A Reinforcement Learning Approach to Reduce Serverless Function Cold Start Frequency

Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya

Cloud Computing and Distributed Systems(CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: siddhartha@student.unimelb.edu.au, {maria.read, rbuyya}@unimelb.edu.au

Abstract— **Serverless computing is an event-driven cloud computing architecture for processing requests on-demand, using light weight function containers and a micro-services model. A variety of applications like Internet of Things (IoT) services, edge computing, and stream processing have been introduced to the serverless paradigm. These applications are characterized by their stringent response time requirements, therefore expecting a quick and fault tolerant feedback from the application. The serverless, or Function-as-a-Service (FaaS), paradigm suffers from function 'cold start' challenges, where the serverless platform takes time to set up the dependencies, prepare the runtime environment and code for execution before serving the incoming workload. Most of the current works address the problem of cold start by (1) reducing the start-up or preparation time of function containers, or (2) reducing the frequency of function cold starts on the platform. Recent industrial research has identified that factors such as runtime environment, CPU and memory settings, invocation concurrency, and networking requirements, affect the cold start of a function. Therefore, we propose a Reinforcement Learning (Q-Learning) agent setting, to analyze the identified factors such as function CPU utilization, to ascertain the function-invocation patterns and reduce the function cold start frequency by preparing the function instances in advance. The proposed Q-Learning agent interacts with the Kubeless serverless platform by discretizing the environment states, actions and rewards with the use of per-instance CPU utilization, available function instances and success or failure rate of response, respectively. The workload is replicated using the Apache JMeter non-GUI toolkit and our agent is evaluated against the baseline default auto-scale feature of Kubeless. The agent demonstrates the capability of learning the invocation pattern, make informed decisions by preparing the optimal number of function instances over the period of learning, under controlled environment settings.**

Keywords—Serverless Computing, Faas, Reinforcement Learning, Q-Learning, Cold Start, Kubeless.

I. Introduction

The serverless computing architecture puts forward an event-driven, function-as-a-service model with a fine-grained pay-per-use pricing where costs are incurred only for the actual time that the resources are used. These models define a set of loosely coupled, stateless functions (a piece of code) that are executed on light-weight containers or virtual machines (VMs), having an inherent characteristic of on-demand scalability. Serverless computing completely takes off the burden of resource provisioning and management from the developers or users, thus emphasising solely on the application development. Serverless, in no way means the absence of servers, in fact the complexity of resource management lies solely with the Cloud Service Provider (CSP) [13,14]. The function-based abstraction increases agility in application development, offering lower administrative and ownership costs.

Serverless models execute the client code inside a light-weight function container, spawning the instances as per the function workload. With the ease of deployment and on-demand function scalability, the serverless execution model has attracted a wide range of applications from a variety of fields such as IoT services, REST APIs, stream processing, prediction services, etc. These applications have rigid latency requirements and thus expect a quick and fault tolerant response from the function. Conceptually, the serverless architecture is designed to prepare a new instance for every incoming workload and shut down after serving the request [14]. But, practically, commercial serverless platforms like AWS Lambda, Azure Functions, Google Cloud Function, etc may choose to re-use a function instance or keep the instance running for a limited period of time to serve subsequent requests [1]. Some open source serverless frameworks such as Kubeless [16] and Knative that are implemented over Kubernetes, have similar implementations to retain an instance of a function and re-use it to serve the subsequent requests.

With an incoming workload, new function containers are requested and a process of initialisation precedes the serving of the requests. The serverless platform initialises new containers, downloads the client code, sets up the code dependencies and runtime environment, sets up the worker node and eventually executes the container to handle the incoming request. This process introduces a non-negligible time latency, known as 'cold start', and poses as an existing challenge for the serverless platforms [2,3,5,7]. In other words, cold starts can be understood as the time taken by the platform to start executing an incoming request. A number of application factors as well as the function requirements affect the cold start of a function. Recent studies [6,7,8,9] suggest that factors like programming language, runtime environment, code packaging and deployment size, CPU or memory requirement limits, etc. affect the cold start of a function. The different offerings of serverless platforms allow for capturing the correct underlying resource information and some open source Kubernetes [15] native serverless frameworks like Kubeless take advantage of the native resource metrics. To deal with the function workload, Kubeless supports resource-based auto-scaling, i.e., Kubernetes Horizontal Pod AutoScaler (HPA) to derive the new instances based on the per-instance CPU-utilization of the function. The default auto-scaler starts requesting new instances only when the current function containers runs out of requested memory or the per-instance CPU utilization spikes above a specified threshold value. This leads to function container cold starts to serve the requests and eventually an increased number of failed requests, if the cold start time is greater than the request's time-to-live.

As these observations are solely dependent on the resource utilization values, they pose as an opportunity to explore techniques to understand the process and reduce the frequency of cold starts of a function. In this work, we present a Reinforcement Learning (RL), i.e., a model free Q-Learning agent, which exploits the per-instance CPU utilization, number of available function instances, to represent the environment states and, define the appropriate reward system for the agent to dynamically ascertain the optimal number of function instances for a given workload. In practice, a Q-Learning agent learns through the process of trial and error by interacting with the serverless environment. In each iteration, the agent analyses the current state of the environment and performs a particular action. A delayed feedback is observed, either positive or negative, based upon the realised factor (per-instance CPU-utilization, successful or failed response) and consequently learns about the workload pattern. This strategy does not have any prior knowledge about the workload pattern and dynamically adapts to the changes, thereby reducing the cold start frequency in subsequent invocations. This approach explores the applicability of Q-Learning algorithm for determining the optimal number of function instances in serverless environments in advance, so as to reduce the frequency of function cold starts, during a particular span of time. We compare this work by simulating the workload pattern for the default auto-scaling feature of the Kubeless platform. This helps in performing the analysis and examine the performance of both the configurations.

The **key contributions** of our work are as follows:

1. A Reinforcement Learning Agent implementing model free Q-Learning in a serverless environment setting to reduce the cold start frequencies of a function.
2. Implementing an agent to dynamically learn the function invocation patterns to ascertain optimal number of function instances, reducing cold start occurrences.
3. Evaluation of our proposed agent against the baseline auto-scale policy of the serverless platform for a synthetic function workload pattern.

The rest of the paper is organised as follows. Section II highlights related research studies. In Section III we present the system model and architecture along with the workload specification. Section IV outlines the proposed agent's workflow and describe the design decisions. In Section V we evaluate our technique with the baseline approach and highlight the possible shortcomings. Section VI concludes the paper, highlight the future research directions.

II. Related work

Serverless computing - featuring affordability, on-demand scalability and light-weight containerization, comes with inherent challenges and problems. These challenges can broadly be listed as security, privacy, caching, modes of execution, etc. Among them, the problem of cold start is still prevalent and has attracted academia for realising possible solutions. A current study [1] discusses the ongoing trends of handling the cold starts in commercial as well as open source serverless platforms and present their results by evaluating AWS Lambda offerings. They broadly categorise the approaches to deal with cold starts in two classes: (1) Optimising environments i.e. minimise the cold start delay itself and (2) Pinging i.e. minimising the frequency of cold start occurrences. Among the existing techniques to mitigate the cold start problem, they review the offerings of OpenFaas, OpenWhisk, AWS Lambda and discuss the solutions like cold and warm queue. They further create a case study with I/O intensive and CPU intensive benchmarks for evaluating the AWS Lambda's warm queueing technique and conclude with the absence of any correlation between the warm containers prepared by the platform and time interval of incoming requests.

In [2], an adaptive function container warm up technique is introduced to reduce the cold start latency. It utilises a function chain model, i.e., a sequence of functions to predict the function invocation time, using LSTM networks, and non-first functions to keep the warmed function containers ready in queue. The researchers also propose a container pool strategy that seeks to dynamically adjust the number of empty containers in the container pool to reduce the waste of resources. Both approaches work in synchronisation as the failure of adaptive warmup strategy will automatically launch adaptive container pool strategy by providing a pre-warmed empty container, thus reducing the overall cold start latency. It is highlighted in the study that even though the strategy learns the invocation time of the function chain, the first function in the sequence suffers cold start latency. They test their approaches by comparing the resource utilisation, idle time and overall cluster utilisation with other existing techniques.

Researchers in [3] explain the phenomenon of cold starts with respect to the Knative serverless platform and suggest a pod migration technique to reduce the cold start of the function containers. They posit that the cold start overhead is dependent on the underlying implementation of the function and categorise them in platform dependent and application dependent overheads. To deal with the cold starts, a pool of pre-warmed containers, marked with selector 'app-label', are kept ready. When the requests arrive, first the pool is checked for existing pre-warmed containers and allocated to the application, otherwise new containers are spawned as per the request workload. Using this approach, they conclude with an improvement in the cold start latencies of the containers for a single instance of pool.

Another research [4], studies and exploits the data similarity for reducing the cold starts and proposes a deployment system over a peer-to-peer network, virtual file system and content addressable storage to increase the computing capabilities, storage requirements and prevent network bottlenecks of system. They criticise the current container deployment technique of pulling each new container image from the storage bucket and introduces a live container migration technique over a peer-to-peer network. They propose to transfer blocks of files containing frequently used libraries and packages, over the network when required, and found a 37.9% reduction in the boot-time of containers. Similarly, [5] aims to reduce the number of cold stars by utilising the function composition knowledge. It presents an application side solution based on a light-weight middleware that aims to enable the developers to control the frequency of cold starts. It establishes that applications are generally deployed as a set of functions and proposes three strategies; naïve approach, extended approach and global approach where a dedicated orchestration component invokes all the steps and follow a process of 'hinting' the next batch of functions involved.

Research in [6] explores network creation and network initialisation as the prime contributor to the cold start latency. It explains four stages of container lifecycle: (1) service invocation, (2) startup, (3) run time and (4) cleanup. The cleanup includes stopping the container, disconnecting its network and destroying it and this process demands cycles form the underlying containerisation daemon, hindering with the other three processes. Thus, a pause container pool manager is proposed to pre-create a network for function containers and whenever required, attach the new function container to configured IP and network. Their evaluation on OpenWhisk platform demonstrates a reduction of up to 80% in the cold start times with a negligible memory footprint.

Research [6,7,8] has identified various factors like runtime environment, CPU and memory settings, dependency setting, the effect of concurrency, networking requirements, etc. which affect the cold start of a function. Most works focus on commercial serverless platforms like AWS Lambda, Azure Functions, Google Cloud Functions and fall short to evaluate open source serverless platforms like OpenLambda, Fission, Kubeless, etc. Very few studies [9,10,11] have successfully performed analysis on open source serverless platforms and provided possible solutions by targeting the container level finer-grained control of the platform.

Work in [12] introduces the paradigm of Reinforcement Learning (RL) to the serverless platforms. It is focused towards provisioning VMs or containers on request-based autoscaling in the serverless offerings. The study is conducted using Knative serverless platform that supports parallel processing of requests per instance utilising the Horizontal Pod Autoscaler of Kubernetes. The researchers show that depending upon the workload, different concurrency levels of the container can influence performance and thus, propose a RL based model, specifically model free Q-Learning, to determine the optimal concurrency levels for individual workloads. It evaluates the performance of model based on latency and throughput of the function containers and demonstrated the capability of applying Q-Learning algorithm to the task of auto-scaling in serverless platforms.

As a novel approach, we explore the applicability and capability of RL strategies to reduce the function cold start in a serverless environment. Contrasting to the existing works, we apply the model free Q-Learning algorithm for reducing the cold start frequencies on the serverless platforms, by identifying the invocation patterns of the specific workloads, focus towards learning the optimal number of function instances and evaluate it against the non-intelligent, default auto-scaler strategy responsible for cold starts. A summary of few discussed researches and our methodology is presented in Table 1, highlighting the distinguishing parameters of the individual studies.

Table 1. Summary of Relevant Works.

Parameter	Related Work						Our work
	[2]	[3]	[4]	[5]	[6]	[12]	
Open Source Platform	Yes	Yes	No	Yes	Yes	No	Yes
Commercial Platform	No	No	Yes	Yes	Yes	Yes	No
Function Invocation Pattern	Yes	No	No	No	No	No	Yes
Reinforcement Learning Technique	No	No	No	No	No	Yes	Yes
Pre-Warmed Containers	Yes	Yes	No	No	No	No	No
Other Techniques (Network creation, Migration, etc.)	No	No	Yes	Yes	Yes	Yes	Yes
Cold Start Frequencies	No	No	No	No	No	No	Yes

III. SYSTEM ARCHITECTURE

A. System Model

The overall architecture and system model of the experiment is illustrated in Fig. 1 and Fig. 2. To perform the experiment, a Kubernetes service cluster was setup using Melbourne Research Cloud (MRC) services at The University of Melbourne, Australia.

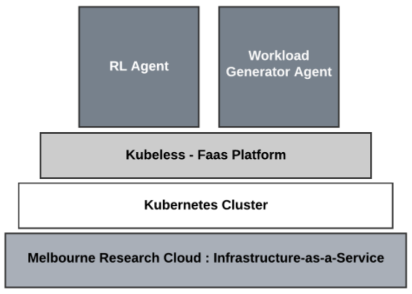

Figure 1. Deployed Stack Architecture.

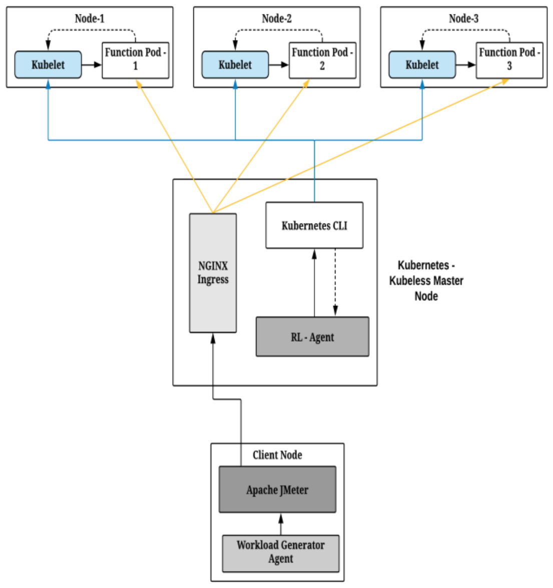

Figure 2. System Model.

The service cluster contains 4 nodes with 4 vCPU and 16 GB memory each and provides sufficient capacity to all the Kubeless components for the experiment. The workload generator agent is also configured on a similar node, responsible for sending the requests to function service cluster to generate the workload using Apache JMeter Non-GUI toolkit. The Kubernetes (version v1.18.6) cluster is configured using Ansible scripts for automatic cluster deployment. Kubeless (version v1.0.6) serverless framework is installed on the service cluster with all its components and inherent from Kubernetes, does not support scale-to-zero functionality (minimum 1 instance) in the implemented version.

The Q-Learning agent is configured to work in parallel to Kubernetes & Kubeless services on the service cluster and continuously monitors and manages the activities of the cluster including metrics collection, update the function deployment state based on collected metrics and logging the required metrics. To extensively test the workload learning capabilities of the agent, a NGINX-Ingress is also configured to handle the load balancing of incoming requests and thus allow the agent to avoid any performance issues while learning.

B. *Workload Specification*

There are a variety of applications which benefit from the serverless execution model including REST APIs, multimedia processing, highly parallel workloads, stream processing, etc. Some of these applications are compute intensive and demand considerable amount of resources, therefore, to investigate the cold start problem, we generate a stable workload from a set quota of function requests to simulate a serverless application. The request simulation uses the thread sleep method that enables the service cluster to serve quota of requests for a set time span and provide the RL agent with a delayed feedback/reward.

We fabricate a compute intensive process of calculating Fibonacci series up to number 38 [8], in order to keep the running instances busy and allow the default auto-scaler of the Kubeless to account for the increased number of function cold starts. Since Kubeless does not cite its concurrency policies, we specify function resource requirements (CPU and Memory) to be allocated and used for the purpose of evaluating the resource metrics. The Fibonacci calculator appropriately fits the compute intensive requirement of the experiment and the RL agent extensively captures the state of the serverless environment for learning the necessary function instances to lower the cold starts.

IV. Proposed agent workflow

The workflow is partitioned into two processes, the workload generator and RL agent. The workload generator is used to simulate a quota of parallel HTTP user requests against the Fibonacci function. We use Apache JMeter, an open source HTTP load testing tool, to simultaneously send a number of requests at a constant rate over a period of time. JMeter features a configurable thread 'ramp-up' period that tells JMeter how long to take for creating the desired number of request threads. In our experiment, a set of requests are sent from the quota of 100 requests with a ramp-up period of 200 seconds, engaging sufficient amount of resources from the function instances. This guarantees the demand for newer instances from the default auto-scaler, providing sufficient time for scaling or acknowledging the RL agent to analyse the workload pattern, observe the environment states and generate the rewards which complement the function cold start evidence.

The RL agent begins with set-up of reinforcement learning environment, i.e., state and action for successful implementation of model free Q-Learning technique to the serverless configuration. The reinforcement learning task is to train the agent that interacts with its environment. The agent transitions between different scenarios of the environment, called states, by performing the valid, available actions. These actions lead to rewards, either positive, negative or zero and the purpose of agent is to maximize the total reward it receives, during the process of learning. Q-learning trains an agent to approximate the value of actions i.e. Q-value, making use of the tabular representation of state-action values known as Q-table that forms the basis of decision making in the learning process.

Q(s,a) represents the Q-value of action *a* an agent performs at state *s* and tries to maximise this cumulative reward using Bellman Equation (1) at each iteration *t*.

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)] \quad ...(1)$$

α is the learning rate that describes the weight of newly observed information over old information, γ is the discount factor that describes the importance of future rewards. To choose between exploration and exploitation of information, the agent is implemented with ε-greedy strategy, where ε is the probability of exploration. Therefore, the agent selects an action that maximises the expected value i.e. an action with maximum Q-value for state *s* with probability of 1- ε.

The Kubeless framework along with its components and resources, including the sample function of Fibonacci calculator, forms the environment for the RL agent to interact. The state of the environment is defined by the combination of (1) number of function instances available and (2) per-instance CPU-utilization of the function. The agent allows maximum number of function instances to be passed as a parameter that controls states explored during the learning process. Since CPU-utilization values are continuous numbers, the per-instance CPU-utilization values are discretised into five bins of equal size – {20, 40, 60, 80, 100}, each representing the maximum value of CPU-utilization in the bin. This decision helps in appropriately limiting the size of the Q-table learned.

The agent interacts with the environment using actions of (1) scale up and (2) scale down the function deployment with valid number of instances. The availability of actions for a state is determined by the state representation, regulating the instances between 1 (minimum instance count supported by Kubeless) to M (maximum instance count). This allows the RL agent to explore and exploit only the possible state-action space, thus reducing the number of explorations and increasing the convergence time. For example, if the maximum number of function instances to be explored are M, the environment states can be represented as a set {x\$y | x ξ 1…M, y ξ (20, 40, 60, 80, 100)} containing M x 5 states. The action set for the agent at state x\$y (x: instance count, y: CPU utilization range) can be represented as {a | x + a > 0, abs (x + a) <= M, a ξ 1…M}. Therefore, the size of the Q-table can be calculated as (M x 5) x (2M – 1).

At each action step, the agent yields delayed rewards, determined after the specific time span, reflecting the appropriateness of the selected action. The agent learns the best actions by updating the Q-values according to the Bellman Equation, as in (1). The immediate reward, 'r', of state transition depends on function instance count, CPU-utilization and request success or failure rate during the observed time span, as in (2). The agent yields positive reward for successfully serving more than half the requests, being inversely proportional to the instance count and gets penalised for the states with CPU-utilization above a threshold of 75% and an undesired failure rate of more than 70%.

$$\textit{Immediate reward} \leftarrow (0.5*\textit{CPU utilisation}) + (0.3*\textit{state}^{-1}) + (0.1*\textit{success rate}) + (0.1*\textit{failure rate}) \quad \ldots(2)$$

The following steps outline the Q-learning approach to train the agent in serverless setting:

1. Input the maximum number of function instances to be explored and time step to consider for observing the rewards.
2. Setup the agent initial state and Q-table for the (state, action) pair.
3. Choose and perform an action according to the ε-Greedy policy with $\varepsilon = 5\%$.
4. Wait for 'sleep time' to observe the reward for current iteration.
    - Gather metrics for current function instances.
    - Calculate and return the immediate reward using collected metrics.
    - Calculate updated Q-value according to the Bellman Equation with $\alpha = 0.4$ and $\gamma = 0.3$.
5. Update the Q-table with new Q-values calculated.
6. Repeat the training procedure.

V. PERFORMANCE EVALUATION

To evaluate the performance of our RL agent, we compare it with the default auto-scaling policy supported by the Kubeless serverless platform. Kubeless supports auto-scaling of the function based upon the CPU metrics i.e. the per instance CPU-utilisation of the active instances. The default policy is implemented as a control loop with a period of 15 seconds, after which the controller scrapes the metrics and perform the required action of scaling up. The platform keeps the allotted resources for a period of 5 minutes to prevent the resources from thrashing, due to dynamic nature of the changing metrics.

We simulate the CPU intensive serverless workload using the function Fibonacci calculator up to number 38 [8]. This ensures a sufficient amount of resources, CPU and memory, are utilised by a single request and therefore results in simulating higher number of cold starts to evaluate the algorithms. To mimic the serverless workload pattern under the experimental conditions, we create a set of a number of requests with a limit of 100 requests per 5 minutes, $\{reqCount \mid reqCount <= 100\}$. The period of 5 minutes is chosen in order to prevent thrashing of the resources while auto-scaling the function instances and thus evaluating both the approaches within a comparable schedule.

To effectively observe the results of default auto-scaling policy and examine the feasibility of model free Q-Learning in the experimental serverless setting, a period of 240 minutes was designated. As the Q-Learning algorithm seeks to explore the available state-action pair in the environment, a maximum of 10 function instances were preferred to comfortably execute the procedure of learning, while abstaining from the explosion of the size of the state-action space represented by the Q-table. In the baseline experiment of default auto-scaling policy responsible for provisioning the function instances on-request, every new instance provisioned represents a potential addition to cold starts frequency. This keeps a part of incoming workload waiting to get an instance allocated and leads to a failed response, while spawning the new instances. Therefore, the rate of failed responses, which could not be served due to requirement of instances (resources) are used to compare both the discussed approaches.

Fig. 3 illustrates the results of the default auto-scaling policy of the Kubeless platform. With the limited number of function instances and considering CPU-utilisation metrics, it can be seen that the baseline approach suffers from a number of failed requests and accounts for approximately 44% of the failed requests out of the workload of approximately 2166 requests. This can be attributed to the following characteristics:

- A control loop period of 15 seconds, after which the auto-scaler checks for the CPU metrics.
- A CPU-intensive function composition that leads to occupied resources and waiting or failed requests.
- Limited scaling of the function instances during the experiment.

Fig. 4, Fig. 5, Fig. 6 and Fig. 7 illustrates the four iterations of the Q-Learning agent. The agent is trained for a period of 240 minutes, in multiple iterations to ascertain the function invocation patterns and learn the number of function instances required for a specific workload. This decision is based upon the rewards experienced by the agent according to the CPU-utilisation, success and failure rate of requests and the number of serving function instances. It is evident from the graphical representation that the agent starts learning the invocation patterns and exploits the experience it gets from the rewards. For instance, it can be inferred from the iteration 3 & 4 that the agent explores different configurations and tries to minimise the cold starts by preparing the required number of function instances, leading to reduced number of failed requests, during the time period 10 – 20 and 30 – 40. The lower variations of failed requests (i.e. the reduction in number of failed requests over the iterations) also signify that the agent is learning the optimal number of function instances to reduce the cold start problems and tries to serve maximum request.

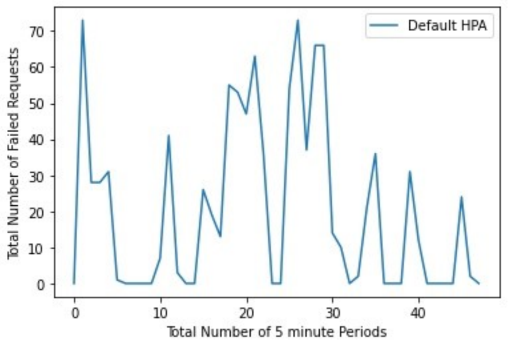

Figure 3. Failed requests using default HPA policy.

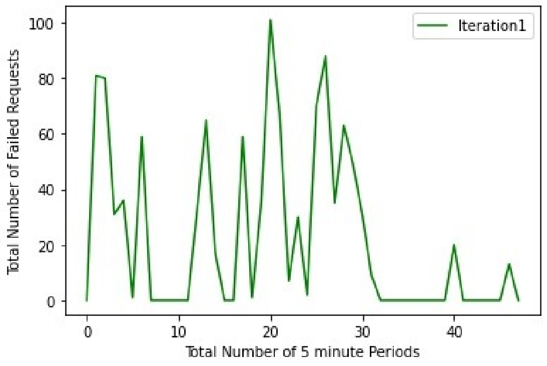

Figure 4. Q-Learning Agent iteration 1 of 240 minutes.

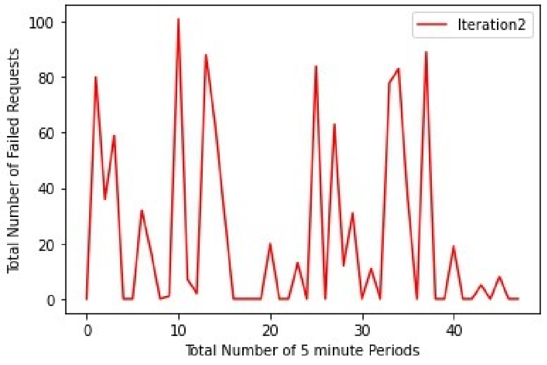

Figure 5. Q-Learning Agent iteration 2 of 240 minutes.

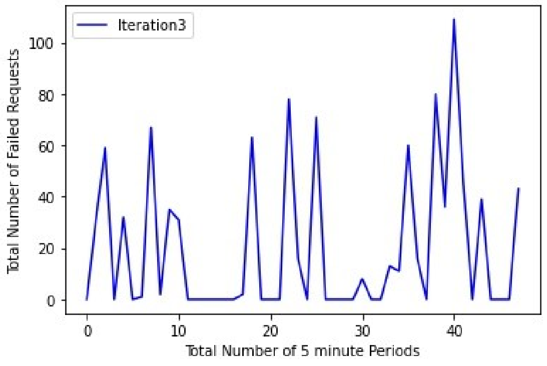

Figure 6. Q-Learning Agent iteration 3 of 240 minutes.

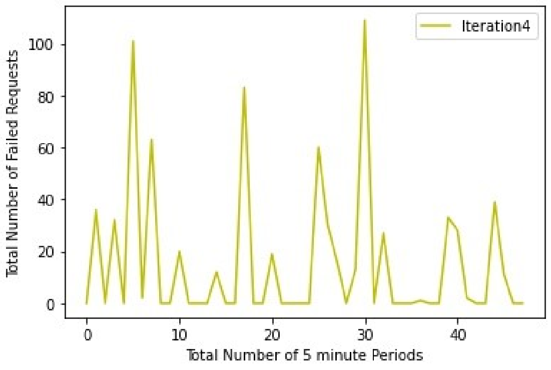

Figure 7. Q-Learning Agent iteration 4 of 240 minutes.

After four iterations of training the RL agent, it manages to serve approximately 50.1% requests successfully and shows a positive indication towards converging to the optimal values. As compared to default auto-scaling technique, the performance of our agent after few iterations of the Q-Learning procedure shows the feasibility and appropriateness of the reinforcement learning strategy to the task of reducing cold start occurrences. The difference of results between both the approaches can be attributed to the following characteristics of the RL agent implemented –

- The elementary reward structure used in the Q-Learning process that can affect the information gain of the agent.
- The values of α, learning rate and γ, discount factor, that plays an important role in learning process. The different combination of these values might result in quicker convergence.
- The underlying CPU-intensive function composition causes the high resource utilisation leading to negative values. Thus, the agent explores different state - action space values taking more time to learn the required values.
- Discretisation of the continuous number values of CPU utilisation for state space representation can hinder with the optimal performance of the agent with Q-Learning techniques.
- The large state-action space also accounts for the longer learning periods and affect the agent's information gains.

In comparison to the baseline approach of default autoscaling, our approach shows practical applicability of the RL algorithm to reduce the cold start occurrences for a specific function workload and closes on the difference between the successfully served requests within few iterations.

VI. CONCLUSIONS AND FUTURE WORK

Serverless computing with its easy-to-go deployment structure, have discharged the application developers from the responsibilities of managing the servers. On the other hand, this execution style increases the responsibilities of the cloud service providers to continuously provide fault tolerant services to their customers. With application response time being one of the most important factors for the end-user, serverless introduces overheads of cold starts i.e. setup time of the function containers to serve the requests. Various approaches have been proposed to reduce the challenge of cold starts both by academia as well as the technology industry like keeping a warm queue of function containers, continuously pinging the functions to keep them running and keeping pre-prepared containers with dependencies ready, etc. These non-intelligent approaches fail to identify the request invocation patterns and therefore lead to failed responses due to resulting cold starts. We present an evidence of using reinforcement learning technique, specifically model free Q-Learning, to the serverless environment setting and propose an intelligent agent that learns from the unknown invocation pattern to ascertain the optimal number of function instances to reduce the cold start frequencies of the function. We compare the result of our approach with the existing auto-scaling technique and successfully observe that with a few or limited number of training iterations, the agent was able to show moderate results by serving approximately 50.1% of the requests.

As part of future work, we plan to extend this approach of Q-Learning using combinations of reward structure, α and γ values and a variety of function compositions. We further plan to include other important factors like memory setting, and function size, etc. in the learning process of the agent to better determine the criticality of the actions in the state space. As this approach requires discretisation of the continuous values for state representation, we plan to extend this approach using DQNs (Deep Q-Learning Networks) to estimate the information about optimal actions without the risk of state-action space explosion.

References

[1] P. Vahidinia, B. Farahani and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," *in Proceedings of the International Conference on Omni-layer Intelligent Systems (COINS)*, Barcelona, Spain, 2020, pp. 1-7.

[2] Z. Xu, H. Zhang, X. Geng, Q. Wu and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *Proceedings of the IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, Tianjin, China, 2019, pp. 9-16.

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

[4] K. Mahajan, S. Mahajan, V. Misra and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies,* Orlando, FL, USA, 2019, pp. 37-39.

[5] D. Bermbach, A.S. Karakaya and S. Buchholz, "Using application knowledge to reduce cold starts in FaaS services," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing,* Brno, Czech Republic, 2020, pp. 134-143.

[6] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, Renton, WA, USA, 2019.

[7] H. Shafiei, A. Khonsari and P. Mousavi, "Serverless computing: A survey of opportunities, challenges and applications," *arXiv preprint arXiv:1911.01296v3,* 2019.

[8] J. Manner, M. Endreß, T. Heckel and G. Wirtz, "Cold start influencing factors in function as a service," in *Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* Zurich, Switzerland, 2018, pp. 181-188.

[9] K. Solaiman and M.A. Adnan, "WLEC: A not so cold architecture to mitigate cold start problem in serverless computing," in *Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E)*, Sydney, NSW, Australia, pp. 144-153.

[10] J. Santos, T. Wauters, B. Volckaert and, F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *Proceedings of the 2019 IEEE Conference on Network Softwarization (NetSoft),* Paris, France, 2019, pp. 351-359.

[11] S. K. Mohanty, G. Premsankar and M. Di Francesco, "An Evaluation of open source serverless computing frameworks," in *Proceedings of the 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom),* Nicosia, Cyprus, 2018, pp. 115-120.

[12] L. Schuler, S. Jamil and N. Kühl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," *arXiv preprint arXiv:2005.14410*, 2020.

[13] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E),* Orlando, FL, USA, 2018, pp. 159-169.

[14] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[15] Kubernetes Documentation | Homepage[Online]. Available: https://kubernetes.io/docs/home/

[16] Kubeless – Kubernetes native serverless[Online]. Available: https://kubeless.io/docs/

[17] Apache JMeter – Getting Started[Online]. Available: https://jmeter.apache.org/usermanual/index.html