Optimizing Container Caching in Serverless Edge Computing with Scalable Reinforcement Learning

Manikanta Sanjay Veera, Faiz Sameer Ahmed, and Genya Ishigaki
Department of Computer Science, San José State University, San José, California 95112, USA
Email: genya.ishigaki@sjsu.edu

***Abstract*—Serverless edge computing realizes low latency and resource-efficient function calls for responsive computing. In cloud-based serverless computing, it is a common practice to cache sufficiently many function containers for future reuse to reduce the overhead of container initiation. In contrast, the capacity limitation of edge nodes poses a complex problem of cache selection in serverless edge computing. In this paper, we propose a scalable container caching agent based on Deep Reinforcement Learning (DRL) to improve the training efficiency and caching performance under dynamic request arrivals. We propose to apply action masking, which eliminates inauspicious caching actions from the DRL exploration and concentrates computing resources on more promising actions, to achieve scalability. Our proposal is evaluated through comparative analysis in simulations with both real and synthetic datasets in terms of the latency and cache efficiency. Our results suggest that DRL with action masking can efficiently solve the combinatorial optimization of container caching.**

***Index Terms*—Serverless Edge Computing, Container Caching, Deep Reinforcement Learning, Scalable Learning.**

I. Introduction

The Edge-to-Cloud Continuum (ECC) interprets the entire data path connecting different computing resources at the network edge and on the cloud as a seamless computing facility [1]. By drastically reducing the propagation delay with the use of edge servers, applications within the ECC enjoy the dual advantages of the low latency of edge computing and the substantial computational power of the cloud.

The ECC also promotes the Function-as-a-Service (FaaS) model, where fine-granularity stateless tasks are executed by multiple workers connected through a network to complete an overall computing task [2]. *Serverless* computing, which supports flexible deployment and mobility of function containers, is a powerful enabler of FaaS platforms. With serverless computing, containers can be deployed anywhere on a data path to optimize the request response time. Since the initialization of the containers involves resource preparation and startup processes, known as the *cold start delay* [3], modern serverless cloud platforms cache container instances for reuse, a strategy called a *warm start* [4].

The scope of this paper is the resource allocation problem of container caching for serverless computing, specifically in the ECC. The limitation in computing resources at the edge servers, which was not present in the serverless cloud computing context, poses a unique resource allocation challenge of selecting an appropriate set of containers to be cached. An effective caching needs to consider both the propagation delay, which is determined by the relative container location on a data path, and the container instantiation delay, which directly depends on the availability of caches at each node.

Recent studies have shown the effectiveness of Deep Reinforcement Learning (DRL) in container cache management. Jeon et al. [5] show that a DRL algorithm, Multi-Agent Deep Deterministic Policy Gradient, surpasses traditional caching methods like FIFO, LFU, and LRU in increasing warm-started containers. This demonstrates DRL's capability to adapt to request patterns and formulate complex caching policies. Our previous work [6] focuses on the explainability of such DRL algorithms for container caching. In particular, we demonstrated that DRL actions that promote the reuse of cached containers can be identified by revealing the causal relationship between caching decisions and the associated rewards. However, as noted in our previous work, DRL approaches such as [5], [6] can face scalability challenges due to the combinatorial action space, which exponentially grows with the number of edge servers in the ECC. The work in [7] avoids the scalability issue by defining caching actions in a progressive manner, where the cached containers at each node are updated only by increments or decrements, relative to the current cache. However, this approach has the drawback of delaying responses to changes in the demand distribution, particularly in dynamic environments.

Hence, this paper proposes a *scalable* DRL-based caching method for serverless computing in the ECC, combining the action masking technique [8] and demand tracking. Our DRL agent with action masking focuses exploration on more effective actions by progressively shrinking the action space according to the observed demand distribution. This scalable property makes our proposal suitable, especially for the ECC settings with a large action space. Evaluation experiments based on real-world function invocation traces demonstrate that our DRL agent with action masking outperforms the same DRL approach without action masking and other common caching methods in the overall delay metric and the learning scalability. This implies our approach's capability of balancing the propagation delay and instantiation delay through optimal cache selection.

II. The Edge-to-Cloud Continuum

A. System Model

We assume a standard Edge-to-Cloud Continuum model, which consists of three types of nodes. The first tier en-

compasses user devices, which are located at the network's periphery and generate function requests. The second tier is formed by edge nodes, which are computing nodes situated in close proximity to the user devices. These edge nodes are interconnected and may also be linked to a cloud node. Each edge node possesses a certain cache capacity, enabling it to store a limited variety of container types. These containers can service specific function requests. For the sake of explanation in this paper, we name edge nodes that directly receive the requests as entry edge nodes. The third tier is made up of cloud nodes, which are characterized by their unlimited cache capacity and the ability to house all types of containers. When edge nodes are unable to process requests, the requests are rerouted to a cloud node.

B. Function Requests and Random Redistribution

We follow a request characterization and arrival process, commonly used in related work such as [4]. Each request $R_t^{(r)}$ is characterized by three metrics: function type t, maxhop, and arrival time step r. The function type t is the demanded function (or container) type. The maxhop represents the maximum number of hops a request can traverse through edge nodes prior to being rerouted to the cloud [9]. The arrival time step r indicates when this request arrived at the serverless edge computing system. We adopt a discrete time step during which the same number of requests arrive at the system.

We adopt a random request redistribution strategy similar to the one described in [10] for load balancing across edge nodes. At an entry node, a request is served through *a warm start* if the cache of requested type t is available; otherwise, it is forwarded to a neighbor with the hop limit reduced. At a non-entry node, the request is fulfilled by a *warm start with request distribution* if the cache exists, or forwarded to a neighbor when maxhop > 0. If maxhop $= 0$, the node either performs a *cold start* if the local capacity is available or resorts to *cloud forwarding* when the capacity is insufficient.

III. The Container Caching Problem

The primary goal of our container caching is to reduce the overall delay experienced from the moment a function request is made until it is matched with the appropriate container on a computing node. When $P(R_t^{(r)})$ represents a path (a sequence of communication links) that a request $R_t^{(r)}$ travels, the propagation delay pd of the request can be computed as the sum of propagation delays at each communication link on the path: $pd(R_t^{(r)}) = \sum_{e \in P(R_t^{(r)})} w(e)$, where $w(e)$ shows the propagation delay of a given link e. As the request propagation path will be determined by the availability of caches at edge nodes closer to the entry node that receives the request, the propagation delay is dependent on caching. If the necessary caches are not present at earlier nodes, the request's propagation path might be extended, possibly reaching the cloud node. Hence, in order to reduce the propagation delay, an appropriate set of container caches should be maintained, capturing the request distribution by container types.

The container instantiation delay cd of a request $R_t^{(r)}$ indicates how long it takes to prepare a container specified by the request at a computing node. When the corresponding cache is available, the instantiation delay is negligible, while the cold start incurs an extra delay: $cd(R_t^{(r)}) = 0$ if the cache is available; otherwise, $D \in \mathbb{N}$, a constant defined relative to the warm start. ($D = 1$ in our experiments.) The container instantiation delay again depends on caching, since a cold start is slower than a warm start. At the same time, a naive increase of caches may harm the flexibility of spinning up demanded containers and increase the forwarding to the cloud.

Formally, the container caching problem can be defined as the minimization problem of the sum of the two types of delays with a scaling constant α:

$$\text{Minimize} \sum_{R_t^{(r)}} \left(pd(R_t^{(r)}) + \alpha \cdot cd(R_t^{(r)}) \right). \tag{1}$$

IV. Deep Reinforcement Learning with Action Masking for Container Caching

A. DQN Algorithm

The Deep Q-Network (DQN) algorithm approximates the action value function with a neural network, enabling agents to learn optimal policies even in a complex environment. Also, the DQN's discrete action space aligns naturally with the action masking mechanism, as discussed in the following section. Our container caching problem is formally represented by a Markov Decision Process as follows.

State Space S: A state at each time step r, denoted by $V_s^{(r)} \in S$, is a two-dimensional vector that represents the number of container caches of each function type t at a node $n \in N$. i.e., $V_s^{(r)}[n][t]$ shows a natural number representing the number of available containers when giving a specific node n and a function type t of interest.

Action Space A: Similar to the state vector, an action vector $V_a^{(r)} \in A$ at each time step r contains decisions regarding the number of cached containers indexed by a pair of node and function type. The most important constraint that restrains caching decisions is the node capacity constraint:

$$\sum_{t \in T} V_a^{(r)}[n][t] \leq C(n) \quad \forall n \in N, \tag{2}$$

where $C(n)$ is the capacity (or the maximum number of containers cached) of an edge node n. Although an action vector resembles the state vector, it defines the agent's intended cache allocation for the next time step.

Reward: The reward function is engineered to incentivize the agent to maximize warm start occurrences, thereby reducing both propagation and instantiation delays. A significant reward is allotted for warm starts at the entry node, while other scenarios are evaluated based on the inverse of the sum of propagation and instantiation delays, as follows. This differential rewarding scheme supports the desirability of warm starts, guiding the agent toward a more effective caching strategy.

$$Reward^{(r)}(R_t^{(r)}) = \begin{cases} U & \text{if warm started,} \\ \frac{1}{dp(R_t^{(r)}) + dc(R_t^{(r)})} & \text{otherwise,} \end{cases} \tag{3}$$

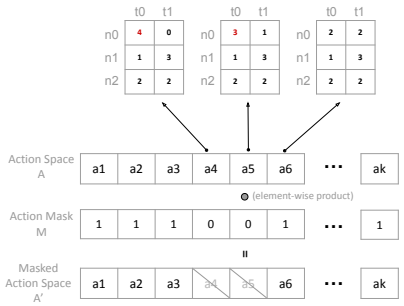

Fig. 1. Demand-based Action Masking (Simplified): $n0$ did not receive many requests for the Type-0 (t0) function; thus, a4 and a5 with many t0 caches at $n0$ will be excluded from the exploration.

where U is a constant greater than 1.0, chosen to exceed the maximum non-warm start reward capped at 1.0. ($U = 2$ in our experiments.) The reward diminishes in scenarios where either the propagation delay $pd(R_t^{(r)})$ or the container instantiation delay $cd(R_t^{(r)})$ increases. This incentivizes the agent to prioritize the request processing at an edge node and further promote warm starts. The total reward at each time step r, $Reward^{(r)}$, is then computed as the sum of rewards for all requests at the time step: $Reward^{(r)} = \sum_{R_t^{(r)}} Reward^{(r)}(R_t^{(r)})$. It should be noted that when a request is accommodated via a warm start, the instantiation delay $dc(R_t^{(r)})$ is markedly reduced compared to that of a cold start in practice.

B. Action Masking

Action masking is a technique to concentrate exploration resources (computing and training time) on meaningful actions during an agent's interactions with an RL environment [8]. In our context, we can apply the masking to two categories of actions. The first category is the prohibited actions in terms of the capacity constraint, defined in Eq. (2). We can explicitly exclude the actions that cache containers more than the node capacity. The second category is the actions that do not produce promising decisions. While these actions are valid in terms of the constraints, they never produce larger immediate rewards. As the exclusion of such actions needs extra attention, we propose a demand-based heuristic for careful, progressive action masking to avoid over-pruning.

1) Demand-based Action Masking: We propose a demand-based action masking where the caching actions misaligned with the demand trend are excluded from the DRL exploration based on the number of requests for each type of function, as shown in Fig. 1.

Algorithmically, each node records the number of arrived requests for each type (e.g., every $k = 10$ time steps). The collective demand state can be represented as a two-dimensional vector $V_d^{(r)}$ similar to the DRL state, in which each value in the vector represents the number of arrived requests for a function type at a node. For the safe and progressive reduction of the action space, we focus on two extreme cases: the minimum and maximum demands: $(t_{\max}, n_{\max}) := \max_{(n,t)\in N\times T} V_d^{(r)}[n][t]$, and $(t_{\min}, n_{\min}) := \min_{(n,t)\in N\times T} V_d^{(r)}[n][t]$, where N is a set of all computing nodes, and T is a set of all function container types available.

In an action space A representing a list of all possible action vectors, let each element $A[i]$ denote one specific caching decision (a two-dimensional vector), which can be selected as an action $V_a^{(r)}$. We construct a mask vector, which has the exactly same shape as the list A of action vectors to indicate which actions to be hidden from the DRL exploration. This construction is done with (i) the minimum and maximum indices and (ii) two hyperparameter thresholds (minThresh and maxThresh). The thresholds are always relative to the capacity of a node and control how radically the action space should be updated. For example, when we want to reduce the action space by 20% for a certain pair of node n and type t that experienced either one of the two extreme demand cases, minThresh and maxThresh can be computed as follows: maxThresh $:= 0.2 \cdot C(n)$ and minThresh $:= C(n) - 0.2 \cdot C(n)$, where $C(n)$ is the capacity of a given node n.

With these threshold values, each element $M^{(r)}[i]$ of our mask at a time step r is constructed as follows:

$$M^{(r)}[i] = \begin{cases} 0 & \text{if } A[i][n_{\max}][t_{\max}] < \text{maxThresh} \\ 0 & \text{if } A[i][n_{\min}][t_{\min}] > \text{minThresh} \\ 1 & \text{otherwise} \end{cases} . \quad (4)$$

Simply put, for the node $n_{\max}$ that experienced excessive demand for the type $t_{\max}$, there will be no need to try a smaller number of caches of $t_{\max}$. Therefore, the number less than the maxThresh value (0 to (maxThresh $-$ 1)) will be excluded from the exploration through the masking. Symmetrically, we set the mask to the node-type pair that experienced minimal demand with minThresh. At the next time step $r+1$ onwards, a DRL action will be selected from the masked action list: $A \circ \sum_r M^{(r)}$, where $\circ$ is the element-wise product. Note that A is an immutable list of all actions, and the masked actions in the past will not be recovered for future exploration.

2) Frequency Adjustment for Action Masking: The frequency of applying the demand-based action mask is a crucial parameter to prevent the masking from mistakenly excluding promising actions due to a lack of demand statistics. We introduce a linear decaying function with a lower bound to start the masking slowly and increase the masking frequency along with the accumulation of the demand statistics. Formally, the action masking is applied to a time step r if and only if $r \mod F(r)$ is 0, where $F(r) = \tau_{\text{prune}} \max\left(1.0 - \frac{r}{\tau_{\text{prune}}}, 0.5\right)$ with a hyperparameter τ_{prune} that determines the speed of the decay. Setting a larger value to τ_{prune} slows down the decay and makes the masking less frequent. In our experiment, we use $\tau_{\text{prune}} = 100$, which gives the minimum frequency bound of one masking in every 50 time steps.

V. Evaluation Experiment

A. Experimental Setup

We perform experiments on an exemplary network topology with five nodes. The simple topology does not necessarily represent practical system scenarios; however, the purpose of

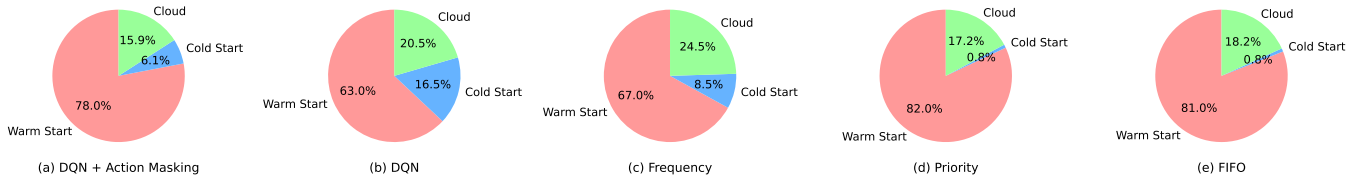

Fig. 2. Distribution of Request Processing Methods (Warm/Cold Starts and Cloud Forwarding) By Different Caching Methods With The Azure Trace 1 Data.

our experiments is to verify the scalability of our DRL agent with action masking in comparison with a traditional DRL model and other common caching methods. The topology consists of two entry edge nodes, three intermediate edge nodes, and a cloud node that is connected to the intermediate edge nodes. An entry edge node is connected to two of the intermediate edge nodes and forwards the received requests to the intermediate when it cannot process them. The capacity $C(n)$ of each edge node n is set to 4. Requests are assumed to be generated by pseudo devices connected to the two entry edge nodes. Also, the edge weights representing the propagation delay are set to 1 and 50 for a connection among edge nodes and to a cloud node, respectively. Based on the ECC model, we assume the forwarding to the cloud incurs a longer delay than the propagation delay among edge nodes.

The DQN agent is equipped with a neural network (NN) comprising two hidden layers (300 and 240 neurons) with the ReLU activation and the output layer with the linear activation. The NN is optimized in terms of MSE by the Adam optimizer over 450 episodes. Using the replay buffer, the agent stores experiences and samples a mini-batch of 64 experiences from the buffer to update its NN, which reduces the correlation among the sampled experiences. With hyperparameter tuning, we set the RL learning rate to 0.005 and the discount factor γ to 0.99. Also, the epsilon-greedy algorithm is used for exploration with the initial ϵ value of 1.0, which is decayed by a rate of 0.998 for each episode until $\epsilon = 0.001$.

The request generation process is simulated based on a real workload trace dataset on Microsoft Azure functions [11] and synthetic static request distribution. In all experiments, 20 random requests for two types of functions (Type-0 and Type-1) are generated at every time step based on the trace or distribution. The Azure trace provides a record of function invocations over 24 hours. We preprocessed the original data to create two datasets (Azure Trace 1 and 2) so that each dataset represents the dynamic request distribution of the original dataset at every simulation time step. The **Azure Trace 1** dataset consists of requests for functions Type-0 and Type-1, where Type-0 is requested 26% more than Type-1 on average. In **Azure Trace 2**, Type-0 is requested 16% more on average.

The synthetic experiment was conducted with a static request generation process, which enables us to analytically estimate the performance of RL algorithms. We used a random distribution where requests for Type-0 and Type-1 are generated by the probabilities of 0.1 and 0.9, respectively. The maxhop of each request is set to 2 since the topology is small. The generated requests are forwarded to an entry node uniformly randomly.

TABLE I
AVERAGE DELAY EXPERIENCED BY A REQUEST WITH THE AZURE DATA.

	Azure Trace 1	Azure Trace 2
DQN + Action Masking	**6.781**	**9.292**
DQN	14.727	11.7450
Frequency	13.339	14.757
Priority	9.645	10.531
FIFO	10.740	9.995

All experiments were conducted in a Python simulator on a Linux server with an Intel i9-13900KF CPU (24 cores), NVIDIA GeForce RTX 4080, and 64GB of RAM.

B. Comparison

We compare our RL approach with the original implementation of DQN without the masking and other three common caching algorithms. The DQN hyperparameters are set to the same values as the ones used for our DQN agent, as summarized in Section V-A. **Frequency**-based caching is a caching approach that adjusts the distribution of caches to match the request arrival distribution. **Priority**-based caching [12] considers the recency and frequency of function invocations, cold start time, and the size of containers. **FIFO**-based caching is commonly used in AWS Lambda [4]. It replaces older container caches based on recency in a FIFO manner.

C. Experiment Results

Table I summarizes the average request delay with the Azure traces. Our DQN agent with action masking outperformed other methods, reducing the overall delay. Fig. 2 illustrates the distribution of the ways that requests based on the Azure Trace 1 were processed. Comparing (a) DQN with action masking and (b) original DQN, the action masking helps the agent improve the caching by reducing both cloud forwarding and cold starts. Similar observations can be found in the comparison to (c) Frequency-based algorithm. A nontrivial result can be found in the comparison to (d) Priority or (e) FIFO-based algorithms. Both the Priority and FIFO succeeded in processing requests with more warm starts than our DQN. However, DQN with action masking realized lower delays than the delays of the two methods (See Table I). This is because DQN with action masking has the lowest usage of cloud forwarding with a larger propagation delay. When the propagation delay to the cloud is considerably larger than

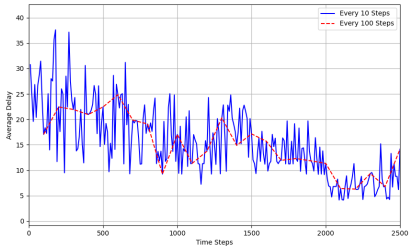

Fig. 3. The Smoothed Total Request Delay of the DQN Agent with Action Masking (Synthetic Data).

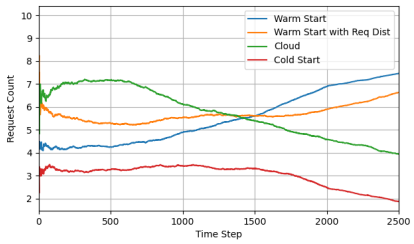

Fig. 4. The Number of Requests that Experienced Each Type of Request Processing (Synthetic Data).

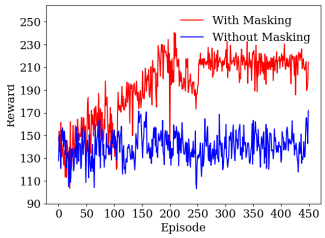

Fig. 5. Rewards with/without Action Masking (Synthetic Data).

the container instantiation delay (related to cold starts), it is reasonable to set aside some spare capacity for cold starts and enable the instantiation of requested functions to avoid cloud forwarding. Our experiment shows that our agent learns the balance between warm and cold starts better than the other. Our DQN requires about 2,900 seconds (≈48 minutes) for training, but once deployed, it can make caching decisions in less than a second, comparable to other methods.

Fig. 3 illustrates the delay of our DQN agent with the static dataset. While the average delay fluctuates due to the random request generation, the overall trend indicates a reduction in the delay. The average delay decreased from about 25 at the start of learning to between 5 and 15 after learning.

Focusing on individual requests, Fig. 4 shows the breakdown of the total delay as a request distribution among the four possible request processing scenarios (See Section II-B). The trend indicates that our DQN agent with action masking learns an optimal caching decision that increases the number of requests served by warm starts and decreases the use of cold starts or the cloud. In particular, the upward trend of the warm start (blue line) and the downward trend of the cloud (red line) empirically prove the caching improvement.

Finally, we analyze the scalability provided by our action masking based on the comparison between DQN with and without action masking. Fig. 5 illustrates the cumulative rewards obtained by the two DQN agents over 450 episodes with the synthetic request data. Note that the maximization of the reward will lead to the targeted minimization of the delay. Without action masking, the DQN agent failed to obtain meaningful insights from rewards within a small number of episodes. In contrast, our agent with action masking succeeded in continuously improving the reward roughly between episodes 0 and 200. The plot also indicates that the agent with action masking occasionally experienced drastic drops in performance for a short time span (e.g., after episodes 100 and 200). These drops could be attributed to the changes in its action space by the masking. However, the performance drop did not happen after episode 250. As promising actions are correctly preserved in the masked action space, the learning seems to be stabilized along with the policy convergence.

VI. Conclusion

This paper addresses a container caching problem in serverless edge computing. In serverless computing, it is common to cache and reuse running function containers to avoid container initiation delay. However, the limited capacity of edge nodes provides additional constraints to the caching. While DRL approaches have been proposed to devise an optimal caching of containers at edge nodes, our previous work hinted at the scalability issue of the learning approach. We developed an action masking module that suppresses DRL's trials of ineffective caching actions and increases the exploration of more promising caching actions, which correlate with request arrival patterns. Our simulation results show that the action masking improves the scalability of the existing DRL and enables the agent to learn an optimal caching policy with a smaller number of experiences. A future extension includes the implementation of further scalable alternatives, such as factorized action for massively large edge environments.

References

[1] J. Santos et al., "Towards low-latency service delivery in a continuum of virtual resources: State-of-the-art and research directions," *IEEE Comm. Surveys & Tutorials*, vol. 23, pp. 2557–2589, 2021.

[2] C. Cicconetti et al., "A decentralized framework for serverless edge computing in the Internet of Things," *IEEE Transactions on Network and Service Mgmt*, vol. 18, pp. 2166–2180, 2021.

[3] A. Tzenetopoulos et al., "FaaS and curious: Performance implications of serverless functions on edge computing platforms," in *High Performance Computing*, pp. 428–438, Springer International Publishing, 2021.

[4] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pp. 1069–1078, 2022.

[5] H. Jeon, S. Shin, C. Cho, and S. Yoon, "Deep reinforcement learning for QoS-aware package caching in serverless edge computing," in *2021 IEEE Global Comm. Conf. (GLOBECOM)*, pp. 1–6, 2021.

[6] D. Jayaram, S. Jeelani, and G. Ishigaki, "Container caching optimization based on explainable deep reinforcement learning," in *2023 IEEE Global Comm. Conf. (GLOBECOM)*, pp. 7127–7132, 2023.

[7] M. Bensalem, E. Ipek, and A. Jukan, "Scaling serverless functions in edge networks: A reinforcement learning approach," in *2023 IEEE Global Comm. Conf. (GLOBECOM)*, pp. 1777–1782, 2023.

[8] O. Vinyals et al., "StarCraft II: A new challenge for reinforcement learning," arXiv, 1708.04782, 2017.

[9] A. Yousefpour, G. Ishigaki, R. Gour, and J. P. Jue, "On reducing iot service delay via fog offloading," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 998–1010, 2018.

[10] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 386–400, ACM, 2021.

[11] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference*, pp. 205–218, USENIX, July 2020.

[12] C. Chen et al., "S-cache: Function caching for serverless edge computing," in *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '23, p. 1–6, ACM, 2023.