Demeter: Fine-grained Function Orchestration for Geo-distributed Serverless Analytics

Xiaofei Yue[§], Song Yang[§], Liehuang Zhu[§], Stojan Trajanovski[†], Xiaoming Fu[‡]
[§] Beijing Institute of Technology, China
[†] Microsoft, London, United Kingdom
[‡] University of Göttingen, Germany
Email:{xfyue, S.Yang, liehuangz}@bit.edu.cn, sttrajan@microsoft.com, fu@cs.uni-goettingen.de

Abstract—**In the era of global services, low-latency analytics on large-volume geo-distributed data has been a regular demand for application decision-making. Serverless computing facilitates fast function start-up and deployment, making it an attractive way for geo-distributed analytics. We argue that the serverless paradigm holds the potential to breach current performance bottlenecks via *fine-grained function orchestration*. However, how to configure it for geo-distributed analytics remains ambiguous. To fill this gap, we present Demeter, a scalable fine-grained function orchestrator for geo-distributed serverless analytics systems. Demeter aims to minimize the composite cost of co-existing jobs while meeting the user-specific Service Level Objectives (SLO). To handle the volatile environments and learn the diverse function demands, a Multi-Agent Reinforcement Learning (MARL) solution is used to co-optimize the per-function placement and resource allocation. The MARL extracts holistic and compact states via hierarchical graph neural networks, and then designs a novel actor network to shrink the huge decision space and model complexity. Finally, we implement Demeter and evaluate it using realistic workloads. The experimental results reveal that Demeter significantly saves costs by 23.3%$\sim$32.7%, while reducing SLO violations by over 27.4%, surpassing state-of-the-art solutions.**

I. Introduction

Many global services, such as video streaming [1] and social network [2], continuously produce substantial volumes of data, including user 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], [4]. Researchers have sought efficient ways to analyze these geo-distributed data, pursuing the expected Service Level Objectives (SLO) in terms of response time. Otherwise, the results could be outdated [5]. Recently, serverless computing in the form of Function-as-a-Service (FaaS) has flourished in analytics offerings, due to its promise of fine-grained elasticity and billing [6]. Many serverless systems [7], [8] with distinct architectures are designed to provide traditional analytics services [9] in a centralized DC. They free developers from fussy infrastructure management. Nevertheless, it is still challenging to leverage the benefits of FaaS to provide similar performance across DCs, *i.e.*, Geo-distributed Serverless Analytics (GSA). Meanwhile, most of the existing studies [4], [10], [11] cannot be directly applied to the serverless environment, due to a lack of adaptation to its uniqueness.

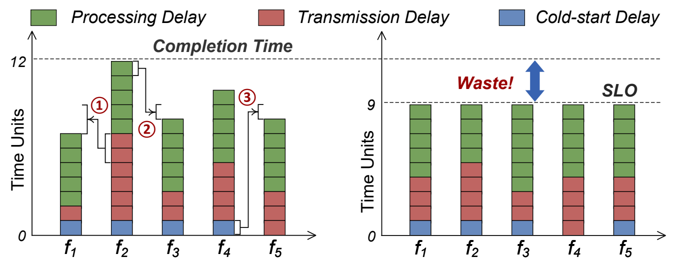
Serverless analytics systems decouple a monolithic job into stages, each consisting of stateless functions (*i.e.*, tasks) that execute in parallel [6]. Due to the extensive communication between stages (*a.k.a.*, *shuffle*) via a scarce WAN, carefully placing each function over geo-distributed DCs is necessary. Given the lightweight and rapid start-up nature of functions, the preference is to move them to where the data is located for local processing, as in prior works [4], [10]. This is because migrating geo-distributed data into a single DC for centralized processing has been generally limited by privacy regulations. Notably, the heterogeneous WAN brings potential variations in resource demands for each function. In traditional server-centric architectures, all the tasks stick to slots that are *evenly* divided from a fixed resource pool [12], which easily suffers from resource under- or over-provisioning. In contrast, serverless computing allows *elastic resource scaling* [13], [14], [15] to match the demands of stages that generally have varying input sizes in a job [6]. This coarse-grained way, however, is still inflexible and misses the huge potential of function-level on-demand provisioning, leading to wasted costs, especially in *wide-area settings* [4]. In fact, each function runs individually in an isolated container, loosely-coupled in a job. They can be allocated suitable resources for their unique demands.

To ensure the GSA job's end-to-end SLO while saving its cost, we advocate enabling *fine-grained function orchestration*, which consists of careful per-function placement and multi-resource allocation (*e.g.*, CPU and memory). To achieve this goal, we tackle three basic challenges. Firstly, both resource allocation and strategic placement for each function are critical to the Job Completion Time (JCT) and cost [16]. However, it is unclear how to handle such complexity of *co-optimization* and inter-function data dependencies. Secondly, the cost and performance of functions are inherently relevant to the allocated resources. However, the function cost is further affected by its duration [17], which will be amplified in the geo-distributed analytics due to more complex cost components (*e.g.*, WAN cost). Thirdly, considering such detailed orchestration for each parallel function from widespread co-existing jobs will result in an exponential growth of the decision space.

In this paper, we propose **Demeter**, a scalable fine-grained function orchestrator in the geo-distributed serverless analytics

Song Yang is the corresponding author.

This work has been partially supported by the Beijing Natural Science Foundation (No. 4232033), the National Natural Science Foundation of China (No. 62172038 and No. 62232002), the Horizon Europe CODECO Project (No. 101092696), and the Horizon Europe COVER Project (No. 101086228).



(a) Stage granularity. (b) Function granularity.

Fig. 1. An illustrative example of our fine-grained function orchestration. The stage has five parallel functions, and the green bars are equal (*i.e.*, 5 units) on the left and not identical on the right.

system. We first clarify the manner and goal of our fine-grained orchestration. To learn from the actual working pattern of the system, Demeter uses a Multi-Agent Reinforcement Learning (MARL) algorithm, which makes per-function placement and resource allocation decisions. Concretely, it extracts compact cross-DC states from multi-level features using hierarchical Graph Neural Networks (GNN). Also, we decouple the joint actions to reduce the model complexity and action space. The separated actions are encoded in sequence by different NNs, to collaborate on the orchestration policy. Finally, we develop several tailor-made components to implement Demeter based on Pheromone [7], an open-source serverless platform.

To sum up, this paper makes the following contributions:

- We formulate the first fine-grained function orchestration problem, whose goal is to minimize the composite cost of co-existing geo-distributed serverless analytics jobs while ensuring their respective SLOs.
- We propose Demeter, which designs a MARL algorithm with customized state representation and action encoding mechanism that jointly makes the placement and resource allocation decisions at function granularity.
- We build a system prototype of Demeter, which outperforms state-of-the-art solutions in terms of the job's SLO compliance and cost savings via extensive experiments.

II. Background and Motivation

A. Serverless Data Analytics

Resembling traditional systems [9], [12] a serverless analytics job can be expressed as a Directed Acyclic Graph (DAG), where nodes denote function stages and edges mean their data dependencies. Following the event-driven mode of serverless, a stage is triggered only when all of its dependent stages are completed. It then invokes parallel functions to consume a set of data partitions individually. This stage finishes only when the last component function concludes. In contrast to tasks in fixed resource slots divided evenly, these functions are loosely-coupled [17], giving us the freedom to decide the *configurable* amount of resources for *each of them*. In addition, the benefits of serverless analytics are: (*i*) Intra-stage, the resources held by each function are instantly released upon completion (from FaaS itself); (*ii*) Inter-stage, matching resource demands for each stage can yield better resource utilization [6].

Indeed, the serverless functions have struggled to build analytics applications, as direct communication between functions is not allowed by cloud providers [17]. Yet, this is no longer the case with advances in recent serverless systems [7], [18]. For instance, Pheromone [7] is a very promising solution that enables intra-node functions to exchange data via local shared memory, and inter-node functions over the network. Thus, the *position* of functions with dependencies directly affects their data I/O efficiency, especially across DCs.

B. Benefits and Challenges of Fine-Grained Orchestration

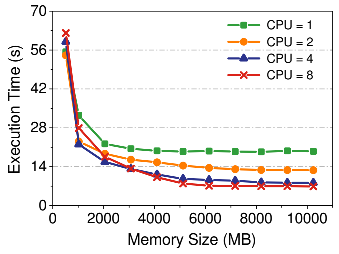
Benefits. Supposing that the cost of a function is inversely proportional to its duration, Fig. 1 illustrates an ideal example. As shown, due to existing remote data I/O, each function may incur a non-negligible transfer delay. Prior works [13], [14] always allocate the same computing resource for *all functions within a stage*, leading to a completion time of 12 units, even without any data or task skewing. The lagging critical function f_2 dominates the completion time, so it is almost impossible to get better performance from *inter-stage benefits*.

As shown in Fig. 1(a), we prefer to skew resources toward the critical function f_2, while functions f_1 and f_3 are slowed down to control the budget (Steps ① and ②). The roles of functions within a stage, however, might switch due to their interrelation. When function f_4 becomes the new bottleneck, it is migrated into a warm container (Step ③) to re-maintain the *balanced structure* in Fig. 1(b). The completion time becomes 9 units, which is 25% lower. In contrast, the overall execution time of functions (*i.e.*, total cost, though not entirely realistic due to changes in resource unit prices) does not undergo a notable increase. As such, a good orchestrator should identify each function's role, and make fine-grained function-level on-demand placement and configuration, contributing to stabilizing end-to-end JCT around the SLO with optimal cost.

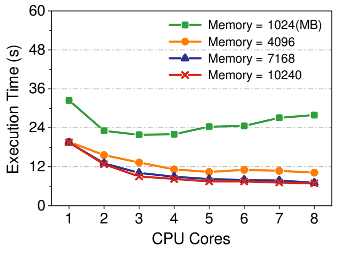
Challenges. Even ignoring the *correlation* between the two decisions for a while, we still tackle some technical challenges:

(*i*) *Volatile geo-distributed serverless environments*. Intense remote function interactions (*e.g.*, during the all-to-all *shuffle* phase) must traverse WAN links between DCs [4]. As we 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]. Besides, serverless providers are inevitably enduring unexpected cold-start when facing bursty functions from uncertain triggered stages, even with techniques such as *keep-alive* [19]. Such the volatility of WAN bandwidth and the burstiness of function execution jointly make it non-trivial to place large-scale functions within triggered stages.

(*ii*) *Diverse function resource demands*. We run a MapReduce sort workload over 10 GB data on Pheromone-MR [7]. As shown in Fig. 2, a data-intensive function exhibits varying sensitivities to different resource combinations. Specifically, increasing CPU cores for the function with a fixed memory size can effectively enhance its performance, but CPU over-provisioning enables multi-process overhead as a bottleneck, resulting in an execution time bounce. The multi-core advantage can only be fully utilized when there is sufficient memory capacity. However, any inside information about user functions



(a) Varying with memory size.



(b) Varying with CPU cores.

Fig. 2. The execution time of a data-intensive function under different CPU-memory combinations.

(packed as a black box [20]) is denied access, which makes it more challenging to estimate the resource demands of diverse functions with distinct amounts of input data.

III. Preliminaries

A. System Model

We consider a system $\mathcal{G} = (\mathcal{N}, \mathcal{L})$ consisting of N geo-distributed DCs in set $\mathcal{N}$ that enable serverless data analytics services, with the assumption that each DC can provide ample computing resources [4]. The user will request to query data stored on a subset of DCs, which is parsed into an analytics job comprising serverless functions distributed on these DCs [6]. Existing function interactions across DCs proceed over scarce and dynamic WAN links $\mathcal{L} = \{l_{i,j} \mid 1 \le i, j \le N, i \ne j\}$. As previously discussed, we consider the decoupled function configuration of any CPU-memory combination, without forcing a proportional allocation [17]. Thus, we define an allocation type as $p = (p_c, p_m) \in \mathcal{P}$, where p_c and p_m denote a set of CPU cores and memory limits, respectively. Here p is non-preemptive, *i.e.*, p_c and p_m are consistently available to a function until the end or timeout.

B. Function Orchestration for Geo-Distributed Analytics

We consider a total of T timesteps $\mathcal{T} = [1, \ldots, t, \ldots, T]$, each of which is divided evenly. A total of J jobs will arrive continuously at any time in $t \in \mathcal{T}$. In practice, the user tends to launch the same query (*i.e.*, job DAG) periodically on the freshest data [5]. We thus do not distinguish a user and a job below. The query request Ω_j of job j submitted by user j is:

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

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_j^n, l_j^n sign the path and offset of data bucket in DC $n \in \mathcal{N}_j$.

Upon Ω_j is received by the system, its profile $\mathcal{P}_j$ is parsed as 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 the cascade of upstream durations, each stage will also be triggered at any time in $t \in \mathcal{T}$. To avoid functions with long waits to be orchestrated, we enable distributed scheduling. Thus, 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 to all the DCs in $\mathcal{N}_j$, where $f_{j,k,i}$ indicates 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. Naturally, we restrict each function to be responded to by one DC, which requires a coordinator with a global view to decide. Finally, the placed functions are flexibly configured with tailored resources to adapt to the task complexity.

Note that, the total number of functions in triggered stages may exceed the limits of batch scheduling. So, each DC 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. For DC $n \in \mathcal{N}$, it extracts functions in the current "*wave*" (*i.e.,* pending function set $P_{n,t}$) by dequeuing functions in $Q_{n,t}$ until $P_{n,t}$ is full or $Q_{n,t}$ is empty. In essence, given J job profiles with arbitrary dependencies, each DC n needs to orchestrate all functions in $P_{n,t}$ at timestep t, which requires considering two decision variables: (*i*) $x_{j,k,i}^n$ in set X, indicating whether the function $f_{j,k,i}$ is placed on DC n $(= 1)$ or not $(= 0)$, and (*ii*) $y_{j,k,i}^p$ in set Y, denoting whether the function $f_{j,k,i}$ selects a type-p allocation for its container runtime $(= 1)$ or not $(= 0)$.

C. Function Performance and Cost Models

1) Function Duration Time: In practice, parallel functions in a "*wave*" are orchestrated in sequence, thus not started simultaneously. The reason is that batch function scheduling, based on the control-flow paradigm, remains in a First-Come-First-Served (FCFS) basis at the micro-level [18]. Besides the time components in Fig. 1, a function is also accompanied by an inevitable waiting time before formal execution.

Intuitively, the execution time of a data-intensive function is related to our allocation decision Y (see Fig. 2) 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 [14], 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.*, $\phi_{j,k} = \max_{i \in [|\mathcal{F}_{j,k}|]} \phi_{j,k,i}$. Note that 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 highly correlated with the function duration time. We denote the price per GB-second and core-second 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 $C_{j,k,i,n,p}^{op}$ of function $f_{j,k,i}$ with type-p allocation at DC n is:

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

Transmission cost. The transmission cost is expended on reading remote results via expensive WAN links, as local data

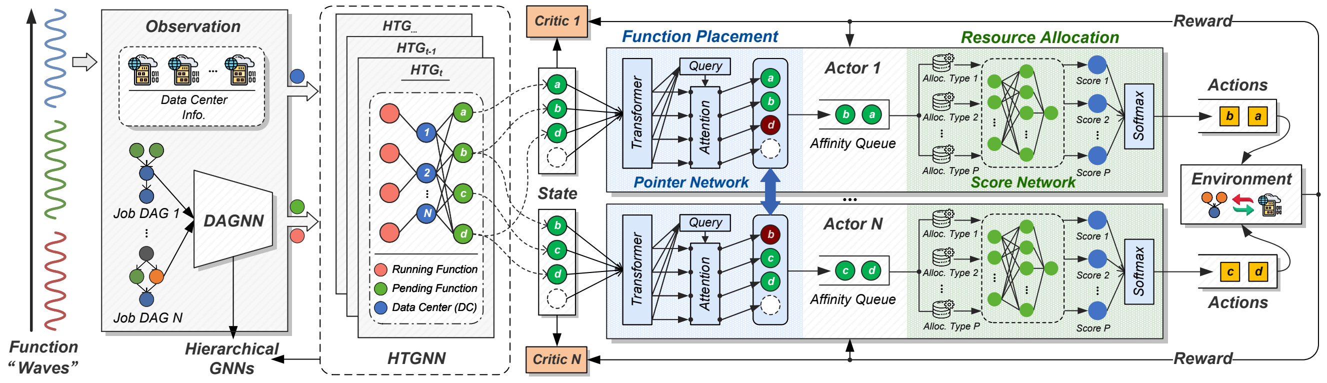


Fig. 3. The model workflow of Demeter.

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 function $f_{j,k,i}$ 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. Serverless users generally incur extra costs for ephemeral storage, which arises from: (*i*) The data outputted by functions that finish early needs to be stored, awaiting the current stage finishes; (*ii*) Fault-tolerant system retains all data produced by the current stage until the dependent stages end, ensuring that data 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 function $f_{j,k,i}$ 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 formulate our fine-grained orchestration problem in (6) – (14), which co-optimizes per-function placement and resource allocation for geo-distributed serverless analytics. The goal is to minimize the cost of J jobs while satisfying their SLOs and constraints of resource demands and data dependencies.

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

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

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

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

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

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

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

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

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

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-living limit on functions [17], *i.e.*, their durations should not exceed the timeout Γ.

IV. MARL-Based Function Orchestration

A. Algorithm Overview

Orchestrating functions across event-driven stages with (7) poses a challenging *sequential decision-making* problem. We thus have the opportunity to explore the unknown performance objective $B(\cdot)$ online by replaying experiences, while enhancing the distributed scheduling in Section III-B. As illustrated in Fig. 3, Demeter is designed as a MARL-based solution, in which each agent $n \in \mathcal{N}$ serves as the scheduler in DC n. With the state of a function "*wave*" as input, each agent produces a series of three-dimensional actions to place pending functions within the "*wave*" and allocate suitable resources to them.

To **handle the volatile GSA environments**, Demeter generates holistic and compact states for each agent using scalable hierarchical GNNs. These GNNs efficiently extract in-depth semantic relations between large-scale functions and DCs over multi-level features, capturing the impact of network changes, function dependencies and interferences. To **learn the diverse function demands**, we design a *pointer-score* network that wisely decouples the joint action for each agent. For function placement, the agents decode input states via a pointer network and then collaborate to make an *affinity-guided* decision. For resource allocation, the placed functions' raw states and their valid allocation types are input into a score network, producing the corresponding resource priorities. These two aspects are presented in Sections IV-B and IV-C, respectively.

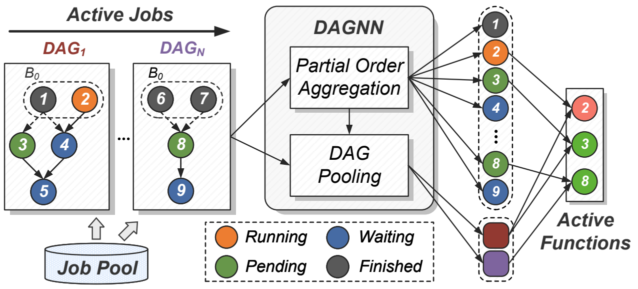


Fig. 4. The embedding process of our DAGNN. For ease of display, we only give an example of single-function stages. Note that, the dashed (solid) lines in a DAG depict data dependencies that have been (un)satisfied, and the "waiting" function denotes a function that has not yet been triggered.

B. Scalable State Representation via Hierarchical GNNs

Upon receiving a function "*wave*" at timestep t, Demeter collects information from two levels: function and DC, obtaining the desired features based on the factors in Section III-C. For the function level, we first classify the functions of active job j based on their status at t. Next, we separately construct their features, *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 follows:

- $\boldsymbol{\mathcal{R}_{j,t}}$ includes the type, 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, placement, resource peak, input data size, and duration time for every *finished* function in job j.

For the DC level, Demeter captures the number of remaining functions and warm containers in each DC $n \in \mathcal{N}$, its available bandwidth per connection with other DCs, and the unit prices of resources in the current region. They are concatenated into a flattened vector $d_{n,t}$ with a dimension of $(|\mathcal{N}| + 7)$.

DAGNN. A job DAG [1] 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 performance of event-driven functions. DAGNN [21] 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.

[1]The job DAG in our paper is a function-level DAG (see Section V-A).

Note that, *topological batching* is capable of scaling to multi-DAG cases for better parallel concurrency, as shown in Fig. 4. 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. 4, 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. 3. 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 [22] 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. 3, forming a set $\mathcal{P}_{n,t}$ as the input state of the actor network (see Section IV-C) in agent n.

C. Action Decoupling Based on Pointer-Score Network

The design of our actor networks is based on a key insight that decouples joint actions by their serial relation. We only assign resources for favorably placed functions, avoiding combining two actions into a large space. To this end, we design a *pointer-score* network to encode the decoupled actions, while reducing the complexity of training agents.

Pointer network. Given the relevance of parallel functions described in Section II-B, functions in a "*wave*" are critical contextual information for each other. Yet, their number $|P_{n,t}|$ is unknown for each DC n, due to the uncertainty of stage triggering. While the length of the input state sequence $\mathcal{P}_{n,t}$ can be fixed to the maximum by *padding* method in the Seq2Seq model, it cannot be used to fix the output sequence. In contrast, the pointer network [23] can learn the conditional probability of an output sequence against the element positions in an input sequence, which is well-suited for encoding our placement action. We re-design it 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:

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

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 $\mathrm{e}_{n,t} = \text{avg}_{f \in P_{n,t}} e_{f,n,t}$. The obtained attention score $h_{f,n,t}$ is used as a pointer to the corresponding function. Given (10), the pointer $h_{f,n,t}$ is converted into the conditional probability $\Pr\left(f|n\right)$ via a Softmax operator [23], 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 [24].

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

Considering DC load balancing 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):

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

Score network. As opposed to imposing a penalty term, we must filter allocation types that fail to satisfy the resource hard constraint (13) before formal decision-making. The reason is that we have no margin for trial-and-error, invalid allocations will lead to the function crashing directly [6].

Once the pointer network completes, agent n further handles the raw HTGNN embeddings of favorably placed functions in ascending order of posterior probability. The intuition is that functions with lower affinity need to be scheduled as early as possible to enhance performance. Concretely, each embedding concatenates with the total of P valid potential allocations to form 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 the type-p allocation for function f. Then, Demeter feeds $s^p_{f,n,t}$ into the score network to obtain a scalar value $g^p_{f,n,t}$, which can be interpreted as the score to allocate the type-p resource [25]. Due to its simplicity, the score network not only converges more easily but can also be trained (*i.e.*, P tasks above) in parallel. Finally, Demeter applies a Softmax operation to transform scores in $(g^1_{f,n,t}, \ldots, g^P_{f,n,t})$ into the resource allocation policy:

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

D. Model Training

To learn explicit end-to-end objectives and speed up model convergence, we train Demeter in episodes. Each episode has function "*waves*" from J jobs, with each "*wave*" requiring one or more actions for orchestration.

Reward. In timestep t, each agent receives an immediate reward 0 after it takes an action to orchestrate a function. The time in MARL next moves forward, but the environment states do not change. Until Demeter finishes the orchestration of all functions in the current "*wave*", the states of jobs and DCs shift forward on the time axis. At this point, we feed Demeter with a shared reward r_t, *i.e.*, the agents are rewarded with

$$r_t = -\sum_{j \in \mathcal{J}|_t^{t+1}} \omega_j \Big(\sum_{f \in P_{j,t}} C_f - \beta(S^*_j - l_j)\Big), \tag{20}$$

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 $l_j = \sum_{f \in P_{j,t}} \max\left(\delta_f - \Gamma, 0\right)$ for penalizing bad actions, where δ_f and ϕ_f are the duration time and finish timestamp of function f, respectively. The goal of the agent n is to maximize the cumulative rewards given by $\mathcal{J}(\theta_n) = \mathbb{E}\left[\sum_{i=1}^{T} \gamma^{t-1} r_t\right]$. We set the discount factor γ to be 1 as these actions are of equal importance over time [26].

Training algorithm. Due to the high concurrency and short-living 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) [27], to jointly learn the near-optimal orchestration policy. We mainly focus on two benefits: (*i*) PPO is a stable on-policy method that can better handle large-scale samples while being much simpler to tune; (*ii*) The smooth policy updates in PPO can alleviate the nonstationarity issue in MARL [28].

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 in (21),

$$\mathbb{E}\left[\min\left(\text{clip}\left(p_{n,t}(\theta_n), \epsilon\right) \hat{A}_{n,t}, p_{n,t}(\theta_n)\hat{A}_{n,t}\right)\right], \tag{21}$$

where ϵ is a clip threshold, 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 (GAE) [29] 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 the 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.

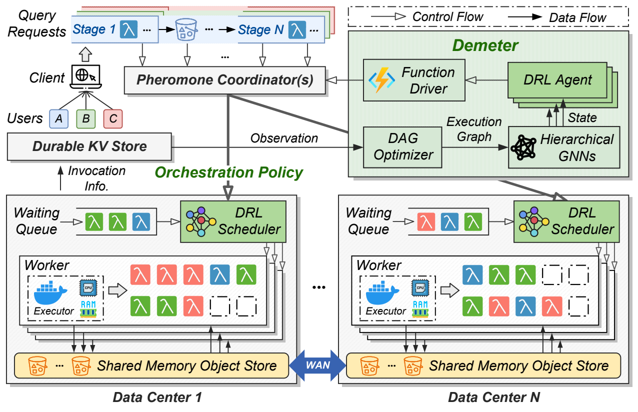


Fig. 5. The architecture of Demeter.

V. Implementation

We implement Demeter atop Pheromone [7], a data-centric serverless system that enables fine-grained function data exchange, such as all-to-all *shuffle*, via zero-copy shared memory in the form of data buckets. As shown in Fig. 5, we re-architect it and develop several components of Demeter.

A. DAG Optimizer for Execution Graph Generation

The DAG optimizer first identifies stage types (*i.e.,* function types) based on trigger primitives used in the job profile $\mathcal{P}_j$, *e.g.*, `DynamicJoin`→map, `DynamicGroup`→reduce [7]. It then generates the initial `LogicGraph`, where nodes denote stages and edges are their trigger modes. If a stage is triggered by the parent in a one-to-one *forward* manner, they will be chained as an all-new stage to further generate the `JobGraph`. In this way, upon a parent function finishes, its container will be reused by `fork()` new processes of the related child, thus avoiding start-up and data transmission overheads. Ultimately, the DAG optimizer splits a stage node into multiple parallel function nodes and decides inter-function linking relations to generate the `ExecutionGraph` (*i.e.*, function-level DAG), which serves as the DAG-based state representation.

B. Function Driver with Multi-process Support

To take advantage of multi-core performance, we implement a function driver that enables multi-process support of Python. The driver first gets the number of CPU cores p_c allocated for a function. When the input data is read from S3 [30] or shared memory, it evenly splits the data into p_c sets and `fork()` the corresponding number of processes. Each process will call the function code to consume a set of data. For Inter-Process Communication (IPC), we reuse the relevant data bucket that was initially used for inter-function IPC in Pheromone. This way, our intra-function IPC requires no additional memory and eliminates the overhead of collecting the multi-process output.

C. The Running Process of Demeter

Given the centralized state representation and shared reward in Demeter, we do not disperse each agent to its corresponding DC, thus reducing the communication overhead. Illustrated in Fig. 5, the query requests are created by the client based on user codes. The Pheromone coordinator admits and parses them into a sequence of event-driven stages, then dispatches parallel function invocations for triggered stages to the waiting queues across all potential DCs asynchronously. Meanwhile, Demeter inputs observations (*e.g.*, necessary invocation history) from the durable KV store into the DAG optimizer, generating the `ExecutionGraph`, and then makes per-function orchestration decisions to notify the DRL schedulers. Notably, the pending data sizes are directly obtained via a Pheromone API. Finally, the codes encapsulated by the function driver run through forcing the allocated resource for executors.

VI. Experimental Evaluation

A. Experimental Settings

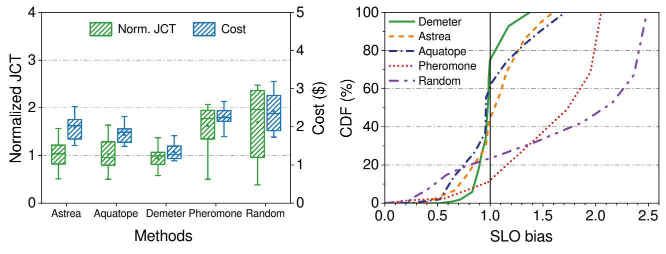
Model configurations. We train the MARL using PyTorch. The model depth L of both GNNs is set to 3. For HTGNN, we set its time window size as 5 and accelerate it using a memory module [31] to match the parallelized DAGNN. In the pointer network, its Transformer encoder is configured with 3 layers and 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}.

Testbed. 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. We also deploy 4 shared coordinators as the ratio in [7] to improve the system scalability, each of which manages a batch of disjoint jobs. The coordinators and workers are run at c5.4×large and c5.9×large instances, respectively. Each function can be configured with 8 CPU cores at most and memory limits from 128 MB to 10 240 MB with 64 MB increments [17].

Workloads. We generate serverless job sets using analytics benchmark suites: TPC-DS [32] and BigData [33]. The stages invoke parallel functions based on the size of intermediate data [34]. Each function is written in Python and calls system interfaces using `ctypes` [35]. 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 adjusting the *scaling factor* of the 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.

Job arrivals. Our MARL model is trained with job arrival rates within 24 hours following the Facebook Hadoop Workload Trace [36]. We select three modes: `slow` (50 jobs arriving in one hour), `normal` (80 jobs arriving in 30 minutes), and `burst` (100 jobs arriving in only 15 minutes).

Baselines. We compare Demeter with the following well-assembled baselines. Before that, we introduce Fair [10], a task scheduling algorithm for multi-job across geo-distributed DCs, to our FaaS setup and rewrite its objective function. To this end, we use the parametric model in [16] to provide rough function performance estimates for Fair.



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

Fig. 6. Overall performance under the 4-agent cluster setting.

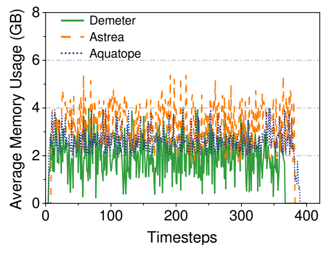


Fig. 8. The long-term average memory usage.

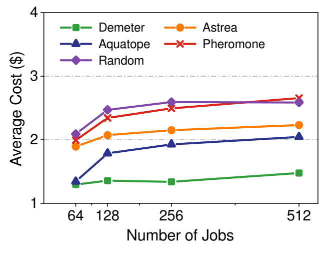
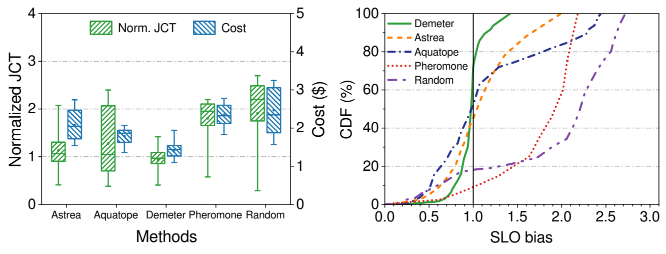
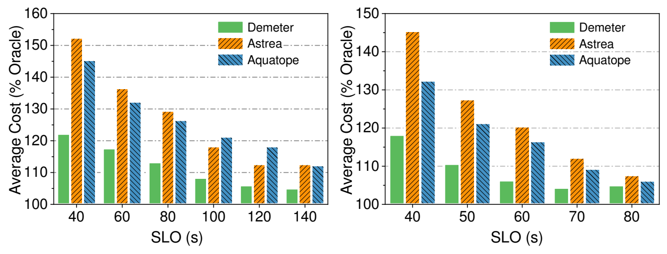


Fig. 9. The average cost with various job scales.



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

Fig. 7. Overall performance under the 8-agent cluster setting.



(a) TPC-DS Q94. (b) BigData Q3.

Fig. 10. The performance of two typical jobs with various SLOs.

- **Astrea** [13]: It poses a shortest-path-based algorithm on a DAG composed of configurable parameters, to find the optimal allocation for a serverless analytics job with user-defined constraints. We extract the sub-algorithm aligned with our goal and add Fair as its function scheduler.
- **Aquatope** [14]: It searches for appropriate resources of warm-start functions within stages to ensure the workflow's SLO, via uncertainty-aware Bayesian optimization. Likewise, we add Fair as its function scheduler.
- **Pheromone** [7]: It natively adopts a data locality-aware function scheduling and greedily allocates the minimum but valid amount of resources to functions.
- **Random**: It randomly samples a potential DC and feasible allocation type for every function.

B. Performance Evaluation

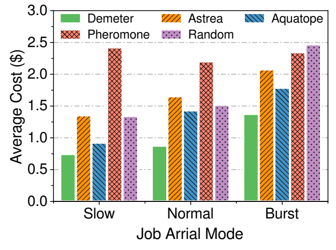
We first duplicate Demeter and assign the replicas to other shared coordinators to share inference pressure. Each function requires at least 1 CPU core and the lowest memory limit satisfying (13). The loss factor α is set to 0.8. The resource prices for each region, as discussed in Section III-C2, follow the AWS Lambda [17] and EC2 [37] pricing models. Notably, the CPU allocation on Lambda roughly follows a ratio (*i.e.*, 1.728 GB/core) of the memory limit. Therefore, we set the per-core price as $1.728\times$ that of per-GB memory within the same region. Also, the orchestration interval is set to 5 seconds, and the maximum number of functions is limited to 20.

Overall performance. We first evaluate Demeter in terms of average cost and SLO compliance against four baselines. Fig. 6(a) and 7(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 orchestration. It reduces average costs by at least 32.7% and 23.3%, respectively, compared to the baselines. Moreover, part of the gap is attributed to the higher non-execution costs incurred by the baselines, resulting from a significant number of functions being stranded, especially in the 8-agent cluster. In contrast, Aquatope, which considers uncertainty in FaaS platforms similar to ours, performs better than other baselines. However, it confines the allocation search to warm-start behaviors only, which makes it struggle to adapt to unexpected function cold-starts from bursty stages, resulting in inferior SLO compliance under the 8-agent cluster.

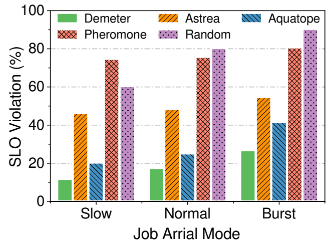
Fig. 6(b) and 7(b) show the SLO violation ratios and their distributions. We define the SLO bias (*i.e.*, normalized JCT) as measured JCT / SLO, which is higher for more critical SLO violations. As shown, Demeter outperforms all baselines, ensuring better SLO compliance with relatively smaller performance fluctuations. Specifically, it eliminates another 27.4% and 39.1% of the SLO violations compared with Aquatope and Astrea, respectively, and brings the total to below 20%. It's noteworthy that Aquatope's lower cost comes with the risk of a high violation magnitude, despite having a relatively low SLO violation ratio. This is because of its active allocation search without explicit penalties, which tends to pursue minimal cost when jobs are judged not to be completed on time.

Given the correlation between the JCT and cost, we further analyze whether Demeter truly reduces resource consumption. Fig. 8 displays the results with the 8-agent. As shown, Astrea allocates CPU cores in a fixed proportion to the memory limit, leading to memory over-provisioning to accelerate compute-intensive functions. In contrast, Demeter strikes a good balance via flexible multi-resource allocation, neither stacking large amounts of resources for lower JCTs to ensure their SLOs, nor reducing resource usage time to save costs.

The impact of workloads with various SLOs. We track the trajectories of two typical workloads on the 8-agent cluster: TPC-DS Q94 and BigData Q3. The former has more stages

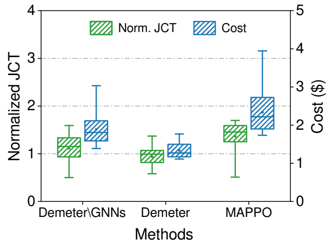


(a) Average cost.

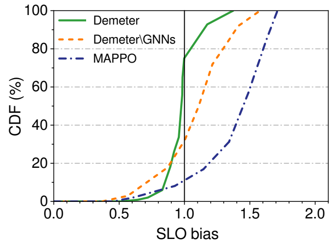


(b) SLO violation.

Fig. 11. The average cost and SLO violation with different arrival modes.



(a) Normalized JCT and cost.



(b) CDF of SLO bias.

Fig. 12. Overall performance among various model settings.

and pronounced intermediate data changes. Fig. 10 plots their average costs with three SLO-aware algorithms. When SLO is relatively low, Demeter demonstrates better cost-effectiveness across the two workloads, with savings of 18% to 51% compared to the second-best Aquatope. Due to the wide variety of function resource demands, the cost gap between Demeter and the baselines is larger in Q94 than in Q3. 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 SLO is relatively high, all methods perform well. Overall, Demeter maintains stable performance and achieves average costs within 12.3% of the optimal allocation of Oracle, which exhaustively explores the entire profile space without any performance fluctuations.

Scalability evaluation. To evaluate the robustness of Demeter, we vary the number and mode of job arrivals on the 8-agent cluster. The results are shown in Fig. 9 and 11, respectively. We observe that Aquatope is better suited to the `slow` mode and small-scale jobs, as its pre-warm container pool has more breathing time to improve the hit rate of warm-starts. Static Astrea is non-sensitive to changes in the environment, and its instability stems from the inherent noise in the system. In contrast, Demeter exhibits no significant performance degradation as the job scale increases, and reduces the average cost and SLO violation by at least 23.4% and 9.7%, respectively, under different job arrival modes. Our performance benefits from the efficiency of our MARL model and the training with sufficient workloads.

Ablation evaluation. We dive into Demeter and compare it with a vanilla MAPPO algorithm that orchestrates pending functions in an FCFS manner, and a MAPPO algorithm only with the *pointer-score* network. Due to the absence of state compression via GNNs, the control group has to input stacking features of pending functions and DCs to the eight agents. As shown in Fig. 12, the vanilla MAPPO struggles to find suitable function-level orchestration strategies with a large state-action space. The addition of both the *pointer-score* network and hierarchical GNNs, however, can further improve performance, suggesting that the state representation and action decoupling mechanism we have designed are efficient.

VII. Relate Work

(Geo-distributed) serverless analytics. Geo-distributed analytics in traditional framework optimize JCT or WAN costs by task scheduling [10], [11] or data placement [4], [38]. Due to the limitations of serverful architecture, they cannot allocate resources at task granularity like Demeter. In the serverless context, while some studies [39], [40] place functions among cross-region nodes by sensing network, they are not specific to analytics jobs. More studies aim to optimize the performance and cost of serverless analytics in one region [6], [34]. Astrea [13] finds optimal allocation for a user-oriented analytics job with budget and SLO demands based on holistic performance modeling. Kassing *et al.* [16] also consider factors unique to serverless analytics, as we do, to make the trade-off between JCT and cost with Pareto optimal. However, they do not implement resource allocation and placement at function granularity. Other studies [5], [41], [42] deploy functions at the network edge, and consider network variations and resource limits to optimize analytics services. These works are yet not suitable for large-scale query services across DCs.

SLO-aware resource allocation. StepConf [15] dynamically configures function steps (*i.e.*, stage) based on the critical path in the job DAG. Aquatope [14] 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 heterogeneous needs of functions due to various placements. Moreover, since Mao *et al.* [25], [26] proposed Deep Reinforcement Learning (DRL) to enable multi-resource scheduling for data processing clusters, *DRL for systems* has gained a surge of attention. Yu *et al.* [20], [43] were the first to use DRL algorithms to harvest idle resources and alleviate function slowdown in FaaS systems. Qiu *et al.* [44] apply MARL with SLO violation and resource utilization as part of the reward, to reduce function duration in multi-tenant platforms. Our solution is different from [20] and [44], as 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.

VIII. Conclusion

In this paper, we have presented Demeter, a fine-grained function orchestrator that decides the per-function placement and resource allocation for geo-distributed serverless analytics. Given the volatility of GSA environments, Demeter has used a MARL algorithm with scalable state representation and action encoding, aiming to enhance cost savings while ensuring the job's end-to-end SLO. Extensive experiments with realistic workloads have validated that Demeter can effectively reduce average job costs and improve SLO compliance, outperforming 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. of the IEEE INFOCOM*, pp. 1–10, 2023.

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

[3] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon, "Follow the sun through the clouds: Application migration for geographically shifting workloads," in *Proc. of the ACM SoCC*, pp. 141–154, 2016.

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

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

[6] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable analytics on serverless infrastructure.," in *Proc. of the USENIX NSDI*, pp. 193–206, 2019.

[7] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. of the USENIX NSDI*, pp. 1489–1504, 2023.

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

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

[10] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 488–500, 2018.

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

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

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

[14] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. of the ACM ASPLOS*, pp. 1–14, 2022.

[15] Z. Wen, Y. Wang, and F. Liu, "StepConf: SLO-aware dynamic resource configuration for serverless function workflows," in *Proc. of the IEEE INFOCOM*, pp. 1868–1877, 2022.

[16] S. Kassing, I. Müller, and G. Alonso, "Resource allocation in serverless query processing," *arXiv preprint arXiv:2208.09519*, 2022.

[17] "AWS Lambda." https://aws.amazon.com/lambda/.

[18] Z. Li, C. Xu, Q. Chen, J. Zhao, C. Chen, and M. Guo, "Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration," *arXiv preprint arXiv:2304.14629*, 2023.

[19] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," *arXiv preprint arXiv:2003.03423*, 2020.

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

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

[22] Y. Fan, M. Ju, C. Zhang, and Y. Ye, "Heterogeneous temporal graph neural network," in *Proc. of the SIAM-SDM*, pp. 657–665, 2022.

[23] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[24] Y. Li, X. Zhang, T. Zeng, J. Duan, C. Wu, D. Wu, and X. Chen, "Task placement and resource allocation for edge machine learning: A gnn-based multi-agent reinforcement learning paradigm," *arXiv preprint arXiv:2302.00571*, 2023.

[25] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. of the ACM SIGCOMM*, pp. 270–288, 2019.

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

[27] 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," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24611–24624, 2022.

[28] C. S. de Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson, "Is independent learning all you need in the starcraft multi-agent challenge?," *arXiv preprint arXiv:2011.09533*, 2020.

[29] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[30] "AWS S3: Object storage built to retrieve any amount of data from anywhere." https://aws.amazon.com/s3/.

[31] 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.

[32] "TPC-DS Benchmark." http://www.tpc.org/tpcds/.

[33] "Big Data Benchmark." https://amplab.cs.berkeley.edu/benchmark/.

[34] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics.," in *Proc. of the USENIX NSDI*, pp. 653–669, 2021.

[35] "ctypes — A foreign function library for Python." https://docs.python.org/3/library/ctypes.html.

[36] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *Proc. of the IEEE MASCOTS*, pp. 390–399, IEEE, 2011.

[37] "AWS EC2 Pricing." https://aws.amazon.com/cn/ec2/pricing.

[38] 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," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2155–2169, 2019.

[39] F. Rossi, S. Falvo, and V. Cardellini, "GOFS: Geo-distributed scheduling in openfaas," in *Proc. of the IEEE ISCC*, pp. 1–6, 2021.

[40] G. Zheng and Y. Peng, "Globalflow: A cross-region orchestration service for serverless computing services," in *Proc. of the IEEE CLOUD*, pp. 508–510, 2019.

[41] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *Proc. of the IEEE/ACM CCGRID*, pp. 41–50, 2020.

[42] 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. of the IEEE INFOCOM*, pp. 1–10, 2023.

[43] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, "Faasrank: Learning to schedule functions in serverless platforms," in *Proc. of the IEEE ACSOS*, pp. 31–40, 2021.

[44] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Reinforcement learning for resource management in multi-tenant serverless platforms," in *Proc. of the ACM EuroMLSys*, pp. 20–28, 2022.