Exploiting Wide-Area Resource Elasticity with Fine-Grained Orchestration for Serverless Analytics

Xiaofei Yue, *Graduate Student Member, IEEE,* Song Yang, *Senior Member, IEEE,*
Liehuang Zhu, *Senior Member, IEEE,* Stojan Trajanovski, *Member, IEEE,*
Fan Li, *Member, IEEE,* Xiaoming Fu, *Fellow, IEEE*

***Abstract*—With the flourishing of global services, low-latency analytics on large-volume geo-distributed data has been a regular requirement for application decision-making. Serverless computing, with its rapid function start-up and lightweight deployment, provides a compelling way for geo-distributed analytics. However, existing research focuses on elastic resource scaling at the stage granularity, struggling to heterogeneous resource demands across component functions in *wide-area settings*. The neglect potentially results in the cost inefficiency and Service Level Objective (SLO) violations. In this paper, we advocate for fine-grained function orchestration to exploit *wide-area resource elasticity*. We thereby present *Demeter*, a fine-grained function orchestrator that saves job execution costs for geo-distributed serverless analytics while ensuring SLO compliance. By learning from volatile and bursty environments, *Demeter* jointly makes per-function placement and resource allocation decisions using a well-optimized multi-agent reinforcement learning algorithm with a pruning mechanism. It prevent the irreparable performance loss by function congestion control. Ultimately, we implement *Demeter* and evaluate it with the realistic workloads. Experimental results reveal that *Demeter* outperforms the baselines by up to 46.6% on cost, while reducing SLO violation by over 23.7% and bringing it to below 15%.**

***Index Terms*—Serverless computing, data analytics, function placement, resource allocation, reinforcement learning.**

I. Introduction

MANY global services, such as video streaming [1] and social network [2], continuously produce large amounts of data, such as user session and system operation logs. These data are often stored in respective local Data Centers (DC), awaiting mining for advertising decision-making, system fault diagnosis, and so on [3]. Researchers have explored effective ways to process the geo-distributed data, pursuing an agreed-upon Service Level Objective (SLO) in terms of completion time. Otherwise, the results could be outdated [4]. Recently, serverless computing with the form of Function-as-a-Service (FaaS) [5] has flourished in data analytics offerings, due to its promise of fine-grained resource elasticity and billing mode. Many serverless systems [6], [7], [8] with distinct architectures are devised to offer traditional data analytics services [9] in a centralized DC. They effectively free developers from fussy infrastructure management. Nonetheless, it remains challenging to harness the benefit of FaaS to provide comparable performance across multiple geo-distributed DCs, what we refer to as Geo-distributed Serverless Analytics (GSA), connected by a Wide-Area Network (WAN). Existing wide-area schedulers [10], [11], [12] struggle to be directly applied to the serverless scenario, due to a lack of adaptation to its uniqueness.

Serverless systems decouple a monolithic analytics job into stages, each consisting of stateless functions (*i.e.*, tasks) that execute in parallel [13]. They are stated as a Directed Acyclic Graph (DAG) to implement specific logic. Owing to frequent communication between stages (*a.k.a.*, *shuffle*) through volatile WAN links, carefully placing functions in a GSA job across DCs is naturally necessary. Given the lightweight and rapid start-up nature of functions, the intention is to move them to where the data resides for local processing, as in prior works [11], [12], to minimize WAN traffic. This is because migrating geo-distributed data to a single DC for centralized processing is limited by privacy regulations [10]. Nonetheless, the WAN potentially brings unexpected variations in resource demand of each function. In traditional serverful architectures, all tasks stick to slots divided evenly from a fixed resource pool [14], which is prone to resource over-provisioning [7]. In contrast, serverless computing allows elastic resource scaling [15], [16], [17], [18] to match the demands of stages that typically have different input sizes. Despite the benefits, this coarse-grained way remains inflexible and misses the potential of function-level on-demand provisioning, resulting in cost inefficiency or SLO violations in *wide-area settings*. Actually, each function runs individually in an isolated container, loosely-coupled in a job. They have the opportunity to be configured with tailored resources based on heterogeneous demand. Thus, it is essential to rethink the orchestration way of these functions.

In this paper, we advocate that wide-area resource elasticity should be exploited via *fine-grained function orchestration*.

The work of Song Yang was supported partially by the National Natural Science Foundation of China (NSFC) under Grants 62472028 and 62172038, in part by the Beijing Natural Science Foundation under Grant 4232033, and in part by the National Key Research and Development (R&D) Program of China under Grant 2023YFB3107300. The work of Liehuang Zhu was supported in part by the Yunnan Provincial Major Science and Technology Special Plan Projects under Grant 202302AD080003. The work of Fan Li was supported in part by the NSFC under Grants 62372045 and 62072040. The work of Xiaoming Fu was supported in part by the Horizon Europe CODECO project under Grant 101092696, and in part by the Horizon Europe COVER project under Grant 101086228. The preliminary version of this paper was published in the IEEE INFOCOM 2024 [DOI: 10.1109/INFOCOM52122.2024.10621303]. *(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).

Stojan Trajanovski is with the Microsoft, W2 6BD London, U.K. (e-mail: sttrajan@microsoft.com).

Xiaoming Fu is with the Institute of Computer Science, University of Göttingen, 37077 Göttingen, Germany (e-mail: fu@cs.uni-goettingen.de).

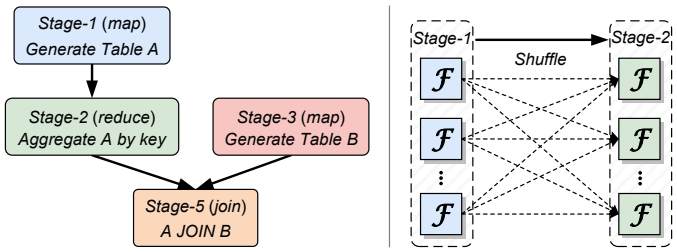


Fig. 1. The execution DAG (left) and communication between stages (right) of a serverless analytics job. The solid lines indicate stage-level dependencies and the dashed lines mean fine-grained function-level dependencies.

That is, pursuing the on-demand placement and multi-resource allocation (*e.g.*, CPU and memory) at the function granularity. There are three basic challenges in achieving this goal. *First*, decisions regarding the placement and configuration for each function are critical to the end-to-end Job Completion Time (JCT) and cost. Nonetheless, it remains unclear how to handle the complexity of such co-optimization and data dependencies between stages. *Second*, the performance and cost of a function are inherently related to the allocated resources. Additionally, the cost is further affected by its duration time [19], which is amplified in volatile wide-area settings due to more complex cost components (*e.g.*, WAN cost). *Third*, decoupling a data analytics job into serverless architectures spawns a variety of data-intensive functions with diverse resource demands. Thus, accurately predicting the execution time of each function under different input sizes and resource combinations is critical to estimating both JCT and cost.

We thus propose *Demeter*, a fine-grained function orchestrator that minimizes job execution costs for GSA while ensuring JCT compliance with the agreed-upon SLO. First, we clarify the manner and goal of the fine-grained orchestration. To glean insights from the system operational pattern, *Demeter* adopts a Multi-Agent Reinforcement Learning (MARL) algorithm that jointly makes per-function placement and resource allocation decisions. Specifically, it distills comprehensive and compact cross-DC states from multi-level features through hierarchical Graph Neural Networks (GNN). We then decouple the joint actions according to their serial relation to shrink the decision space. They are encoded by different NNs. To reduce model complexity and the risk of mis-sampling, we devise a pruning mechanism to further focus *Demeter* on a portion of the extensive configuration pool. Besides, we enable elastic parallelism, adjusting the number of functions (*i.e.*, Degree of Parallelism (DoP)) of the triggered stages, for function congestion control. Finally, we implement *Demeter* based on Pheromone [6], an open-source serverless system.

To the best of our knowledge, we are the first to implement GSA, which exploits wide-area resource elasticity with fine-grained function orchestration. In particular, the contributions of this paper are summarized as follows:

- We first formulate the fine-grained function orchestration problem, whose objective is to minimize the job execution costs for GSA while guaranteeing SLO compliance.
- We propose *Demeter*, which makes on-demand orchestration decisions at the function granularity via a customized MARL algorithm with a pruning mechanism.
- We further design a DoP tuning algorithm for *Demeter*, which dynamically decides the DoPs of triggered stages to solve the function congestion issue.
- We build a prototype system of *Demeter*, which outperforms state-of-the-art solutions in terms of the job's SLO compliance and cost savings via extensive experiments.

The remainder of this paper is organized as follows. In Section II, we provide a brief overview of serverless analytics and explain the motivation behind our insights. Section III describes the system model and formulates the fine-grained function orchestration problem. The detailed model design of *Demeter* and the DoP tuning algorithm are introduced in Sections IV and V, respectively. Section VI presents the system architecture of *Demeter*, while Section VII displays and analyzes the experimental results in performance and overhead. The related work is reviewed and discussed in Section VIII, with the paper concluded in Section IX.

II. Background and Motivation

In this section, we provide a brief background on serverless analytics, and the huge potential of generalizing it to wide-area settings, followed by the motivation behind our insights.

A. Serverless Analytics

Similar to traditional big-data systems [9], [14], a serverless analytics job contains a DAG whose nodes are function stages and edges denote their data dependencies. Fig. 1 (left) illustrates an example. Following the event-driven nature, a stage is triggered when all of its dependent stages are completed. As depicted in Fig. 1 (right), stages run in parallel over multiple shuffled partitions of input data, and each partition is handled by a corresponding function. Also, a stage finishes when the last component function concludes. This makes data analytics effectively benefit from serverless computing: (*i*) intra-stage, the resources held by each function are instantly released upon completion (from FaaS itself), and (*ii*) inter-stage, matching resource demand of each stage can yield better utilization [16]. In contrast to the serverful tasks, which are stuck in resource slots divided evenly, serverless functions are loosely-coupled in a job [19]. This affords developers the freedom to decide a *configurable* number of functions (*i.e.*, DoP) in a stage and amount of resources for *each of them*.

New opportunities. The serverless paradigm has struggled to build analytics workloads due to the stateless formulation of functions. Explicitly, direct communication between functions is prohibited, as their containers are not addressable beforehand [5]. Prior studies [7], [8] depend on external storage as an intermediate data relay, along with the *double transfer* issues. Recent advances in serverless systems [6], [20] have, however, altered this case. For instance, Pheromone [6] is a promising solution that enables intra-node functions to communicate via local shared memory, and inter-node functions over networks. This methodology unifies system and data exchange without unstable third-party services, paving the way to expand serverless analytics into wide-area settings. Meanwhile, the *position* of functions with dependencies affects their data I/O efficiency, especially across geo-distributed DCs.

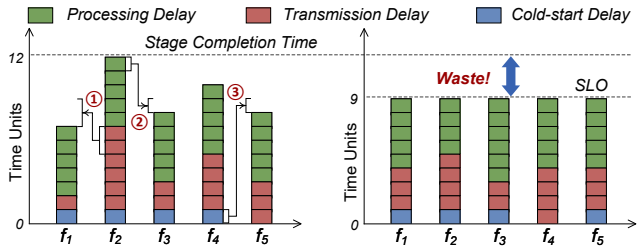


Fig. 2. An illustrative example of the fine-grained function orchestration for a single stage. The stage's DoP is 5, with the green bars equal (*i.e.*, 5 units) on the left and not identical on the right.

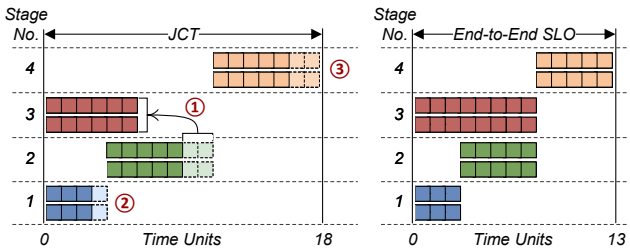


Fig. 3. An illustrative example of the fine-grained function orchestration for a multi-stage job (see Fig. 1).

B. Why Fine-Grained Function Orchestration?

For single stages. Supposing that the investment cost of a function is inversely proportional to its duration, Fig. 2 shows the impact of fine-grained function orchestration on a single stage. Due to the presence of remote data I/O over WAN links, each function could incur a non-negligible transmission delay. Allocating the equivalent computing resource for *all functions within a stage* [15], [16], [17], [18], resulting in their execution times to differ, even without any input skew [21]. Specifically, the most lagging function f_2 shown in Fig. 2 (left) dominates the stage completion time (*i.e.*, 12 units), which hinders the potential performance improvements from *inter-stage benefits*. Additionally, the earlier completed functions incur inevitable output storage costs while waiting for function f_2.

Fig. 2 (left) displays an ideal solution. As shown, it prefers to skew more resources toward the critical function f_2 to speed it up, while over-provisioned functions f_1 and f_3 are slowed down to conserve the budget (steps ① and ②). The roles of different functions within a stage, however, might switch due to their interrelation. For example, when function f_4 becomes the new bottleneck, it is migrated into a warm container (step ③) to maintain the *balanced status* in Fig. 2 (right). That is, in-stage functions finish at similar times. By such means, the stage completion time becomes 9 units (25% lower), while the total execution time remains at 45 units. Thus, the solution is reasonable, as the total cost, despite variations in resource unit prices, would not undergo a significant increase and is fully utilized to satisfy the SLO. Otherwise, more resources should be invested for acceleration.

For multi-stage jobs. A general job involves a stage-wise DAG with complex data dependencies, but it can benefit from fine-grained orchestration as well. Fig. 3 shows this advantage for a multi-stage job with the DAG in Fig. 1. In particular, the naive solution in Fig. 3 (left) is to maintain all stages in balance directly, leading to a JCT of 18 units. However, the start time of stage 4 is determined by the lagging stage 2, implying an inefficient allocation of resources across stages. Fig. 3 (right) portrays a well-optimized solution through achieving the *balanced status* of the entire job. Using an end-to-start method, parallelizable stages (*e.g.*, stages 3 and 2) are also supposed to finish as close as possible (step ①). This is because they share the same downstream dependency (*i.e.,* stage 4), and can be treated as one large *single stage* that follows the above insight. Due to the cascading of dependencies, the end-to-end SLO can be met by further accelerating stages 1 and 4 (steps ② and ③). This way reduces the JCT to 13 units, which is 28% lower than that of Fig. 3 (left). As such, a good orchestrator should accurately identify the roles of functions, and make on-demand orchestration (*i.e.*, placement and configuration) for each one, contributing to JCT within the SLO at optimal cost.

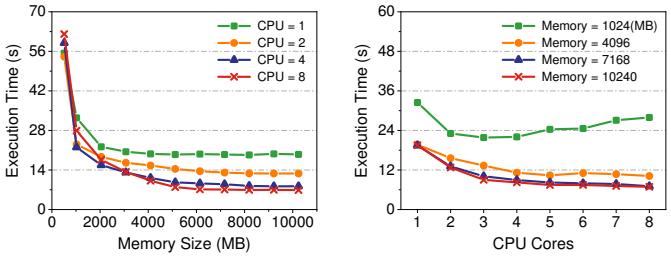
C. Challenges

The complexity of the joint optimization and the impact of function performance on JCT pose three challenges:

Challenge 1: Complex intra- and inter-function correlations. Both resource allocation and strategic placement of each function will collectively affect its performance. The intuition is to enumerate all the possible combinations to seek the best orchestration plan. However, a serverless analytics job involves a variety of functions belonging to various stages, each with a large two-dimensional configuration pool (*e.g.*, AWS Lambda [19] allows memory limits within $[128, 10240]$MB and up to 6 cores). To achieve the above *balanced status* of jobs involving complex data dependencies, such fine-grained decisions of all functions require to be considered collectively. As a result, the search range for enumeration will be huge.

Challenge 2: Volatile and bursty geo-distributed serverless analytics environments. The remote function communication (*e.g.*, during the all-to-all *shuffle* phase) requires traverse WAN links between DCs [10]. As observed, both the total and per connection WAN bandwidths are severely limited and in sharp fluctuations (*i.e.*, 24% to 76% deviation from the mean), which is also reported to be distinctly tidal [3]. Such volatile WANs exacerbate the function expenses. For example, it must bear dual costs in terms of blocking execution and incurred WAN traffic, during remote data reading. Moreover, serverless platforms inevitably undergo cold starts and congestion when faced with bursty-parallel functions [22], even with techniques such as *keep-alive* [23]. The burstiness and volatility of GSA environments jointly make it non-trivial to orchestrate large-scale functions within triggered stages.

Challenge 3: Diverse function types and resource demands. Functions in an analytics job are data-intensive, with their execution times being heavily related to type and input size [24]. Nevertheless, we still find that a function exhibits varying resource sensitivities, even with a specific input. We run a MapReduce *sort* over 10 GB data on Pheromone-MR [6] repeatedly. As illustrated in Fig. 4, increasing CPU cores for a function with a fixed memory capacity can effectively enhance the performance. Yet, CPU over-provisioning can exacerbate the multi-process overheads (*e.g.*, startup and blocking), which become significant bottlenecks and result in an execution time



(a) Varying with memory size. (b) Varying with CPU cores.

Fig. 4. The execution time of a data-intensive function under different CPU-memory combinations.

bounce. On the other hand, the multi-core benefits can only be fully utilized when there is ample memory size. Besides, inside information about user functions (packed as a black box [25]) is denied access, which further complicates the estimation of the resource demands for each function within jobs.

III. Preliminaries

In this section, we first elucidate the system model and the function orchestration methodology within a GSA job. Subsequently, we formulate our fine-grained function orchestration problem. For clarity, the vital notations are listed in Table I.

TABLE I
The Notations and Their Definition.

Notation	Definition
$\mathcal{G} = (\mathcal{N}, \mathcal{L})$	A system with set of DCs $\mathcal{N}$ and network links $\mathcal{L}$.
$n \in \mathcal{N}$	The index of the n-th geo-distributed DC.
$l_{i,j} \in \mathcal{L}$	The WAN link between the i-th and j-th DCs.
$(p_c, p_m) \in \mathcal{P}$	The index of the p-th resource allocation type with p_c cores and memory limit p_m.
$t \in \mathcal{T}$	The index of the t-th wall-clock timestep.
$\mathcal{N}_j$	The subset of DCs where the input data of job j lies.
$\mathcal{S}_j$	The expected end-to-end SLO of job j.
$v_{j,k} \in \mathcal{V}_j$	The index of the k-th stage of job j.
$f_{j,k,i} \in \mathcal{F}_{j,k}$	The index of the i-th function of stage $v_{j,k}$.
$Q_{n,t}$	The set of functions in the waiting queue of DC n at time t.
$P_{n,t}$	The set of functions in the "*wave*" of DC n at time t.
$x^n_{j,k,i} \in X$	A binary variable $x^n_{j,k,i} = 1$ if function $f_{j,k,i}$ is placed on DC n; and 0 otherwise.
$y^p_{j,k,i} \in Y$	A binary variable $y^p_{j,k,i} = 1$ if function $f_{j,k,i}$ selects a type-p allocation; and 0 otherwise.
$\tilde{\phi}_j, \tilde{\phi}_{j,k,i}$	The arrival time of job j and function $f_{j,k,i}$, respectively.
$\phi_j, \phi_{j,k}, \phi_{j,k,i}$	The finish timestamps of job j, stage $v_{j,k}$ and function $f_{j,k,i}$, respectively.
$C_{j,k,i}, C_j$	The cost of function $f_{j,k,i}$ and job j, respectively.

A. System Model

We consider a system $\mathcal{G} = (\mathcal{N}, \mathcal{L})$ as illustrated in Fig. 5, consisting of N geo-distributed DCs in the set $\mathcal{N}$ that enable serverless analytics services, with the assumption that they can provide sufficient computing resources [10]. Contrary to the relatively stable network with adequate bandwidth inside them, these DCs are inter-connected via volatile and limited WAN links $\mathcal{L} = \{l_{i,j} \mid 1 \le i, j \le N, i \ne j\}$. The user will request to query data stored on a subset of DCs, which is parsed into a serverless analytics job comprising stateless functions potentially distributed on these DCs [7]. As previously discussed, we consider the decoupled function configuration of any CPU-memory combination, without forcing a proportional allocation [19]. So we define an allocation type as $p = (p_c, p_m) \in \mathcal{P}$, where p_c and p_m are the number of CPU cores and memory limits, respectively. Here, p is non-preemptive and consistently available to a function until the end or timeout.

B. Function Orchestration for Geo-Distributed Analytics

Serverless systems typically operate in a multi-tenant environment, with new requests from various users being registered continuously [26]. We thus consider a total of T wall-clock timesteps $\mathcal{T} = [1, \ldots, t, \ldots, T]$ that is divided evenly based on the system performance. A total of J GSA jobs will arrive continuously at any time in $t \in \mathcal{T}$. In practice, each user tends to launch the same query periodically on the freshest data [4]. Therefore, we do not distinguish between a user and a job below. The query request Ω_j of job j, submitted by user j, is represented as a tuple, given by:

$$\Omega_j = \left\{ \tilde{\phi}_j, \mathcal{P}_j, \mathcal{S}_j, \mathcal{N}_j, \left\{ m^n_j, l^n_j \right\}_{\forall n} \right\}, \tag{1}$$

where $\tilde{\phi}_j$ is the arrival time, $\mathcal{P}_j$ is the user-defined job profile, which includes metadata of stages and their trigger modes. $\mathcal{S}_j$ is the expected job's end-to-end SLO, $\mathcal{N}_j \in \mathcal{N}$ is the subset of DCs where the pending data is located, and m^n_j, l^n_j sign the path and offset of data bucket in DC $n \in \mathcal{N}_j$.

Upon receiving Ω_j, the system parses the job profile $\mathcal{P}_j$ into multiple event-driven stages $\mathcal{V}_j = \{v_{j,1}, \ldots, v_{j,k}, \ldots, v_{j,|\mathcal{V}_j|}\}$, where $v_{j,k}$ denotes the k-th stage of job j. Affected by cascades of upstream dependencies, each stage may be triggered at any time in $t \in \mathcal{T}$. To avoid functions with long waits to be orchestrated, we enable the distributed scheduling: each function of stage $v_{j,k}$ in $\mathcal{F}_{j,k} = \{f_{j,k,1}, \ldots, f_{j,k,i}, \ldots, f_{j,k,|\mathcal{F}_{j,k}|}\}$ sends its invocation request to all the DCs in $\mathcal{N}_j$, where $f_{j,k,i}$ denotes the i-th function in the k-th stage of job j. They are appended to $Q_{n,t}$, which is the set of functions in the waiting queue of DC n at t. Each DC schedules the functions in $Q_{n,t}$ independently, but each of which is restricted to be assigned to one DC. To avoid conflicts, it is essential to determine by a coordinator with a global view (see Section IV-C). Finally, the assigned functions are flexibly configured with customized resources to adapt to the task complexity.

Notably, the number of functions in triggered stages could exceed the limits of batch orchestration. Each DC thus will continuously receive an uncertain number of function "*waves*" during T timesteps, each of which denotes a batch of parallel functions from co-existing jobs, *i.e.*, the basic orchestration unit. For each DC $n \in \mathcal{N}$, it extracts functions in the current "*wave*" by dequeuing functions in $Q_{n,t}$ until pending function set $P_{n,t}$ is full or $Q_{n,t}$ is empty. In a nutshell, given J job profiles with arbitrary dependencies, each DC n needs to batch orchestrate all functions in $P_{n,t}$ at timestep t, which considers two decision variables: (*i*) $x^n_{j,k,i}$ in set X, indicating whether function $f_{j,k,i}$ is placed on DC n $(= 1)$ or not $(= 0)$, and (*ii*) $y^p_{j,k,i}$ in set Y, denoting whether function $f_{j,k,i}$ selects a type-p allocation for its container runtime $(= 1)$ or not $(= 0)$.

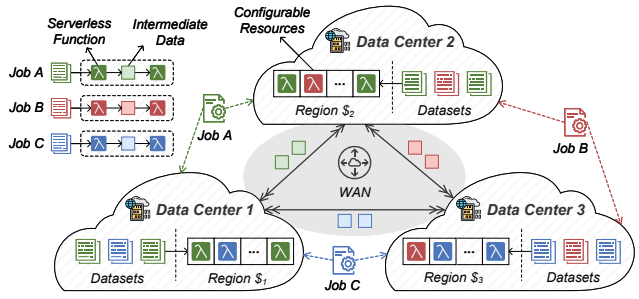


Fig. 5. The system infrastructure of serverless analytics over DCs located in distinct geographic regions. The functions and data with the same color belong to a job A, B, or C. The unit price ($) of resource (*e.g.*, CPU, memory, and WAN bandwidth) in various regions are generally heterogeneous [19], [27].

C. Function-Level Models

1) Function Duration Time: In practice, parallel functions in a "*wave*" are orchestrated in sequence, and thus not started simultaneously. The reason is that batch function orchestration remains in a First-Come-First-Served (FCFS) paradigm at the micro-level [28]. Besides the time components illustrated in Fig. 2, functions are accompanied by an inevitable and skewed waiting time before formal execution.

Intuitively, the execution time of a data-intensive function is related to our allocation decision Y (see Fig. 4) and the input size. In addition, it also suffers from non-negligible cold-start and existing remote input transfer over WAN links $\mathcal{L}$, which depends on our placement decision X. Due to the presence of inter-function interference and inherent cloud noise [16], the decisions for widespread co-existing functions will also affect it. Thus, the duration of function $f_{j,k,i}$ can be formulated as:

$$\delta_{j,k,i} = \Delta_{j,k,i} + B\left(X, Y, d_{j,k,i}\right), \tag{2}$$

where $B(\cdot)$ is regarded as a black-box function that expresses the execution time, $\Delta_{j,k,i}$ and $d_{j,k,i}$ are the waiting time and input data size of $f_{j,k,i}$, respectively. Its finish timestamp can be calculated as $\phi_{j,k,i} = \tilde{\phi}_{j,k,i} + \delta_{j,k,i}$, where $\tilde{\phi}_{j,k,i}$ is the arrival time. Further, the finish timestamp of stage $v_{j,k}$ relies on its last completed function, *i.e.*, $\phi_{j,k} = \max_{i\in[|\mathcal{F}_{j,k}|]}\phi_{j,k,i}$. Here we use $[S]$ to denote the set $\{1, 2, \cdots, S\}$.

2) Function Cost Model: In our problem setting, there are operating costs, transmission costs, and storage costs.

Operating cost. The operating cost is correlated with the function duration. We denote the price per core-second (CPU) and GB-second (memory) of function at DC (*i.e.*, region) n as μ_m^n and μ_c^n, respectively, and the price for each function request and orchestration as σ_n. The operating cost $C^{op}_{j,k,i,n,p}$ of function $f_{j,k,i}$ with type-p allocation at DC n is:

$$C^{op}_{j,k,i,n,p} = \delta_{j,k,i}\left(\mu_c^n \cdot p_c + \mu_m^n \cdot p_m\right) + \sigma_n. \tag{3}$$

Transmission cost. The transmission cost is expended on reading remote results via expensive WAN links, as local data exchange that briefly uses shared memory is generally free. Let $q_{i,j}$ be the price per GB of data transferred via $l_{i,j} \in \mathcal{L}$. The transmission cost $C^{trans}_{j,k,i,n}$ of $f_{j,k,i}$ at DC n is:

$$C^{trans}_{j,k,i,n} = \sum\nolimits_{l\in US_{j,k,i}} s^l_{j,k,i} \cdot q_{n_l,n}, \tag{4}$$

where $s^l_{j,k,i}$ is the amount of data read from upstream function $l \in US_{j,k,i}$, and n_l is the DC where l is deployed.

Storage cost. Users incur extra costs for ephemeral storage, which arises from: (*i*) the output of functions that finish early is stored, awaiting the stage finishes, and (*ii*) the fault-tolerant mechanism retains produced data until the dependent stages end, ensuring it can be retransmitted in case of anomalies. Let h_n denote the price for storing data per GB-second at DC n. The storage cost $C^{store}_{j,k,i,n}$ of $f_{j,k,i}$ at DC n is:

$$C^{store}_{j,k,i,n} = (\underbrace{\phi_{j,k} - \phi_{j,k,i}}_{(i)} + \underbrace{\max_d \phi_{j,d} - \phi_{j,k}}_{(ii)}) \cdot m_{j,k,i} \cdot h_n, \tag{5}$$

where $m_{j,k,i}$ is the amount of data output by $f_{j,k,i}$, and $\phi_{j,d}$ is the finish timestamp of its downstream stage $v_{j,d} \in \mathcal{DS}_{j,k}$.

D. Problem Formulation

We now formulate the fine-grained function orchestration problem in (6) – (14), which jointly optimizes the per-function placement X and resource allocation Y to exploit wide-area resource elasticity. The objective is to minimize the execution costs of J jobs that arrive continually over a period $\mathcal{T}$, while meeting the respective SLOs, and hard constraints of resource demands and data dependencies. In view of the volatility and burstiness of the environment (*i.e.*, volatile WAN bandwidths and bursty-parallel functions), we ensure each job is completed within the expected end-to-end SLO by orchestrating component stages with fine-grained and best-effort.

$$\min_{X,Y} \sum_{j=1}^{J} \omega_j \cdot C_j\left(X, Y\right). \tag{6}$$

$$\text{s.t.} \quad \phi_{j,k} \le \phi_j, \quad \forall v_{j,k} \tag{7}$$

$$C_j = \sum_{n\in\mathcal{N}_j} \sum_{i\in[|\mathcal{F}_{j,k}|]} \sum_{k\in[|\mathcal{V}_j|]} x^n_{j,k,i} \sum_{p\in\mathcal{P}} y^p_{j,k,i} C_{j,k,i}, \quad \forall j \tag{8}$$

$$\phi_j - \tilde{\phi}_j \le \mathcal{S}_j, \quad \forall j \tag{9}$$

$$\sum\nolimits_{n\in\mathcal{N}_j} x^n_{j,k,i} = 1, \quad \forall f_{j,k,i} \tag{10}$$

$$\sum\nolimits_{p\in\mathcal{P}} y^p_{j,k,i} = 1, \quad \forall f_{j,k,i} \tag{11}$$

$$x^n_{j,k,i}, y^p_{j,k,i} \in \{0, 1\}, \quad \forall n, p, f_{j,k,i} \tag{12}$$

$$\alpha \sum\nolimits_{p\in\mathcal{P}} y^p_{j,k,i} \cdot p_m \ge d_{j,k,i}, \quad \forall f_{j,k,i} \tag{13}$$

$$\delta_{j,k,i} \le \Gamma. \quad \forall f_{j,k,i} \tag{14}$$

where ω_j is a non-negative weight that denotes the priority of job j, and $C_{j,k,i} = C^{op}_{j,k,i,n,p} + C^{trans}_{j,k,i,n} + C^{store}_{j,k,i,n}$. Inequality (7) ensures that the finish timestamp ϕ_j of job j depends on the last completed stage. Equation (8) defines the cost C_j of job j as the aggregated cost of the full composition function. Inequality (9) requires the JCT of job j to obey its SLO $\mathcal{S}_j$. Also, each function is placed on only one DC (constraint (10)), and selects only one allocation type to deliver its container runtime (constraint (11)). Constraint (12) ensures that both the placement and allocation decisions are binary. To avoid *out-of-memory* errors, each function should be allocated a memory limit with a loss factor α to ensure that it is larger than the input size $d_{j,k,i}$ (constraint (13)). Finally, constraint (14) stems from the serverless platform's short-lived limit on functions [19], *i.e.*, their durations should not exceed the timeout Γ.

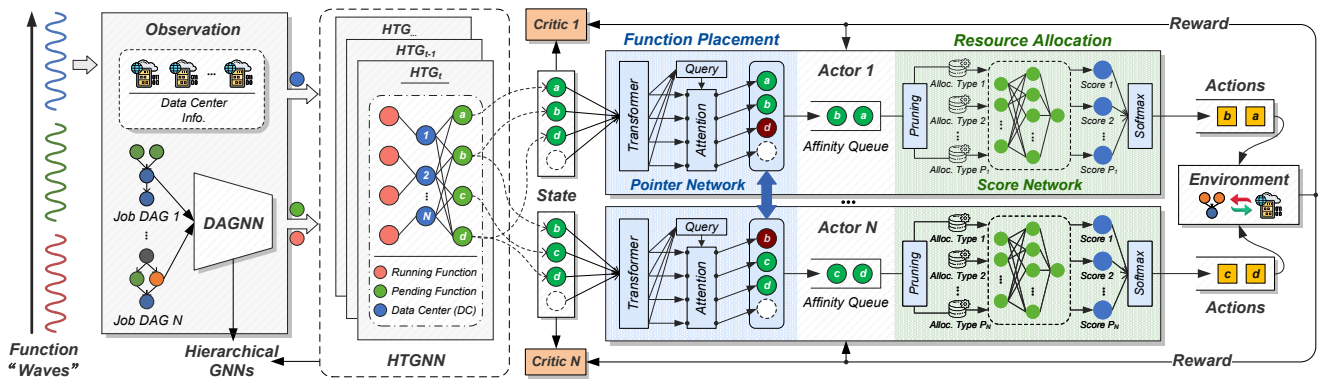


Fig. 6. The model workflow of *Demeter*.

IV. MARL-Based Function Orchestration

In this section, we present the design of *Demeter*, which exploits wide-area resource elasticity with fine-grained function orchestration for serverless analytics.

A. Algorithm Overview

Orchestrating functions across event-driven stages in "*wave*" poses a challenging *sequential decision-making* problem. We have the opportunity to explore the unknown performance objective $B(\cdot)$ and the impact of data dependencies (7) online by replaying experiences, while accommodating the distributed orchestration in Section III-B. Coupled with the generally fixed and limited number of DCs for a cloud provider, *Demeter* is devised as a MARL-based solution. Fig. 6 shows the workflow of *Demeter* with an actor-critic architecture. Each agent $n \in \mathcal{N}$ serves as the local scheduler for the DC n. With the state of a function "*wave*" as input, each agent outputs a series of three-dimensional actions, to place all pending functions within the "*wave*" and allocate suitable resources to them.

To cope with the volatile and bursty GSA environments, *Demeter* generates holistic and compact states for each agent using scalable and hierarchical GNNs. These GNNs efficiently extract in-depth semantic relations among functions and DCs over multi-level features, capturing the time-varying impact of network and function dependencies. ***To handle the complex intra- and inter-function correlations***, we introduce an actor network with a *pointer-score* architecture to decouple the joint optimization (6) based on the serial relation between decisions X and Y. Also, the network facilitates batch orchestration to further tap the inter-function correlation. ***To learn the diverse resource demands of various functions***, the decoupled actions are encoded by different NNs. For function placement, agents decode input states via a pointer network and collaborate to make an affinity-guided decision. For resource allocation, the raw states of placed functions and their pruned allocation types are fed into a score network, spawning the corresponding resource priorities. The state representation and action encoding are detailed in Sections IV-B and IV-C, respectively.

B. State Representation via Hierarchical GNNs

1) Feature Construction: Upon receiving a function "*wave*" at time t, *Demeter* collects information from two levels: DC and function, obtaining the desired raw features based on the factors in Section III-C.

At the function level, we first classify all the functions in active job j based on their current status. Next, we separately construct raw features for functions in each group, *i.e.*, $\mathcal{H}_{j,t} = \{\mathcal{R}_{j,t}, \mathcal{P}_{j,t}, \mathcal{W}_{j,t}, \mathcal{F}_{j,t}\}$, where $\mathcal{R}_{j,t}$, $\mathcal{P}_{j,t}$, $\mathcal{W}_{j,t}$ and $\mathcal{F}_{j,t}$ are the feature sets of functions in running, pending, waiting and finished status, respectively. The vectors of $(|\mathcal{V}_j| + |\mathcal{N}_j| + 4)$ dimensions in $\mathcal{H}_{j,t}$ are as listed below:

- $\boldsymbol{\mathcal{R}_{j,t}}$ includes the type, current placement, resource configuration, processed data size, and elapsed time for every *running* function in job j.
- $\boldsymbol{\mathcal{P}_{j,t}}$ includes the type, pending data size, and invocation log that records the previous placement, resource peak, and duration time for every *pending* function in job j.
- $\boldsymbol{\mathcal{W}_{j,t}}$ includes the type, and invocation log that records the previous placement, input data size, resource peak, and duration time for every *waiting* function in job j.
- $\boldsymbol{\mathcal{F}_{j,t}}$ includes the type, and invocation log that records the placement, resource peak, input data size, and duration time for every *finished* function in job j.

At the DC level, *Demeter* captures the number of remaining functions and warm containers in each DC $n \in \mathcal{N}$, and its available bandwidth per connection with other DCs at t. In addition, they are concatenated with a series of scalars, *i.e.*, the unit prices of required resources in the current region, into a flattened vector $d_{n,t}$ with a dimension of $(|\mathcal{N}| + 7)$.

2) Hierarchical Embedding: As illustrated in Fig. 6 (left), our state representation proceeds in a hierarchical and centralized manner. Initially, the DAGs of active jobs[1] are embedded by a GNN. Subsequently, the heterogeneous deployment graph, formed by the aforementioned GNN embeddings of active functions and the associated potential DCs, is further encoded by another GNN. Finally, the generated embeddings across DCs are divided into input states for each agent.

DAGNN. A job DAG[2] defines fine-grained data dependencies via edges, which are partial ordering over function nodes. We naturally desire to integrate this strong inductive bias into the DAG representation, as it covers vital information for the

[1]The arrival of a "*wave*" will activate multiple jobs, and the functions that are running or runnable in these jobs are called active functions.

[2]The job DAG in our paper is a function-level DAG (see Section VI-B).

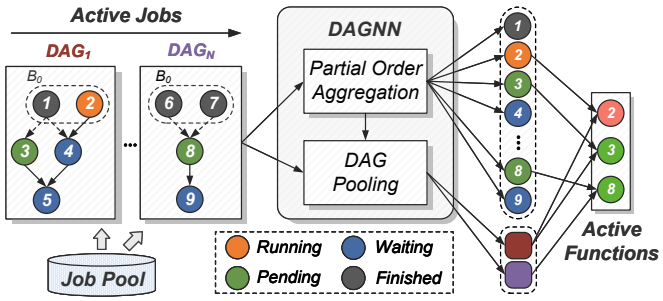


Fig. 7. The embedding process of our DAGNN. For ease of display, we give an example of single-function stages. Note that, the dashed (solid) lines in a DAG depict data dependencies that have been (un)satisfied, and the "waiting" function denotes a function that has not yet been triggered.

correlation between event-driven functions. DAGNN [29] is driven by DAG-induced partial ordering, aligning well with our goal. It aggregates only the predecessor to generate the embedding h_v^l for each node v in DAG $\mathcal{G}$ at each layer l, and readouts the final graph embedding $h_{\mathcal{G}}$ via DAG pooling.

Moreover, *topological batching* is incorporated in DAGNN to process nodes in parallel. It divides a DAG based on the topological order, where nodes without dependencies join the same sequential batch $\{B_i\}_{i\geq 0}$. All nodes in batch B_i can be handled concurrently if their predecessors have all completed. Note that, *topological batching* is capable of scaling to multi-DAG cases for better parallel concurrency, as shown in Fig. 7. By treating multiple DAGs as one disconnected large DAG, we can merge their batches B_i of the same i for processing simultaneously. This scalable way enables efficient embedding for co-existing jobs. To this end, we combine the node features $\mathcal{H}_{j,t}$ of each job DAG $\mathcal{G}_j$ as DAGNN's initial input $\mathcal{H}_t^{(0)}$.

$$\mathcal{H}_t^{(0)} = \left\{\mathcal{R}_t^{(0)}, \mathcal{P}_t^{(0)}, \mathcal{W}_t^{(0)}, \mathcal{F}_t^{(0)}\right\} = \bigcup\nolimits_{j\in\mathcal{J}|_t^{t+1}} \mathcal{H}_{j,t}, \quad (15)$$

where $\mathcal{J}$ is the set of active jobs at t. After L-layer message propagation with *topological batching*, we can get a function embedding set $\mathcal{H}_t^{(L)} = \{\mathcal{R}_t^{(L)}, \mathcal{P}_t^{(L)}, \mathcal{W}_t^{(L)}, \mathcal{F}_t^{(L)}\}$ and a job embedding set $\{h_{\mathcal{G}_j,t}\}_{\forall j}$. As shown in Fig. 7, we concatenate each embedding in $\{\mathcal{R}_t^{(L)}, \mathcal{P}_t^{(L)}\}$ with its job embedding, thus forming the final DAGNN representations of active functions, denoted as $\{\bar{\mathcal{R}}_t, \bar{\mathcal{P}}_t\}$. They serve as raw features for function nodes in another GNN that will be presented soon.

HTGNN. Note that directly stacking the features of pending functions and potential DCs for each agent n will produce a redundant state space (*e.g.*, the same function in various $P_{n,t}$). Also, the network condition observed at any time in t cannot accurately reflect the real function runtime, due to the WAN fluctuation and cross-traffic among DCs [3]. Given the hidden graph structure in function deployment, we use another GNN to learn dynamic node representations, while compressing the state space via information aggregate across DCs.

Since active functions cover the most realistic information reflecting their roles (*i.e.*, resource demands) at the moment, we connect them to associated DCs using edges, thus forming a heterogeneous deployment graph G_t in Fig. 6. Precisely, a running function node $f \in \bar{\mathcal{R}}_t$ connects with a DC node n if function f is running on DC n, and a pending function node $f \in \bar{\mathcal{P}}_t$ connects with a DC node n if it is in the pending set $P_{n,t}$. We further design a Heterogeneous Temporal Graph (HTG) $G = \{G_{t'}\}_{t'=1}^{t}$, where the slice $G_{t'}$ is the deployment graph at t'. The intuition is that features $\{d_{n,t'}\}_{t'=1}^{t}$ of DC node n in the HTG are temporal-dependent [3], and the heterogeneous structures among graph slices are spatial-dependent due to the execution evolution of active function nodes.

Fittingly, HTGNN [30] integrates spatial and temporal dependencies to learn from historical evolution over our HTG G. In each layer l, it first aggregates intra- and inter-relations with neighbors for each node $v \in G_{t'}$, then further embeds shared DC nodes via temporal aggregation across $G_{t'}$. Let $\{\bar{\mathcal{R}}_{t'}, \bar{\mathcal{P}}_{t'}\}$ and $\{d_{n,t'}\}_{\forall n}$ be the raw features of the function nodes and DC nodes in $G_{t'}$, respectively. After stacking L layers, we can derive the embeddings $\bar{\mathcal{P}}_t^{(L)}$ of pending function nodes at t. Finally, we extract the HTGNN embeddings of all functions in $P_{n,t}$ from $\bar{\mathcal{P}}_t^{(L)}$, as in Fig. 6, forming a set $\mathcal{P}_{n,t}$ as the input state of the actor network (stated in Section IV-C) in agent n.

C. Action Decoupling Based on Pointer-Score Network

The architectural design of our actor networks is based on the key insight that decouples joint placement and allocation actions through their serial relation in (8). More specifically, we only assign resources for favorably placed functions, which avoids combining two actions into a large space for each agent. To this end, we design a well-assembled pointer-score network to encode the decoupled actions, while further decreasing the computing complexity of training agents.

1) Pointer Network: Given the relevance of parallel functions in Section II-B, those in a "*wave*" are critical contextual information for each other, and should be processed together. However, their number $|P_{n,t}|$ is unknown for each DC n, due to the uncertainty of the stage triggering. Although the length of the input state sequence $\mathcal{P}_{n,t}$ is capable of being fixed to the maximum by *padding* method in the Seq2Seq model, it cannot be used to fix the output. We thereby cannot precisely extract the information associated with each function from the output sequence. In contrast, the pointer network [31] can learn the conditional probability of an output sequence against the element positions in an input sequence, which is well-suited for encoding the placement action. Since functions in a "*wave*" are equal, we re-design the pointer network for agent n using Transformer without *positional encoding* as the encoder and a single-head attention mechanism as the decoder, given by (16) as follows:

$$h_{f,n,t} = \begin{cases} -\infty, & \text{MASK}\,(f) = 0, \\ v^T \tanh\left(W_1 e_{f,n,t} + W_2 e_{n,t}\right), & \text{o/w}, \end{cases} \quad (16)$$

where W_1, W_2, and v^T are learnable parameters, $e_{f,n,t}$ is the further Transformer embedding of each function $f \in P_{n,t}$ over their states $\mathcal{P}_{n,t}$, to discover correlations within "*waves*", and $e_{n,t} = \text{avg}_{f\in P_{n,t}} e_{f,n,t}$. The final attention score $h_{f,n,t}$ is used as a pointer to the corresponding function f. Given restriction (10), pointer $h_{f,n,t}$ is turned into the conditional probability $\Pr(f|n)$ via a Softmax operator [31], which can be regarded as the action that DC n responds to function f individually. Note that, the *padding* will never be visited, as its score is masked to $-\infty$ and thus has probability 0. Finally, we utilize

Bayes' theorem to compute the posterior probability $\Pr(n|f)$ in (17), guiding the final placement of function f [32].

$$\Pr(n|f) = \frac{\Pr(f|n)\Pr(n)}{\sum_{n\in\mathcal{N}_j}\Pr(f|n)\Pr(n)}. \tag{17}$$

Considering DC load balancing [33] and less inter-function interference, we approximate $\Pr(n)$ with the number of functions running on DC n. Indeed, we place the function f on the DC with the largest posterior probability (*i.e.*, affinity):

$$x^n_{f:f_{j,k,i}} = \begin{cases} 1, & n = \text{argmax}_{n^*}\Pr(n^*|f), \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

2) Score Network: It is essential to filter out the allocation types that violate the hard constraint (13) prior to the formal decision-making. This is because too low memory limits lead to functions crashing directly, with no latitude for trial-and-error [34]. While the output actions involving the above invalid allocations can be simply masked by the actor network [32], it still has to take unnecessary effort for them. To prevent this, we introduce a score network, implemented as a multi-layer perceptron with a single output neuron, to encode our resource allocation action by dropping the invalid types.

Fig. 6 (right) visualizes the workflow of the score network. Once the pointer network completes, agent n further handles the HTGNN embeddings of functions placed on DC n via an affinity queue. That is, functions with lower affinity should be orchestrated as early as possible to enhance performance by reducing waiting time. Specifically, each embedding concatenates with P' valid potential allocations in turn to produce a batch of states $s_{f,n,t} = (s^1_{f,n,t}, \ldots, s^p_{f,n,t}, \ldots, s^{P'}_{f,n,t})$, where $s^p_{f,n,t}$ refers to the state to select type-p allocation for function f. Then, *Demeter* feeds $s^p_{f,n,t}$ into the score network to get a scalar value $g^p_{f,n,t}$, which is interpreted as the score of type-p allocation [25]. Finally, *Demeter* applies a Softmax operation to convert the scores in $(g^1_{f,n,t}, \ldots, g^{P'}_{f,n,t})$ into the resource allocation policy as follows:

$$y^p_{f:f_{j,k,i}} = \begin{cases} 1, & p = \text{argmax}_{p^*}\text{Softmax}(g^{p^*}_{f,n,t}), \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

Besides eschewing redundant computations, the score network converges more easily due to its architectural simplicity. It also facilitates model-parallelism (for the P score models), accelerating resource allocation for each function.

Pruning mechanism. Despite the advantages of the score network, allocating resources for a "*wave*" across wide function configuration spaces still incurs significant inference overhead and makes it likely to sample in-appropriate allocation types. Runaway function performance may cascade across the entire job once mis-provisioned, with this influence potentially leading to SLO violations. In light of this, we devise a pruning mechanism to guide decision-making by focusing *Demeter* on a certain segment of each function's configuration pool.

Algorithm 1 illustrates the pruning mechanism that builds on function invocation records from recent job history. Recall that job runs over time generally have varying input sizes, which also applies to any of its functions. For simplicity, we categorize the invocation records for each type of function by input size. An *input size class* refers to a scope of input sizes that exhibit similar resource demand [24]. As such, *Demeter* only queries the records that fall within the belonging class (line 10) and prunes the configuration pool for the functions in the affinity queue accordingly. More specifically, we first calibrate the resource lower bound based on the over-provisioned invocations (lines 13–14). If usage peaks exceed 80% of the allocation, an invocation is deemed to be overloaded and needs extra resources. Afterward, a function is assessed by treating each of its invocations as the *baseline* (lines 16–19). During orchestration, each invocation has spawned two properties: (*i*) affinity β_1, which reflects the function performance in terms of startup and communication, and (*ii*) progress β_2, denoting the urgency for job completion, derived from (20) as follows,

$$\beta_2 = \frac{\max_{f\in F_{j,t-1}}\phi_f - \tilde{\phi}_j}{\text{SLO}}, \tag{20}$$

where $F_{j,t-1}$ is the finished function set for job j until $t-1$, and thus the numerator counts its elapsed time. If the current function has higher affinity and slower progress, its configuration range should naturally be looser than the baseline. Conversely, it is supposed to be tightened. Nonetheless, there are partial invocations whose resource demands are ambiguous. That is, they are above or below the current function in both properties. Given this, we combine the two above properties into a unified metric β, specified by $\beta_1(1-\beta_2)$. *Demeter* can prune the function configuration pool by comparing its β with that of the baseline. In case of urgency, it is reasonable to search upwards from the *baseline type* to accelerate function execution, or downwards in reverse to reduce costs. During this period, the pool is guaranteed not to be empty. Otherwise, the pruning will be terminated immediately (lines 20–22).

Algorithm 1: The pruning mechanism of *Demeter*.

Input : Affinity queue of DC n at time t, $\mathcal{A}_{n,t}$;
Function allocation type set, $\mathcal{P}$;
Output: Pruned configuration pool, M;

1 **while** $\mathcal{A}_{n,t}$ is not empty **do**
2 $func \leftarrow \mathcal{A}_{n,t}$.`dequeue` ();
3 **Initialize** the configuration pool: $M \leftarrow [1, |\mathcal{P}|]$;
4 **for** $p \in \mathcal{P}$ **do**
5 **if** $\alpha \times p_m \geq func.input\_size$ **then**
6 $M = M \cap [p, |\mathcal{P}|]$; **break**;
7 $\beta_2 \leftarrow$ current job progress obtained by (20);
8 $func$.`Set_Progress` (β_2);
9 Get the unified metric: $\beta \leftarrow func.\beta_1 \times (1-\beta_2)$;
10 $records \leftarrow$ `Query_History` $(func.stage\_id, func.input\_size)$;
11 **for** $rec \in records$ **do**
12 $baseline\_type \leftarrow rec.alloc\_type$;
13 **if** *rec.peak / rec.alloc* $\geq$ *0.8* **then**
14 $temp \leftarrow M \cap [baseline\_type, |\mathcal{P}|]$;
15 **else**
16 **if** $rec.\beta_1 \times rec.\beta_2 > \beta$ **then** ▷ **Urgency**
17 $temp \leftarrow M \cap [baseline\_type, |\mathcal{P}|]$;
18 **else** ▷ **Non-urgency**
19 $temp \leftarrow M \cap [1, baseline\_type]$;
20 **if** *temp* is empty **then**
21 **break**;
22 $M \leftarrow temp$;

D. Model Training

To learn the explicit end-to-end goal, we train *Demeter* in episodes. An episode contains function "*waves*" from J jobs, with each requiring one or more actions for orchestration.

Reward. In timestep t, each agent receives an immediate reward 0 after taking an action to orchestrate a function. The time in MARL next advances forward. Until *Demeter* finishes the orchestration of the current "*wave*", the environment state does not move on the time axis. Recall that job cost and JCT are considered in our problem, where the former is the direct optimization goal (6), and the latter deals with the SLO constraint (9). They both necessitate global information to be determined, we thus feed *Demeter* with a shared reward r_t,

$$r_t = -\sum_{j\in\mathcal{J}|_t^{t+1}} \omega_j\Big(\sum_{f\in P_{j,t}} C_f - \beta(S_j^* - l_j)\Big), \tag{21}$$

where $\omega_j = 1/|\mathcal{J}|$, $\mathcal{J}$ is the set of active jobs at t, $P_{j,t}$ is the pending function set of job j at t, C_f is the total cost of function f, and β is the penalty factor. The remaining SLO $S_j^* = \mathcal{S}_j - (\max_{f\in P_{j,t}}\phi_f - \tilde{\phi}_j)$ of job j for awarding good and total timeout $l_j = \sum_{f\in P_{j,t}} \max(\delta_f - \Gamma, 0)$ for penalizing bad actions, where δ_f and ϕ_f are the duration and finish timestamp of function f, respectively. Note that *Demeter* is guided to ensure that JCT of job j complies with the SLO by optimizing the gap S_j^* between them continuously, instead of imposing a penalty upon final violation. This is because such sudden and large penalties cannot provide timely and effective feedback to "*wave*"-wise orchestration actions, and may even hinder the model convergence. The objective of agent n is to maximize the cumulative reward given by $\mathcal{J}(\theta_n) = \mathbb{E}\left[\sum_{t=1}^{T}\gamma^{t-1}r_t\right]$, where the discount factor γ is set to 1 as these actions are of equal importance over time [35].

Training algorithm. Due to the high concurrency and short-lived nature of serverless functions in jobs, lots of invocation information is available in every timestep. To relieve training instability, we select a cooperative policy gradient-based algorithm, Multi-Agent Proximal Policy Optimization (MAPPO) [36], to jointly learn the near-optimal orchestration policy. We primarily focus on two benefits: (*i*) PPO is a stable on-policy method that can better handle large-scale samples while being much simpler to adjust, and (*ii*) the smooth policy updates in PPO can alleviate the nonstationarity issue in MARL [37].

For each agent n, the MAPPO algorithm defines a ratio function $p_{n,t}(\theta) = \frac{\pi_{\theta_n}(a_{n,t}|s_{n,t})}{\pi_{\theta'_n}(a_{n,t}|s_{n,t})}$ to prevent the current policy π_{θ_n} from getting far from the old policy $\pi_{\theta'_n}$. It updates the policy with a clipped loss function:

$$\mathbb{E}\left[\min\left(\text{clip}\left(p_{n,t}(\theta_n),\epsilon\right)\hat{A}_{n,t}, p_{n,t}(\theta_n)\hat{A}_{n,t}\right)\right], \tag{22}$$

where ϵ is a clip threshold, $\text{clip}(\cdot)$ ensures that $p_{n,t}(\theta_n)$ falls in the interval $[1-\epsilon, 1+\epsilon]$, and $\hat{A}_{n,t}$ is the generalized advantage estimation with advantage normalization.

Explicitly, we train *Demeter* with 2000 episodes, using a clipping threshold ϵ of 0.2, a penalty factor β of 0.3, and a function timeout Γ as 2 minutes. When an episode ends, agent n updates its policy π_{θ_n} using a set of trajectories $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r})$ collected from each "*wave*" in a batching manner. Here, $\boldsymbol{s}$, $\boldsymbol{a}$ and $\boldsymbol{r}$ denote the states, actions and rewards, respectively.

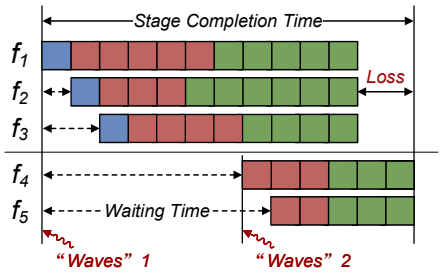


Fig. 8. The potential performance loss due to stage splitting.

V. Function Congestion Control

A. Practical Issues

In contrast to some single-task schedulers [25], [32], *Demeter* orchestrates batch functions in the form of "*wave*". Stable and moderate function requests are vital for *Demeter*, but this is not realistic in serverless analytics scenarios. The triggering of each stage is uncertain because it is jointly determined by all upstream dependencies. This burst-parallel nature [22] can induce varying degrees of function congestion, when arriving functions are out of the "*wave*". For example, either lots of stages or a few stages with lots of functions are triggered. In turn, there arise the following two practical issues:

(i) Potential performance loss due to stage splitting. Recall that if a stage cannot be accommodated by a "*wave*", then its overflowing functions will be split into one or more subsequent "*waves*". As shown in Fig. 8, functions f_4 and f_5 are congested into "*wave*" 2. This stage splitting may exacerbate the inherent waiting skew described in Section III-C1, which entails a potential performance loss. Despite efforts to eliminate cold-start by pre-heating and allocating saturated resources, functions f_4 and f_5 fail to catch up with others owing to prolonged waits. Hence, it is non-trivial for *Demeter* to remedy this loss within stages through fine-grained orchestration.

(ii) Asymmetrical arrival of jobs and functions. The DoP of stages as metadata for job profiles needs to be set before submission. In serverless scenarios with nearly unlimited resources, developers can only configure defensively according to previous experience (*e.g.*, input share [13]). Coupled with the short-lived limits, numerous functions are completed and invocated in each orchestration interval. Even with ideal job arrivals, there is still significant function stacking that leads to persistent congestion. Especially for jobs with a longer elapsed time, they have a lower *buffer time* before an SLO violation. Thus, *Demeter* may not even have an opportunity to salvage them via aggressive orchestration across stages.

B. DoP Tuning Algorithm

Benefiting from the fine-grained resource elasticity provided by serverless computing, we can enable elastic parallelism to determine the number of functions (*i.e.*, DoP) for any stage [15]. Unlike in serverful analytics, this process can be done at runtime due to the event-driven nature. Thus, our congestion control solution is based on the idea: at each orchestration, we tune the DoPs of triggered stages so that the sum falls within the "*wave*" size while maximizing performance. This synchronizes function arrivals with orchestration, thus eliminating the

potential function congestion and offering *Demeter* latitude to pursue the end-to-end goal with the fine-grained orchestration. Besides, a lower DoP can promote intra-function parallelism, facilitating to reduction of overhead in initializing containers and language runtime repeatedly [21].

Intuitively, the stages within a "*wave*" are inherently parallelizable, and free of data dependencies. If their job DAGs are connected into one large DAG via a dummy start node, these parallel stages should also be balanced to contribute to this large job, as described in Section II-B. Based on the existing stage execution time model $T(v_i, d_i) = \alpha_i/d_i + \gamma_i$ [18], we model the DoP tuning of k stages for a DC as:

$$\frac{\alpha_1}{d_1} + \gamma_1 = \frac{\alpha_2}{d_2} + \gamma_2 = \cdots = \frac{\alpha_k}{d_k} + \gamma_k, \tag{23}$$

$$\text{s.t.} \quad \sum_{i=1}^{k} d_i = \text{wave\_size}, \tag{24}$$

$$d_i \in \mathbb{Z}^+, \ i = 1, 2, \ldots, k, \tag{25}$$

where d_i is the DoP of stage v_i, and α_i and γ_i are the fittable parameters indicating the serialization time and inherent overhead, respectively. The term α_i/d_i is thus the parallelized time. Ignoring the slight effect of γ_i for now, we can obtain $d_i/d_j = \alpha_i/\alpha_j$ that guide the dividing of DoP within a limited "*wave*" size. We round down the computed d_i to the nearest integer and set it to one if $d_i < 1$. Note that, this ratio also proved to be an optimal solution for

$$\min_{(d_1,\ldots,d_k)} \max\left\{\frac{\alpha_1}{d_1} + \gamma_1, \ldots, \frac{\alpha_k}{d_k} + \gamma_k\right\}, \tag{26}$$

$$\text{s.t.} \quad (24), (25), \tag{27}$$

which can provide optimal balanced performance for all parallel stages under finite DoPs [18].

First, DoP tuning is conducted on each DC that experiences congestion, *i.e.*, when the functions in its waiting queue exceed the orchestration batch. If the number of triggered stages is greater than the "*wave*" size, late-arriving stages that cannot be accommodated are assigned to the next "*wave*". Recall that function requests are dispatched to all the potential DCs, so the tuning results need to be synchronized for global consistency. More concretely, each stage selects the lowest DoP as the final value to ensure (24) across all DCs. Afterward, *Demeter* repartitions the input data and calibrates function requests for the stages in which the DoP changes (see Section VI-B). Finally, the uncertain impact of heterogeneous function demands and data dependencies (*e.g.*, stages 1 and 3 in Fig. 2) will be further handled by the MARL model atop job DAGs updated partially with new DoPs. This is also the reason why we can omit the inherent overhead, even though it can be apportioned by higher (intra- and inter-function) parallelism for most stages.

VI. Implementation

We implement *Demeter* atop Pheromone [6], a data-centric serverless system that enables fine-grained data exchange, such as all-to-all shuffle. Concretely, intermediate data between two consecutive stages is placed into the *zero-copy* shared memory in the form of a bucket abstraction. The data in each bucket is divided based on their metadata (*e.g.*, specified key), with each group being consumed by an associated function. We re-architect it on a Kubernetes [28] cluster and develop several tailor-made components to drive *Demeter*.

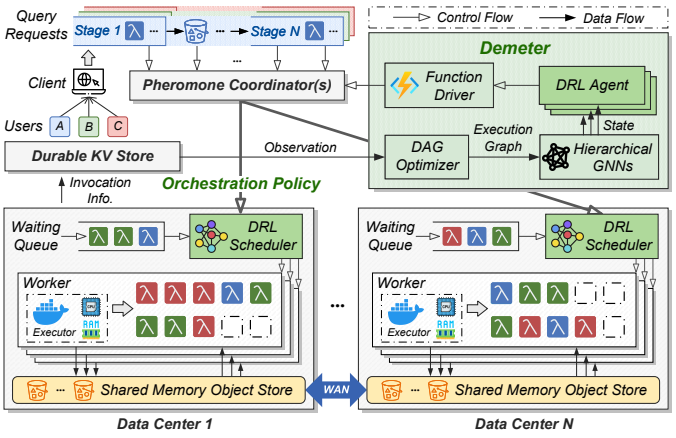


Fig. 9. The architecture of *Demeter*.

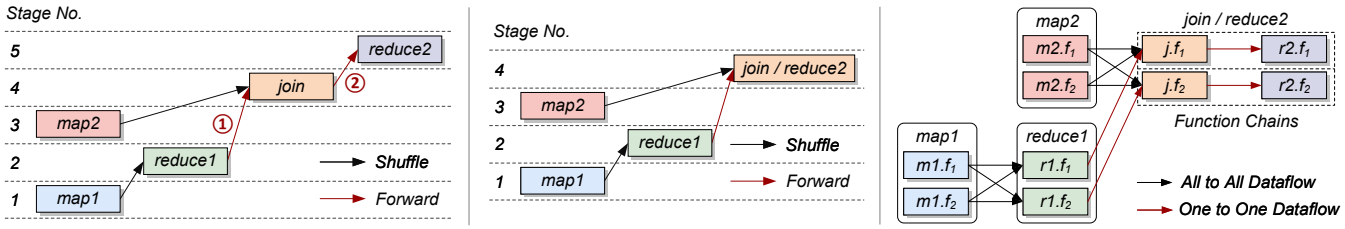
A. Overview

Fig. 9 displays an architecture overview of *Demeter*. Given the design of centralized state representation and shared reward in *Demeter*, we do not disperse each agent to its corresponding DC, thus reducing the communication overhead between the global coordinator and DRL agents. The desired information of *Demeter* is piggybacked on the bucket status synchronization during inter-node scheduling. Besides, we also deploy multiple coordinators in a shared fashion, each integrating a replica of *Demeter* and managing a batch of disjoint jobs, so as to further improve the system scalability.

The running process of *Demeter* is shown. First, the query requests are created by the client based on user codes. The coordinator then admits and parses them into a sequence of event-driven stages. It routes the parallel function invocations to all potential DCs asynchronously, when a stage is triggered. Before batch orchestration, each DC draws up the DoP tuning plan via the coordinator. The plan is delivered by the bucket partitioner. Then, the DAG optimizer inputs the observations (*e.g.*, necessary invocation history) from the durable key-value store, generating or updating the `ExecutionGraph` for the DAG-driven state representation. Next, *Demeter* makes per-function orchestration decisions to notify each DC. Owing to the data-centric design of Pheromone, fine-grained knowledge of data-to-function dependencies (*e.g.*, the pending data size) can be exposed via system APIs. Finally, the codes encapsulated by the function driver execute by forcing the allocated resource for executors.

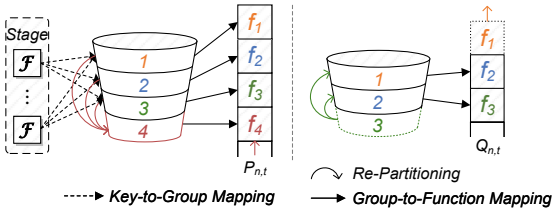
B. Components

DAG optimizer. The DAG optimizer identifies stage types based on the trigger primitives (*e.g.*, `DynamicJoin`→map, `DynamicGroup`→reduce) in job profile [6]. It first constructs the initial `LogicGraph` (Fig. 10(a)), where nodes are stages and edges mean the dependency modes. For successive stages with the same partition layout (*i.e.*, co-partitioning [38]), intermediate data between them flows in a one-to-one mapping,



(a) LogicGraph. (b) JobGraph. (c) ExecutionGraph.

Fig. 10. An overview of the job DAG optimization process. (a) The job DAG consists of 5 stages, where stages with a repeated trigger primitive are still recognized as being of different types (*e.g.*, map1 and map2). (b) The stages reduce1 and join connected by *forward* ① cannot be chained, since the child stage join does not have an in-degree of 1. (c) The loosely-coupled nature of the functions is not altered by the two chains in the all-new stage join / reduce2, each chained function still holds a separate node in the ExecutionGraph.



(a) Scale-out (3→4). (b) Scale-in (3→2).

Fig. 11. Bucket repartitioning with JumpHash.

meaning one downstream function exclusively reads data from an upstream counterpart. This unusual shuffle is replaced as *forward*. At the same time, the optimizer chains them as an all-new stage to further generate the `JobGraph` (Fig. 10(b)). Specifically, upon a parent function finishes, its container is repurposed by `fork()` new processes of the child, avoiding cold start and transmission delays. By such means, a *function chain* is formed as a placement granularity, but its component functions remain separate in execution. Finally, the optimizer splits each stage node into parallel function nodes, and decides inter-function links to generate the `ExecutionGraph` (*i.e.*, function-level DAG, see Fig. 10(c)).

Function driver. Given function configurations with decoupled CPU and memory, it is essential to take full advantage of multi-core performance. To this end, we build the function driver to enable multi-process support. Concretely, the driver first gets the number of CPU cores p_c allocated for a function. When the input data is retrieved from the external storage or memory, it evenly divides the data into p_c sets and `fork()` the corresponding number of processes. Each process handles a batch of data by calling the function code. For Inter-Process Communication (IPC), we reuse the relevant bucket designed for inter-function IPC in Pheromone. This intra-function IPC approach incurs no additional memory, while eliminating the overhead in aggregating multi-process output.

Bucket partitioner. Data grouping in a bucket is intricately tied to the DoP of the downstream stage and arranged prior to triggering. Hash partitioning, commonly used in traditional data analytics systems [9], [14], becomes impractical for the elastic parallelism. Whenever the downstream DoP alters, the entire data bucket must undergo a complete repartitioning. To obviate this, *Demeter* partitions buckets based on JumpHash [39], a fast *consistent hashing* algorithm that ensures balanced key-to-group mappings while minimizing perturbation. Thanks to its monotonicity, only partial data is required to be migrated into the new groups (Fig. 11(a)) or out of unavailable groups (Fig. 11(b)). However, JumpHash only permits groups to be regulated at the end of buckets. Given this, new functions can be triggered and appended to $P_{n,t}$ directly. For invalid functions, *Demeter* skips the corresponding number of elements in $Q_{n,t}$ when extracting "*waves*".

VII. Experimental Evaluation

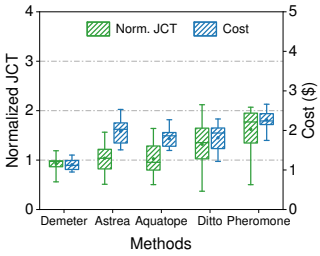
This section evaluates the performance of *Demeter* against the baselines, and the effectiveness of its components.

A. Experimental Settings

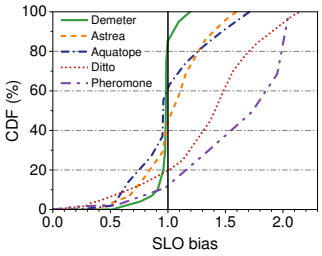
Model configurations. We train the MARL using PyTorch. The model depth L of both GNNs is set to 3. For HTGNN, we set the time window size as 5 and accelerate it using a memory module [40] to match the parallelized DAGNN. In the pointer network, its Transformer encoder is configured with 3 layers and 8 attention heads. The score network has two fully connected hidden layers with 32 and 16 neurons, using Tanh as the activation function. Note that critics share the same neural and input structure as the score network, and they are updated with the corresponding average *score*. Both actors and critics use Adam optimizer with a learning rate of 3×10^{-4}.

Setup. We deploy *Demeter* in an AWS EC2 cluster which consists of 28 instances across 8 regions (Paris, Ohio, Oregon, London, Virginia, Singapore, Sydney, and Tokyo), and the DC in one of these regions has 3 worker instances. In addition, we deploy 4 shared coordinators following the ratio in [6]. The coordinators and workers are run at c5.4×large and c5.9×large instances, respectively. Each function can be configured with 8 cores at most and memory limits from 128 MB to 10 240 MB with 64 MB increments.

Workloads. We generate serverless job sets using analytics benchmark suites: TPC-DS [41] and BigData [42]. By default, the stages invoke functions based on the size of intermediate data [13]. Each function is written in Python and calls system interfaces using `ctypes` [43]. For the datasets, we put relatively small tables in an S3 bucket in one of the regions and evenly split large tables into multiple regions. Moreover, we run the jobs with varying input sizes by tuning the *scaling factor* in suites. To expedite *Demeter*'s training by reducing function duration while ensuring the continuous availability of resources, the adjustments are made at a lower-level.

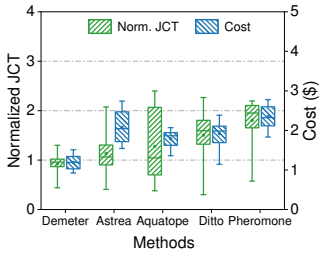


(a) Normalized JCT and cost.

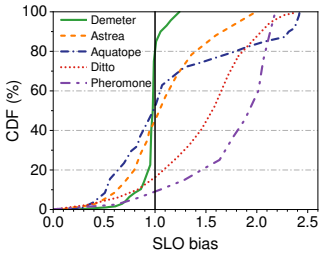


(b) CDF of SLO bias.

Fig. 12. The overall performance under the 4-agent cluster setting.



(a) Normalized JCT and cost.



(b) CDF of SLO bias.

Fig. 13. The overall performance under the 8-agent cluster setting.

Job arrivals. Our MARL model is trained with job arrival rates within 24 hours following the Facebook Hadoop Workload Trace [44]. We select three modes: `slow` (50 jobs arriving in one hour), `normal` (80 jobs arriving in 30 min), and `burst` (100 jobs arriving in only 15 min).

Baselines. We compare *Demeter* with the following baselines. Before that, we introduce Fair [11], a multi-job scheduling algorithm across geo-distributed DCs, to our FaaS setup and rewrite its objective function. To this end, we use the parametric model in [45] to provide rough performance estimates.

- **Aquatope** [16] is a general serverless Resource Manager (RM) that searches for suitable allocation for warm-start stages to satisfy the workflow's SLO, via a uncertainty-aware Bayesian optimization method. Note that we integrate Fair as its function scheduler.
- **Astrea** [17] is a serverless analytics RM that configures resources for stages with end-to-end limits. It employs a shortest-path algorithm over a DAG composed of configurable parameters. We extract the sub-algorithm aligned with our goal and add Fair as its scheduler.
- **Ditto** [18] is a state-of-the-art serverless analytics scheduler that configures elastic parallelism for each stage with greedy placement jointly, thereby reducing data shuffling overhead to optimize for JCT and cost.
- **Pheromone** [6] natively offers a data locality-aware function scheduling. Also, it greedily allocates the minimum but valid amount of resources to functions.

Metrics. We focus on two metrics: normalized JCT (*a.k.a*, SLO bias) and cost to evaluate the performance of *Demeter*. For a given job, the former is measured JCT / SLO, which is higher for more critical SLO violations. The JCT is defined as the duration from the job's arrival time to the completion of its last function. Regarding the latter, we measure the aggregated monetary cost across all component functions in the job.

B. Performance of Demeter

We compare the overall performance of *Demeter* in cost and SLO compliance against the baselines. Each function requires at least 1 CPU core and the lowest memory satisfying (13). The loss factor α is set to 0.8. The resource prices for each region, as discussed in Section III-C2, follow the AWS Lambda [19] and EC2 [27] pricing models. Note that the CPU allocation on Lambda adheres to a ratio (*i.e.*, 1.728 GB/core) of the memory limit. We thus set the per-core price as 1.728× that of per-GB memory in the same region. Besides, the orchestration interval (timeout) is set to 5 seconds, with a batch size limit of 20.

1) Overall Performance: Fig. 12(a) and 13(a) show the results in 4- and 8-agent (DC) clusters with 2 and 4 coordinators, respectively. The 4-agent with `normal` mode tests 64 jobs containing more than 3000 functions, while the 8-agent with `burst` mode tests 128 jobs covering over 5000 functions. As shown, only *Demeter* consistently stabilizes the JCT around the corresponding SLO and performs well in cost-saving under both cluster settings, due to its unique fine-grained function orchestration. Specifically, it decreases average cost by up to 45.3% and 46.6%, respectively, compared with the baselines. Part of the gap is due to the higher non-execution costs caused by numerous functions being stranded, especially with the 8-agent cluster. Note that the stage-granular scheduling in Ditto struggles to effectively reduce data transmission overhead and cost over WAN, leading to lower performance than the baselines with Fair (*i.e.*, Astrea and Aquatope). Aquatope considers uncertainty in the system, similar to ours, and thus outperforms other baselines. Nevertheless, it confines configuration search to warm-start functions, which makes it non-trivial to handle unexpected cold-start from bursty stages, resulting in inferior SLO compliance under the 8-agent cluster.

Fig. 12(b) and 13(b) show the distribution of SLO bias accordingly. *Demeter* outperforms all baselines, ensuring better SLO compliance with relatively smaller performance fluctuations. It can eliminate another 23.7% and 36.7% of SLO violations compared with the second-best Aquatope, respectively, bringing the total down to below 15%. In contrast to Astrea, Aquatope's low cost comes with the risk of a high violation magnitude, despite a relatively low SLO violation rate. This is due to its active allocation search without explicit penalties, which tends to prioritize minimal cost when jobs are deemed unlikely to finish on time. Ditto does not account for resource sharing across jobs, leading to resource under-utilization. This is because different stages could not overlap in time, leaving resource idle during non-overlapping periods. Moreover, Ditto has to select higher parallelism to mitigate straggler functions within stages, which further exacerbates resource wastage and significant SLO violation in frequency and magnitude.

2) Resource Efficiency: Given the correlation between JCT and cost, we further analyze whether *Demeter* actually saves cost by improving resource efficiency. Fig. 14 shows the mean memory usage per stage under the 8-agent cluster. Note that we include the resource overhead of the solutions themselves in the measurement. As shown, Astrea requires memory over-provisioning to expedite the compute-intensive data analytics functions. This is because it allocates CPU cores with a fixed

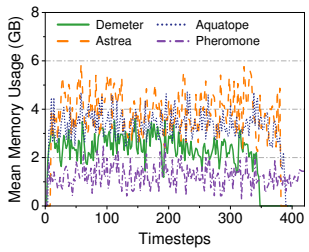


Fig. 14. The long-term mean memory usage.

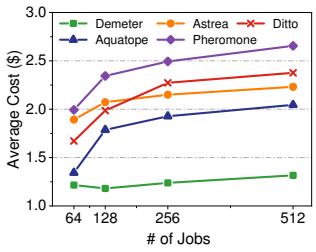


Fig. 15. The mean cost under varying numbers of jobs.

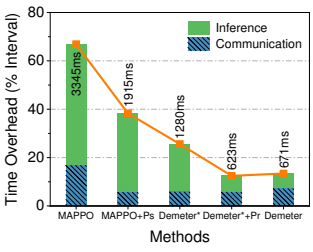


Fig. 16. The orchestration overhead under various algorithm settings.

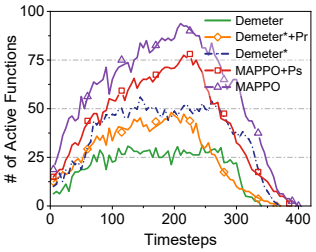
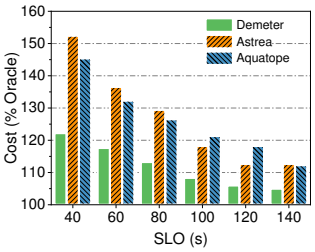
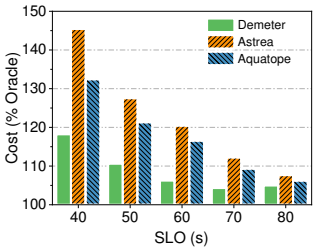


Fig. 17. The function stacking under various algorithm settings.

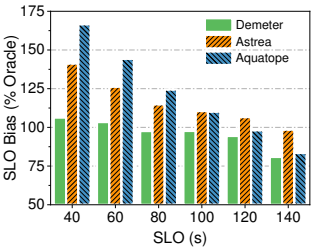


(a) TPC-DS Q94.

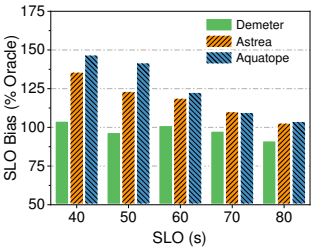


(b) BigData Q3.

Fig. 18. The cost guarantee for two typical workloads.



(a) TPC-DS Q94.



(b) BigData Q3.

Fig. 19. The SLO guarantee for two typical workloads.

proportion of the memory capacity. In contrast to Aquatope, which uses a similar decoupled configuration, *Demeter* strikes a better balance by scaling this approach to each function. It neither stacks large redundant resources to satisfy SLOs, nor opts for cheap configurations to cut costs. Last but not least, Pheromone significantly increases the resource occupancy time of jobs, without our low-overhead MARL orchestrator. This in turn leads to high costs, despite its minimal allocation.

3) Scalability: To evaluate the scalability of *Demeter*, we change the number and mode of job arrivals under the 8-agent cluster. The results are shown in Fig. 15 and 20, respectively. Note that Aquatope and Ditto perform better with the `slow` mode and small-scale jobs. For Aquatope, its pre-warm container pool has more margin to improve the hit rate of warm-starts. For Ditto, more stages from different jobs may run in overlap to mitigate resource idleness. With the bursty-parallel nature of jobs, however, Ditto is more likely to delay partial stages due to resource starvation, resulting in increased non-execution costs. This is because the free-standing parallelism configuration gradually shrinks their available resource spaces in theory, as the job scale increases. Moreover, static Astrea is non-sensitive to the environment and its instability mainly arises from the inherent noise. In contrast, *Demeter* exhibits no notable performance degradation with different job scales. It also reduces the average cost and SLO violation by over 23.7% and 10.6%, respectively, under various job arrival modes. Our performance benefits from the efficiency of our MARL model trained with sufficient workloads.

C. Performance Guarantees

We evaluate the effectiveness of *Demeter* against the SLO-aware baselines in providing performance guarantees. We track the trajectories of two typical workloads (TPC-DS Q94 and BigData Q3) on the 8-agent cluster. Note that the former has more stages and pronounced changes in intermediate data.

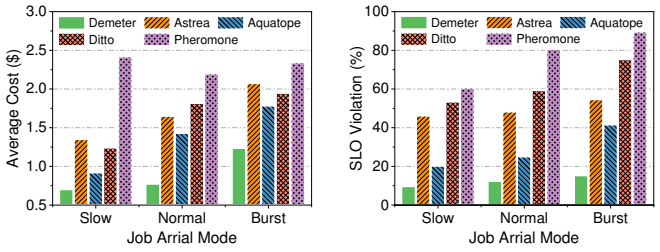
1) Cost Guarantee: We vary the SLO constraint to evaluate the cost guarantee of *Demeter*. As shown in Fig. 18, *Demeter* exhibits superior cost-effectiveness across both jobs, achieving savings ranging from 20.8% to 62.7% compared to the second-best Aquatope. When the SLO is relatively low, the cost gap between *Demeter* and the baselines is larger in Q94 than in Q3, due to the wide variety of function demands. Astrea can break up large jobs into multi-round executions, resulting in slightly lower performance degradation than Aquatope. These results highlight *Demeter*'s efficiency in handling jobs with complex topologies and SLO constraints. When the SLO is relatively high, all algorithms perform well. Overall, *Demeter* maintains stabilized performance and achieves costs within 10.1% of the optimal Oracle, which exhaustively explores the configuration space without any performance fluctuations.

2) SLO Guarantee: We also evaluate the SLO guarantee of *Demeter*, as depicted in Fig. 19. Similar to the cost, *Demeter* is capable of guaranteeing SLO for both jobs. When we relax the constraint, the SLO bias remains relatively stable, denoting that *Demeter* can adapt to varying requirements. However, the baselines struggle to accelerate jobs to satisfy stringent SLOs (*e.g.*, $<$ 80 seconds for Q94), even with a high cost. When the SLO is relatively loose, the actual bias decreases abruptly. This is because the optimization goal has shifted to cost. Shrinking costs does not further increase JCT.

D. Deep Dive of Demeter

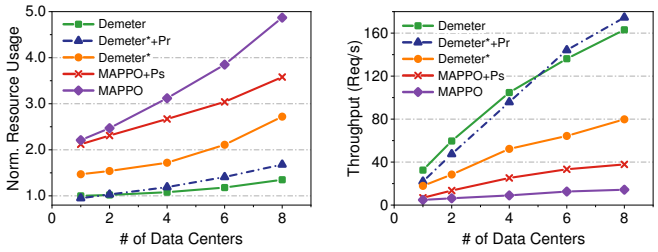
We evaluate the effectiveness of *Demeter* components with the following methods with different incremental techniques:

- **MAPPO** [36]: A vanilla MAPPO algorithm that orchestrates pending functions in an FCFS manner.
- **MAPPO+Ps**: The MAPPO with pointer-score network.
- ***Demeter**** [46]: The MAPPO+Ps with hierarchical GNNs, *i.e.*, the preliminary version of our work.
- ***Demeter***+Pr*: The *Demeter** with pruning mechanism.



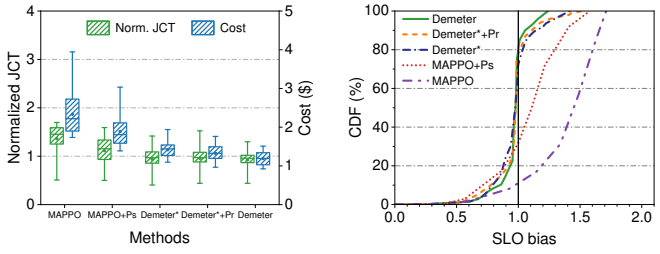
(a) Average cost. (b) SLO violation.

Fig. 20. The average cost and SLO violation with varying arrival modes.



(a) Resource usage. (b) Throughput.

Fig. 22. The resource usage and throughput with varying numbers of DCs.



(a) Normalized JCT and cost. (b) CDF of SLO bias.

Fig. 21. The overall performance among various algorithm settings.

1) Ablation Evaluation: As shown in Fig. 21, MAPPO+Ps outperforms due to the multi-negotiated function placement by exploring their correlation. However, it inputs stacked features of pending functions and DCs to each agent, thus struggling to make efficient policies on this redundant state space. With the pointer-score network and hierarchical GNNs, *Demeter** reduces the mean cost by 40.1% and SLO violation by 63.5% compared to MAPPO. *Demeter**+Pr drops the sampling probability of un-suitable allocations via the pruning mechanism. Further, *Demeter* alleviates unnecessary waiting costs due to function congestion while also sparing some jobs potentially in SLO violation. It outperforms *Demeter** by 14.6% in average cost and 9.4% in SLO violation.

2) Orchestration Overhead: Low-latency is vital for *Demeter* to be applied in real-world systems. Fig. 16 measures the orchestration time in inference and communication. *Demeter* is significantly accelerated with the compact state and decoupled action spaces. The inference time is further reduced by pruning the score model to narrow down the configuration pool. While DoP tuning requires synchronization across agents, it aims to hold functions within a "*wave*", and does not bring notable communication pressure. In general, the orchestration time is quite tolerable relative to the interval. Coupled with the sub-second absolute time (671 ms), this further showcases the fast orchestration potential of *Demeter* in wide-area settings.

Resource overhead is another key factor. Fig. 22(a) shows the normalized resource usages. By applying HTGNN on the deployment graph, function information is aggregated across DCs along the edges. This not only decreases state processing overhead by compressing space, but also enhances scalability via GNN. Also, the pointer-score network decouples the joint actions to prevent merging into a huge decision space. Coupled with the pruning, it effectively reduces the compute complexity and overhead of agents. Thus, the resource demand of *Demeter* does not increase significantly as the number of DCs grows.

3) Throughput: Demeter handles the function requests that scale with the number of DCs due to its distributed orchestration. However, it effectively avoids significant communication and scheduling bottlenecks, due to low orchestration overhead. As depicted in Fig. 22(b), *Demeter* thus can achieve sublinear scalability in terms of throughput.

4) Function Stacking: As Fig. 17 shown, the job arrivals are accompanied by the delayed triggering of numerous functions. With the DoP tuning, *Demeter* relieves function congestion by orchestrating triggered stages in the current "*wave*". Moreover, other techniques can also alleviate function stacking to varying degrees by enabling early execution.

VIII. Related Work

Serverless systems. Serverless computing attractes stateful and parallel offerings, due to fine-grained elasticity and billing. Many systems [6], [20], [22], [47] with distinct architectures are devised to fill the gap between stateless functions and stateful computing. To enhance data locality, SAND [47] places all functions in an application-level sandbox, and Wukong [22] partitions its DAG by execution path, assigning each path to a container. Nevertheless, this *function fusion* [21] expands the allocation granularity and blocks on-demand scalability across stages. In contrast, *Demeter* merges only paths with the same partition layout at the function granularity, without influencing the fine-grained orchestration. In addition to mapreduce-based analytics jobs, *Demeter* can be extended to general serverless workflows as they also have DAG statements with dependencies and event-driven nature [15], [48].

(Geo-distributed) serverless analytics. Geo-distributed analytics in traditional systems [14] optimize JCT or WAN costs by task scheduling [11], [12] or data placement [10], [49]. Due to the limitations of serverful architecture, they cannot manage task granularity like *Demeter*. In the serverless context, while some studies [50], [51] deploy functions across regions, they are not specific to complex analytics jobs. More research on serverless analytics in a single region [13], [7]. For instance, Astrea [17] finds the optimal allocation for an analytics job with budget and SLO demands based on the holistic modeling and graph theory. Kassing *et al.* [45] consider factors unique to serverless analytics, as we do, to make the trade-off between JCT and cost with Pareto optimal. Ditto [18] determines the optimal parallelism distribution across stages, and places stage groups to minimize shuffle traffic. However, they more or less ignore varying function performance, and do not orchestrate at the function granularity. Other works [4], [48], [52] deploy

functions at the network edge, and take network variations and resource limits into account. These works are not yet suitable for large-scale query services across DCs.

SLO-aware resource allocation. StepConf [15] configures function steps (*i.e.*, stage) dynamically based on the critical path in the job DAG. Aquatope [16] also considers uncertainty to optimize cold-start and resource allocation for stages using Bayesian models. While they aim to achieve end-to-end goals similar to ours, they do not consider the performance impact of function placements. WiseFuse [21] bundles parallel functions within stages and enables in-container CPU sharing to mitigate their execution time skew. However, it is only suitable for co-located functions. Since Mao *et al.* [35], [53] integrated Deep Reinforcement Learning (DRL) for multi-resource scheduling in data processing clusters, *DRL for systems* has attracted a surge of attention. Yu *et al.* [25] employ DRL algorithms to harvest idle resources and alleviate function slowdown in FaaS systems. Qiu *et al.* [26] propose MARL with SLO violation and resource utilization as part of the reward in multi-tenant platforms. Our solution is different from [25] and [26], since *Demeter* focuses on orchestrating functions with dependencies in analytics jobs across DCs, rather than a series of relatively independent functions in a single-region cluster.

IX. Conclusion

This paper proposes *Demeter*, a fine-grained function orchestrator that jointly decides the per-function placement and resource allocation for geo-distributed serverless analytics. The goal is to enhance cost savings while ensuring the job's end-to-end SLO. Specifically, it utilizes a MARL algorithm with scalable state representation and decoupled action encoding, along with a pruning mechanism. In addition, we tackle the potential function congestion by enabling elastic parallelism. Extensive experiment results validate that *Demeter* can effectively exploit wide-area resource elasticity. It reduces by up to 46.6% on cost and over 23.7% on SLO violation, outperforming the state-of-the-art solutions.

References

[1] 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.

[2] J. Zhou, J. Fan, J. Jia, B. Cheng, and Z. Liu, "Optimizing cost for geo-distributed storage systems in online social networks," *J. Comput. Sci.*, vol. 26, pp. 363–374, 2018.

[3] Z. Shen *et al.*, "Follow the sun through the clouds: Application migration for geographically shifting workloads," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 141–154.

[4] Z. Xu, Y. Fu, Q. Xia, and H. Li, "Enabling age-aware big data analytics in serverless edge clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

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

[6] 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.

[7] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable analytics on serverless infrastructure." in *Proc. USENIX Symp. Netw. Syst. Des. Implement.*, 2019, pp. 193–206.

[8] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 445–451.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 421–434, 2015.

[11] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 3, pp. 488–500, 2018.

[12] K. Oh, M. Zhang, A. Chandra, and J. Weissman, "Network cost-aware geo-distributed data analytics system," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 6, pp. 1407–1420, 2021.

[13] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics," in *Proc. USENIX Symp. Netw. Syst. Des. Implement.*, 2021, pp. 653–669.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Workshop Hot Top. Cloud Comput.*, 2010, pp. 1–7.

[15] 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.

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

[17] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Astrea: Auto-serverless analytics towards cost-efficiency and QoS-awareness," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 12, pp. 3833–3849, 2022.

[18] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf.*, 2023, pp. 406–419.

[19] AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/.

[20] Z. Li *et al.*, "FaaSFlow: Enable efficient workflow execution for function-as-a-service," in *Proc. ACM Int. Conf. Archit. Support Program Lang. Oper. Syst.*, 2022, pp. 782–796.

[21] 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.

[22] B. Carver *et al.*, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 1–15.

[23] 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.

[24] 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.

[25] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, "Accelerating serverless computing by harvesting idle resources," in *Proc. ACM Web Conf.*, 2022, pp. 1741–1751.

[26] H. Qiu *et al.*, "Reinforcement learning for resource management in multi-tenant serverless platforms," in *Proc. Workshop Mach. Learn. Syst.*, 2022, pp. 20–28.

[27] EC2 Pricing. [Online]. Available: https://aws.amazon.com/ec2/pricing.

[28] Kubernetes. [Online]. Available: http://kubernetes.io.

[29] V. Thost and J. Chen, "Directed acyclic graph neural networks," *arXiv preprint arXiv:2101.07965*, 2021.

[30] Y. Fan *et al.*, "Heterogeneous temporal graph neural network," in *Proc. SIAM Int. Conf. Data Min.*, 2022, pp. 657–665.

[31] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Adv. neural inf. proces. syst.*, vol. 28, pp. 1–9, 2015.

[32] Y. Li *et al.*, "Task placement and resource allocation for edge machine learning: A gnn-based multi-agent reinforcement learning paradigm," *arXiv preprint arXiv:2302.00571*, 2023.

[33] A. Sahraei *et al.*, "XFaaS: Hyperscale and low cost serverless functions at meta," in *Proc. ACM Symp. Oper. Syst. Principles*, 2023, pp. 231–246.

[34] 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.

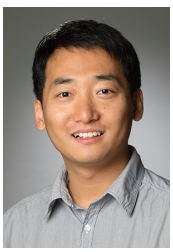
[35] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. ACM HotNets*, 2016, pp. 50–56.

[36] C. Yu, A. Velu, E. Vinitsky, J. Gao, Y. Wang, A. Bayen, and Y. Wu, "The surprising effectiveness of ppo in cooperative multi-agent games," *Adv. neural inf. proces. syst.*, vol. 35, pp. 24 611–24 624, 2022.

[37] C. S. de Witt *et al.*, "Is independent learning all you need in the starcraft multi-agent challenge?" *arXiv preprint arXiv:2011.09533*, 2020.

[38] W. Rödiger *et al.*, "Locality-sensitive operators for parallel main-memory database clusters," in *Proc IEEE Int. Conf. Data Eng.*, 2014, pp. 592–603.
[39] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," *arXiv preprint arXiv:1406.2294*, 2014.
[40] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *arXiv preprint arXiv:2006.10637*, 2020.
[41] TPC-DS Benchmark. [Online]. Available: http://www.tpc.org/tpcds/.
[42] Big Data Benchmark. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/.
[43] Ctypes. [Online]. Available: https://docs.python.org/3/library/ctypes.
[44] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Proc. IEEE Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2011, pp. 390–399.
[45] S. Kassing, I. Müller, and G. Alonso, "Resource allocation in serverless query processing," *arXiv preprint arXiv:2208.09519*, 2022.
[46] 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.
[47] I. E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 923–935.
[48] Z. Xu *et al.*, "Stateful serverless application placement in MEC with function and state dependencies," *IEEE Trans. Computers*, vol. 72, no. 9, pp. 2701–2716, 2023.
[49] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, and J. Zhou, "Yugong: Geo-distributed data and job placement at scale," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2155–2169, 2019.
[50] F. Rossi, S. Falvo, and V. Cardellini, "GOFS: Geo-distributed scheduling in openfaas," in *Proc. IEEE Symp. Comput. Commun.*, 2021, pp. 1–6.
[51] G. Zheng and Y. Peng, "GlobalFlow: A cross-region orchestration service for serverless computing services," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2019, pp. 508–510.
[52] X. Shang, Y. Mao, Y. Liu, Y. Huang, Z. Liu, and Y. Yang, "Online container scheduling for data-intensive applications in serverless edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.
[53] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM SIGCOMM Conf.*, 2019, pp. 270–288.

Xiaofei Yue (Graduate Student Member, IEEE) received the M.E. degree from Northeastern University, Shenyang, China, in 2022. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. His current research interests include distributed systems, cloud/serverless computing, and data analytics.

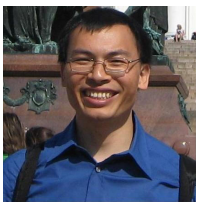


Song Yang (Senior Member, IEEE) received the BS degree in software engineering and MS degree in computer science from the Dalian University of Technology, Dalian, Liaoning, China, in 2008 and 2010, respectively, and the PhD degree from the Delft University of Technology, The Netherlands, in 2015. From 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.

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, China. 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.

Stojan Trajanovski (Member, IEEE) received the master's (with distinction) degree in advanced computer science from the University of Cambridge, Cambridge, U.K., in 2011, and the PhD (cum laude) degree from the Delft University of Technology, Delft, The Netherlands, in 2014. He is currently a senior applied scientist with Microsoft in London, U.K. He was in a similar role with Philips Research in Eindhoven, The Netherlands from 2016 to 2019. Before that, he spent some time as a postdoctoral researcher with the University of Amsterdam and with the Delft University of Technology. He has published more than 50 articles, including papers at top-tier conferences and journals such as NeurIPS, INFOCOM, NAACL, IEEE/ACM Trans. on Networking, IEEE Trans. on Mobile Computing, and Physical Review. He successfully participated at the International Mathematical Olympiad (IMO), winning a bronze medal in 2003. His main research interests include machine learning, LLMs, network science, game theory, and optimization algorithms.

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.



Xiaoming Fu (Fellow, IEEE) received the PhD degree in computer science from Tsinghua University, Beijing, China, in 2000. He was then a research staff with the Technical University Berlin until joining the University of Göttingen, Germany in 2002, where he has been a professor in computer science and heading the Computer Networks Group since 2007. He has spent research visits at universities of Cambridge, Uppsala, UPMC, Columbia, UCLA, Tsinghua, Nanjing, Fudan, and PolyU of Hong Kong. His research interests include network architectures, protocols, and applications. He is currently an editorial board member of the IEEE Network and IEEE Transactions on Network and Service Management, and has served on the organization or program committees of leading conferences such as INFOCOM, ICNP, ICDCS, SIGCOMM, MOBICOM, MOBIHOC, CoNEXT, ICN and Networking. He is an ACM Distinguished Member, a fellow of IET, and member of the Academia Europaea.