrated with $\theta \in \Theta_i$ at runtime $(= 1)$ or not $(= 0)$. Equations (9) and (10) represent the end-to-end workflow latency T as the total latency of stages along the *critical path*, and the cost C as the aggregated cost of all stages. Inequality (11) guarantees that each stage cannot be over-saturated. Constraints (12) and (13) refer to the domain constraints.

In the special case of a single-path workflow, this problem can be reduced to the classical multi-choice knapsack problem [14]. Given the well-known NP-hard property of the knapsack problem, it is similarly NP-hard for more general workflows. To tackle this, we divide the problem into two phases: workflow deployment and coordination. The first phase is to find the minimal end-to-end latency T_{min} quickly. The second phase then focuses on reducing the cost C without altering T_{min}. They are detailed in Sections V-A and V-B, respectively.

A. Graph Partitioning-Based Workflow Deployment

Recall that co-locating adjacent stages can maximize memory efficiency to reduce communication overhead, but with the increased scaling overhead. However, dispatching their parallel instances can amortize the scaling pressure by facilitating load-balancing, while preserving efficient intra-server communication. Switching to our setups, we need to carefully deploy the workflow across servers by juggling both the transfer latency of I/O stages and scaling latency of compute stages, in pursuit of the optimal overall performance.

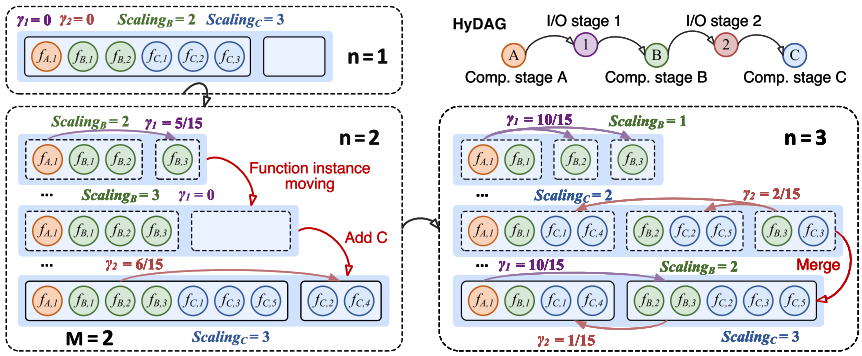


Fig. 10. An illustrative example of LHP. The HyDAG consists of two I/O stages 1, 2, and three compute stages A, B, C. We plot the updates of the scaling stragglers for the compute stages and γ for the I/O stages. The dashed (solid) rounded rectangular boxes denote the first-level (second-level) sub-HyDAGs.

The workflow deployment is tantamount to partitioning its HyDAG at the instance granularity. Function instances within each sub-DAG are co-located to interact using local memory, while various sub-DAGs are deployed and served individually, communicating via external storage. Unlike common balance [5] and min-cut-oriented [22] methods, we must juggle the two complementary goals. To this end, we present LHP based on a lightweight graph partitioning method, Fiduccia-Mattheyses (FM) [34], as shown in Algorithm 1. Given the HyDAG of a *saturated workflow*, LHP makes its deployment plan over M servers. That is, the value of each HyDAG node is initialized to the saturated latency of its corresponding hybrid stage, with parallel instances evenly dispatched for compute stages (lines 1–6). The key idea of LHP is "*split then merge*":

- *First-level. horizontal splitting:* In topological order, each compute stage v_i is split into n function groups, greedily minimizing $t_i + \max_{v_j \in Pre(v_i)}(t_j)$, i.e., the total latency of the compute stage and its associated I/O stages (lines 7–11). These groups are then combined across stages to form n horizontal sub-DAGs (line 12). This can increase the likelihood of finding the optimal deployment.
- *Second-level. vertical merging:* Beginning with the well-initialized partitioning from the first-level, LHP vertically merges the n sub-DAGs into M ones by partitioning the indexes, effectively optimizing the end-to-end latency of the HyDAG (lines 13–15). That is, the sub-DAGs indexed within the same group will be deployed to a server.

The HyDAG is updated with the partitioning results (lines 11 and 16). LHP iterates the above procedure by refining the function group granularity (line 3), until the workflow latency no longer drops (lines 17–20). Note that the core partitioning tasks are borne by the tailored FM method (lines 10 and 15). As a bipartitioning algorithm, its goal is to find and conduct a chain of *instance-moving* actions that maximize the cumulative *gain*, between each pair of the n groups (first-level) or M sub-DAGs (second-level). The gain means the reduction in stage or workflow latency. As depicted in Fig. 10, for an I/O stage, we re-calculate γ by counting the share of data to be moved between sub-DAGs, and update the transfer latency based on the new S'. For a compute stage, we update its scaling latency by measuring the maximum number of instances in the same sub-DAG (i.e., the straggler). Based on this, we calculate the corresponding latency changes to take the gains. Overall, the bottleneck of LHP is the iterative n-way partitioning (lines 3–11), which includes $\frac{n(n-1)}{2}$ pair-wise FM for each stage. Each FM moves a $\frac{2}{n}$ fraction of instances up to once. Thus, the time complexity is $\mathcal{O}(\sum_{n=1}^{N} n^2 \cdot \frac{|V|}{n}) = \mathcal{O}(N^2|V|)$. Here, $|\mathcal{V}|$ is the number of instances in the HyDAG.

Fig. 10 displays an example of LHP. When the number of function groups $n = 1$, LHP tries to co-locate all instances to speed up communication, but their limited number results in high compute latency. By increasing n from 1 to 3, LHP adds and scales up compute stages A, B, C gradually, while moving their instances among n groups or two servers (after merging). Finally, LHP finds the optimal deployment that minimizes the end-to-end workflow latency. Each pair of proxy instances for I/O stages 1,2 can also be deployed based on the ratio of its γ component on each server.

B. Branch-Aware Workflow Coordination

By allocating the saturated resources to each stage and carefully deploying them, HyFaaS identifies the minimal workflow latency (i.e., the critical path of its HyDAG). Nonetheless, this orchestration plan is sub-optimal, because of the implied cost efficiency. We must coordinate the workflow by scaling down non-critical stages, so as to balance the total latency of non-critical paths to approach that of the critical one. Recall that a HyDAG holds complex control flow dependencies, making the workflow coordination NP-hard. However, the inclusion of I/O stages prolongs the paths of its DAG, which yields more branches with greater optimization depth. With this intuition, HyFaaS divides the whole problem into multiple independent (NP-hard) branch coordination sub-problems.

Inspired by [35], we first treat the hybrid stages as a series of *branches* and *synchronizations* (sync):

- *Branch:* A branch consists of sequential compute and I/O stages, which form a disjoint path in the HyDAG.
- *Sync:* Parallel branches might have the same downstream (e.g., *join*) or upstream (e.g., *switch*) branch, i.e., a sync.

The example HyDAG in Fig. 8 (middle) contains two branches $\{2, B, 4\}, \{3, C, 5\}$ and two syncs $\{1, A\}, \{D, 6\}$. Thus, it can be transformed into the *branch-wise HyDAG* shown in Fig. 8 (right), where each node is either a branch or sync. As shown

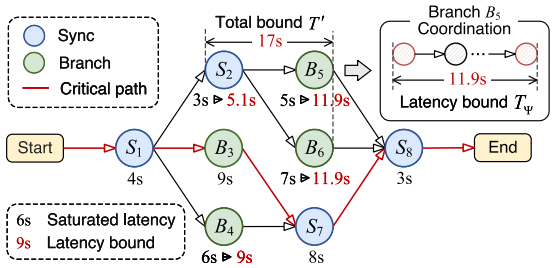


Fig. 11. An illustrative example of workflow and branch coordination.

in Fig. 11, we assign loose latency bounds to non-critical syncs (S_2) and branches (B_4, B_5, B_6). If they are completed within the bound at optimal cost (i.e., the branch coordination), the entire workflow will achieve cost efficiency. Note that sync is a special kind of branch; we do not distinguish either below. Next, we discuss how HyFaaS assigns the latency bound for each branch, and coordinates them accordingly.

1) Latency Bound Assignment: Clearly, a weaker performing branch requires a looser bound. Algorithm 2 thereby uses the *saturated latency* to guide proportional assignment, while avoiding the bound below it. Specifically, this algorithm takes a branch-wise HyDAG $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ with the optimal deployment plan as input. It first calculates the Earliest Start Time (EST) and Latest Finish Time (LFT) for all branches according to the current saturated latency (lines 1–2), and then calls the core `AssignBound` method to assign the latency bound $T_\Psi^{B_i}$ for each branch $B_i \in \mathcal{V}'$ (line 3). Note that the saturated latency of branch B_i is defined as the sum of those of its component stages, i.e., $T_\Phi^{B_i} = \sum_{v_j \in B_i} t_j^\Phi$.

Due to the control dependency, each sync is determined by the latest completed predecessor (e.g., branch B_6 *w.r.t.* sync S_8 in Fig. 11). Hence, this method starts from the end node v_E and seeks an unassigned path P along this direction (lines 7–9). On this path, latency bounds are assigned to each branch by dividing the total bound in proportion to its saturated latency (lines 10–12). The total bound T' of this path is the difference between the LFT of the last branch and the EST of the first branch. Once a branch has been assigned, its bound is recorded as the new saturation latency, and the ESTs of its successors and the LFTs of its precursors need to be updated accordingly (lines 14–15). Next, this method is called recursively to resume the path search starting from the unassigned predecessors of each node (line 16). Finally, each branch is assigned once and its precursors and successors are updated once, so the time complexity of Algorithm 2 is $\mathcal{O}(|\mathcal{V}'| + |\mathcal{E}'|)$.

2) Branch Coordination: For a non-critical branch B, our goal is to coordinate its n sequential stages using cost-efficient configurations. These stages are completed within the assigned latency bound T_Ψ, which can be formulated as:

$$\min_X \sum_{v_i \in B} \sum_{\theta \in \Theta_i} x_i^\theta \cdot c_i(\theta). \tag{14}$$

$$\text{s.t.} \sum_{v_i \in B} \sum_{\theta \in \Theta_i} x_i^\theta \cdot t_i(\theta) \le T_\Psi, (11), (12) \text{ and } (13).$$

Algorithm 2: Latency Bound Assignment Algorithm.

Input : Branch-wise HyDAG, $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$;
Minimal workflow latency, T_{min};
Output: Latency bound set, $\{T_\Psi^{B_i}\}_{\forall B_i \in \mathcal{V}'}$;

1 `EST`$(v_S) \leftarrow 0$, `LFT`$(v_E) \leftarrow T_{min}$, mark v_S, v_E as assigned;
2 Calculate EST and LFT for branch $B_i \in \mathcal{V}'$;
3 `AssignBound`(v_E);
4 **return** $\{T_\Psi^{B_i} \leftarrow T_\Phi^{B_i}\}_{\forall B_i \in \mathcal{V}'}$;
5 **Function** `AssignBound`(B)**:**
6 $P \leftarrow \{\}$, $T_\Phi^{Agg} \leftarrow 0$;
7 **while** B has an unassigned precursor **do**
8 $B \leftarrow \arg\max_{B_i \in Pre(B)}$(`EST`$(B_i) + T_\Phi^{B_i}$);
9 P.`Append_head`(B), $T_\Phi^{Agg} \leftarrow T_\Phi^{Agg} + T_\Phi^B$;
10 $T' \leftarrow$ `LFT`$(P[|P|-1]) -$ `EST`$(P[0])$;
// *Proportional latency bound assignment on path* P
11 **for** branch $B_i \in P$ **do**
12 $T_\Phi^{B_i} \leftarrow (T_\Phi^{B_i}/T_\Phi^{Agg})T'$, and mark B_i as "assigned";
13 **for** branch $B_i \in P$ has an unassigned precursor **do**
14 Update EST for precursors of B_i;
15 Update LFT for successors of B_i;
16 `AssignBound`(B_i);

Here, Θ_i is only the Pareto front of stage v_i. Its deployment still follows the instance distribution across servers of Algorithm 1. If $n = 1$, B is a single-stage branch, we search down for the target configuration directly. If $n > 1$, this problem fits well with the *multiple-choice knapsack problem*. Specifically, the knapsack's capacity is the latency bound, while the items make up the Pareto front. For each stage, an item's value and weight refer to the cost and latency under the corresponding configuration. We adopt Dynamic Programming (DP) to cover this case. Let $d_{i,j} \in D$ be the lowest cost of the last i stages with total latency up to j, we thereby need to explore d_{n,T_Ψ}. The state transition equation is as follows:

$$d_{i,j} = \begin{cases} \min_\theta \left(d_{i,j},\ d_{i-1,\,j-t_i(\theta)} + c_i(\theta)\right), & j > T_\Phi^i \\ \sum_{v_k \in B,\, k \in [i]} c_k^\Phi, & j = T_\Phi^i \end{cases} \tag{15}$$

where T_Φ^i denotes the cumulative saturated latency of the last i stages. According to (15), each $d_{i,j}$ is iteratively solved by a bottom-up DP with nested loops. We use $conf_{i,j}$ to facilitate maintaining the decomposition path of $d_{i,j}$. By tracking from $conf_{n,T_\Psi}$, we can reconstruct the selected configuration for each stage $v_i \in B$. Considering the dual impact of latency and resource price on the cost (see (6)), those configurations that fall into a latency window (e.g., 10 ms [24]) can be perceived as cost-equivalent. Hence, the time complexity of the branch coordination is $\mathcal{O}(n|\Theta|\widetilde{T}_\Psi)$, where $\widetilde{T}_\Psi$ denotes the number of latency windows. In practice, 95% of workflows have fewer than 8 stages with a mean latency of 670 ms [2]. For the non-critical branches in $\mathcal{V}'$ coordinated individually, the incurred overhead is effectively constant in scale.

3) Runtime Fine-Tuning: Due to the event-driven nature of functions, it is not feasible to reserve resources for the entire workflow [1]. Resource availability on each server varies over time, with the frequent destruction and creation of instances [22]. The instance's CPU quota, decided in advance, could be rejected,

causing it to wait until ample resources are available [36]. Coupled with uncertain runtime cases (e.g., warm-start and error-retry), any stage may complete earlier or later than expected. To this end, we propose an *event-driven* fine-tuning method at runtime, which proceeds on two levels:

i) Upon *a stage* ends, we re-coordinate the remaining stages in the respective branch. First, we narrow their latency bound by subtracting the total latency of the k completed stages from the branch bound, i.e., $T'_\Psi = T_\Psi - \sum_{i\in[k]} t_i$. Next, the new configurations can be directly obtained by iteratively tracking from $conf_{n-k,T'_\Psi}$. *ii*) Upon a ***full branch*** ends, we re-run the Algorithm 2, marking it as "assigned" with the actual latency, and re-assigning for those un-triggered. For a branch with a new bound $T'_\Psi \in [T^n_\Phi, T_\Psi]$, we similarly get the new configurations by tracking from $conf_{n,T'_\Psi}$. Even if $T'_\Psi > T_\Psi$, we just *incrementally* calculate d_{n,T'_Ψ} on top of the existing DP table D with (15), and obtain the new configurations as before. Note that a stage rarely benefits from warm-start to complete earlier due to straggler functions, let alone a branch. In most cases,[2] $T'_\Psi < T_\Psi$, reducing the time complexity of the runtime fine-tuning to $\mathcal{O}(n)$, thanks to efficient path tracking.

VI. Implementation

HyFaaS is implemented based on OpenWhisk [10], providing a hybrid resource management service. Based on previous studies [5], [20], the shared memory is treated as a directory, mounted and mapped to the memory region of each instance. The self-contained CouchDB [37] is used as external storage while logging functions. Besides, we adopt the CPU-oriented configuration according to [38].

The HyDAG translator retrieves the DAG statement generated by OpenWhisk Composer [10] and inserts the invocations of I/O stages as described in Section III-B. The *action* of the I/O stages is pre-registered and can be reused across different workflows by issuing respective arguments (e.g., the ID of the calling compute stage and its γ component). The performance profiler uses the *RandomForestClassifier* and *GaussianMixture* in the scikit-learn library [39] to build the Prob. RF models. Each model needs to re-learn with the up-to-date training set in sliding windows, if the function's *solo-run* behavior varies. Alternatively, we can support in-container CPU sharing [11], [24] by simulating this shared execution to adjust (4), so as to mitigate the blocking influence. The saturated profiling results are recorded in a configuration table.

In the core optimizer engine, the workflow deployer overrides the load balancer in the controller to integrate LHP. Note that the native consistent hash algorithm is reused to initialize the two-level partitioning of LHP, without missing the server health checks. The deployer uses a routing table to record the generated deployment plan and dispatches function instances at runtime accordingly. For the workflow coordinator, it incorporates the branch-aware algorithm to proactively coordinate the entire workflow. Once a stage or a full branch completes, it is notified again to re-coordinate the remaining stages and branches by reading the recent invocation logs. Accordingly, the routing and configuration tables are updated.

[2]In the worst case, the time complexity is $\mathcal{O}(n|\Theta|\Delta\widetilde{T}_\Psi)$, where $\Delta\widetilde{T}_\Psi = \widetilde{T}'_\Psi - \widetilde{T}_\Psi$ is the increment in the number of latency windows.

VII. Experimental Evaluation

In this section, we first detail the experimental setup. Next, we evaluate the overall performance of HyFaaS and analyze the effectiveness of its components.

A. Experimental Environment

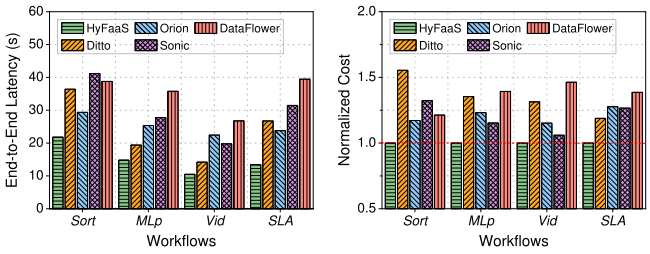
Testbed setup: Our evaluations were conducted on an AWS EC2 cluster consisting of 9 instances. We deploy one controller and eight invoker servers for HyFaaS, with the client hosted on a separate instance. The controller and invoker servers are provisioned through the c5.4×large and c5.9× large instances, respectively. In our experiments, each stage can be configured with a maximum fan-out degree of 100, and up to 6 CPU cores per function instance [3]. Following [19], we bundle 40 Mbps bandwidth per 0.1 core-sized instance.

Evaluated workloads: We adopt the following four widely-used workflow workloads, each with a distinct DAG topology. (1) *TeraSort (Sort)* [5] is a typical data-intensive job that sorts a dataset [40] based on the specified key. It consists of a map stage and a reduce stage, linked by a shuffle operation. (2) *ML pipeline (MLp)* is an ML training workflow with 4 sequential stages. It trains a random forest model over given images. The training process consists of image set splitting, PCA for feature extraction, model training and testing. (3) *Video analytics (Vid)* is a ML serving workflow shown in Fig. 2. It extracts a frame from each video chunk and classifies it through a well-trained model. Some frames are pre-processed using a *switch* branch. The above two workloads are generated from [16]. (4) *SLApp (SLA)* [18] is a synthetic workflow composed of eight stages with various compute and I/O sensitivities. We remove cycles and scale the input to simulate real-world complex cases.

Baselines: We evaluate HyFaaS against the following state-of-the-art serverless baselines.

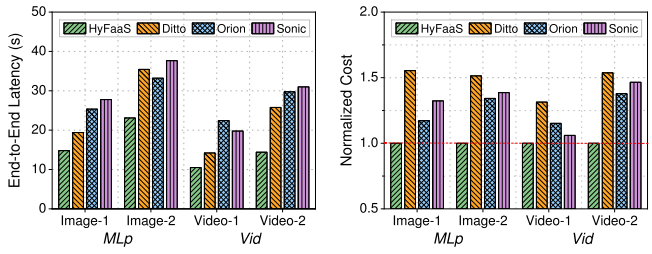
- Ditto [13] is a data analytics job scheduler that configures elastic parallelism for each stage with grouped deployment jointly to minimize latency.
- Orion [11] is a workflow orchestrator that bundles parallel instances into fewer containers and right-sizes them with explicit performance guarantees.
- Sonic [17] is a communication-aware workflow orchestrator that uses dynamic data-passing method selection and function placement to optimize performance and cost.
- DataFlower [19] is a decentralized workflow orchestrator that decouples and overlaps computation and communication with pressure-aware scaling.

Metrics: We evaluate HyFaaS with two main metrics: end-to-end latency and normalized cost of workflow execution. For a given workflow, the end-to-end latency is the duration from the submission to the completion of its last instance. For the latter, we measure the aggregated cost across all instances and normalize them to avoid reporting precise dollars. In addition, we consider the cost of data persistence in shared memory, as



(a) End-to-end latency. (b) Normalized cost.

Fig. 12. The overall performance of various methods with four workflows.



(a) End-to-end latency. (b) Normalized cost.

Fig. 13. The sensitivity of various methods to input content.

memory is scarce and expensive in serverless computing [13]. This facilitates avoiding unfair competition for baselines and commercial systems that do not support memory interaction.

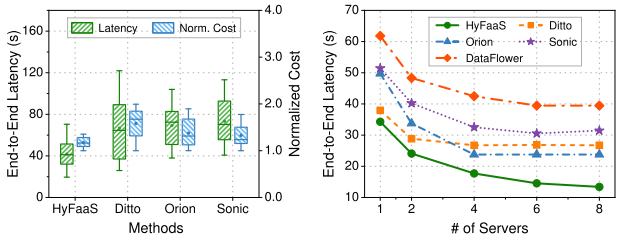
B. Performance Evaluation

We deploy CouchDB on the controller server for fairness in remote data I/O. The cluster remains idle before submitting a single workflow with the default input. Each instance requires a minimal memory limit larger than the peak memory usage, which benefits harvesting for shared memory. By default, the knob α is set to a floating-point number close to 0, and the parameter K in Prob. RF is set as 3, as discussed later.

1) Overall performance: We compare the overall performance of HyFaaS in end-to-end workflow latency and normalization cost. Following [11], we set the latency goal for Orion to the minimum achievable latency, matching that of HyFaaS. Also, we only retain fan-out of the Scatter type for Sonic [17].

End-to-end latency: Fig. 12(a) illustrates that HyFaaS can significantly accelerate these four workflows with distinct I/O and compute characteristics. It reduces end-to-end latency by 32.7%–50.4%, due to the unique *computation-communication-separated orchestration*. This is because the baselines conduct stage-level monolithic resource provisioning, despite existing communication optimization. Like, Sonic dynamically selects different storage-based data transfer methods for stages, while Ditto replaces Sonic's zero-copy method with a faster shared-memory-based one. However, they ignore the scaling overhead of straggler instances, particularly for highly parallel *Sort*. In contrast, Orion bundles parallel instances to lower the number of spawned containers, alleviating latency skews due to over-scaling. Nonetheless, it limits resource elasticity and thereby struggles to fully expedite the workflows. DataFlower enables decoupling, but can overlap computation and communication, not explicitly orchestrate them. By applying both storage and memory-based transfer methods, HyFaaS flexibly orchestrates I/O and compute stages to juggle communication and scaling overheads to minimize the end-to-end latency.

Cost: Fig. 12(b) depicts the corresponding normalized cost of HyFaaS against the baselines. As illustrated, HyFaaS outperforms these baselines by up to 1.37× in cost savings, but the margin is smaller than that of end-to-end latency. This is because of the lower resource costs associated with inadequate scaling in the baselines. Besides, the results show that fewer resource inputs do not necessarily save costs with fine-grained



(a) Input size sensitivity. (b) Performance scalability.

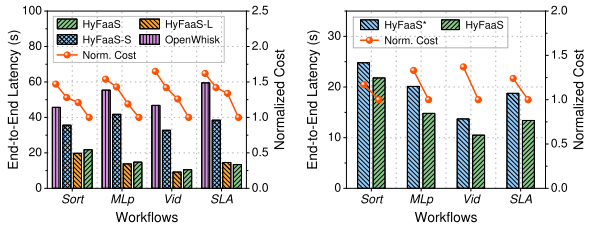
Fig. 14. The (a) sensitivity to input size and (b) scalability of various methods.

billing, as explicitly shown by DataFlower, whose latency and cost are at a high level. Others intend to balance DAG paths like HyFaaS to avoid cost wastage. By maximizing memory efficiency, further, Ditto greedily colocates stages to eliminate imbalance in communication. Sonic alleviates this imbalance in computation by flexibly exploiting cheaper storage, which exhibits a cost advantage. However, they cannot maintain the balance in both computation and communication by exploring the non-linear latency-cost relationship for non-critical hybrid stages, making it difficult to fully harvest cost efficiency.

2) The impact of input sensitivity: Since *Sort* is used to sort structured data, whereas *MLp* and *Vid* handle unstructured data (i.e., images and videos), we use two workflow sets to evaluate sensitivity to input size and content, respectively. Note that we exclude DataFlower as it is unaware of the input.

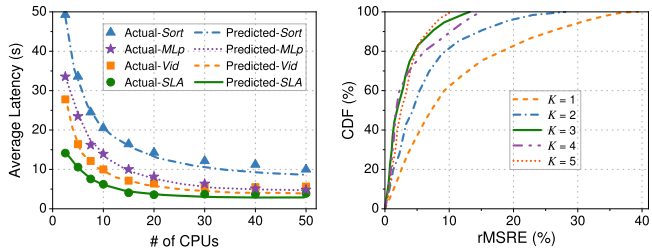
Input content: We use two image sets from CIFAR-10 and MNIST, as well as two video sets with frame rates of 60 and 30 FPS, respectively. For fairness, the sets with the same type are of equal size. The results are depicted in Fig. 13. Note that MNIST has a higher resolution and channel count, while a low frame rate yields more frames extracted [41]. These baselines treat the same workload with various input contents as distinct workflows, requiring temporal, manual retraining for profiling models. Nonetheless, Ditto and Sonic cannot fully explore the shifts in function complexity behind the content, resulting in sub-optimal workflow orchestration with the increased latency and cost. In contrast, Orion's bundling mechanism enables in-container resource sharing, partially mitigating the impact of the ensuing stragglers with comparable performance.

Input size: We change the input dataset for *Sort*, generated by Sort Benchmarks [40], increasing its size from 10GB to 50GB. Note that each run is standalone, and we showcase all results at once in Fig. 14(a). The normalization criterion is the respective



(a) HyFaaS components. (b) With/without decoupling.

Fig. 15. The overall performance among various HyFaaS settings.



(a) I/O stage. (b) Compute stage.

Fig. 16. The effectiveness of the gray-box profiling model for various stages.

minimal observed cost. As shown, HyFaaS performs better with its *gray-box* model, achieving up to 40.7% average latency and 25.9% cost improvements. This is because these baselines suffer the significant scaling overheads from sorting larger datasets with higher fan-out degrees, while lacking the in-depth profiling of function execution characteristics. Apart from Sonic, they are not explicitly aware of variations in input size, requiring the aforementioned model retraining.

3) Scalability evaluation: We run *SLA* with varying numbers of servers to evaluate the scalability. Fig. 14(b) illustrates the variation in performance. As the cluster expands, HyFaaS can unleash the advantages of distributed function serving to juggle scaling and communication overheads, thus reducing end-to-end latency. This provides more margin for cost optimization. Ditto, Sonic and Orion prioritize deploying function instances to fewer servers or containers, causing the workflow to exceed saturation before the server does. In contrast, DataFlower can achieve decentralized orchestration and pipelined in-memory data transmission, thereby delivering similar performance improvements as HyFaaS.

4) Ablation evaluation: To verify the effectiveness of HyFaaS components, we evaluate it with three incremental variants and vanilla OpenWhisk with the default configuration.

- HyFaaS-S: The modified OpenWhisk that allocates saturated resources for each stage using the *gray-box* model.
- HyFaaS-L: HyFaaS-S along with the LHP algorithm.
- HyFaaS*: HyFaaS without the decoupling of communication and computation. It *holistically* orchestrates the computation and communication operations for each stage.

As shown in Fig. 15(a), HyFaaS-S significantly mitigates the dual increase in latency and cost caused by mis-provisioning in OpenWhisk. Coupled with LHP, HyFaaS-L further reduces the end-to-end latency over 44.6% by optimizing both scaling and communication overheads. Despite the increased memory cost in data movement of I/O stages, it also mitigates that in waiting for straggler instances of compute stages. Furthermore, HyFaaS outperforms HyFaaS-L by up to 1.43× on cost using branch-aware coordination with a negligible drop in latency. For *Sort* and *MLp*, with just one branch, can still benefit from the stage event-driven fine-tuning. Even without the separated orchestration, Fig. 15(b) illustrates that *HyFaaS** still delivers comparable latency with a mean gap of only 17.6%. However, it struggles to match the varying resource demands of the two parts, leading to a 27.8% cost wastage.

TABLE III
ORCHESTRATION TIME WITH VARIOUS CLUSTER SIZES

Workflows	Orchestration Time			
	$M = 2$	$M = 4$	$M = 6$	$M = 8$
Sort	19.7 ms	21.6 ms	25.4 ms	24.7 ms
MLp	26.5 ms	30.6 ms	34.7 ms	35.2 ms
Vid	23.6 ms	31.4 ms	35.6 ms	36.1 ms
SLA	33.1 ms	36.3 ms	35.2 ms	35.4 ms

5) Profiling performance: We evaluate the profiling model for I/O and compute stages across various workflows, respectively. Fig. 16(a) illustrates the real I/O stage latency (point) and the predicted distribution (line) against the total number of used CPUs. As shown, the latency of all I/O stages can be precisely estimated due to the isolated and stable bandwidth. Although the growth in fan-out degree leads to high concurrent requests to storage, the error keeps within 6%. For the compute stage, Fig. 16(b) shows the error distribution of the *gray-box* model with various K. We evaluate all compute stages in these four workflows under various configurations, and the metric is root Mean Square Relative Error (rMSRE). As depicted, the error drops sharply as K increases from 1 to 3. Beyond this point, the improvement plateaus, with 90% of the rMSREs falling below 10%. We thus set $K = 3$ by default.

6) Orchestration overhead: Table III shows the orchestration time of HyFaaS for each workflow. It does not notably grow as the cluster size M increases. This is because the bottleneck of LHP lies in the partitioning of function groups. Even for *SLA*, which holds the most stages (i.e., eight) and a complex DAG, the overhead is in sub-second and tolerable relative to total latency. Recall that 95% of workflows have fewer than 8 stages [2], HyFaaS thus can be applied in most cases.

VIII. RELATED WORK

Serverless resource configuration: Resource configuration has always been vital for both performance and cost optimization of cloud applications [42], [43], especially for serverless computing with the fine-grained elasticity and billing. Existing research either builds analytical models with prior knowledge [13], [14], [31] or black-box models (e.g., Bayesian optimization [15] and ML [44], [45]) to estimate function performance, further guiding optimization. However, they are tough to find the optimal configuration, due to unknown function complexity and inherent variability. Jolteon [16] proposes a stochastic performance model to solve this issue. However, it ignores the effects

of multi-core processing and input content. Considering content in generic models comes at the cost of losing function execution characteristics [26], [46]. In contrast, our gray-box model merges white-box modeling and black-box prediction, yielding an accurate performance estimation.

Serverless workflow orchestration: Serverless workflows have been orchestrated in various ways to enhance efficiency. In the area of improved communication, SpecFaaS [47] and Chitu [48] pipeline data transfer between stages. HyFaaS can support them by tuning the I/O stage profiling with pipeline annotation. Some studies [20], [21], [22], [26] use local memory to accelerate intra-server transmission. For deployment, some efforts reduce the number of spawned instances through hybrid process/thread execution [30], [49] and parallel instance packing [23], [24]. In terms of resource configuration, many works [14], [15], [16] aim to scale each stage to ensure performance. Like HyFaaS, some works jointly consider the deployment and configuration of workflows. Based on the right-sizing for each stage, Orion [11] further bundles parallel instances (like [23] and [24]) to improve resource efficiency; Sonic [17] reduces communication latency via dynamic data passing strategy and function placement. Ditto [13] combines stage placement and parallelism configuration to optimize latency or cost, failing to reconcile both like ours. These methods either ignore notable scaling overhead or shift it inside containers, and much less co-scheduling with hybrid resources. Though DataFlower [19] also decouples and overlaps computation and communication in functions, it cannot orchestrate them separately.

IX. Conclusion

This paper designs HyFaaS, a computation-communication-separated orchestrator for serverless workflows. It re-expresses the workflows as hybrid I/O and compute stages via a HyDAG abstraction. HyFaaS has two key goals. (1) High performance: It identifies Pareto-optimal saturated configurations via a gray-box model for the hybrid stages, while deploying them across servers with LHP to juggle communication and scaling overheads. (2) Cost efficiency: It further determines cost-efficient configurations for non-critical stages via branch-aware latency bound assignment and coordination, along with runtime fine-tuning. Experimental results display that HyFaaS can unleash the hybrid resource elasticity, with a significant improvement in terms of end-to-end latency and cost.

Overall, we highlight some key findings: (1) It is critical to unleash hybrid resource elasticity with *vertical heterogeneous configuration* for communication and computation operations. (2) HyDAG of I/O and compute stages can offer fine-grained information for hybrid resource management. (3) While each stage has a saturated latency, dispatching its instances further shares scaling pressure and keeps fast intra-server communication. (4) Non-critical stages along prolonged branches with optimization depth imply cost-efficiency, regardless of overall performance. Besides, given the latency skew among parallel instances due to potential content variations, it is also critical to pursue *horizontal heterogeneous configuration* for them in the future. This will facilitate greater resource efficiency.

References

[1] E. Jonas et al., "Cloud programming simplified: A berkeley view on serverless computing," 2019, *arXiv: 1902.03383*.

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

[3] "AWS step functions," 2024. [Online]. Available: https://aws.amazon.com/step-functions/

[4] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proc. ACM Int. Conf. Manage. Data*, 2020, pp. 115–130.

[5] T. Li, Y. Li, W. Zhu, Y. Xu, and J. C. Lui, "MinFlow: High-performance and cost-efficient data passing for I/O-intensive stateful serverless analytics," in *Proc. USENIX Conf. File Storage Technol.*, 2024, pp. 311–327.

[6] M. Zhang, Y. Zhu, J. Liu, F. Wang, and F. Wang, "CharmSeeker: Automated pipeline configuration for serverless video processing," *IEEE/ACM Trans. Netw.*, vol. 30, no. 6, pp. 2730–2743, 2022.

[7] B. Hou, S. Yang, F. A. Kuipers, L. Jiao, and X. Fu, "EAVS: Edge-assisted adaptive video streaming with fine-grained serverless pipelines," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

[8] C. Lu et al., "SMIless: Serving DAG-based inference with dynamic invocations under serverless computing," in *Proc. Int. Conf. High Perfor. Comput. Netw. Storage Anal.*, 2024, pp. 590–606.

[9] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ML workflows," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 13–24.

[10] "OpenWhisk apache," 2024. [Online]. Available: https://openwhisk.apache.org/

[11] A. Mahgoub et al., "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2022, pp. 303–320.

[12] H. Fingler, A. Akshintala, and C. J. Rossbach, "USETL: Unikernels for serverless extract transform and load why should you settle for less?," in *Proc. ACM SIGOPS Asia-Pac. Workshop Syst.*, 2019, pp. 23–30.

[13] C. Jin et al., "Ditto: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf.*, 2023, pp. 406–419.

[14] 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.

[15] Z. Zhou et al., "Aquatope: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. ACM Int. Conf. Architect. Support Prog. Lang. Operating Syst.*, 2022, pp. 1–14.

[16] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2024, pp. 167–183.

[17] A. Mahgoub et al., "Sonic: Application-aware data passing for chained serverless applications," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 285–301.

[18] 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.

[19] Z. Li et al., "DataFlower: Exploiting the data-flow paradigm for serverless workflow orchestration," in *Proc. ACM Int. Conf. Architect. Support Prog. Lang. Operating Syst.*, 2023, pp. 57–72.

[20] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. USENIX Symp. Netw. Syst. Des. Implement.*, 2023, pp. 1489–1504.

[21] S. Qi et al., "SPRIGHT: High-performance eBPF-based event-driven, shared-memory processing for serverless computing," *IEEE/ACM Trans. Netw.*, vol. 32, no. 3, pp. 2539–2554, 2024.

[22] Z. Li et al., "FaasFlow: Enable efficient workflow execution for Function-as-a-Service," in *Proc. ACM Int. Conf. Architect. Support Prog. Lang. Operating Syst.*, 2022, pp. 782–796.

[23] S. Mohanty, V. M. Bhasi, M. Son, M. T. Kandemir, and C. Das, "FAASt-loop: Optimizing loop-based applications for serverless computing," in *Proc. ACM Symp. Cloud Comput.*, 2024, pp. 943–960.

[24] A. Mahgoub et al., "WiseFuse: Workload characterization and DAG transformation for serverless workflows," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 2, pp. 1–28, 2022.

[25] "Docker container update," 2024. [Online]. Available: https://docs.docker.com/engine/reference/commandline/update/

[26] D. Mvondo et al., "OFC: An opportunistic caching system for FaaS platforms," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 228–244.

[27] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. ACM Int. Conf. Architect. Support Prog. Lang. Operating Syst.*, 2020, pp. 467–481.
[28] Q. Liu et al., "Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu," in *Proc. USENIX Annu. Tech. Conf.*, 2024, pp. 1–17.
[29] M. Kiener, M. Chadha, and M. Gerndt, "Towards demystifying intra-function parallelism in serverless computing," in *Proc. Int. Workshop Serverless Comp.*, 2021, pp. 42–49.
[30] Y. Li, L. Zhao, Y. Yang, and W. Qu, "Rethinking deployment for serverless functions: A performance-first perspective," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 1–14.
[31] V. M. Bhasi et al., "Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proc. ACM Symp. Cloud Comput.*, 2022, pp. 257–272.
[32] Q. Ding et al., "Bc2FL: Double-layer blockchain-driven federated learning framework for agricultural IoT," *IEEE Internet Things J.*, vol. 12, no. 4, pp. 4362–4374, 2024.
[33] M. Bilal, M. Canini, R. Fonseca, and R. Rodrigues, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. Eur. Conf. Comput. Syst.*, 2023, pp. 381–397.
[34] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. Des. Autom. Conf.*, 1988, pp. 241–247.
[35] Z. Hu, D. Li, Y. Zhang, D. Guo, and Z. Li, "Branch scheduling: DAG-aware scheduling for speeding up data-parallel jobs," in *Proc. Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
[36] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. ACM Int. Conf. Architect. Support Prog. Lang. Operating Syst.*, 2022, pp. 753–767.
[37] "CouchDB," 2024. [Online]. Available: https://couchdb.apache.org/
[38] "Add optional cpu limit to spawned action containers," 2024. [Online]. Available: https://github.com/apache/openwhisk/pull/5443/
[39] F. Pedregosa et al., "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[40] "Sort benchmarks," 2024. [Online]. Available: https://sortbenchmark.org/
[41] R. Zeng, C. Zeng, X. Wang, B. Li, and X. Chu, "Incentive mechanisms in federated learning and a game-theoretical approach," *IEEE Netw.*, vol. 36, no. 6, pp. 229–235, Nov./Dec. 2022.
[42] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 363–378.
[43] H. Al-Sayeh et al., "Juggler: Autonomous cost optimization and performance prediction of Big Data applications," in *Proc. ACM Int. Conf. Manage. Data*, 2022, pp. 1840–1854.
[44] T. Li and Z. Zhao, "Moirai: Optimizing quantum serverless function orchestration via device allocation and circuit deployment," in *Proc. IEEE Int. Conf. Web Serv.*, 2024, pp. 707–717.
[45] X. Yue, S. Yang, L. Zhu, S. Trajanovski, and X. Fu, "Demeter: Fine-grained function orchestration for geo-distributed serverless analytics," in *Proc. IEEE Conf. Comput. Commun.*, 2024, pp. 2498–2507.
[46] P. Sinha, K. Kaffes, and N. J. Yadwadkar, "Online learning for right-sizing serverless functions," in *Proc. Workshop Architecture Syst. Support Transformer Models (ASSYST, ISCA)*, Orlando, Florida, United States of America, 2023, pp. 1–6.
[47] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "SpecFaaS: Accelerating serverless applications with speculative function execution," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2023, pp. 814–827.
[48] Z. Lei, X. Shi, C. Lv, X. Yu, and X. Zhao, "Chitu: Accelerating serverless workflows with asynchronous state replication pipelines," in *Proc. ACM Symp. Cloud Comput.*, 2023, pp. 597–610.
[49] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.

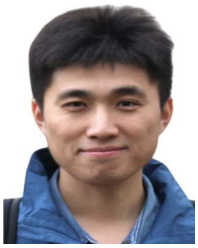
Xiaofei Yue received the ME degree in computer science from the Northeastern University, Shenyang, China, in 2022. He is currently working toward the PhD degree with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His research interests include distributed systems, cloud/serverless computing, data analytics, and model inference optimization.

Song Yang (Senior Member, IEEE) received the BS degree in software engineering and MS degree in computer science from the Dalian University of Technology, Dalian, Liaoning, China, in 2008 and 2010, respectively, and the PhD degree from the Delft University of Technology, The Netherlands, in 2015. From August 2015 to July 2017, he worked as postdoc researcher with the EU FP7 Marie Curie Actions CleanSky Project in Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), Göttingen, Germany. He is currently an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. His research interests include data communication networks, cloud/edge computing, and network function virtualization.

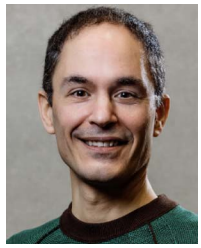
Fan Li (Member, IEEE) received the BEng and MEng degrees in communications and information system from the Huazhong University of Science and Technology, Wuhan, China, in 1998 and 2001, respectively, the MEng degree in electrical engineering from the University of Delaware, Newark, Delaware, in 2004, and the PhD degree in computer science from the University of North Carolina at Charlotte, Charlotte, North Carolina, in 2008. She is currently a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. Her current research focuses on wireless networks, ad hoc and sensor networks, and mobile computing. Her papers won best paper awards from IEEE MASS (2013), IEEE IPCCC (2013), ACM MobiHoc (2014), and Tsinghua Science and Technology (2015). She is a member of the ACM.

Liehuang Zhu (Senior Member, IEEE) received the BE and ME degrees from Wuhan University, Wuhan, China, in 1998 and 2001, respectively, and the PhD degree in computer science from the Beijing Institute of Technology, Beijing, China, in 2004. He is currently a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing. He has published more than 150 peer-reviewed journal or conference papers. He has been granted a number of IEEE best paper awards, including IWQoS 17', TrustCom 18', and ICA3PP 20'. His research interests include security protocol analysis and design, blockchain, wireless sensor networks, and cloud computing.

Xu Wang received the PhD degree from the University of Oxford, in 2012. Then he returned to College of Big Data and Information, Guizhou University, in 2013 as Professor of Physics and Electronics. Since 2019, he has also become a graduate tutor of Tsinghua University in Beijing, China. His research and teaching focuses on the areas of AI related energy materials. He has achieved considerable success in these fields and won numerous awards.



Zhen Feng received the PhD degree from the Dalian University of Technology, China, in 2014. He is currently a senior research engineer with Jinan Inspur Data Technology. His research interests include cloud computing, server virtualization and software-defined storage.



Fernando A. Kuipers (Senior Member, IEEE) received the PhD degree (cum laude) from the Delft University of Technology (TU Delft), in 2004. He was a visiting scholar with the Technion-Israel Institute of Technology, in 2009, and Columbia University, New York, NY, USA, in 2016. He is currently a full professor with TU Delft, where he established and leads the Networked Systems Group and the Laboratory on Internet Science. He cofounded the Do IoT Fieldlab and the PowerWeb Institute. His research interests include network optimization, network resilience, quality-of-service, and quality-of-experience and address problems in computer networks, software-defined networking, 6G, and the Internet of Things. His work on these subjects includes distinguished papers at IEEE INFOCOM 2003, Chinacom 2006, IFIP Networking 2008, IEEE FMN 2008, IEEE ISM 2008, ITC 2009, IEEE JISIC 2014, NetGames 2015, and EuroGP 2017. He has served as the General Chair and the TPC Chair for flagship conferences, such as ACM SIGCOMM (2021 and 2022) and IEEE INFOCOM (2024). He is the vice chair of the ACM SIGCOMM Executive Committee. He served on the Board of the TU Delft Safety and Security Institute. He is the Co-PI of the Dutch 6G Flagship Project Future Network Services, where he leads the program line Intelligent Networks.