Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach

Parichehr Vahidinia, Bahar Farahani, and Fereidoon Shams Aliee

***Abstract*—Serverless computing has revolutionized the world of cloud-based and event-driven applications with the introduction of Function as a Service (FaaS) as the latest cloud computing model. This computational model increases the level of abstraction from the infrastructure and breaks the program into small units called functions. Thus, it brings benefits, such as ease of development, saving resources, and reducing product launch time for enterprises and developers. Thanks to the scale-to-zero feature of this computational model, idle functions with no traffic will be depreciated from memory. However, this cost-saving approach adversely impacts delay leading to the cold start problem. Unfortunately, the existing solutions to alleviate the cold start delay are not resource efficient as they follow a fixed policy over time. Thereby, this article proposes a novel two-layer adaptive approach to tackle this issue. The first layer utilizes a holistic reinforcement learning algorithm to discover the function invocation patterns over time for determining the best time to keep the containers warm. The second layer is designed based on a long short-term memory (LSTM) to predict the function invocation times in the future to determine the required prewarmed containers. The experimental results on the Openwhisk platform show that the proposed approach reduces the memory consumption by 12.73% and improves the execution invocations on prewarmed containers by 22.65% compared to the Openwhisk platform.**

***Index Terms*—Cold start delay, Function as a Service (FaaS), memory consumption, openwhisk, reinforcement learning, serverless computing, serverless platforms.**

I. Introduction

With the advent of the digital revolution and the entry into the information age in the mid-20th century, traditional organizations changed, e-commerce emerged, and computing and communications were based on the Internet. Therefore, the use of information technology became inevitable, and organizations needed infrastructure, network-based technologies, computing resources, and storage. This path was followed by the advent of client/server architecture and centralized computing. However, implementation costs, human resources, infrastructure maintenance concerns, and growing data volumes led to the idea of cloud computing.

In cloud computing, infrastructure, computing, and storage resources, platforms and services are shared virtually, dynamically, and on-demand over the Internet. Cloud computing has three models of services. These models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [1].

In 2014, Amazon introduced the latest cloud computing model called serverless computing. This computational model reduces operational costs and the complexity of system development, and also increases agility in an enterprise. Because any responsibility for preparing, implementing, and managing infrastructure for product development is on the cloud provider. By reducing these responsibilities, the enterprise can focus on its business. It also allows developers to focus only on writing the core logic of their functions and finally, place them on a serverless platform [2]–[4].

Despite all the benefits of serverless computing, it faces some challenges. These challenges are briefly divided into three categories: 1) system; 2) performance; and 3) software engineering challenges. One of the most critical performance challenges is the cold start delay, which is considered as a delay in preparation of the function execution environment [5]. The most common methods used by popular platforms to reduce cold start delay are using a pool of warm containers, reusing the containers, and calling functions periodically. However, these methods waste resources such as memory, increase cost, and do not have any insight into function invocation patterns over time. In other words, these methods provide fixed mechanisms to reduce cold start delay, and they are not suitable for the dynamic cloud environment.

The convergence of serverless computing and the Internet of Things (IoT), particularly for edge applications, is an emerging paradigm that has attracted researchers' attention in recent years. Many IoT devices are resource constrained by processing and storage capacity as well as power consumption requirements. Serverless computing and the corresponding serverless functions ensure that the number of resources consumed by an application running on a resource-constrained IoT device can be controlled on-demand and dynamically. In addition, despite the scale-to-zero feature, edge device resources such as energy sources are saved. Another main advantage that serverless computing can provide to the edge is the auto scalability feature, as the cloud provider handles load balancing and

Manuscript received 14 November 2021; revised 11 February 2022; accepted 20 March 2022. Date of publication 5 April 2022; date of current version 20 February 2023. *(Corresponding author: Bahar Farahani.)*

Parichehr Vahidinia and Fereidoon Shams Aliee are with the Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran 983969411, Iran.

Bahar Farahani is with the Cyberspace Research Institute, Shahid Beheshti University, Tehran 983969411, Iran (e-mail: b_farahani@sbu.ac.ir).

Digital Object Identifier 10.1109/JIOT.2022.3165127

scalability. Moreover, serverless computing can play an important role in deploying IoT applications. Due to the lightweight properties of serverless functions, they can be easily deployed and placed independently anywhere in the cloud, edge, or fog [6].

Although serverless computing alleviates some of the major challenges of IoT, still these converging technologies suffer from specific limitations such as cold start delay that must be approached holistically. Indeed, real-time interaction, prompt processing, and low-latency response are the primary needs for a wide variety of context-sensitive IoT applications from healthcare to online games. Thereby, tackling the cold start delay is the key to the successful integration of IoT and serverless computing [7].

In this article, in order to reduce cold start delay and consider resource consumption, we propose an intelligent approach that determines the best policy for keeping the containers warm, according to the function invocations over time.

The remainder of this article is organized as follows. In Section II, the basic concepts and features of serverless computing are explained. In Section III, the related works for reducing cold start delay are categorized and compared. Section IV describes the proposed two-layer approach to balance the number of cold start occurrences and memory consumption. Section V presents and discusses the experimental results. Finally, Section VI concludes this article.

II. Serverless Computing

Defining serverless computing can be a bit tricky. However, one way to define it is to consider the amount of control the developer has over resources. The IaaS model is where the developer has the most control over both the application and the cloud operating infrastructure. In the PaaS model, the developer is not aware of the infrastructure, therefore, has no control over it. But has access to a ready platform to manage the entire software stack. Finally, SaaS hosts applications as a service in the cloud, and users across the Internet can use them without installing them on the system. However, serverless computing is somewhere between PaaS and SaaS. As shown in Fig. 1, this computational model increases the granularity of applications. According to it, an application is broken down into independent units called functions and introduces a new service model called Function as a Service (FaaS). In other words, serverless computing is a general concept, and one way to perform it is Function as a Service.

Each function is a small, short lived, stateless, and task-specific piece of code that is executed in response to events. In other words, the functions are event driven. These events include the application interface requests (such as an HTTP request), changes to the database, and events from IoT devices [5], [8], [9].

Serverless functions are deployed on containers. As soon as a function is triggered, a new container is created, and the function with all its dependencies is deployed on it. It should be noted that if several invocations occur simultaneously to execute a function, new instances of the function are created according to the number of requests, and each instance is deployed and executed independently on a separate container. According to Fig. 2, when an invocation occurs from an event source, the following steps must be performed to execute the function [10], [11].

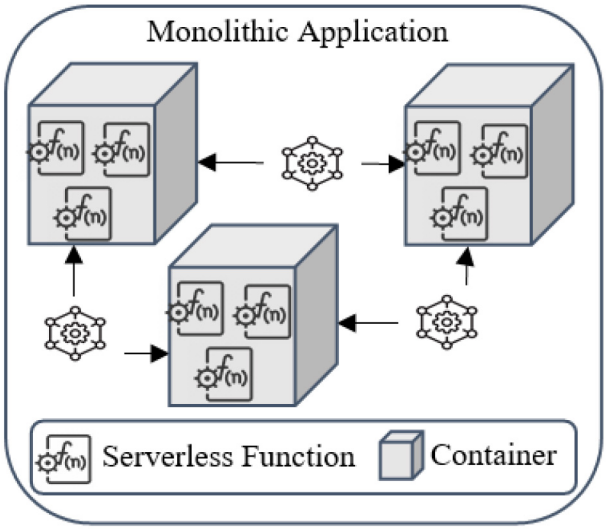

Fig. 1. Increasing the granularity of applications by serverless computing.

1) Initialize a new container that includes preparing the container environment, such as setting environment variables, connecting to the database, user authentication, etc.
2) Allocate memory and computing resources.
3) Load the function and required libraries.
4) Execute the function.
5) Return the results to the user and save the logs in the database.

The first three steps take time and lead to a delay in the overall response time. This delay is called cold start. At the end of the execution, the container with all its resources will be released. This is called scale-to-zero that is an important feature of serverless computing and leads to cost reduction. According to this, users do not pay for times when the functions are idle. On the other hand, it also leads to cold start challenge. Because when new invocations occur, all previous steps must be repeated. On the other hand, if the number of simultaneous invocations exceeds the number of available containers, several instances of a function must be created, and each instance must go through the above steps separately. Because as mentioned, each instance of the function is placed on a separate container. Therefore, the functions have a cold start delay at the first invocation and when the scale increases [10], [12], [13].

III. Related Work

To reduce the cold start delay, two general approaches can be considered. These approaches include: reducing cold start delay and occurrence.

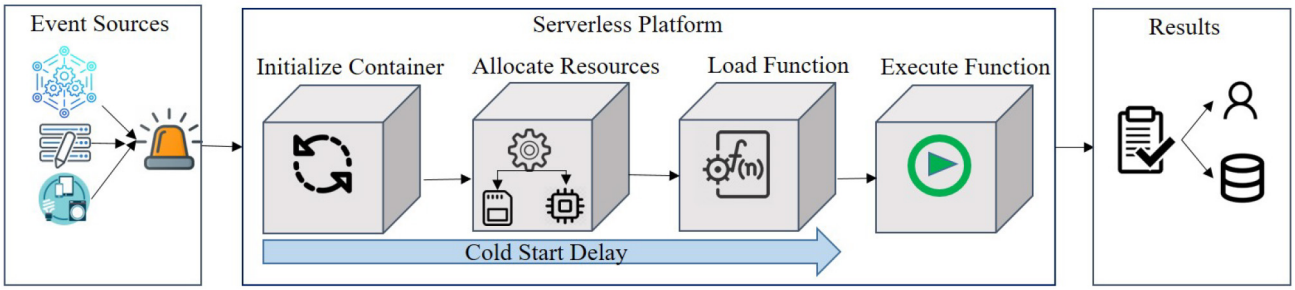

Fig. 2. FaaS execution workflow.

A. Reduce Cold Start Delay

The reduce cold start delay approach includes solutions that focus on reducing the delay in each step of the function execution mentioned in Section I.

1) *Reduce Container Preparation Delay:*
   a) The Openwhisk platform uses prewarmed stem cell containers, which categorizes the containers according to the memory and programming language. With this approach, the preparation time of the container and consequently, the cold start delay are reduced. By default, there are two of them with Node.Js execution environment. Each time one of them is assigned to a request, another prewarmed container is created immediately [14].
   b) Akkus *et al.* [15] reduced container preparation delay by providing a platform called SAND that performs logic separation at the application level. Furthermore, Oakes *et al.* [16] provided a platform called SOCK to deal with performance bottlenecks of Linux kernels used in containers. Moreover, Silva *et al.* [17] introduced a method called Prebaking that reuses the snapshots created for the previous functions.
   c) Xu *et al.* [18] used an adaptive container pool scaling strategy (ACPS). According to it, there is a pool of prelaunched containers that are classified according to different programming languages. Moreover, the number of cold starts for each category is predicted to determine the number of container instances adaptively.
2) *Loading Function Code and Other Dependencies Into Memory:*
   a) The application-level separation method introduced by Akkus *et al.* [15] placed all the functions of an application on the same container. In addition, Oakes *et al.* [16] used the Zygote technique for preimporting required libraries.
   b) To reduce the delay of calling functions by each other, Akkus *et al.* [15] also used a messaging queue to reduce the delay of internal events.

B. Reduce Cold Start Occurrence

The aim of reducing the cold start occurrence approach is to reduce the number of times the function suffers from this delay. For this approach, two general solutions are presented in the literature: 1) prepare function containers and 2) container reuse.

1) *Prepare Function Containers:* To reduce the cold start occurrence, one approach is to prepare a ready container before the request for a function is entered.
   a) Tools, such as CloudWatch [19], Thundra.io [20], Dashbird.io [21], and Lambda Warmer [22], were used to set rules to periodically invoke functions and warm-up them (for example, every 5 min).
   b) Xu *et al.* [18] presented a strategy called Adaptive Warm-UP that uses the long short-term memory (LSTM) network to predict future invocation times. In addition, Gunasekaran *et al.* [23] provide a resource management framework called Fifer that predicts future workloads using the LSTM network.
   c) Agarwal *et al.* [24] used the Q-learning algorithm to determine the number of function instances dynamically. The number of available function instances and the discretized CPU consumption of each function are considered as environmental states, and the action is scaled up or down of function instances.
2) *Container Reuse:* According to this approach, containers of the previous execution are reused for new request.
   a) In some platforms, such as Google Cloud Functions, AWS Lambda, Microsoft Azure, Open Lambda, and Openwhisk, after completing a function execution, the container and its resources are paused for a few minutes. Thus, if a request is received during this period, the container will be reused and it will have a warm start. More specifically, in the openwhisk platform, the container stays running or hot for 50 ms after an execution. After this time, the container and its resources are paused, and after 10 min they are completely released. This time is called the idle-container window. This method can be effective in reducing delay. Because restarting the paused container has less delay than allocating a new container [25].
   b) OpenFaaS gives the developer the option to use an always-running container for a function [26]. Knative and Fission platforms have a pool of warm containers [27], [28]. Lloyd *et al.* [29] also used

an approach called Keep-alive to keep containers warm at all times. McGrath and Brenner [30] used a queue of warm containers per function. In addition, Solaiman and Adnan [31] presented a container management architecture called WLEC that has warm and template queues. The template queue keeps a copy for each warm container.

c) The AWS Lambda platform provides a provisioned concurrency mechanism that the developer can determine the number of required warm function instances for future requests [32].

d) Shahrad *et al.* [33] presented a resource management policy called the Hybrid Histogram Policy that determines the optimal value of the idle-container window according to the characteristics of the functions.

e) Gias and Casale [34] performed capacity planning for serverless functions using a queueing-based approach and considering the effect of cold start delay. Therefore, the optimal idle time of a container is determined by considering the resource consumption.

f) Banaei and Sharifi [35] provided a function scheduler called ETAS, which predicts the execution time of an invocation and assigns the best worker node to it based on the availability of a warm container or the availability of the requested resources.

g) Wu *et al.* [36] proposed a container lifecycle-aware scheduling called CAS. This approach has three strategies that monitor the state of the containers and select the best container to execute the input invocation to prevent the creation of new containers and cold start occurrence.

C. Comparison and Analysis of Related Work

In this section, we analyze the related work in terms of resource consumption and adaptability.

1) *Resource Consumption Analysis:* Resource consumption analysis aims to determine whether reducing cold start delay in a method leads to higher resource consumption. Note that processes such as keeping the containers warm, pausing and reusing them, creating a pool of warm containers, and calling functions waste resources. The reasons are as follows.

a) *Container Reuse:* This method has no threshold to determine the best time to release the container after completing a function. Therefore, if subsequent requests are received long after the container has been released, keeping the container warm will only lead to additional resources and costs.

b) *Pool of Warm Containers:* These containers are always running and occupy memory and computing resources in the presence or absence of requests.

c) *Calling Functions Periodically:* This method does not have any insight into the time and number of future requests. There may not be a request in a while, thus we do not need to restart the function container. On the other hand, the requests may be bursty, and we have not considered enough instances for all of them [37].

2) *Adaptability:* The purpose of adaptability analysis is to determine whether a method for reducing cold start delay considers the pattern of invocations using machine learning algorithms. Because given the dynamics of the cloud environments, the prediction of the function invocations over time cannot be ignored. This proves the importance of using machine learning algorithms. Methods that provide fixed mechanisms to reduce cold start delay may not respond to such a changing environment. Due to the variability of function invocation times and the dynamic nature of the cloud environment, we need a self-adapting strategy that can provide automated decisions by monitoring the execution environment. Therefore, reinforcement learning-based algorithms seem to be suitable for this area. Because they do not need prior knowledge of the environment and learn the optimal action when the system is running. On the other hand, classical reinforcement learning algorithms, such as Q-learning, have a discrete state and action space. However, when it comes to reducing cold start delay, environmental states, such as memory and CPU consumption, and invocation times are continuous; therefore, it is best to use deep reinforcement learning algorithms. A comparison of the methods is given in Table I. Each row of this table represents the features and information of a method. Table I contains five columns. The first column is the reference number of the presented method. The second column includes the name of the platform. The third and fourth columns indicate that the purpose of this method is to reduce the cold start occurrence or reduce the cold start delay. The fifth column compares the methods in terms of resource consumption and adaptability. According to Table I, most methods only either reduce the cold start delay or reduce its occurrence. However, the Openwhisk platform and Xu *et al.* [18] provided methods to reduce the cold start occurrence and delay. On the other hand, according to the efficiency of keeping the containers warm in reducing the cold start delay, there is a need to determine the value of keeping the container warm according to the invocation patterns. Because the advantage of these methods is reusing the existing containers instead of creating new containers.

IV. Proposed Two-Layer Approach

The most common way to reduce cold start delay among popular platforms is to keep the containers warm for a fixed period of time after completing a function. This is called the idle-container window. However, among these platforms, only Openwhisk has provided solutions to reduce both the time and number of cold start delays. However, since the pattern of function invocations varies over time, considering a constant value for this window is not an efficient method. This proves

TABLE I
COMPARISON OF RELATED WORK

Reference	Platform/Method Name	Reduce Container Preparation Delay			Reduce Cold Start Occurrence		Analysis	
		Reduce Container Preparation Delay	Function Chain Calling	Loading Dependencies into Memory	Prepare Function Containers	Container Reuse	Optimal Resource Consumption	Adaptability
[14]	Openwhisk	✓	x	x	x	✓	x	x
[15]	SAND	✓	✓	✓	x	x	✓	x
[16]	SOCK	✓	x	✓	x	x	✓	x
[17]	Prebaking	✓	x	x	x	x	x	x
[18]	Adaptive Warm-up and Adaptive Container Pool Scaling Strategies	✓	x	x	✓	x	✓	✓
[19-22]	CloudWatch, Thundra.io, Dashbird.io, Lambda Warmer	x	x	x	✓	x	x	x
[23]	Fifer	x	x	x	✓	x	✓	✓
[24]	-	x	x	x	✓	x	✓	✓
[25]	Open Lambda	x	x	x	x	✓	x	x
[26]	OpenFaaS	x	x	x	x	✓	x	x
[27]	Knative	x	x	x	x	✓	x	x
[28]	Fission	x	x	x	x	✓	x	x
[29]	Keep-alive	x	x	x	x	✓	x	x
[30]	-	x	x	x	x	✓	x	x
[31]	WLEC	x	x	x	x	✓	x	x
[32]	AWS Lambda	x	x	x	x	✓	x	x
[33]	Hybrid Histogram Policy	x	x	x	x	✓	✓	✓
[34]	COCOA	x	x	x	x	✓	✓	x
[35]	ETAS	x	x	x	x	✓	x	x
[36]	CAS	x	x	x	x	✓	x	x
-	Two-layer Proposed Approach	✓	x	x	x	✓	✓	✓

the importance of using machine learning algorithms to predict future function invocation times. Therefore, the Openwhisk platform method is considered the baseline of this article, and in this section, a two-layer approach is proposed to make the platform adaptable.

The first layer aims to determine the best time for the idle-container window to reduce the number of cold start delays and resource consumption using the reinforcement learning algorithm. Because there is a tradeoff between the number of cold start occurrences and resource consumption. Therefore, the longer this window is, the shorter cold start delays and more consumption of resources.

The purpose of the second layer is to reduce the cold start delay time. When requests are received bursty, there may not be a warm container for each one, and some requests will inevitably be delayed. Therefore, in this layer, the number of requests that will be delayed in the future is predicted using LSTM and based on them, the number of prewarmed containers required in the future is determined. Fig. 3 shows an overview of this approach.

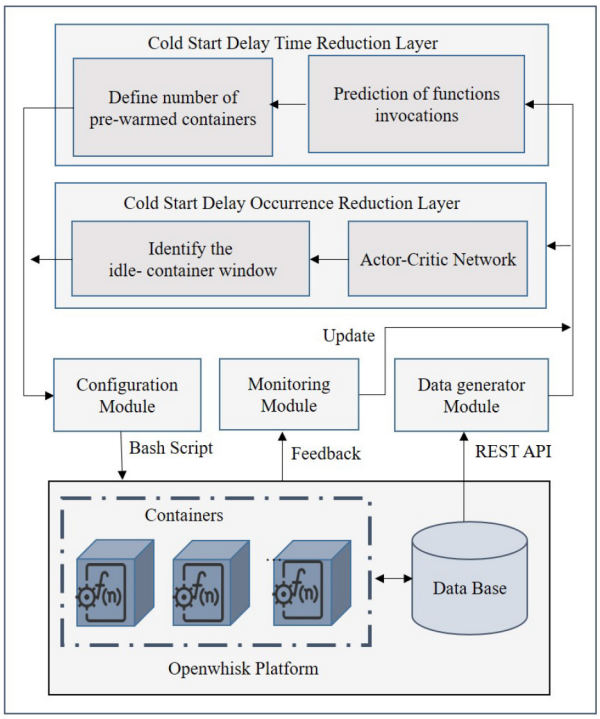

Fig. 3. Two-layer proposed approach.

A. Cold Start Occurrence Reduction Layer

As mentioned, the purpose of this layer is to reduce the number of cold start occurrences over a period of time. Given that the state and action space is continuous, there is a need to receive immediate rewards at each step instead of each episode, and also considering that the approach used to optimize the agent behavior should be on-policy, the temporal difference (TD) advantage actor–critic algorithm, which is a deep reinforcement learning algorithm, is used. Therefore, the intelligent agent learns the invocation pattern of the function by observing the time intervals between invocations in the past and determines the value of the idle-container window in the future. The steps of this algorithm are as follows.

1) *Determine the State of the Environment:* The function execution log is received from the platform and is considered as state space. Therefore, the state space includes the time intervals between the arrival of invocations and whether it had a cold or warm start.
2) *Determine Appropriate Action:* The action is to determine the value of idle-container window. In each step, we give the state as input to the actor neural network. This neural network then returns two values of μ and σ as the mean and standard deviation at the output. μ and σ are functions of state s. Since we are interested in the case where the action is a continuous variable, we used a stochastic policy $\pi = P(a|s)$, where $P(a|s)$ is the probability distribution of taking action a given state s. Thereby, P will be a probability density. We use a Gaussian probability distribution to represent our stochastic policy, where μ is the mean, σ is the standard deviation of the Gaussian (normal) distribution abbreviated as $N(a|\mu, \sigma)$, and a is a continuous action drawn from real numbers. Therefore, the action is randomly selected from a normal distribution with mean μ and standard deviation σ.
3) *Take Action on the Environment:* The serverless computing platform is then updated with a new value for the idle-container window parameter by the reconfiguration module.
4) *Receive Rewards:* At the end of a time period, and based on the information received from the monitoring module, the reward function is calculated based on (1). As mentioned, if the size of the idle-container window is large, the number of delays is reduced. However, resource consumption is increased because the idle container wastes memory. Therefore, the reward function is based on the ratio of the number of cold start delays to total invocations and a penalty based on the amount of memory wastage when keeping the container warm. If the number of delays is high, the wasted memory is reduced. However, the reward is a high negative number. Conversely, if the number of delays is low, the wasted memory will increase, and the reward will still be a high negative number. Hence, this algorithm will balance the number of delays and memory wastes. The number of cold start occurrences, the total number of invocations, and the penalty are indicated by Cold, N, and P, respectively.

$$\text{Reward} = -(\text{Cold})/N - P. \tag{1}$$

B. Cold Start Delay Reduction Layer

A cold start delay is inevitable. Because some invocations occur at long intervals, keeping the containers warm will result in memory wastage. Therefore, some invocations that enter outside the idle-container window will suffer a cold start delay. Moreover, invocations may be occurring simultaneously, and there may not be enough warm containers for all of them. The purpose of this layer is to adapt the number of stem cell containers. This layer predicts the maximum number of concurrent invocations over a period of time, and accordingly, determines the number of prewarmed containers. This is a time-series problem because the number of concurrent invocations in the future depends on the pattern of the previous invocations. Thus, in this approach we will use the LSTM.

V. Evaluation

In this section, the efficiency of the proposed approach is evaluated through the simulations. The following is a description of the experimental setup and performance evaluation.

A. Experimental Setup

This section includes the description of the data set, computing platform, and training setup.

1) *Data Set:* We considered two types of data sets. One includes sequential invocations, and the other includes concurrent invocations. To evaluate the first layer of the proposed approach, a data set containing sequential invocations is required. According to the data set simulation introduced by Gias and Casale [34], we expect invocations to enter based on the Poisson process. According to this process, the distribution of invocations entry time is random and exponentially distributed. We created 10 000 entry times with different 5, 10, and 20 average entry rates of invocations per hour. These entry times represent the time interval between invocations. To evaluate the second layer of the proposed approach, a data set containing concurrent invocations is required. We randomly created 10 000 invocations, including concurrent invocations, to determine how this concurrency affects the idle-container window and the number of prewarmed containers, as well.
2) *Computing Platform:* As mentioned in Section III, the Openwhisk platform method is used as the baseline for this article. The purpose of this article is using the reinforcement learning algorithm to make this method adaptable in order to reduce memory consumption. The Openwhisk platform is implemented on an Ubuntu server by a Kubernetes cluster. Then, the configuration parameters corresponding to the parameters of the proposed approach on the platform should be determined. This platform uses a parameter called idle container to keep the containers warm. However, by default this parameter cannot be configured to the desired value. Therefore, in this step, we first configured this parameter. In addition, this platform has an adjustable configuration file for the number of its prewarmed containers.
3) *Training Setup:* Each layer of the proposed approach has its own hyperparameters. As mentioned in Section IV, in the first layer of the proposed approach, the TD advantage actor–critic algorithm is used to determine the idle-container window. To implement the actor network, three hidden layers have been used. Each layer contains 32 neurons, a ReLU activation function, and a uniform weight function. This network also has two output layers for μ and σ as mean and standard deviation, which have Softplus activation function. Moreover, the critic neural network has two layers with 32 and 16 neurons

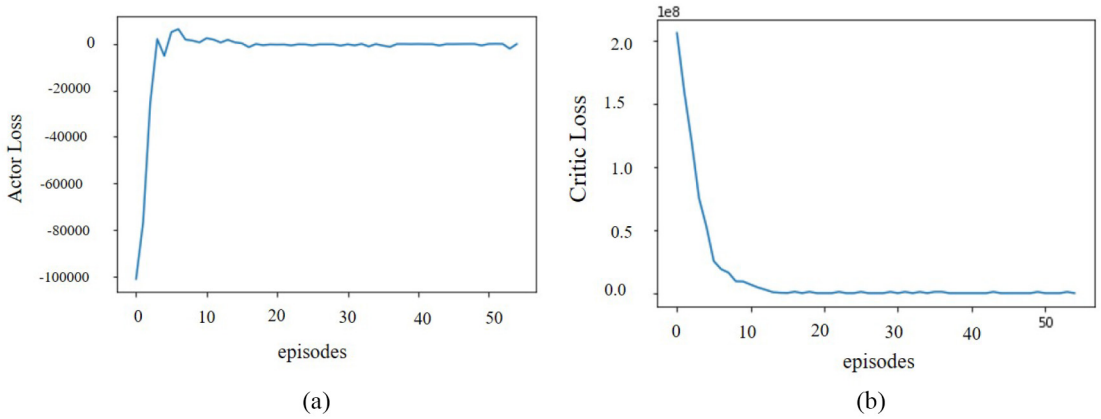

Fig. 4. Training loss functions without regularizer. (a) Actor loss function. (b) Critic loss function.

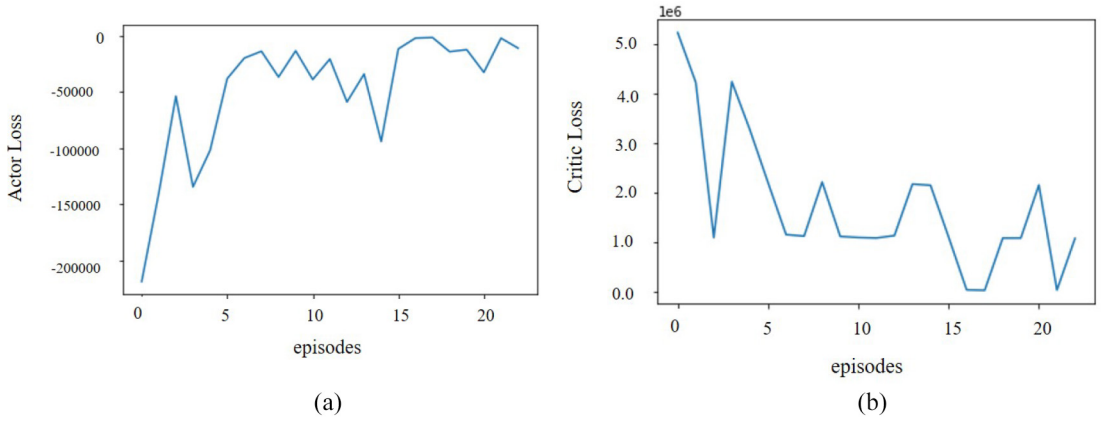

Fig. 5. Training loss functions with regularizer. (a) Actor loss function. (b) Critic loss function.

and a ReLU activation function. The output layer also has a linear activation function to calculate the value of the next state. It should be noted that by adding a regularizer, the training will stop faster, and the loss function will fluctuate. Fig. 4(a) and (b) is the actor loss function and critic loss function for training without regularization, respectively. In addition, Fig. 5(a) and (b) are the actor loss function and critic loss function for training with regularization, respectively. The LSTM network has five hidden layers, each with 32 neurons and a ReLU activation function. In addition, to prevent overfitting, a Dropout layer with a value of 0.5 is used. The network also has one output and a linear activations function. The mean squared error (MSE) is also used as a loss function. Note that the training was done in Google Colab using a TPU v2 node with 8 cores and 64 GB of total memory, and it took 2 h and 20 min. We have considered a monitoring module that activates the trigger to retrain the model if the model accuracy decreases or the data distribution changes. In addition, we can utilize continuous learning to adapt to change.

B. Performance Evaluation

In order to perform evaluation, we used an I/O bound function that sends an HTTP request to a Web page and receives a text file. The response time of this function is about 6 s. Then, according to the simulated intervals, asynchronous requests are sent to the function through the REST API of the Openwhisk platform. The result of each request will be recorded in the Openwhisk platform log service. Two simulations are performed: 1) using the Openwhisk platform with its default parameters and 2) setting new parameters for Openwhisk obtained from the proposed approach. In the second simulation, the results of each layer of the proposed approach are compared to the results of the first simulation. Note that the test data set includes 200 requests.

1) *Cold Start Occurrence Reduction Layer:* Comparison metrics in this layer are the value of the idle-container window, number of cold start occurrences, and memory consumption.
   a) *Idle-Container Window:* The idle-container window of the Openwhisk platform is fixed at 10 min. However, according to the proposed approach, this window has different values for each average entry rate at different time intervals. Fig. 6 compares the idle-container window of the proposed approach and the Openwhisk platform over time for three different average invocations per hour (5, 10, 20). The horizontal axis shows the number of time intervals of 200 invocations according to each average entry rate. For example, when we have 5

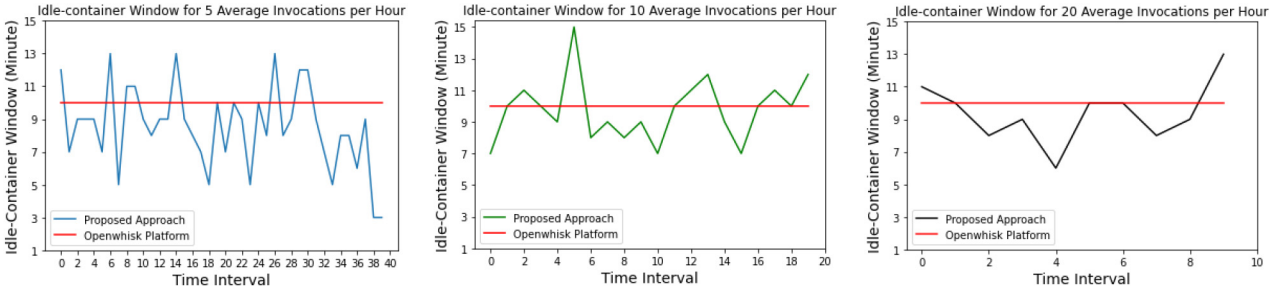

Fig. 6. Idle-container window according to proposed approach and openwhisk platform. (a) Five invocations per hour. (b) Ten invocations per hour. (c) Twenty invocations per hour.

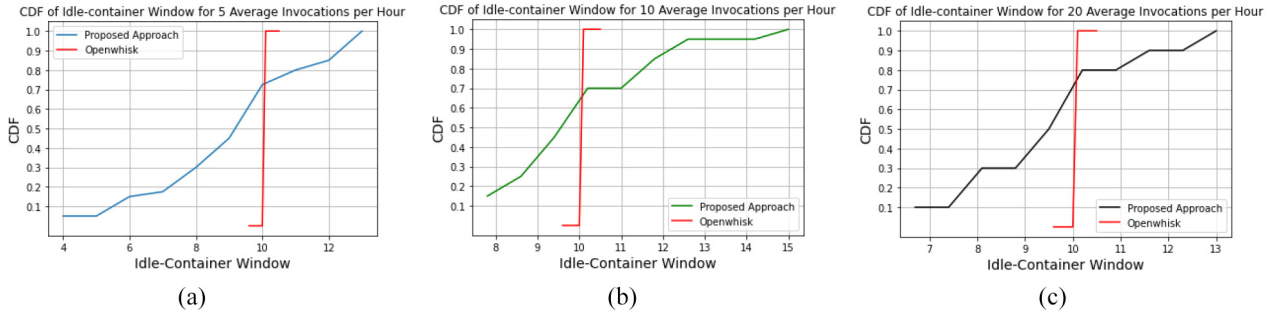

Fig. 7. CDF of idle-container window according to proposed approach and openwhisk platform. (a) Five invocations per hour. (b) Ten invocations per hour. (c) Twenty invocations per hour.

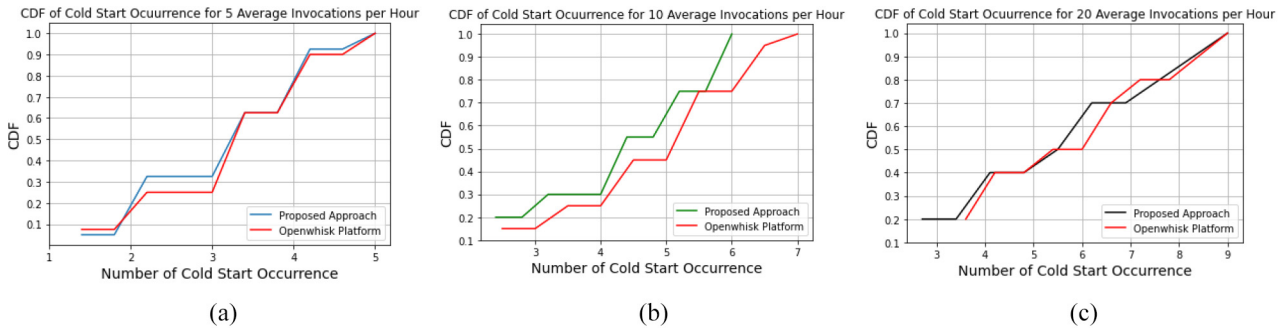

Fig. 8. CDF of number of cold start occurrence according to proposed approach and openwhisk platform. (a) Five invocations per hour (b) Ten invocations per hour. (c) Twenty invocations per hour.

average invocations per hour, 200 requests have 40 time intervals (200/5). Also, the idle-container window varies from 3 to 13 min. It should be noted that, this window varies from 7 to 15 min, and 8 to 13 min for average invocations per hour of 10 and 20, respectively. According to the figures, the idle-container window is determined according to the invocations entry pattern using the proposed reinforcement learning algorithm. Fig. 7 compares the cumulative distribution function (CDF) plot of the idle-container window determined by the proposed approach and the Openwhisk platform for three different average invocations per hour (5, 10, 20). According to Fig. 7(a), 73% of invocations have less than 10 min idle-container window. This value is equal to 67% and 76% for average invocations per hour of 10 and 20, respectively. Therefore, most of the time a container does not need to stay warm for up to 10 min. In order to reduce the cold start occurrence, different values can be considered for the idle-container window depending on the different invocation patterns. It can be interpreted that the proposed approach expands the window for times when the intervals are longer and decreases it when the intervals are shorter.

b) *Number of Cold Start Occurrences:* Fig. 8 compares the CDF of number of the cold start occurrences for the proposed approach and the Openwhisk platform. This simulation is performed for three different average invocations per hour of 5, 10, and 20. As shown in Fig. 8(a), for both CDFs, 75% of invocations have less than four cold start occurrences within 1 h when the average invocations per hour is 5. According to Fig. 8(b),

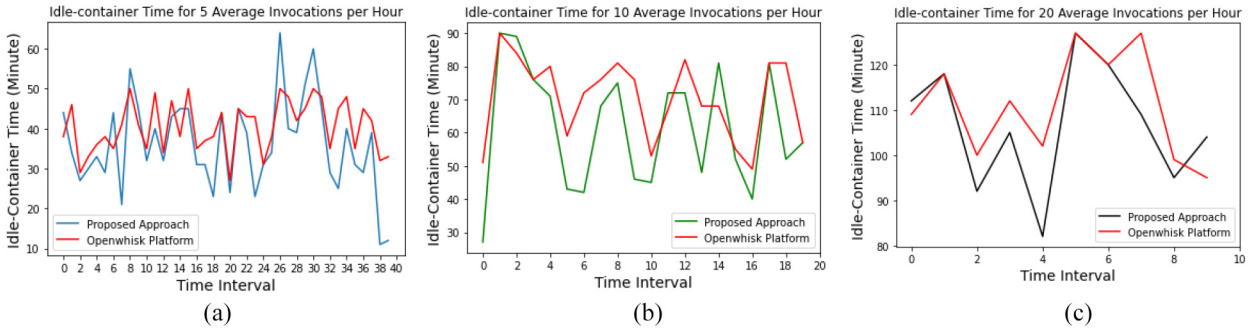

Fig. 9. Idle-container time according to proposed approach and openwhisk platform. (a) Five invocations per hour. (b) Ten invocations per hour. (c) Twenty invocations per hour.

TABLE II
NUMBER OF COLD START OCCURRENCE

Average Invocation Rate per Hour	Proposed Approach	Openwhisk Platform	Memory Improvement
5	123	126	11.11%
10	84	89	12.73%
20	53	55	4.05%

when the average invocations per hour is 10, in the proposed approach, 65% of invocations have less than five cold start occurrences. This value equals to 47% for the Openwhisk platform. Finally, according to Fig. 8(c), for both CDFs, 42% of invocations have less than five cold start occurrences within 1 h when the average invocations per hour is 20. Table II compares the number of cold start delays for the proposed approach and the Openwhisk platform for the average invocations per hour of 5, 10, and 20. As shown in Table II, the proposed approach results in a negligible improvement in the number of cold start occurrences. However, the proposed approach and the Openwhisk platform have a similar impact on reducing cold start occurrences. Note that this result can be concluded from Fig. 7 as well. The improvement in the number of cold start delays is low due to memory control. If we want to reduce delay more than the Openwhisk platform, we have to choose a larger idle-container window. However, this is contrary to the main purpose of this article (i.e., control memory consumption while reducing cold start delay.)

c) *Memory Consumption:* We have compared the memory consumption according to the idle-container time. That is, the length of time interval a container waits until the next invocation arrives. If the next invocation is warm, this time is equal to the idle-container window according to the proposed approach and 10 min according to the Openwhisk platform. Fig. 9 compares the idle-container time of the proposed approach and the Openwhisk platform for average invocations per hour of 5, 10, and 20. As shown in the Table II, our proposed approach has 11.11%, 12.73%, and 4.05% improvements in memory consumption time for average invocation per hour of 5, 10, and 20, respectively.

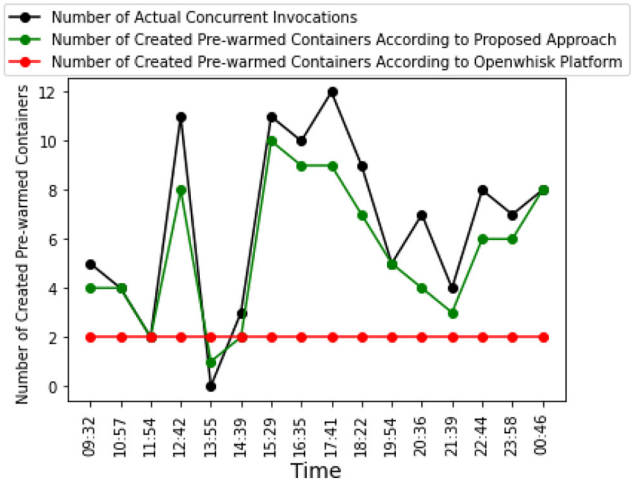

Fig. 10. Number of invocations executed on the prewarmed containers.

2) *Results for Cold Start Delay Reduction Layer:* The purpose of the cold start delay reduction layer is to determine the number of prewarmed containers based on the number of concurrent invocations. Fig. 10 compares the number of invocations executed on the prewarmed containers for the proposed approach and the Openwhisk platform. The test data set has 200 invocations, and the maximum number of concurrent invocations per hour is considered. The data set has a total of 106 concurrent invocations. Based on the results, the proposed approach was able to correctly predict 56 invocations and execute them on the prewarmed containers. However, the Openwhisk platform, which has two prewarmed containers at all of the time, has executed 32 requests on prewarmed containers. Therefore, it is clear that the proposed approach has 22.65% improvement in the execution of invocations on prewarmed containers.

VI. CONCLUSION

Cold start delay is one of the most important challenges in serverless computing. Reducing this delay in real-time applications is critical because it directly affects performance and customer satisfaction. However, the most common solutions used by serverless platforms, such as keeping the containers warm, have a fixed policy and lead to memory waste.

Therefore, we need a method that determines the time required to keep the containers warm by discovering the invocation pattern of the functions. It should also balance reducing cold start latency and memory consumption. To meet these goals, in this article, we propose a two-layer approach. In the first layer, the reinforcement learning algorithm is used to discover the function invocation pattern and determine the idle-container window. In the second layer, according to the prediction of the next invocation time by LSTM, the required number of prewarmed containers is determined. Evaluations prove that it is possible to determine the amount of idle-container window by discovering the pattern of invocations. This reduces the number of cold start occurrences and controls memory consumption. In addition, the number of cold start delays that occurred according to the proposed approach is approximately equal to the Openwhisk platform results. Moreover, the cold start occurrence reduction layer of the proposed approach has 11.11%, 12.73%, and 4.05% improvement in memory consumption time for the average invocations per hour of 5, 10, and 20, respectively. Finally, the cold start delay reduction layer of the proposed approach has 22.65% improvement for execution invocations on prewarmed containers.

REFERENCES

[1] V. V. Arutyunov, "Cloud computing: Its history of development, modern state, and future considerations," *Sci. Techn. Inf. Process.*, vol. 39, no. 3, pp. 173–178, 2012.

[2] N. Kratzke, "A brief history of cloud application architectures," *Appl. Sci.*, vol. 8, no. 8, p. 1368, 2018.

[3] I. Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Singapore: Springer, 2017, pp. 1–20.

[4] M. Sewak and S. Singh, "Winning in the era of serverless computing and function as a service," in *Proc. 3rd Int. Conf. Converg. Technol. (I2CT)*, 2018, pp. 1–5.

[5] E. Van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 21–24.

[6] S. R. Chaudhry, A. Palade, A. Kazmi, and S. Clarke, "Improved QoS at the edge using serverless computing to deploy virtual network functions," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 10673–10683, Oct. 2020.

[7] V. Kjorveziroski, S. Filiposka, and V. Trajkovic, "IoT serverless computing at the edge: Open issues and research direction," *Computers*, vol. 10, p. 130, Dec. 2021.

[8] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uţă, and A. Iosup, "Serverless is more: From PaaS to present cloud computing," *IEEE Internet Comput.*, vol. 22, no. 5, pp. 8–17, Sep./Oct. 2018.

[9] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Gener. Comput. Syst.*, vol. 83, pp. 50–59, Jun. 2018.

[10] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2019, p. 21.

[11] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, 2018, pp. 181–188.

[12] C. Denninnart and M. A. Salehi, "Efficiency in the serverless cloud computing paradigm: A survey study," 2021, *arXiv:2110.06508*.

[13] S. Eismann *et al.*, "Serverless applications: Why, when, and how?" *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, Jan./Feb. 2021.

[14] "Apache/Openwhisk: Apache Openwhisk Is an Open Source Serverless Cloud Platform." Apache. [Online]. Available: https://github.com/apache/openwhisk (Accessed: Oct. 10, 2021).

[15] I. E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Techn. Conf. (USENIX ATC)*, 2018, pp. 923–935.

[16] E. Oakes *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Techn. Conf. (USENIXATC)*, 2018, pp. 57–79.

[17] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proc. 21st Int. Middlew. Conf.*, 2020, pp. 1–13.

[18] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2019, pp. 9–16.

[19] "Amazon CloudWatch—Application and Infrastructure Monitoring." [Online]. Available: https://aws.amazon.com/cloudwatch/ (Accessed: Oct. 10, 2021).

[20] "Thundra | Monitoring for Your CI Pipelines & Tests." [Online]. Available: https://www.thundra.io (Accessed: Oct. 10, 2021).

[21] "Serverless Observability and Intelligence Platform." [Online]. Available: https://dashbird.io/ (Accessed: Oct. 10, 2021).

[22] "A Module to Optimize AWS Lambda Function Cold Starts." [Online]. Available: https://github.com/jeremydaly/lambda-warmer (Accessed: Oct. 10, 2021).

[23] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proc. 21st Int. Middlew. Conf.*, 2020, pp. 280–295.

[24] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proc. IEEE/ACM 21st Int. Symp. Clust. Cloud Internet Comput. (CCGrid)*, 2021, pp. 797–803.

[25] S. Shillaker, "A provider-friendly serverless framework for latency-critical applications," in *Proc. 12th Eurosys Doctoral Workshop*, 2018, p. 71.

[26] "OpenFaaS Makes it Simple to Deploy Both Functions and Existing Code to Kubernetes." [Online]. Available: https://www.openfaas.com/ (Accessed: Oct. 10, 2021).

[27] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," 2019, *arXiv:1903.12221*.

[28] "Open Source, Kubernetes-Native, Serverless Framework." [Online]. Available: https://docs.fission.io/ (Accessed: Oct. 10, 2021).

[29] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving application migration to serverless computing platforms: Latency mitigation with keep-alive workloads," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, 2018, pp. 195–200.

[30] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, 2017, pp. 405–410.

[31] K. Solaiman and M. A. Adnan, "WLEC: A not so cold architecture to mitigate cold start problem in serverless computing," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, 2020, pp. 144–153.

[32] "AWS Lambda Announces Provisioned Concurrency." 2019. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/

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

[34] A. U. Gias and G. Casale, "COCOA: Cold start aware capacity planning for function-as-a-service platforms," in *Proc. 28th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. (MASCOTS)*, 2020, pp. 1–8.

[35] A. Banaei and M. Sharifi, "ETAS: Predictive scheduling of functions on worker nodes of Apache OpenWhisk platform," *J. Supercomput.*, vol. 78, pp. 5358–5393, Sep. 2021.

[36] S. Wu *et al.*, "Container lifecycle-aware scheduling for serverless computing," *Softw. Pract. Exp.*, vol. 52, no. 2, pp. 337–352, 2022.

[37] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *Proc. Int. Conf. on Omni-Layer Intell. Syst. (COINS)*, 2020, pp. 1–7, doi: 10.1109/COINS49042.2020.9191377.

Parichehr Vahidinia received the M.S. degree in information technology from Shahid Beheshti University, Tehran, Iran, in 2021.

Her current research focuses on serverless computing and artificial intelligence.

Bahar Farahani received the Ph.D. degree in computer engineering from the University of Tehran, Tehran, Iran, in 2017.

She was a Postdoctoral Fellow of Computer Engineering with Shahid Beheshti University, Tehran, where she is an Assistant Professor with Cyberspace Research Institute. She authored several peer-reviewed conference/journal papers as well as book chapters on IoT, big data, and AI.

Dr. Farahani has served as a Guest Editor for several journals, such as IEEE INTERNET OF THINGS JOURNAL, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *Future Generation Computer Systems* (Elsevier), *Microprocessors and Microsystems* (Elsevier), *Journal of Network and Computer Applications* (Elsevier), and *Information Systems* (Elsevier). Besides, she has also served on the Technical Program Committee of many international conferences/workshops on AI/IoT/eHealth as well as the Technical Chair of the IEEE COINS Conference.

Fereidoon Shams Aliee received the Ph.D. degree in software engineering from the Department of Computer Science, Manchester University, Manchester, U.K., in 1996, and the M.S. degree from Sharif University of Technology, Tehran, Iran, in 1990.

He is currently a Professor with the Software Engineering Department, Shahid Beheshti University, Tehran, where he is also heading two research groups, namely, Service-Oriented Enterprise Architecture Reference Laboratory and Information Systems Architecture. His major interests are software architecture, enterprise architecture, service-oriented architecture, agile methodologies, ultra-large-scale systems, and ontological engineering.