WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing

Khondokar Solaiman*, Muhammad Abdullah Adnan†
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh
Email: *googlisolaiman@gmail.com, †abdullah.adnan@gmail.com

***Abstract*—As serverless computing gains popularity among developers for its low costing and elasticity, it has emerged as a promising research field in computer science. Despite its popularity, the cold start remains an issue that needs more attention. In this paper, we address the cold start problem of the serverless platform. We propose WLEC, a container management architecture to minimize the cold start time. WLEC uses a modified S2LRU structure, called S2LRU++ with an additional third queue. We implement WLEC in OpenLambda and evaluate it in both AWS and Local VM environment with six different metrics in addition to one real-time image resizing application. Among improvements in all metrics, 50% less memory consumption compared to the all-warm method and 31% average cold start duration reduction compared to the no-warm method are the most notable ones.**

***Index Terms*—container, lambda functions, cold start, warm up queue**

I. Introduction

The advancement of cloud computing now makes it possible to define the new concept of serverless computing. It often refers to applications that can be run without managing any server dependencies. They do not require any provisioning scheme and management of servers. It enables us to focus on the core product and business logic instead of responsibilities like an operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability [4]. By building an application on a serverless platform, we can get automatic scaling without any server management, high availability and fault tolerance, pay as you go features that allow paying only for active capacity.

Serverless computing is just the latest edition of the "as a service" model of cloud computing. Different versions of the model like Infrastructure as a service (IaaS), Software as a service (SaaS) and Platform as a service (PaaS) have been provided by the cloud providers offering customers different levels of resources like software subscriptions to full computing systems. Serverless is in the fourth category: Functions as a service (FaaS) [25] which is based on runtime level virtualization. In Function as a service model, the developer does not need to think about the runtime that the code uses. He just needs to write code based on the platform and use a way that the platform supports and use the library provided by the platform [4].

In earlier settings, applications needed their environment including both hardware and software. The introduction of virtualization has made it possible to share the same hardware for multiple operating systems (OS) known as Virtual Machine (VM) [12]. Later container-based virtualization has allowed us to use the same OS and H/W for different applications. The repackaging of Unix-style processes combined with distribution tools has gained huge favor among developers which is known as Docker [11]. However, the Lambda model is the latest virtualization model that allows sharing runtime resources. It has revolutionized the concept of Serverless Computing.

With the Lambda model, applications are seen as a set of functions called lambda functions instead of a collection of servers and developers. Different lambda handlers are written to handle different kinds of requests. As we browse available lambda services, we find that lack of full control over the lambda model hinders the implementation of the lambda model in real life. All the core sections: request distribution, code files and code executions- all are hidden from the developer because of security, consistency, resource management and performance issues [19]. The lack of control in code execution on specific containers results in increased start-up time, also known as cold start problem. To be specific, the cold start mainly refers to the time delay from receiving a lambda request in the application to the start of the execution of code for that request. Alternatively, the time of preparing a container to serve any request is called cold start. As a result, the absence of any built-in mechanisms to counter cold start leads to the same problem for all lambda models present in the industry. Thus, the only possible way-out from the developers' end, is to send continuous hello request actively to keep those lambda containers warm.

After the introduction of OpenLambda-an open-source lambda model implementation, developers and researchers now have more control over the architecture of the lambda model than ever before [1]. Here, we propose WLEC to counter the cold start problem and integrate it with the conventional lambda model and implement it in the OpenLambda Platform. Pannier [3], a container-based caching policy encourages the architectural idea for WLEC. WLEC uses S2LRU++ instead of standard two queue S2LRU model. The specialty of S2LRU++ is that it is a container-ware S2LRU model and it uses three queues rather than two. The three queues are called cold, warm and template queue in S2LRU++. These queues hold containers that run the lambda functions.

978-1-7281-1099-8/20/$31.00 ©2020 IEEE
DOI 10.1109/IC2E48712.2020.00022

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

Containers are placed in these queues based on their usage, invocation time, wake up time, state and other parameters and managed by the CMS (Container Management Service). CMS continuously monitors the state of the containers and can move, destroy and initialize them accordingly. CMS consists of three main functions: *Initialization* initiates a container with corresponding flags and variables, *QueuePlacing* handles the transition of a container from one queue to another when certain conditions are met and *OnRequest* handles the request and selects the best container to serve the request at any given time, are also discussed in detail later.

We test the performance of WLEC architecture from our comparison of six different metrics with the ubiquitous lambda model. We define these metrics and compute them for the OpenLambda platform in both the Local VM and AWS VM setup. By further computation of these metrics, we show that WLEC reduces average cold start invocation cases by 31% and the average duration of cold start by 23.5% in AWS VM. In the case of memory consumption, we find WLEC architecture consumes about half of the memory of traditional All-Warm methods. Besides, we observe that the number of invocation per container is increased by about 31.25% while start-up latency is reduced by 70.2% comparing to the fresh start scenario for AWS VM. Then we use a serverless image resizing application to delineate its performance on a real-time application. The main contributions of this paper are as follows:

- We discuss various reasons, contributors and impacts of cold start in the OpenLambda platform. Besides we explore the traditional approaches and recent studies to minimize cold start time for containers and their suitability in the lambda model.
- To counter cold start, we propose an integrated, structured approach for the first time to our knowledge. We develop WLEC, an S2LRU++ based architecture to handle the containers in a more structured fashion to ensure less start-up latency and more concurrency than the ubiquitous lambda model.
- We describe the design, components, workflow, and implementation of WLEC in the OpenLambda platform and evaluate it using Local VM and AWS VM with six different metrics. We also show its congruity over a real-time serverless application on image resizing.

II. Background & Motivation

The container-based virtualization has now become a cornerstone of cloud platforms and data centers. But the need for runtime level virtualization to make easily deployable and elastic applications with lightweight bundling opens up a new door in the virtualization concept named lambda model. So, the recent development of OpenLambda has made it possible to take a closer look at many problems and research areas of runtime level virtualization. The three key components of the lambda model are: 1) the application packaging system which bundles server runtime with required library (e.g. execution engine), 2) the memory and server time-sharing between applications, 3) the registry store which stores the codes for applications. These codes are called lambda functions. The packaged execution engine forms a sandbox also known as workers/containers. Most other platforms keep the worker creation and code execution management out of developers' hands for security and maintenance reasons [23]. The workers are managed by the platform providers and hence developers have no control over container management. As a result, the cold start problem remained an issue to be resolved.

To improve performance overheads of these secured serverless platforms, concepts like load balancing and packaging have been proposed [22]. Beside built-in load balancer also helps to reduce response times of the workers. Pipsqueak [24], a shared packaging tool to reduce the start-up time of cloud functions has also been introduced. It caches required packages of functions at each worker at the sleeping state of the function. It has a wake up and forking mechanism along with cache-tree to support multiple dependencies. Nonetheless, its unscalability and cache loading on each worker seems like an overhead for large libraries like Pandas [26]. For load balancing, function scheduling is also a popular choice to reduce the response time. As no intelligent decision-making method is available to minimize the number of workers or to reuse the already packaged worker, they find little usability in real-life applications.

To avoid the large cold start delay, the current trend is to keep warm workers. We have found that AWS lambda service has this feature enabled where it keeps some warm workers(around 10) per application in the memory to minimize the latency [23]. These workers are always kept in memory by AWS. These naive workers cause large memory consumption, less re-usability, non-intelligent lambda function assigning. Even all serverless platforms do not have this kind of feature. We show the state-wise average timing for a worker of OpenLambda in Fig. 1 by making 1000 lambda requests with 20 workers. We find fresh-start and restart incur 1000 times and 10000 times more delay than unpause state respectively. Here, WLEC will try to ensure warm workers for every request along with intelligent lambda function assigning to the workers, concurrent request handling and re-usability of

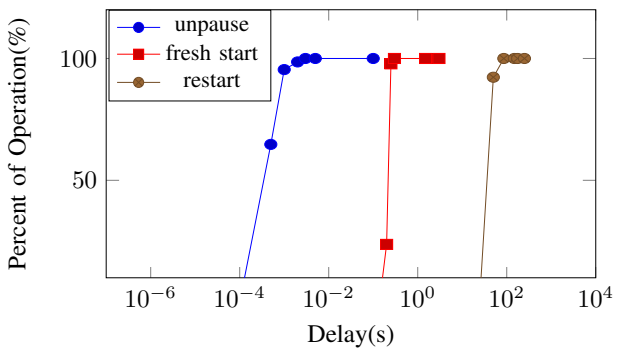

Fig. 1. Readiness latency in unpause, fresh start and restart case for OpenLambda

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

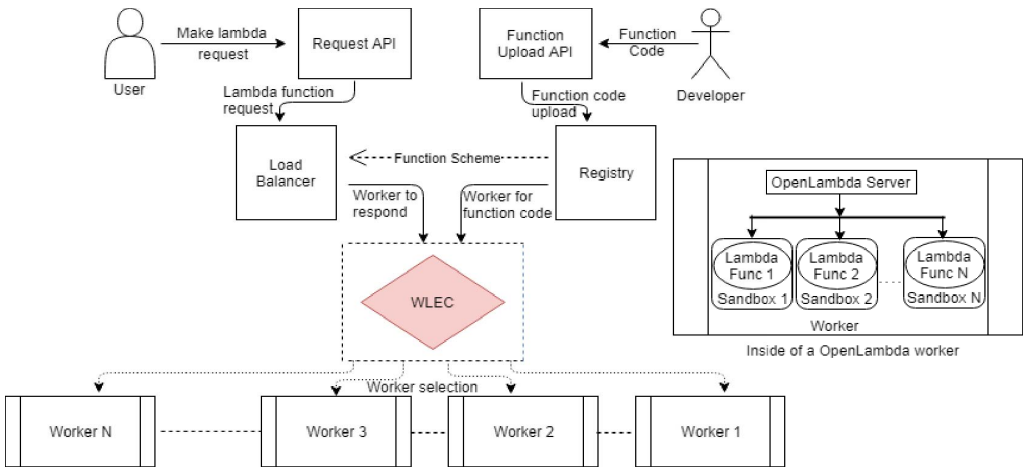

Fig. 2. OpenLambda with WLEC

the workers with proper worker lifetime.

III. WLEC: Our Proposed Architecture

Fig. 2 shows the architecture of OpenLambda with WLEC. It consists of 3 main components. a)Load balancer based on Nginx which receives the client request and assigns this request to a worker. The load balancing mechanism highly relies on the locality management of the system. The localities that were considered in this case were session locality, code locality, and data locality. b)Lambda store is also known as a registry that holds all the codes to handle different kinds of user requests. c)Execution engine which consists of a number of workers. Each worker works as a sandbox based on Linux container, to execute code files the function code associated with the lambda request issued by the client. We can see, WLEC works as a middleware for both load balancer and registry to provide the best worker available in any given time for function execution. The internal architecture of WLEC is shown in Fig. 3 including all the components. The components are described in subsection III-A. One of the most challenging parts of the design of WLEC is concurrent request and late request(after 15min) handling. The most challenging perspectives in this research are to address both design challenges with a single system and to manage the system accordingly. We assume that we can control the invocation of containers in lambda functions.

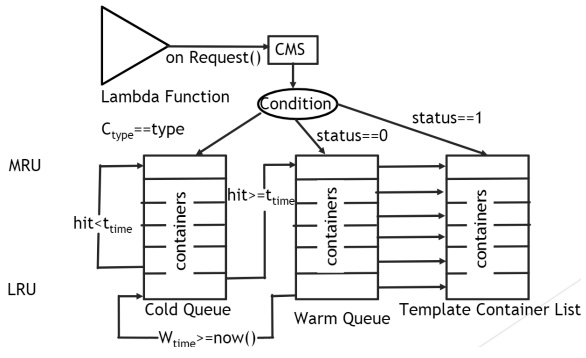

Fig. 3. WLEC Architecture.

A. WLEC Components

In this section, we describe the main components of WLEC. We define the terms and components used in architecture. Then we narrate CMS and its sub-functions and their algorithms. Next, we delineate the workflow of WLEC.

1) S2LRU++: S2LRU is a partition technique used to make two segments, one for the probationary and the other for the protected segments. In the standard S2LRU algorithm, new data are inserted into the most recently used (MRU) position in the probationary segment. On a hit, data are promoted from the probationary segment to the MRU position in the protected segment. If data in the protected segments have got a hit, they are promoted to the MRU position of the same segment. When the protected segment exceeds its predefined size (e.g. half of the cache size), the LRU data are migrated to the MRU position in the probationary segment. For clarity, we rename the protected segment and probationary segment as the warm queue and cold queue [14] respectively shown in Fig. 3. Here we use containers instead of data. For S2LRU++, we maintain another queue named template container list, a copy of warm containers for each type of request of any application to serve any concurrent request. In our case, instead of data, we are going to place the lambda containers with respective lambda functions in the hot and cold queues. The term "Data hit" is replaced with "Container invocation". So, container placement is managed using the number of invocations of a container.

2) Template Container List: A list of container which holds the duplicate container of all the containers in the current warm-up list. Here, the term duplicate container means another container with the same library packaging and environment configurations of a container that it is cloned from, to serve any similar incoming lambda requests. If any concurrent request

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

is issued for any lambda functions currently serving another request would be passed to the copy container of this list. It reduces the cold start as the container is already packaged with necessary libraries. As this list of containers is processed in the background of the system, no cold start incurs for the request. After completing the response, the container is not pushed to any other queue as it is created just to handle the concurrent requests only. This copy container is destroyed if it is not busy and the respective warm-up container moves to cold queue at the same time. As we try to minimize the cold start time, concurrency handling is a big factor. We use this template list specifically to maximize the concurrency factor. However, the size of this template container list depends on the application. The total number of supported lambda request type defines its size. We also introduce a variable named concurrency factor. It defines the number of copies of a worker will have in the template container list. For example, the video transcoding application in [32] supports 2 different lambda requests from clients for HLS and DASH video coding. So, there will be 2 duplicate containers in the template container list with concurrency factor 1. Hence the size of the template container list size will be only 2 in this case.

3) Warm time: For every container in the warm queue, there should be a warm time in that container header. It indicates the time stamp when a container should leave the warm-up queue. Warm time is maintained in container header using Hr:Min:Sec format. The warm time of a lambda container will change based on its last invocation time. So, the warm time needs to be updated according to its last invocation time. For a container its warm-up time is calculated as follows:

$$warmTime = invocationTime + a_{time}.$$

In the case of the first invocation, the invocation time will be the initialization time of the container. The a_{time} is the default time, a container can live in the warm queue. This is configurable which varies from platform to platform from 15min to 45min. We assume a container can stay active for 15 min from its last invocation. Because 15 min is considered as standard alive time in other serverless platforms like AWS, Azure, etc.

4) Container Header: To maintain, monitor and keep track of the containers, we define some flags and variables in the header section. We propose five headers for every container in OpenLambda. The header parameters can be divided into two parts: Function parameters and Container parameters. Function parameters are maintained for each lambda but container parameters are maintained for each container. When a new lambda function is invoked in any container, the container parameters get updated if it is already initialized. Because lambda parameters get updated every time upon invocation. The f_{type} and $status$ variables are maintained for the lambda functions. The other three q_{type}, w_{time} and hit are used for the container which holds any lambda function. These flags are assigned and initialized by the Container Management Service in the initialization period and they are also maintained and updated by the CMS in runtime. Every container has a container header with the fields shown in Table I.

TABLE I
TABLE OF HEADER COMPONENTS

f_{type}	meta data about lambda function
q_{type}	the location queue of the container
$status$	execution status of lambda function
w_{time}	warm time of the container
hit	hit count of the container

B. Container Management Service

CMS is the system that manages all the containers and their placement in queues. Whenever a lambda container is created, all the management of the container is done using CMS. CMS is a unique service that also incorporates and manages all the requests in WLEC. From requests handling-issued by a client, to serve the request-all the functionalities are maintained and served by CMS. As a container is created, the related flags are created by CMS and through these flags and related algorithms defined in the CMS function section, the container is then transferred in its rightful position. The architecture of CMS consists of two queues and one container list named warm, cold and template container list respectively. The containers which are kept in these structures are the main concern of our architecture. If any container becomes invalid, it is also tracked by CMS and evicted from these container structures. The symbols used in CMS algorithms are listed in Table II.

1) CMS Function: We define basic CMS functions as three smaller functions and their algorithms in this section. The three functions are Initialization, QueuePlacing, and OnRequest. The functions, their functionalities, and descriptions are given below:

Initialization(): When a new container is invoked the headers are set accordingly. The hit count is incremented and is placed in the Q_c. Thus the q_{type} is set to 0. The f_{type} is set from the metadata of the lambda function provided by the function writer. Status is set 1 when it is in the execution phase. Back to 0 when it has finished the execution. w_{time} is updated according to the algorithm 1. The initialization function is invoked every time a new lambda function is created. To serve any request from the client application, the

TABLE II
TABLE OF SYMBOLS OF CMS

Q_c	cold queue
Q_w	warm queue
L_t	Template container list
c	a container
LRU	least recently used
MRU	most recently used
λ_{type}	lambda function type
t_{value}	threshold number to change queue
a_{time}	alive time(15 min)

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

lambda function needs to be packaged in a container. The new container selection is determined in the OnRequest function. Any container which is not found in the queues will have to be created through the initialization method. The algorithm for initialization of containers is given in 1.

Function `Initialization()`**:**
  $create$ container c
  $hit \longleftarrow hit$++
  $q_{type} \longleftarrow 0$
  $w_{time} \longleftarrow now() + a_{time}$
  $Q_c.push(c)$
  $return$ c
End

Algorithm 1: Algorithm for container initialization

QueuePlacing(): The QueuePlacing function works as a caretaker of these lambda containers initialized earlier. When a container has hit value 0, it is placed in Q_c in the LRU position. If it gets another hit, the container c is then moved to the MRU position of Q_c. If it gets more requests then the hit count gets incremented every time. If the hit count crosses t_{value}, it is then moved to the MRU position of Q_w. Each time a_{time} is updated on a hit. When the $now()$ crossed the a_{time} then c is pushed to the Q_c. The algorithm is given in 2.

Function `QueuePlacing(c)`**:**
  if $c.hit = 0$ *or* $c.hit < t_{value}$ **then**
    $Q_c.LRU \longleftarrow c$;
    $c.w_{time} \longleftarrow now() + a_{time}$;
  else if $c.hit < t_{value}$ *and* $q_{type} = 0$ **then**
    $Q_c.MRU \longleftarrow c$;
    $c.w_{time} \longleftarrow now() + a_{time}$;
  else if $c.hit > t_{value}$ *and* $q_{type} = 0$ **then**
    $Q_c.pop()$;
    $Q_w.MRU \longleftarrow c$;
    $c.w_{time} \longleftarrow now() + a_{time}$;
    $L_c \longleftarrow c$;
  else if $c.a_{time} < now()$ *and* $q_{type} = 1$ **then**
    $Q_c.MRU \longleftarrow c$;
    $c.w_{time} \longleftarrow now() + a_{time}$;
  else
    $No\ Change$
  end
End

Algorithm 2: Algorithm for container placement

OnRequest(): This function is called whenever a request is issued from the client-side. After validation check of the request, a lambda container is picked from the queues using this function for executing the requested lambda function.

The logic is to first search in the cold queue. If found then it is returned instantly as it was kept warm by warm-up calls. If the cold queue does not have the container then it is searched in the warm queue. If the same type of container is found in the warm queue, then the state of the container is evaluated to determine if it is busy or not. If the status flag shows that it is not busy then it means currently no request is in the processing state for that container then that container is returned. If it is busy, then this request is a concurrent request. So, the corresponding template container list is searched. When the corresponding container is found then that template container is returned to serve the request. If no suitable container in all the queues is found, then a new container is needed. Thus the initialization function is invoked to provide the new container to serve the request. The algorithm is given in 3.

Function `OnRequest()`**:**
  if *request is valid* **then**
    foreach $c \in Q_w$ **do**
      if $c.f_{type} = \lambda_{type}$ *and* $c.status = 0$ **then**
        $c.hit \longleftarrow c.hit$++
        return c
      else if $c.f_{type} = \lambda_{type}$ *and* $c.status = 1$ **then**
        $c = Initialization()$
        $L_t.LRU \longleftarrow c$
        return c from L_t
    end
    foreach $c \in Q_c$ **do**
      if $c.f_{type} = \lambda_{type}$ **then**
        $c.hit \longleftarrow c.hit$++
        return c
    end
    $c = Initialization()$
    $c.hit \longleftarrow c.hit$++
    return c
  else
    invalid request
  end
End

Algorithm 3: Algorithm for container selection

2) Work Flow: When a request is issued from the client-side, the request is sent to the OnRequest method of CMS. This client request can be one of get, post or put requests. The OnRequest method searches the cold queue and then the warm queue for the container that can serve the client request optimally. If any container is found then that container is returned to serve the request. If the container is warm but busy serving any concurrent request then the template container list provides the temporary container restored in the list. Then a new template container of the same type is created in the template list and new pointer points from the same type of busy containers in the warm queue to handle the next concurrent request for the same container. If the request is served by the previously assigned container, that container gets deleted from the template list.

If no container is found in lists, then a new container needs to be issued. So, the initialization method is called and a new container is returned to serve the request. All the corresponding flags are updated according to the algorithm of the initialization phase.

After serving a request, all the flags like hit, w_{time} are updated

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

by CMS. Then the QueuePlacing method is called to place the lambda container in its respective position. The method checks the flags and variable value to determine which queue should hold the container. If it belongs to the warm queue, another template of the container is created in the template container list to serve any concurrent request. All containers in these queues are maintained, updated and served by CMS using only CMS functions.

IV. Evaluation & Results

The different numbers of workers are used to evaluate the performance of the WLEC approach. All the metrics are plotted to compare the performance with traditional approaches like All warm approach or No warm approach. Here, All warm approach means a customized system of warming every worker initialized in OpenLambda for request handling. Details are described by Cordova [10]. Periodic ping requests are issued to each worker to keep them alive for a specific amount of time which is mentioned in the warming system. On the other hand, No warm approach means the exclusion of any customized warming mechanism. This approach uses only typical OpenLambda setup.

In this section, first, we discuss our experiment setup and methodology. Then we define the metrics and narrate how these metrics are calculated and plotted to delineate the performance of our architecture. Next, we take WLEC to a real-time application to determine its impact on a pragmatic case.

A. Experiment Testbed

All the experimentation was done in the OpenLambda environment with 2 different virtual machine setup. One of them was on a local laptop computer and another setup was on Amazon Web Server with an EC2 instance. In both cases, we used Ubuntu 14.01 as the Operating System. For simplicity, we used the default setup of OpenLambda from the Github repository [15]. The lambda functions were written in Python [15]. So, we also took the warm-up functions in the same language. The local machine VM was assigned 40GB hard disk space with 2GB RAM with 4 vCPU. The AWS VM was a t2.micro instance with 1 vCPU, 1GB memory along with EBS type storage with 8GB storage space. We used AWS's US East region for our computation.

B. Methodology

For every experiment, we varied the total number of workers as 500, 1000, 1500, 2000, 2500, 3000. We used 10 different lambda functions from [36] as our workload and invoked them randomly. The request number was kept as same as the worker number. For warming, we used the Thundra [2] default warm-up mechanism using a lambda function call without any parameter. The metrics were calculated according to the AWS metrics definition provided in their documentation [20]. We took about 10 simulations with each parameter set and then calculated the mean. In our case, we kept 10 requests per min as a standard request policy for both Local and AWS simulation. We set the cold and warm queue size maximum of 1/3 of the total available workers for each. Besides client requests to these lambda functions were random to ensure a bias-free result. The value of the concurrency factor was 2 throughout the whole experiment. The parameters and their respective values which were used for our experimentation are shown in Table. III.

TABLE III
Table of parameters and their values, default values are in bold

Parameter	Value
f_{type}	**get**, post, apiCall, dbOp
q_{type}	**0**,1,2
$status$	**0**,1
w_{time}	**15**,20,25
hit	**0**, 1, 2...
t_{value}	**25**,50,100
a_{time}	**15**, 16, 17 ..
$concurrencyFactor$	0, 1, **2**, 3 ..

C. Results

Now, we illustrate the results of our evaluation regarding WLEC with six different metrics and then show a performance comparison. We define each metrics first and then compute them with our testbed setup in both platforms with OpenLambda. Next, we provide a quantitative improvement report against each stated metric with variances and standard deviations in the case of both Local and AWS VM in Table IV. Finally, an analogy of WLEC over a real-time image resizing serverless application is incorporated.

1) Number of Cold Start Invocation: The number of cold start invocation metric shows the count of workers that faced cold start in the time of invocation, initiated by the lambda functions. In our experiment, we count the number of cold starts occurred with the change in the number of workers. Fig. 4(i) shows the number of workers that suffer cold start problem. We find that the number grows with the available worker number proportionally. As more requests are sent, more containers are initialized which increased invocations that suffer cold start in case of both Local and AWS VM. But with our WLEC, we find that the cold start occurrence is decreased by about 27% and 31%. Besides, we also find that AWS VM shows slightly better results. It is caused by the better package management of AWS VM. Though initially, we see a decline in the number of cold start invocation, as the number of workers grows, the superiority of WLEC gets clearer and we find 4x fewer containers that suffered a cold start at 3000 available workers.

2) Average Cold Start Duration: This metric shows the average time taken by the cold started worker to respond to lambda requests. It was calculated by the ratio of total cold start duration to the number of cold start for all the workers. Fig. 4(ii) provides the comparison between AWS and local VM where we find that WLEC not only reduces the number of cold start occurrence but also the duration of cold start at the

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

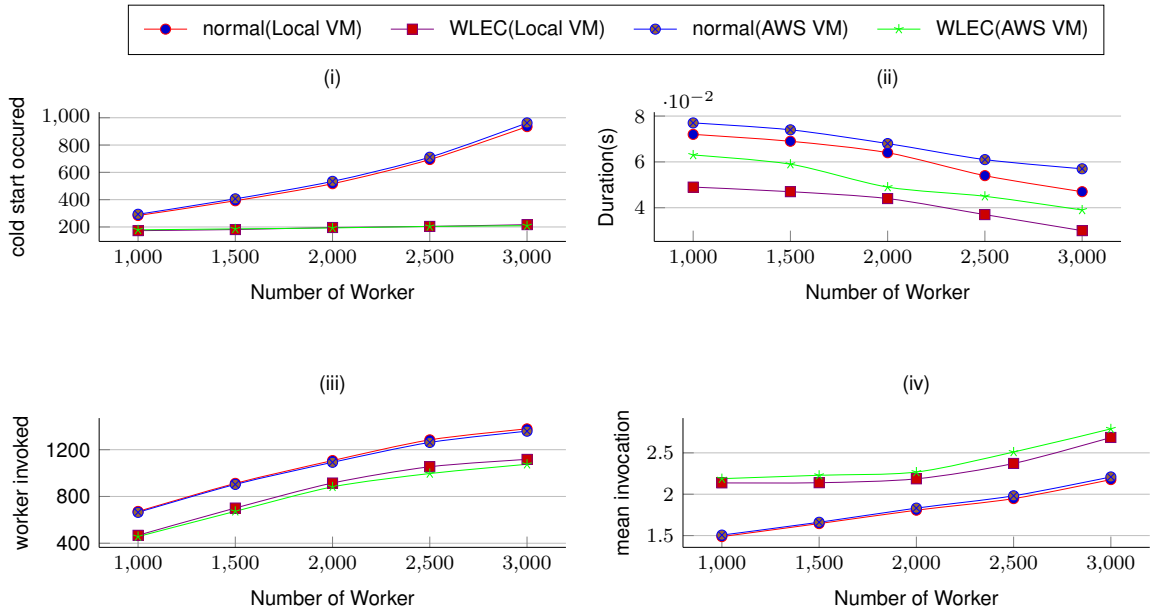

Fig. 4. Different metrics against different number of workers on x axis with same number of request made from client side with normal(without warm up) and with WLEC on OpenLambda in Local and AWS VM with y axis showing (i) number of invocation suffered cold start, (ii) Mean cold duration during cold start invocation of workers, (iii) total number of worker invoked during function request, (iv) Mean invocation per invoked container of (iii).

same time. As the cold start is found, the latency is shown on Y-axis. We find that the cold start duration decreases slightly with the number of available workers for all cases. Initially, the cold start duration is longer with less number of workers. We assume that the cold start duration shortens because of the caching and reuse of workers by WLEC. We can see 21% improvement for Local VM whereas 23.5% improvement for AWS VM with WLEC in addition to around 4.5% standard deviation.

3) Number of Container Invoked: This metric counts the total number of containers that are invoked by the requests from any event or call from the pool of available workers. Like lambda function invocation count, it is also a metric in the serverless platform but only limited to the containers themselves. For the OpenLambda environment, The number indicates how many workers are invoked to the corresponding requests which correspond to lambda events [21]. We change the number of available workers, then make the same number

metrics	*environment*	*mean*	*variance*	*S.D.*
Total number of Container Invoked	Local VM	21.534	23.544	4.852
	AWS VM	23.558	18.771	4.332
Average number of Invocation Per Invoked Container	Local VM	27.964	71.324	8.445
	AWS VM	31.256	61.336	7.832
Number of Cold Start Invocation	Local VM	60.53	174.026	13.191
	AWS VM	60.53	174.027	13.192
Average duration of Cold Start Invocation	Local VM	32.544	3.352	1.830
	AWS VM	25.44	33.970	5.828
Maximum StartUp Latency	Local VM	70.218	644.589	25.389
	AWS VM	77.63	374.009	19.339
Occupied Memory∗	Local VM(WLEC to No Warm)	39.288	40.272	6.346
	Local VM(WLEC to All Warm)	50.36	7.942	2.818

TABLE IV
MEAN, VARIANCE AND STANDARD DEVIATION OF PERCENTAGE IMPROVEMENT BETWEEN NO WARM AND PROPOSED STRATEGY ACROSS DIFFERENT METRICS ON BOTH EXPERIMENTAL ENVIRONMENTS. ∗ WAS TAKEN ONLY ON LOCAL VM AND THE IMPROVEMENT WERE SHOWN BETWEEN WLEC TO NO WARM AND WLEC TO ALL WARM STRATEGY

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

of requests to the lambda functions. We do not select any specific lambda function. We rather keep it random. We do this simulation for both the Local VM and AWS VM. In Fig. 4(iii), we took the different number of available workers and made the same number of requests to the lambda functions. The functions then invoked the workers. From the plot, we can see as the number of workers increases the number of invocation also increases. The reason here is as the number of workers is increased, more fresh workers are available for a lambda function. The invocation increase also indicates that more workers are invoked by the requests and less concurrency occurs in the workers. But as the workers' number grows larger than 2000, we see a slight sluggishness at the invoked container number. It must be the result of the saturation of the required number of free workers. Yet, the WLEC architecture implementation always invokes less number of containers to respond to the requests because of its reuse principle of workers. Between Local and AWS VM, we find AWS invoked little more container invocation which may be caused by network delay.

4) Average Invocation per Invoked Container: The Average Invocation per Invoked Container metric is calculated by the ratio of total lambda request to the number of the invoked container. This metric provides an approximation of the number of times a worker is invoked by the requests. Dealing with burst requests requires the repeated invocation of available workers. High invocation per invoked container indicates a reduction of fresh starts of workers resulting in less cold start occurrence. From this metric, we can predict if any burst has come, how many workers may need to be warmed to deal with the burst. We show the mean of the number of invocations for each invoked container in Fig. 4(iv). We can also denote it as a mean of the number of requests handled by each worker. We find, in both AWS and Local VM the mean of invocation per invoked container is low with the small number of available workers. As it gets more available workers, the mean gets lower. However, WLEC uses the same workers as much as possible which provides higher invocation per worker. The mean increases with the increase of request number because more workers can be alive. Even so, this leads to more invocations in the same workers. From our calculation, we see 27.96% and 31.25% mean invocation increase per container with 8.4% and 7.8% S.D. respectively for Local and AWS instances.

5) Maximum StartUp Latency: Startup latency [20] means the time workers take to take the first fresh start. Maximum start-up latency was calculated from the maximum time taken by any of the given workers. This metric is directly linked to our cold start reduction as the start-up time contributes most of the cold time for the workers. Low latency indicates that the system takes less time to prepare for request handling.Fig. 5 shows how the startup latency is affected by the number of available workers in case of cold and warm conditions. As we can see, initially at the cold state, the start-up latency is low, but as the number of workers increases the latency also increases proportionally as expected. But for a higher

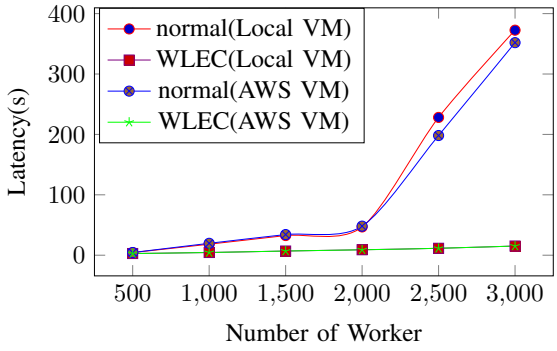

Fig. 5. Maximum Startup Latency vs Number of Worker for OpenLambda

number of workers especially from 2500 workers the latency graph jumped and continues upward from thereon. The latency becomes a major issue as it starts to take around 8 min for 3500 workers in our setup. So, if we use WLEC, we observe that the curve does not jump up like before for any number of workers. Even for 3000 workers, it only takes around 15sec in both AWS and VM instead of 372sec maximum delay shown earlier-resulting 77.6% and 70.2% start-up time reduction.

6) Occupied Memory: We use another metric named occupied memory to describe memory consumption for the different numbers of workers. It indicates the total sum of the memory occupied by all warm containers. It shows the lightness of the architecture which leads to higher memory availability for concurrent executions. Besides a better architecture will always cause less memory occupation despite serving the same number of requests. Fig. 6 shows how the OpenLambda platform behaves in case of the different number of available workers in the Local VM environment. We find that no warm setup requires less memory to function but the performance degrades rapidly because of no caching. For all warm architecture, the memory requirement is unrealistic for large applications as it keeps all the containers warm. Instead

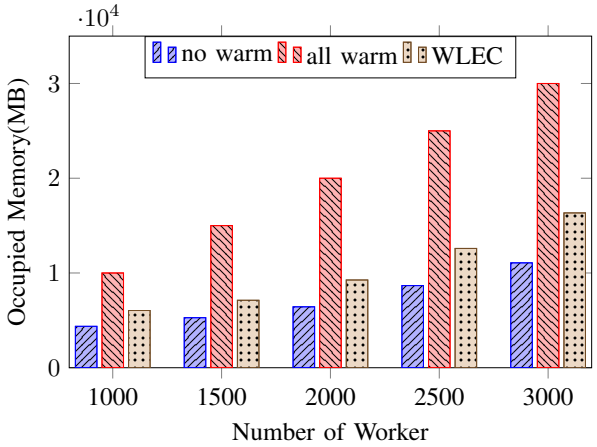

Fig. 6. Occupied Memory vs Number of Worker

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

of WLEC, we can see that the memory requirement is halved from all warm approache though the performance remains acceptable. We keep the container size at 10MB. We also find around 39.28% more memory required for WLEC over no warm approach. On the other hand, we reduce total occupied memory by 50.36% comparing to all warm strategy.

D. Case Study : Image Resizing

To realize the impact of WLEC, we evaluate WLEC by implementing it in a real-time serverless application. For that, we consider an application from the AWS lambda application pool. Then we modify the application based on OpenLambda architecture rather than using its typical AWS build. Next, we implement it in the OpenLambda platform and calculate both platforms and compute latency without and with WLEC to present a proper comparison.

We use on-demand image resizing the application from [33] to evaluate WLEC with the real-time application. To implement it in OpenLambda, we use the Python-based Pillow package [35] instead of AWS Linux based Sharp package from the original application. We use our Local VM set up for simulation. In Fig. 7, We show the platform and compute the latency of Image Resizing application with an average of 20 runs. We can see that the platform latency has been reduced by 6.6 times using WLEC from the basic OpenLambda setup. We also find the compute latency with WLEC is 33ms only where without WLEC it is 361ms-almost 11 times larger. The ready containers from Template Container List in WLEC with pre-installed Pillow packages have made it possible. Because of the nature of WLEC, all package management happened before the lambda request is made. So, we get the lowest compute cost with WLEC.

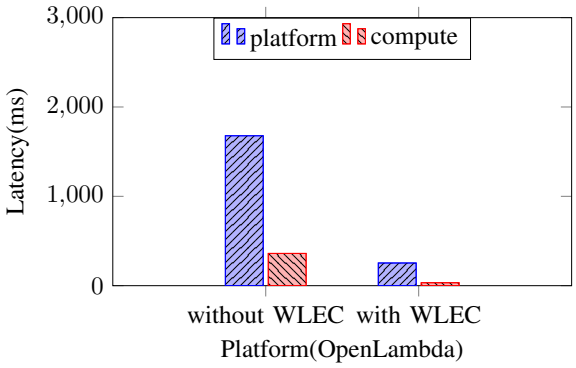

Fig. 7. Latency vs Platform

V. Related Works

Various efforts to reduce the start-up cost of containers have been proposed recently. Oakes *et al.* proposed SOCK [28], an optimized lambda container system with modified Zygote [31] provisioning along with a three-tier caching mechanism to counter kernel overhead. But the implementation was bounded by only python package caching specifically PiPy repository [30]. Chen *et al.* proposed BIG-C [29] a container-based preemptive task scheduling for clusters with heterogeneous workloads. Immediate and Graceful, two types of preemption were proposed to handle task scheduling for both short and long jobs in a shared cluster. For VM level cold boot, we find LightVM [27], a lightweight virtualization technique that has lower boot time than Docker-based container and constant boot time-no matter how many VMs are already running. However, issues like portability, complexity, and generality have made LightVM less inspiring to replace ubiquitous containers yet. McGrath *et al.* [6] made an analysis where they showed that different platforms suffer from cold start time problem. They showed Big platforms like Azure, OpenWhisks, Google, AWS faces cold start around 15 min mark of the containers last invocation. They designed a high performance focused serverless platform and implemented it in .NET where they used Windows containers as a function execution environment. However, their main focus was single function execution. Baldini *et al.* showed the challenges of the serverless platform and defined cold start as a problem that occurs from the scaling to zero feature [5]. In their article, they made a survey of all existing serverless platforms like academia, industry and also open-source projects and identified their key characteristics.

Baird *et al.* [7] showed the performance difference between warm containers and cold containers in the AWS platform. The article also provided a detailed overview of lambda handlers and event-based lambda invocations in the AWS platform. Malishev [13] showed the comparison of different languages like golang, python, java, .net, node js in cold start timing. He also tested with different memory sizes like 128mb, 1024mb, 3008mb of a container. McAnlis [8] tried out to reduce cold start by trimming out the dependencies. Yet it proved to be very language-dependent. Another approach was dependency caching which also suffers from the same problem in addition to non-scalability. He also described a lazy loading idea that can be implemented in a lambda function to reduce the cold start time. Cordova [10] introduced warming up function and delay mechanism. But warming up all lambda functions costs a large memory and is non-scalable. He also proposed to allocate more memory for the lambda containers but memory allocation is costly in the serverless platform which makes the idea impractical for large applications. Cui [9] proved that the cold start improvement techniques of lambda functions are much effective in the case of python and node js because of the pip and npm package management tools respectively. He showed that java and C# suffer 100 times more start time than python. He also mentioned that the code size and memory size improves cold start time linearly. [16] showed how the concurrency can affect the cold start and showed the huge impact on the response time for lambda functions. Daly *et al.* created Lambda Warmer [34] a lightweight module which uses the“ping” method of CloudWatch Event to keep specific lambda functions warm. Lakhsman [17] showed a way where he warms up all the functions every 20 min to make them warm and calls it a hack to reduce the start-up latency.

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.

Serhat [18] described the pros and cons of cold start and elaborated on the reason behind them. However, none of the previous works provides a platform and language independent serverless architecture to ensure reduced cold start delay and high concurrency support at the same time.

VI. Future Work

Though we try to make WLEC as advanced as possible, we believe more researches can be done to eliminate the cold start problem. Elimination may require proper and effective redesigning of serverless platforms like AWS, OpenWhisks and Google Cloud which is costly. Large containers and divergent containers prove a big challenge for our architecture. To be compatible with other lambda platforms can also be challenging. It is also an open problem for Warm Up architecture to solve the random container picking issue in the serverless platform. Instead of a warm queue structure, a warm stack can also be another topic of interest.

VII. Conclusion

As the popularity of serverless computing rises, the cold start has become a more significant issue in the case of request-response architecture. Moreover, developers like to shift from server to serverless platform for an enjoyable and hustle free experience by reducing the cold start time at a minimum. Using WLEC, we are able to reduce the cold start time significantly and make a possibility to eliminate the problem for good except for the first fresh start time. It can also be robust to be integrated with other popular lambda services. As the first attempt of reducing the cold start, we make the first lambda model with built-in dynamic warming up mechanism.

References

[1] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, "Serverless Computation with OpenLambda" in 8th USENIX Workshop on Hot Topics in Cloud Computing(Hot Cloud'16), June 20-21, 2016.

[2] "Thundra: Serverless Observability for AWS Lambda." in https://docs.thundra.io/

[3] Cheng Li, Philip Shilane, Fred Douglis, Grant Wallace, "Pannier: A Container-based Flash Cache for Compound Objects," Middleware '15 Proceedings of the 16th Annual Middleware Conference, Pages 50-62, December 07 - 11, 2015.

[4] Neil Savage, "Going Serverless," in Communications of ACM, No.2, vol. 61, February 2018.

[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter, "Serverless Computing: Current Trends and Open Problems,"in Research Advances in Cloud Computing, pp 1-20, December 2017.

[6] Garrett McGrath, Paul R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in IEEE 37th International Conference on Distributed Computing Systems Workshops, 2017.

[7] Andrew Baird, George Huang, Chris Munns, Orr Weinstein, "Serverless Architectures with AWS Lambda," November 2017.

[8] Colt McAnlis, "Improving Cloud Function cold start time," in in https://medium.com/@duhroach/improving-cloud-function-cold-start-time-2eb6f5700f6.

[9] Yan Cui, "Does Coding Language, Memory or Package Size Affect Cold Starts of AWS Lambda?," in https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76.

[10] Aaron Cordova, "Cold starting AWS Lambda functions," in https://read.acloud.guru/cold-starting-lambdas-2c663055589e.

[11] DirkMerkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." Linux Journal, 2014.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces."in Arpaci-Dusseau Books, 0.91 edition, May 2015.

[13] Nathan Malishev, "Lambda Cold Starts, A Language Comparison" in https://medium.com/@nathan.malishev/lambda-cold-starts-language-comparison-a4f4b5f16a62

[14] Y. Zhou, Z. Chen, and K. Li. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches." in USENIX ATC, 2001.

[15] OpenLambda https://github.com/open-lambda/open-lambda,August 2018

[16] Yan Cui, "Wrong thinking about cold start," https://theburningmonk.com/2018/01/im-afraid-youre-thinking-about-aws-lambda-cold-starts-all-wrong/,January 2018

[17] Lakhsman Diwaakar, "Resolving Cold Start in AWS Lambda," https://medium.com/@lakshmanLD/resolving-cold-start-in-aws-lambda-804512ca9b61

[18] Serhat Chan, "Everything you need to know about cold starts in AWS Lambda," https://hackernoon.com/cold-starts-in-aws-lambda-f9e3432adbf0

[19] Jorg Thalheim, Pramod Bhatotia,Pedro Fonseca and Baris Kasikci, "Cntr: Lightweight OS Containers,"Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC'18), Pages 199–212, July 2018.

[20] AWS Metrics https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-metrics.html/

[21] Invoking Lambda Functions https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-functions.html

[22] Gustavo Totoy, Edwin F. Boza and Cristina L. Abad. "An Extensible Scheduler for the OpenLambda FaaS Platform." in Min-Move'18,March 2018.

[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart and Michael Swift. "Peeking Behind the Curtains of Serverless Platforms." in Proceedings of 2018 USENIX Annual Technical Conference, USENIX ATC'18:133-145, July 2018.

[24] Edward Oakes,Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. "Pipsqueak: Lean Lambdas with Large Libraries." in IEEE Internatinal Conference Distributed Computer System Workshops(ICDCSW), 2017.

[25] Brendun Burns, Brian Grant, David Oppenheimer, Eric Brewer and John Wilkes. "Lessons Learned From Three Container Management Systems Over a Decade" in Communications of ACM, 2016.

[26] Wes McKinney, "Pandas." https://pandas.pydata.org/

[27] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu and Felipe Huici. "My VM is Lighter (and Safer) than your Container." in Proceedings of the 26th Symposium on Operating Systems Principles, (SOSP'17), Pages 218–233, October 2007.

[28] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers." in Proceedings of USENIX Annual Technical Conference (USENIX ATC 18), Pages 57–67, July 2018.

[29] Wei Chen, Jia Rao and Xiaobo Zhou. "Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization." in Proceedings of USENIX Annual Technical Conference (USENIX ATC 17), Pages 251–263, July 2017.

[30] Python Package Index https://pypi.org/

[31] Kyu Kim, "Android Zygote and Dalvik VM." http://davinci-michelangelo-os.com/2017/02/06/android-zygote-dalvik/

[32] Jaiganesh Girinathan and Robert Breckinridge, "Simple Serverless Video On Demand (VOD) Workflow." https://aws.amazon.com/blogs/networking-and-content-delivery/serverless-video-on-demand-vod-workflow/

[33] Bryan Liston, "Resize Images on the Fly with Amazon S3, AWS Lambda, and Amazon API Gateway." https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/

[34] Jeremy Daly, "Lambda Warmer." https://pillow.readthedocs.io/en/latest/

[35] Alex Clark, "Pillow Python Imaging Library." https://github.com/jeremydaly/lambda-warmer

[36] Joe Block, "AWS-Lambda-List." https://github.com/unixorn/aws-lambda-list

Authorized licensed use limited to: National Institute of Technology Karnataka Surathkal. Downloaded on November 11,2024 at 02:43:04 UTC from IEEE Xplore. Restrictions apply.