Dike: Deep Reinforcement Learning For Function Scheduling in SLO-targeted Serverless Edge Computing

Chen Chen*, Emanuele Carlini †, Richard Mortier*

*Department of Computer Science and Technology, University of Cambridge, UK
†National Research Council of Italy, Italy
Email: cc2181@cam.ac.uk; emanuele.carlini@isti.cnr.it; rmm1002@cam.ac.uk

***Abstract*—Serverless computing is regarded as a good match for distributed edge infrastructures. However, bringing the function-as-a-service model to a highly dynamic, distributed, and heterogeneous pool of resources has its fair amount of challenges. Allocation of functions to the proper resource is an essential operation that avoids over-provisioning and the relative waste of computational resources. In this work, we propose *Dike*, a bi-level function scheduling and resource allocation framework designed to meet end-to-end latency SLOs (service-level objectives). *Dike* leverages deep reinforcement learning to balance resource provisioning and monetary cost by incorporating both composite cost and SLO violations into the reward. Extensive simulations with real-world production workloads demonstrate the superiority of *Dike*. Experimental results show that the proposed algorithms approximate the results of state-of-the-art ILP solver within a factor of 1.08 while dramatically reducing the scheduling time.**

***Index Terms*—Serverless Computing, Edge Computing, Deep Reinforcement Learning**

I. Introduction

Serverless computing, also known as Function-as-a-Service (FaaS) has emerged as a new computing paradigm, providing fine-grained billing in pay-as-you-go model [1, 2], and relieving application developers from spending massive effort to manage virtual machines and containers which precluding them from focusing on the business logic. With serverless computing, function developers are released from complex infrastructure management as they only need to submit function codes to the FaaS platform, where computing resources are allocated by the platform [3, 4]. Serverless has been particularly beneficial for applications with variable workloads, such as those in Machine Learning (ML) applications and Internet of Things (IoT) [5, 6, 7], where the ability to scale up or down seamlessly can lead to better performance and cost efficiency. Furthermore, serverless computing is often utilized to process data close to its source, such as system operation logs and user data [8, 9].

Currently, popular serverless platforms like Microsoft Azure and AWS Lambda enable developers to deploy a large number of function instances in a short period of time in response to events. To ensure a seamless end-user experience, many latency-sensitive applications impose Service-Level Objectives (SLOs) on the end-to-end latency perceived by users. Service providers often resort to over-provisioning resources to enforce these SLOs, but this approach can increase the composite cost of functions and negatively impact their revenue.

A key enabler for more efficient resource utilization is *SLO-aware resource management*. The goal is to continuously optimize the overall cost by allocating functions to more cost-effective resources while still meeting the end-to-end latency SLO. However, modern applications are progressively transitioning from traditional centralized architectures to distributed ones, leveraging the recent progress and availability of Edge Computing infrastructures [10, 11]. User requests often traverse multiple function calls, potentially operated by geographically distributed and heterogeneous nodes, depending on the specific server logic of the applications. This shift introduces two distinct system-level behaviors: macro-level behavior, which includes end-to-end latency and composite cost, and micro-level behavior, which pertains to hardware resource usage (e.g., CPU and memory).

To embrace the distinct levels of behavior brought by edge computing, we decouple resource allocation between service-level SLO constraints and function-level requirements. We propose ***Dike***, a bi-level learning-based resource management framework. The goal of *Dike* is to optimize function scheduling for cost minimization, while ensuring compliance with the SLOs set by application developers. We formulate the resource management of serverless functions at the edge as an Integer Linear Programming (ILP) problem, optimizing the cost of using serverless functions by considering fine-grained resource allocation and application-level SLO targets. Due to the computational complexity of the proposed problem (we prove it to be NP-hard), we have carefully devised two deep reinforcement learning (**DRL**) algorithms: one based on Deep Q Networks (**DQN**) and another based on Proximal Policy Optimization (**PPO**). These algorithms are designed to balance the trade-off between composite cost and end-to-end latency.

Our main contributions can be summarized as follows:

- We formulate the function scheduling and resource allocation problem in edge infrastructures as an Integer Linear Programming (ILP) problem, including composite cost and end-to-end SLO (§III). We also prove that this problem is NP-hard.
- We propose *Dike*, a bi-level learning-based framework (§IV). *Dike* separately devises mechanisms for function resource allocation and application-level SLO, bridging them through the reward function in the DRL system.
- We conducted extensive experiments (§V) in an edge network with capacity constraints to demonstrate *Dike*'s cost savings over the Integer Programming Solver Midaco [12]. Compared to the best-performing benchmark, *Dike* achieves comparable performance to the Midaco solver (within a factor of 1.08) while being significantly more time-efficient (99% less scheduling time).

II. Related work

Since cloud providers have progressively adopted serverless computing as their computing paradigm, recent studies have investigated the trend of applying serverless computing to edge computing. For instance, DisProTrack [1] proposes to combine system and service logs together for provenance tracking over serverless architecture. A universal provenance graph is used to optimize reverse query parsing based on log expression. Demeter [8] uses multi-agent reinforcement learning to configure serverless platforms for geo-distributed analytics. Xu *et al.* [9] enable query evaluations for big data analytics in serverless edge clouds. A parameterized virtualization method is adopted to bridge the short-lived serverless functions and large resource demands of big data queries. Gu *et al.* [7] propose two efficient parameter search methods for serverless data compression, estimating data transmission time and monetary costs.

A good deal of research effort has focused on optimizing the latency, monetary cost and other related areas. For example, Li *et al.* [13] use Zygote container and pre-warming to optimize the container startup time. A randomized rounding method is used to achieve near-optimal solutions with a performance guarantee. QUART [6] optimizes the key stages in pipeline parallelism and uses a bi-level model parameter caching system for response latency. Tutuncuoglu *et al.* [14] propose to dynamically price compute and memory resources for revenue maximization. Peng *et al.* [15] proposes algorithms for cloud-edge video analytics systems, optimizing both communication computation in terms of monetary costs. S-Cache [5] combines LSTM prediction and function caching, optimizing the end-to-end latency and reduces cold-start issues.

Although serverless computing has brought many benefits, it is still challenging to achieve near-optimal function scheduling while considering resource efficiency and SLO violations. We present *Dike*, a bi-level resource management framework that decouples function resource allocation from SLO feedback, bridging them through the notion of reward function in the DRL system.

III. Problem formulation

TABLE I: Symbols and Variables

Symbols	**Description**
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Physical network graph
$\mathcal{V}$	Set of edge nodes
$\mathcal{E}$	Set of links
$\mathcal{N}$	Set of functions
$\mathcal{T}$	Set of time slots
u_n	The required amount of resources for function n
$U_v(t)$	The resource capacity of node v
θ_v^n	The cost of creating a new function n at node v
β_v^n	The cost of operating function n at node v for one unit of time
$\alpha_{v,v'}$	Price for one unit of traffic between node v and v'
O_n	The SLO target for request n
Variables	
$z_v^n(t)$	Decision variable to indicate if function n is assigned to node v
$z_{v->v'}^n(t)$	Decision variable to indicate if function n assigned to node v' from node v
$g_v^n(t)$	Binary variable indicate if n is newly created

A. System model

We consider a cluster $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consisting of $|V|$ geo-distributed edge nodes in $\mathcal{V}$. Each edge node is equipped with a certain amount of hardware resources denoted by $U_v(t)$. Edge nodes are connected by links. For example, we use $(v, v') \in \mathcal{E}$ to represent the link between node v and v'. A user can generate a request n which requires a function n. Each function $n \in \mathcal{N}$ has a resource demand u_n which is the required amount of hardware resources. Finally, we use $t \in \mathcal{T}$ to denote the time in the system. We list all symbols and variables in Table I.

B. Function cost model

Function switching cost:

The function switching cost is incurred when creating a new function instance by pulling images from function repositories and initializing the environment before a function can execute the code. We denote the switching cost of function n on the node v by θ_v^n. Thus, the switching cost $C_s(t)$ of function n is:

$$C_s(t) = \theta_v^n \cdot g_v^n(t), \tag{1}$$

where $g_v^n(t)$ is a binary variable, indicating if function n is newly created on node v.

Function operating cost:

The function operating cost refers to the execution of a specific function incurred by the usage of hardware resources such as CPU and memory. We denote the price per unit of time as $\beta_v^n(t)$ and the duration time of function n as δ_n. Thus, the operating cost $C_o(t)$ is:

$$C_o(t) = \beta_v^n(t) \cdot \delta_n \cdot z_v^n(t), \tag{2}$$

where $z_v^n(t)$ is a binary variable, indicating if function n is allocated to node v.

Transmission cost:

The transmission cost is incurred by the bandwidth usage via expensive WAN links. Let $\alpha_{v,v'}$ be the price per unit of data transferred by link (v, v').

$$C_l(t) = \alpha_{v,v'} \cdot h_v^n \cdot z_{v->v'}^n(t). \tag{3}$$

where h_v^n us the amount of data transferred to function n and v is where n is deployed. We denote the decision variable as $z_{v->v'}^n(t)$, indicating whether function n is generated at node v but offloaded to node v'.

C. Problem formulation

We formulate our function scheduling problem, which optimizes function placement and resource allocation for serverless edge computing. The objective is to minimize the composite cost of N requests while meeting their SLOs and the constraints of resource capacities.

$$\min \sum_{t\in\mathcal{T}}\sum_{v\in\mathcal{V}}\sum_{n\in\mathcal{N}}\Big(C_s(t) + C_o(t) + C_l(t)\Big), \tag{4}$$

s.t.

Constraint 5 guarantees that each function request n must only be assigned to one edge node once.

$$\sum_{v\in\mathcal{V}} z_v^n(t) = 1, \forall n \in \mathcal{N}, \forall t \in \mathcal{T}, \tag{5}$$

Constraint 6 ensures that the variables $g_v^n(t)$, $z_v^n(t)$ and $z_{v->v'}^n(t)$ must be binary.

$$g_v^n(t), z_v^n(t), z_{v->v'}^n(t) \in [0,1], \forall v, v' \in \mathcal{V}, \forall n \in \mathcal{N}, \forall t \in \mathcal{T}, \tag{6}$$

Constraint 7 guarantees that the capacity of each edge node must be sufficient for the demand of each function.

$$\sum_{n\in\mathcal{N}} u_n \cdot z_v^n(t) \le U_v(t), \forall v \in \mathcal{V}, \tag{7}$$

Constraint 8 ensures that the SLO is not violated. O_n represents the SLO of the serverless request n which is the maximum acceptable latency for a service request.

$$\sum_{v\in\mathcal{V}}\left(d_{v,v'}^n \cdot z_{v->v'}^n(t) + d_v^n\right) \le O_n, \forall n \in \mathcal{N}, \forall t \in \mathcal{T}. \tag{8}$$

where $d_{v,v'}^n$ denotes the transmission delay between node v and v'. d_v^n is the total processing delay of function n hosted on node v.

D. NP-hardness

The Generalized Assignment Problem (GAP), which is known to be NP-hard, can be reduced to the proposed problem. The GAP problem refers to assigning a number of $k \in \mathcal{K}$ jobs to a set of $j \in \mathcal{J}$ agents to execute the jobs, optimizing the cost. We denote the size of a job k by ω_k so we can map the job k to a function n with a size u_n. Also, we can map the agent j to a computing node v in the proposed problem with a hardware capacity U_v. Finally, if we map the cost objective of the GAP problem to the composite cost of the proposed problem, the original problem becomes finding the optimal solution for cost minimization in the GAP problem. Hence, the GAP problem becomes a special case of the proposed problem, and thus the problem is NP-hard.

IV. Realizing DRL in *Dike*

A. Markov Decision Process

We have employed model-free reinforcement learning where the probability of state transitions remains unknown. Thus, we use an agent to interact with the environment and learn from the process, addressing the challenges posed by the Markov Decision Process (MDP). Despite this, using control algorithms to solve the MDP remains challenging To solve the proposed problem by deep reinforcement learning, we first model the edge network as an environment with a centralized controller as the DRL agent. We formulate the proposed problem as an MDP where the agent aims to learn the optimal policy π that maps states to actions. By this means, the agent manages to learn a set of scheduling decisions denoted by $a(t)$ according to the observation of the state $s(t)$ in time slot t. Meanwhile, the state $s(t+1)$ of time $t+1$ and the reward $r(t)$ are impacted by the decision $a(t)$. We denote this interaction and learning process by $\{s(t), a(t), r_t, s(t+1), a(t+1), r(t+1), s(t+2), \ldots\}$. Thus, we formulate the state s, the action a, and the reward r as follows.

1) State representation

The state represents the environment in DRL, consisting of a set of states denoted by $s(t) \in \mathcal{S}$. In particular, the state includes the existing functions hosted on each edge node, the hardware capacity on each edge node and the current requests in each time slot t. We denote the remaining hardware capacity of all edge nodes as $\{\phi_v(t) \in \Phi(t)\}$ where $\phi_v(t)$ is the remaining hardware capacity in the edge node v. Also, we use $w_v^n(t)$ to represent the number of existing type n functions in node v. Thus, the set of existing number of functions is denoted by $\mathcal{W}(t)$. The set of requests is denoted by $Req(t) = \{n, i_n(t)\}$, where $i_n(t)$ indicates the index of the node that the request originates from, and n denotes the type of the request. Finally, we formulate the set of states $\mathcal{S}$ in time slot t as:

$$\mathcal{S} = \{\Phi(t), \mathcal{W}(t), Req(t)\}, \tag{9}$$

2) Action space

The action space in DRL refers to the behavior of an agent. While the agent takes an action in time t regarding to the policy π, the state is transferred to a new state. Let $a(t) \in \mathcal{A}$ represent the set of possible actions in the system. The agent can select policies to decide the placement of incoming requests, namely decide the value of $z_v^n(t)$. Thus, the action space can be denoted by:

$$\mathcal{A} = \{z_v^n(t), g_v^n(t)\}, \tag{10}$$

The action $a(t)$ serves two purposes, i.e., specify which edge node to host the function by $z_v^n(t)$; and determine whether create a new function denoted by $g_v^n(t)$. The design of our action space is based on our key insight that decouples joint actions through their serial relationship. We select a feasible edge node with sufficient hardware capacity and then examine if an edge node has idle functions that can be reused, avoiding frequently creating new functions.

3) Reward

In time t, the agent receives a reward $r(t)$ after it takes an action to schedule a request. The reward is a metric to evaluate the performance of the action $a(t)$ under the state $s(t)$, guided by the policy π. In the proposed problem, we feed the agent with a shared reward $r(t)$, i.e., the agent receive a reward as follows.

$$r(t) = -\Bigg(C_s(t) + C_o(t) + C_l(t) + \epsilon \cdot P_n\Bigg), \tag{11}$$

where $C_s(t), C_o(t)$ and $C_l(t)$ are the switching, operating and transmission cost, and ϵ is a penalty factor. P_n is a binary variable indicates if request n violates the SLO for penalizing "bad" actions. In other words, when the SLO is violated, we impose a penalty cost so the agent can try to avoid "bad" actions.

The goal of the agent is to maximize the cumulative rewards given by $R(t)$ as follows. The agent aims to the maximize the expected discounted reward function:

$$R(t) = \mathbb{E}\left[\sum_{j=0}^{\infty} \gamma^j r(t+j)\right]. \tag{12}$$

where γ^j is a discount factor to tune the trade-off between immediate and long-term returns. A value close to 1 indicates the agent prefers the long-term return and vice versa.

B. Algorithm design

Since our action space is discrete and multi-dimensional, we employ two different algorithms to compare their performance, namely Deep Q-learning (***Dike-DQN***) and Proximal Policy Optimization (***Dike-PPO***). *Dike-DQN* is a value-based method that uses a deep neural network to approximate the Q-functions. In contrast, *Dike-PPO* is a gradient-based algorithm with a stable on-policy method that prevents large updates.

In our system, the agent aims to explore and exploit the action space to learn optimized decisions for request scheduling. By employing a Deep Neural Network, *Dike-DQN* can estimate the Q-values for a given action and state, which refers to the expected reward. *Dike-DQN* progressively explores and learns its decision-making capability via experience relay. This feature is particularly useful for discrete and high-dimensional environments. In contrast, *Dike-PPO* uses a gradient policy method for stable, conservative updates of the policy. *Dike-PPO* employs a clipped surrogate objective to restrict the magnitude of policy updates, balancing exploration and exploitation. This makes *Dike-PPO* more versatile and robust, avoiding overly large updates that could destabilize training.

C. Algorithm overview

State Observation: For each time step t, the agent performs observation of the current state $s(t)$, consisting of the remaining hardware capacities and functions on each node.

Action Selection: Following receiving the observation of state, the agent takes an action $a(t) \in \mathcal{A}$ based on its policy π. The action refers to the scheduling decision $z_v^n(t)$ and whether a new function needs to be created by $g_v^n(t)$. The agent interacts with the environment using actions of (1) deciding if creating a new function or using an existing function and (2) assigning a request to a function. The feasibility of actions for a state is determined by the state, constrained by the hardware resources and SLOs.

Reward evaluation: In each time step t, the agent receives an immediate reward after it takes an action to schedule a request, reflecting the appropriateness of the action $a(t)$ in terms of the composite cost. The immediate reward $r(t)$ of state transition is correlated with the switching, operating, transmission cost and the SLO violations.

Policy update: *Dike-DQN* updates the Q-table using the Bellman equation [16] with experience replay to optimize learning stability. For *Dike-PPO*, we compute the policy gradients first and the policy is updated by tuning the gradients, ensuring that the policy updates are within a safe region.

Training: Explicitly, we train *Dike-DQN* with a discount factor γ^j of 0.99, a final random action probability of 0.1 and a learning rate of 0.0001. Also, we train *Dike-PPO* with a discount factor γ^j of 0.99, a clip range of 0.2 and 2048 steps for each update. They can be adjusted accordingly.

Objective and convergence: The objective of *Dike* is to minimize the composite cost which is converted to maximize the reward. We train the algorithm to converge at an optimized policy π^* offline and perform inference online in the edge networks to achieve efficient request scheduling.

V. Performance evaluation

We implement *Dike* with real-world traces in simulations, totalling 2000 lines of code in Python. The experiment is conducted on a server with 32 GB RAM and a 13th Gen Intel(R) Core(TM) i7-13700H processor with 14 cores.

A. Simulation settings

Baselines. To evaluate the performance of *Dike*, we compare it with an Integer Linear Programming (ILP) solver, Midaco [12]. Midaco is suitable for approximating the optimum in complex optimization problems with high dimensions. Midaco solves problems by using objective functions with equality and inequality constraints and has been widely employed by international organizations such as the European Space Agency (ESA) and German Aerospace (DLR). While other ILP solvers like Cplex and Gurobi exist, they cannot find optimized solutions for the proposed problem within a reasonable time due to its complexity and scale. Overall, Midaco can efficiently approximate the optimum after sufficient rounds of iterations.

For the proposed *Dike*, we have used Stable-Baselines3 [17] which is a set of reliable implementations of reinforcement learning algorithms in PyTorch.

Workloads. We generate the serverless requests using a Huawei dataset [18], the memory allocation for each function is in [30, 300] MB. This dataset consists of the invocations in Huawei public clouds for 141 days and 200 functions.

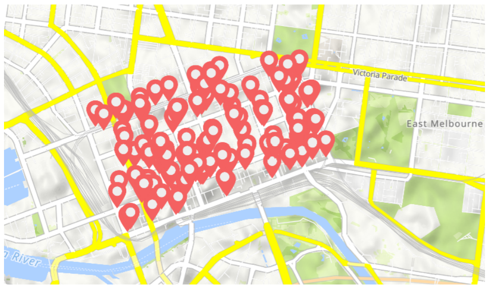

Fig. 1: Map of edge servers in Melbourne CBD area

Edge networks. We selected the EUA dataset [19] that reports the location of 125 edge servers in Melbourne CBD area as illustrated in Figure 1. We defined 5 types of edge nodes with CPU frequency ranges from 2.4 to 3.6 GHz, and the memory capacity is in [16, 24] GB.

Cost model. We use the parameters from [2] for the cost model. In particular, the function operating price $\beta_v^n(t)$ is to be proportional to the CPU frequency of the edge node. The function switching cost is inversely proportional to the CPU frequency of the edge node.

B. Performance evaluation

In the evaluation section, we report the results of Midaco with 100k, 200k and 300k rounds of iterations.

Overall performance. *Dike-DQN* and *Dike-PPO* yield a performance for composite cost within a factor of 1.19 and 1.08 compared to the best performance of Midaco while the scheduling time is 99% less. This result justifies that the proposed *Dike* algorithms can efficiently approximate the results of Midaco solver in a time-efficient manner.

Distribution of composite cost. We first analyze the composite cost distribution over all requests as illustrated in Figure 2. The composite cost is the sum of switching cost, operating cost and transmission cost in monetary form. It is not surprising that Midaco 300k achieves the best performance at $2521.91 for the 99th percentile, while *Dike-DQN* and

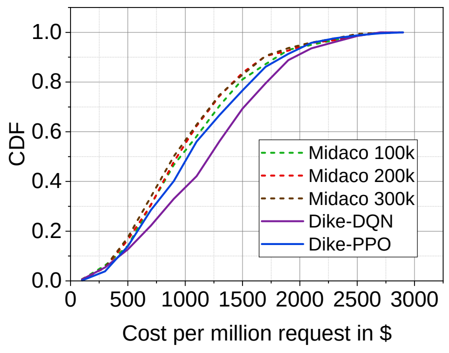

Fig. 2: CDF of composite cost

Dike-PPO achieve $2665.66 and $2628.82, respectively. The average composite cost of Midaco ranges from $1141.53 to $1089.76 when the iteration increases from 100k to 300k rounds. In particular, when the number of iterations increases from 200k to 300k rounds, the performance is only improved by 1.79%, indicating that Midaco has approximately converged at 300k rounds. The average cost of *Dike-DQN* and *Dike-PPO* are $1302.91 and $1184.34, respectively. *Dike-PPO* performs better due to the fact that PPO is a stable, policy-driven method that is more effective for large-scale samples.

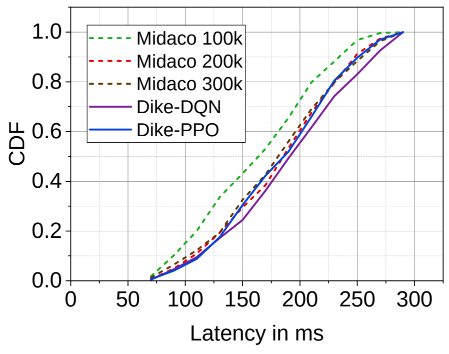

Fig. 3: CDF of end-to-end latency

End-to-end latency. Figure 3 reports the latency distributions of all requests in the simulation. Midaco 100k achieves the best performance with a 99th percentile latency of 269.21 ms, as it sacrifices composite cost for latency. The 99th percentile latency of Midaco 200k and 300k is 345.01 ms and 336.31 ms while that of *Dike-DQN* and *Dike-PPO* is 356.56 ms and 328.26 ms, respectively These four algorithms show similar performance.

SLO violation. The percentage of SLO violations is the ratio between the number of requests that violate the SLO target and the total number of requests. As shown in Figure 4, *Dike-PPO* yields the lowest violation percentage at 0.8% followed by *Dike-DQN* at 3.6%. For Midaco, the percentage of SLO violations ranges from 9% to 11.8%. This is not surprising, as *Dike* can better explore the solution space during the offline training phase.

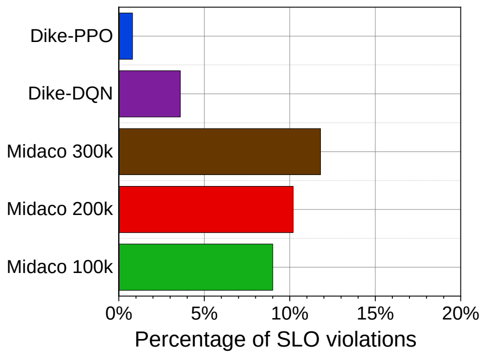

Fig. 4: SLO violations

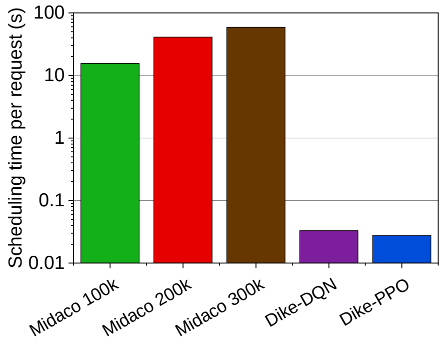

Fig. 5: Scheduling time

Scheduling time. Figure 5 illustrates the scheduling time per request for all the methods, i.e., the time needed to make a scheduling decision. For *Dike*, this is the inference time. The scheduling time of *Dike-DQN* and *Dike-PPO* are 0.03 and 0.02 seconds, respectively. In contrast, the scheduling time of Midaco is in the range of 15.6 to 59.1 seconds. This is because Midaco needs to iterate over a large solution space for convergence. In contrast, *Dike* employs an agent to interact with the solution space and learns the near-optimal policy in the training phase, enabling *Dike* to make decisions efficiently in an online manner.

VI. Conclusion

In this work, we investigated the function scheduling problem with SLO targets for serverless computing at the edge, aiming to reduce monetary costs while satisfying end-user experience. We proposed two different deep reinforcement learning algorithms based on deep Q-networks and proximal policy optimization to achieve efficient online function scheduling. Extensive simulation results indicate that *Dike* yields superior performance for composite cost within a factor of 1.08 compared to the ILP solver Midaco. The scheduling time of *Dike* is 99% less compared to Midaco, showing that *Dike* is well-suited for online scheduling.

Acknowledgment

This work has received funding from the European Union's Horizon Europe research and innovation programme under the Grant Agreement No 101092950 (EDGELESS).

References

[1] Utkalika Satapathy, Rishabh Thakur, Subhrendu Chattopadhyay, and Sandip Chakraborty. Disprotrack: Distributed provenance tracking over serverless applications. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[2] Chen Chen, Manuel Herrera, Ge Zheng, Liqiao Xia, Zhengyang Ling, and Jiangtao Wang. Cross-edge orchestration of serverless functions with probabilistic caching. *IEEE Transactions on Services Computing*, 17(5):2139–2150, 2024.

[3] Yunkai Liang, Zhi Zhou, and Xu Chen. Edgeorcher: Predictive function orchestration for serverless-based edge native applications. In *2023 IEEE ICDCS*, pages 1–2, 2023.

[4] Peiyuan Guan, Chen Chen, Ziru Chen, Lin X. Cai, Xing Hao, and Amir Taherkordi. Context-aware container orchestration in serverless edge computing. In *GLOBECOM 2024 - 2024 IEEE Global Communications Conference*, pages 1041–1046, 2024.

[5] Chen Chen, Lars Nagel, Lin Cui, and Fung Po Tso. S-cache: Function caching for serverless edge computing. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '23, page 1–6, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Yanying Lin, Yanbo Li, Shijie Peng, Yingfei Tang, Shutian Luo, Haiying Shen, Chengzhong Xu, and Kejiang Ye. Quart: Latency-aware faas system for pipelining large model inference. In *2024 IEEE ICDCS*, pages 1–12, 2024.

[7] Rong Gu, Xiaofei Chen, Haipeng Dai, Shulin Wang, Zhaokang Wang, Yaofeng Tu, Yihua Huang, and Guihai Chen. Time and cost-efficient cloud data transmission based on serverless computing compression. In *IEEE INFOCOM 2023*, pages 1–10, 2023.

[8] Xiaofei Yue, Song Yang, Liehuang Zhu, Stojan Trajanovski, and Xiaoming Fu. Demeter: Fine-grained function orchestration for geo-distributed serverless analytics. In *IEEE INFOCOM 2024*, pages 2498–2507, 2024.

[9] Zichuan Xu, Yuexin Fu, Qiufen Xia, and Hao Li. Enabling age-aware big data analytics in serverless edge clouds. In *IEEE INFOCOM 2023*, pages 1–10, 2023.

[10] Yushi Liu, Shixuan Sun, Zijun Li, Quan Chen, Sen Gao, Bingsheng He, Chao Li, and Minyi Guo. Faasgraph: Enabling scalable, efficient, and cost-effective graph processing with serverless computing. In *ASPLOS '24*, page 385–400, New York, NY, USA, 2024.

[11] Zhaorui Wu, Yuhui Deng, Yi Zhou, Jie Li, and Shujie Pang. Faasbatch: Enhancing the efficiency of serverless computing by batching and expanding functions. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 372–382, 2023.

[12] Midaco-solver. https://www.midaco-solver.com/, 2025.

[13] Yuepeng Li, Deze Zeng, Lin Gu, Mingwei Ou, and Quan Chen. On efficient zygote container planning toward fast function startup in serverless edge cloud. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–9, 2023.

[14] Feridun Tütüncüoğlu, Ayoub Ben-Ameur, György Dán, Andrea Araldo, and Tijani Chahed. Dynamic time-of-use pricing for serverless edge computing with generalized hidden parameter markov decision processes. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 668–679, 2024.

[15] Haosong Peng, Yufeng Zhan, Peng Li, and Yuanqing Xia. Tangram: High-resolution video analytics on serverless platform with slo-aware batching. In *2024 IEEE ICDCS*, pages 645–655, 2024.

[16] Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In *2021 IEEE/ACM CCGrid*, pages 797–803, 2021.

[17] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[18] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *2023 ACM Symposium on Cloud Computing*, page 443–458. Association for Computing Machinery, 2023.

[19] Phu Lai, Qiang He, Mohamed Abdelrazek, Feifei Chen, John Hosking, John Grundy, and Yun Yang. Optimal edge user allocation in edge computing with variable sized vector bin packing. In *Service-Oriented Computing*, pages 230–245. Springer International Publishing, 2018.