National College of Ireland

A Deep Learning Based Framework to Initialize New Containers and Reduce Cold Start Latency in Serverless Platforms

MSc Research Project
MSc in Cloud Computing

Surya Kumar Govindan
Student ID: 19103883

School of Computing
National College of Ireland

Supervisor: Manuel Tova-Izquierdo

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Surya Kumar Govindan
Student ID:	19103883
Programme:	MSc in Cloud Computing
Year:	2019
Module:	MSc Research Project
Supervisor:	Manuel Tova-Izquierdo
Submission Due Date:	17/08/2020
Project Title:	A Deep Learning Based Framework to Initialize New Containers and Reduce Cold Start Latency in Serverless Platforms
Word Count:	6850
Page Count:	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

I agree to an electronic copy of my thesis being made publicly available on TRAP the National College of Ireland's Institutional Repository for consultation.

Signature:	
Date:	26th September 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Deep Learning Based Framework to Initialize New Containers and Reduce Cold Start Latency in Serverless Platforms

Surya Kumar Govindan
19103883

Abstract

Serverless computing has recently attracted large significance in the field of Cloud Computing because of its offerings like zero administration, infinite & automatic scaling and fine-grained billing to customers. However, this comes at the expense of having to deal with an unavoidable performance issue - the cold-start latency while initializing new containers. This is an important problem that cannot be ignored as it causes lags and delays in function execution, leaving the platform unsuitable for latency-sensitive applications. To efficiently avoid cold-start latency, this paper proposes a Deep Learning based Serverless (DLS) framework which initializes and sets up containers before the function execution request arrival, as opposed to initializing containers when or after the function execution request arrival. The DLS also makes use of the cache to store and reuse runtime libraries to minimize or avoid the time taken to download and install libraries. The framework provides at least 1.8x times faster function execution times, reduces memory usage up to 33%, and decreases CPU utilization up to 9.5% for applications with recognizable usage patterns, when compared to platforms like Apache OpenWhisk. As platforms like AWS Lambda have started investing towards predictable prewarming of containers, it becomes significant to explore using machine learning in serverless platforms. The reduction in cold start latency benefits both customers and industries in terms of billing and improving resource efficiency respectively. However, the proposed framework's efficiency on user-interaction or dialog-based applications is yet to be analysed.

Contents

1 Introduction

With ever-growing data centers and rapid evolution in the field of Cloud Computing, the resources must be designed and utilized efficiently, as data centers contribute to 2% of greenhouse gas emissions Stenbom [2019]. The latest trend in Cloud - "serverless computing" promises better efficiency in the utilization of resources van Eyk et al. [2018]. Nevertheless, it also brings along certain performance issues, one major issue of them being the latency due to the initialization of new containers, in other words, the "cold start" of containers.

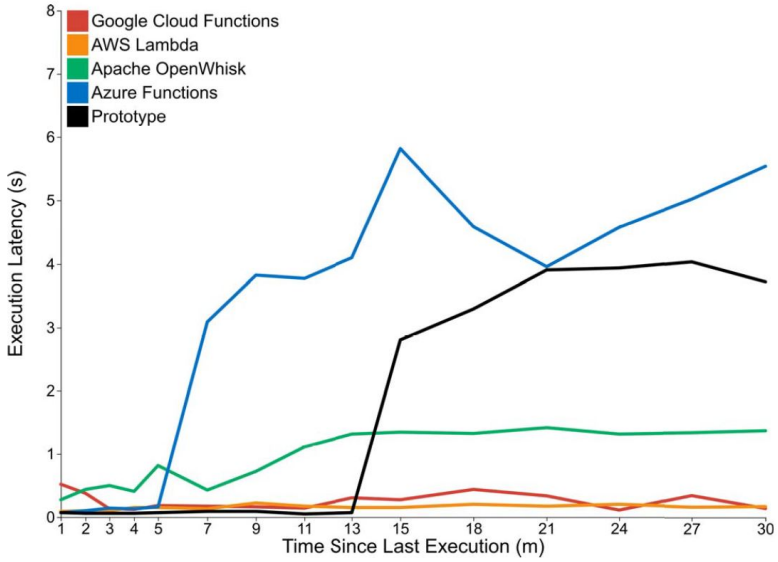

Figure 1: Cold start latency across different Serverless Platforms

1.1 Background

Serverless platforms or Function as a Service (FaaS) are based on an event-driven design, which means that the resources needed to run a stateless function are deployed upon receiving an "event". After the function execution, commercial serverless platforms like AWS Lambda, Google Cloud Functions, IBM Cloud Functions (based on Apache OpenWhisk) keep the resources idle in "warm" state for a grace period (ranging from 10 minutes to 30 minutes) Khondokar Solaiman [2020] to handle future requests. Cold start latency could be of huge concern for latency-sensitive applications like media streaming services, as it impacts the user experience directly. Finding an efficient way to reduce or even avoid cold start latency will certainly be a "win-win" scenario in serverless computing, as it improves the resource utilization efficiency for CSPs and reduces latency issues for customers.

1.2 Importance

It is significant to note that all the serverless platforms suffer from the cold start issue, irrespective of the service provider. Different cold start latency for different service providers could be seen in Figure 1. Existing solutions to cache the runtime libraries efficiently and maintaining a pool of warm containers Khondokar Solaiman [2020] have helped reduce cold starts. However, warm containers will prove beneficial only when

there are consecutive function execution requests. This inspires us to look for a better approach to forecast and predict the times at which the containers may receive an execution request, rather than defining a standard timeout to keep them warm. In this paper, a novel framework for serverless platforms is being proposed, in which the key elements are as follows:

1. Efficiently cache the most used libraries and reuse them to setup new containers
2. A resource manager to direct new function execution requests to warm containers
3. A deep learning model to predict the times at which a function may arrive
4. Pre-initialize the containers at the predicted times with libraries installed

1.3 Research Question

This research project has been directed towards finding whether the approach mentioned in the following research question could solve the cold start latency in serverless platforms:

"Can the latency due to cold start of new containers in serverless platforms be reduced using a deep learning model-based framework to forecast function execution request arrival times along with a novel cache algorithm on serverless platforms like OpenWhisk?"

1.4 Objectives

By implementing this framework alongside serverless platforms (Apache OpenWhisk in our case), the following objectives are aimed to be met.

- Reduce the latency due to download of runtime libraries by efficiently caching and reusing them
- Maintain an efficient cache management system to store only the frequently used libraries in the cache and evict the ones that are not
- Forecast the function execution request arrival times for the next day, based on the previous history using a deep learning model
- Schedule and setup the containers with all the runtime libraries installed beforehand and have the container ready for execution when a new function request arrives

1.5 Limitations

In the scope of this research project and experiments conducted, the dataset is limited to manually generated logs from crontab jobs scheduled for 10 days. A certain pattern in the dataset is needed to train the deep learning model efficiently. Hence, the dataset has been manually generated accordingly and the framework is not tested on real-time dataset. Additionally, the proposed framework is limited only to Python-based executions at the moment and only a set of Python libraries have been used for simplicity in comparison.

1.6 Report structure

The rest of this report is structured as follows. The literature review of related work in this domain (2), followed by methodology used (3), design specifications (4), details about implementation - (5), evaluation of experiments conducted (6), the conclusion and discussion on future work (7) and finally the bibliography.

2 Related Work

Most of the commercial serverless platform providers use Docker as their containerization technology Lloyd et al. [2018] to provision resources faster and on demand. Even though the containerization technology behind AWS Lambda is not publicly disclosed, we saw already that all serverless platforms suffer a lag in resource initialization Amazon Web Services [n.d.b], before they can execute a function. It is also significant to note that the cold-start latency is independent of the type of the runtime, but may differ based on the programming language used Lloyd et al. [2018], as shown in Figure 2 that the average execution time is different for each runtime. Multiple researchers have proposed and tried different techniques to reduce and overcome cold-start latency in serverless platforms. In this section, a few such notable proposals, their effectiveness and results will be discussed and critically analysed. Three main solutions that have been put forth by researchers so far to deal with cold-start latency are as follows:

- Caching and reusing the runtime libraries
- Reusing warm containers
- Prewarming a pool of containers
- Predictive warming of containers

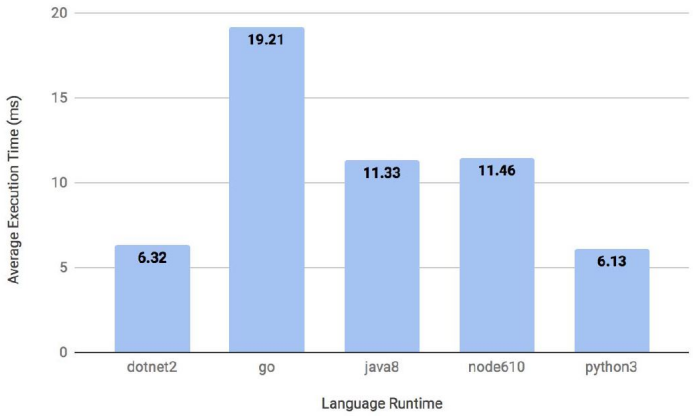

Figure 2: Average execution time for different runtimes on AWS Lambda

2.1 Caching runtime libraries

An obvious and conventional solution in computing to speed up processes is the use of the cache. For a cold-start issue as well, researchers have tried to install runtime libraries faster from the cache, rather than having to download them each time. Oakes et al. mention that executing a function in a serverless platform involves the following - download the runtime libraries, decompress them to disk and then load them onto memory Oakes et al. [2017]. This means that the larger the runtime libraries, the higher the resource initialization time for a container. Nevertheless, the initialization time is unavoidable and expecting the developers to create functions with lightweight containers is not practical. Hence Oakes et al. propose Pipsqueak, which has a set of Python-based interpreters in the cache, where they are already downloaded, decompressed and stored.

This design has proved to be faster in terms of container initialization because it utilizes cache-stored interpreters. However, Pipsqueak does not evict the packages that are not used frequently and hence the cache may become overloaded. Additionally, though Pipsqueak was successful for Python, we do not know its efficiency for runtimes like Java, which depends on JVM, which in itself needs additional time to setup Lloyd et al. [2018].

Even when the runtime libraries are cached and used, typically all resource handlers in today's serverless platforms do not route new function requests to containers where the functions could execute faster. In other words, the schedulers are not aware of which packages are cached in which containers and they route the requests to any container that is available Aumala et al. [2019]. Aumala et al. propose PASch (Package-Aware Scheduling) which aims at making schedulers more aware of cache contents of each container and route the function requests accordingly. This is aimed at identifying the container in which the function could execute faster. PASch has proven to increase the median cache hit ratio to 64.1% while the ratio for a typical scheduler without PASch is 51.2%. Nevertheless, PASch also tends to overload the same container in some cases, thus compromising on load balancing.

To overcome the load balancing issue in Pipsqueak and PASch, Oakes et al. created SOCK Oakes et al. [2018], which uses isolation primitives to make Docker containers more lightweight. SOCK is based on Zygote provisioning strategy, which uses an effective cache mechanism to avoid repeated initialization of Python runtime. This is achieved by what is called as "Import-Cache Eviction" in SOCK. This eviction mechanism identifies libraries that are not needed by the parent process and evicts them from cache to save memory. This ensures that the packages that are not needed are cleaned up eventually. SOCK shows promising results in improving the runtime initialization speed by 45x, compared to typical Docker initialization times. This shows that investing in a direction towards an efficient cache management system could contribute significantly towards the cold-start reduction in containers.

Starting from serverless platforms like AWS Lambda to proposals like SOCK, we should notice that the containers are designed to handle only one request at any given point in time Akkus et al. [2018]. To overcome this, Akkus et al. created SAND, in which the same container could handle more than one request simultaneously. This way, the same cache could be used and accessed by all the function executions from the same

container. SAND forks new processes inside the same container - that is, wherever two containers are needed for two functions typically, SAND needs only one. This eliminates the startup latency that would have been needed for initiating the second container, as the runtime libraries are setup only once. Nevertheless, SAND only supports programming languages in which forking of processes is possible. Additionally, since SAND accommodates multiple functions in a single container, it poses security concerns and inter-functional performance constraints.

However, when we largely depend on cache storage for faster access to runtime libraries, the size of the cache becomes a concern. The question "*How large should a machine's cache be?*", to accommodate all necessary packages is still unanswered by the proposals discussed so far. Increasing the cache size of a machine generously may not be an option, as the cache is very expensive to build and hence they typically tend to be smaller in size. In such cases, we will have to rely on external cache broker systems like Redis alongside the serverless platform to have a dynamically sized cache. As a relatively recent proposal, Ghosh et al. prove that using external cache with serverless platforms reduces the cold start latency significantly Ghosh et al. [2020]. Their proposal used in-memory internal caching, implemented using AWS ElastiCache alongside AWS Lambda. This not only proved to improve the response times of containers, but also improved the cache hit ratio significantly.

2.2 Warm containers

Warm containers are those containers which are kept ON and in an idle state so they could readily execute a new function as and when they arrive, without the need to reinitialize the runtime anew [16]. This definitely is an alternative to new containers, but is not a solution to the cold start latency issue. Lin et al. create a pool of warm containers to overcome cold start latency Lin and Glikson [2019]. These containers are built on Knative, a Kubernetes based open-source serverless platform. In this design by maintaining a pool of warm pods, Lin and Glikson claim that an image classifying application in their evaluation reduced the cold start latency times by a minimum of 50% for concurrent executions. By reusing warm containers in Apache OpenWhisk Apache OpenWhisk [n.d.], Oliver et al. Stenbom [2019] have produced encouraging results up to 20x faster execution times as compared to cold starts. However, the achievement is at the expense of resources. To reduce resource consumption, different Cloud Service Providers maintain a maximum grace time or threshold time for which a container could stay in a "warm" state [18]. It is noteworthy to mention that warm or idle containers also consume the same amount of resources consumed by a container which is actively executing a function.

McGrath and Brenner McGrath and Brenner [2017] propose a novel mechanism in serverless platforms that has queues to maintain the list of cold and warm containers available. By maintaining a queue of resources, the design takes control over the memory allocation of each container. In other words, if a new function requires lesser memory than what is actual memory of the container, then the queue reclaims the memory and allocates it to other containers and functions where more memory might be needed. This way, the design ensures that the resources are optimally allocated. Nevertheless, even in this design, a threshold of 15 minutes for containers in the warm queue is being maintained Amazon Web Services [n.d.c] Esposito et al. [2018]. In an analysis performed by

Wang et al. Wang et al. [2018], behaviour and performance of warm containers were compared across serverless platforms from popular Cloud Service Providers - AWS Lambda, Microsoft Azure Functions and Google Cloud Functions. Upon execution of Node.js functions at a given interval, it was noted that AWS Lambda showed faster execution times than the others. They claim that this could be possible because AWS Lambda has better-preallocated resources, in other words, because of warm containers.

2.3 Prewarmed containers

Another alternate to using warm containers is "prewarmed" containers, which platforms like Apache OpenWhisk offer by default Mohan et al. [2019]. Prewarmed containers are those containers that could be seen at a state that is neither "cold" nor "warm". This means that the prewarmed containers need not be initialized from scratch like in cold containers but also they cannot be used immediately for execution like in warm containers. For instance, when a new function execution request arrives, the runtime libraries are already present Akkus et al. [2018] in a prewarmed container. However, the dependencies of the code will not be loaded yet Mohan et al. [2019]. This solution may not completely eradicate cold start issue like in warm containers, but may help reduce at least sets up the runtime based packages beforehand. In platforms like OpenWhisk, customers get to choose the number of such prewarmed containers to be initiated based on the runtime (Python, Node.js, etc.) and they could be maintained as a pool. Nevertheless, prewarming of containers also reduces the cold start at the expense of resources. Even when there are no function requests, the prewarmed container pool always keeps a list of containers switched on and thus consuming resources. Mohan et al. Mohan et al. [2019] state that the memory footprint in warm and prewarmed containers is high because of caching the libraries. However, they state that instead of precreating the whole container, it would be efficient to precreate network-related tasks like namespace creation, IP allocation, etc., as network setup takes the longest time in container initialization Mohan et al. [2019]. Their experiment also proves that by pre-creating network components alone, nearly 80% of cold start latency could be reduced.

2.4 Predictive warming

Reusing warm containers and prewarming of containers have undoubtedly contributed significantly to avoid cold start latency in serverless platforms. But these techniques help solve the issue at the expense of resources. The resources are not efficiently utilized, as the service providers keep the containers warm or prewarmed even when there are no function execution requests. This makes it important to accurately predict the time at which a container needs to be kept warm. Hence it becomes inevitable to employ machine learning models in serverless platforms Xu et al. [2019] Zhang et al. [2019] to tackle cold-start issues.

The warm-up strategy is a technique used by serverless platforms to restart containers at regular intervals to achieve lesser cold start times. However, this strategy also decreases the efficiency in resource utilization. Xu et al. Xu et al. [2019] propose an adaptive warm-up strategy of containers to predict the frequency at which the containers should be retained in "warm" state, based on the previous history. They employ a Recurrent Neural Networks model - Long Short-Term Memory (LSTM) to analyse the

time-series data and use specialised gating mechanisms to precisely predict the frequency. By doing so, they were not only able to improve the prediction accuracy of invoking a function, but also reduced cold- start latency and memory and CPU utilization. AWS Lambda has also attempted to precisely provision containers based on predictable start-up times of function execution Amazon Web Services [n.d.a]. This means that where a customer knows that certain applications experience heavy loads, then Lambda provisions the resources beforehand according to the runtime and so when these loads actually start to come in, Lambda readily executes them without any cold-start issues.

But in both these attempts of predictable warming of containers, we could observe that very little or no attempts have been made to precisely forecast when a function execution may exactly arrive based on the previous history. If function execution times could be predicted accurately, containers can be made warm beforehand, not only with the runtime environment but also with the specific libraries needed by the function. In this paper, this approach has been experimented by implementing the LSTM deep-learning model, with Python as the runtime. The implementation and evaluation will be discussed in detail in the upcoming sections.

3 Methodology

3.1 Steps

The proposed framework is aimed at making use of the cache, warm containers and deep learning model to reduce the cold-start latency. Hence, these components have been integrated into a novel serverless framework created using Python, named and henceforth will be addressed in this paper as a Deep Learning-based Serverless (DLS) Framework. The DLS framework has been implemented to work only for Python-based functions at the moment. Initially, Apache OpenWhisk OpenWhisk [n.d.a] was installed and Python functions were executed to analyse the cold and warm executions. OpenWhisk by default was observed to provide 10 minutes of threshold time to keep the containers warm after execution. Additionally, users can create a pool of prewarmed containers based on the runtime (Python, NodeJS, Java, Scala, etc.). The DLS framework has been created with Docker as the containerization technology. The framework was tested using three sample Python functions, with basic libraries like sql. The reason behind choosing these two libraries is that they have no dependent packages and hence are simple for analysis and comparison. The execution times of three Python functions were tested on two servers - one with both DLS framework and Apache OpenWhisk installed and the second with only Apache OpenWhisk installed.

To train the deep learning model to forecast the function execution times for the next day ($(t+1)^{th}$ day), crontab jobs were scheduled to generate the required dataset for (t) days. The deep learning model was trained using the generated dataset. The model parameters were tuned to obtain the best possible accuracy, and then the forecast of function execution times for the $(t+1)^{th}$ day was made. The execution times of containers with and without DLS were measured and the memory, CPU utilization were also recorded on these two machines. Final analysis and comparison of the results were performed and concluded whether the DLS framework efficiently reduces the cold-start latency.

3.2 Materials and equipment

For developing the DLS framework and implementation on a Cloud Platform, two virtual machines of size - 4GB memory, 2CPUs and 30GB storage have been used. On the first machine, DLS framework, Apache OpenWhisk & its prerequisite packages were installed and on the second, Apache OpenWhisk was installed as standalone. The setup and installation of prerequisites for the DLS framework and OpenWhisk have been automated using a bash script to keep the manual effort of the user to a minimum. The list of components used for developing and implementing the DLS framework, the corresponding products or vendors chosen and their versions are shown in the table 1.

Table 1: Components and corresponding Products used for DLS Framework

Component	Product
Virtual Machine	Amazon Web Services (EC2)
Operating System	Ubuntu 18.04.4 LTS
Serverless platform	Apache OpenWhisk 0.9.0
Cache manager	Redis 4.0.9
Containerization	Docker* 19.03.12
Docker image	Python:rc-alpine3.12
Programming language	Python 3, Bash
Database	CouchDB* 2.3.1

As for the virtual machines, initially equivalent machines were ordered on the National College of Ireland's OpenStack offering. However, external access was needed on several ports of the components for administrative purposes. Since the OpenStack machines had restricted public access, they were dropped eventually and two AWS EC2 instances of size 't2.medium' were chosen for implementation.

3.3 Data collection and preparation

To train the deep learning model and make predictions of the function arrival time for the next day, the dataset had to be generated. For machine learning models to train efficiently, a certain pattern is necessary for the dataset. For this reason, three simple Python functions (scripts) were created with simple Python packages like sql, datetime. These three Python functions were executed as crontab jobs in intervals of 5,10 and 15 minutes respectively for 10 days (from 28-07-2020 to 06-08-2020) on one of the machines. The timestamp of execution call, the script being executed and the total duration of execution were recorded. This way, upon running the three scripts for 10 days, a dataset of 5277 records was generated. This dataset had to be cleaned and prepared such that it becomes a suitable input to train the deep learning model. For training the LSTM based deep learning model used for this project, we are considering the timestamp of the function execution request arrival as the main variable. Hence, the time parameter in the format HH:MM:SS has been extracted from the timestamp column and added as a new column in the format HHMMSS. Additionally, a number for each unique script has been allocated and a new additional column has been added as "Process".

3.4 Model training and prediction

The Long Short-Term Memory (LSTM) recurrent neural network proves to be a suitable model for time series forecasting A. Sarah and Kim [2018]. As discussed earlier, the fundamental concept behind training the LSTM model is that from the logs generated for 10 days, the data of the first 9 days ((t-1) days) will be used for training and the data of 10^{th} day (t) will be the test data. In this proportion, the deep learning model has been trained and the accuracy of the model is derived. Parameters like the no. of epochs, training batch size, optimizer and the loss were tuned in the model to achieve the best possible accuracy, considering the 'Time' column. Once desired accuracy was obtained, the schedule for the next day (11^{th} day in our case) is forecasted. The framework starts the containers a few seconds before the forecast timestamps, along with the runtime and libraries installed. Hence, when a new function request arrives, the scheduled containers execute the functions without suffering any cold start, thus eliminating delayed executions. To train the model efficiently again to predict the timestamp values for each upcoming day, a rolling forecast scenario can be used. But in this implementation, the model was trained only with 10 days' data and predictions only for the 11^{th} day was made. Thus a fixed approach has been followed, making predictions for each new day, one step at a time.

3.5 Measurements

To compare the cold-start behavior while executing a given Python function on Apache OpenWhisk and the proposed DLS framework, the total execution times of the functions triggered on the 11^{th} day (07-08-2020) were measured. A logging mechanism has been created on the DLS framework - the timestamp, total execution time and type of execution (cold or warm) are logged and stored. In the DLS framework, besides the time taken to spin up the the docker container, we are defining the total time taken for executing a function into two parts - t_{setup}, the time taken to set up Python libraries of a function and $t_{execute}$, the actual time taken to execute the function after the libraries have been set up in the container. But as per Oakes et. al Oakes et al. [2017], the t_{setup} is made of two actions:

1. Downloading the dependant libraries ($t_{download}$)
2. Installing the libraries onto the container ($t_{install}$)

Hence, the DLS framework calculates t_{setup} as the sum of download and installation times of each library present in the function or script to be executed. Hence, t_{setup} could be denoted as shown below, where n is the number of dependent libraries.

$$t_{setup} = \sum_{i=1}^{n} (t_{download} + t_{install})$$

An important point to be noted is that when the library does not need to be downloaded and is already available in the cache, then $t_{download} = 0$. As per the design of DLS, the containers should be started t_{setup} seconds before the function arrival. The value of t_{setup} depends on the number of dependent libraries in a Python script to be executed. The t_{setup} time for each library will also be calculated and stored in Redis. These calculations are represented in the following equations.

Total time for executing a function in DLS, $t_{total} = t_{setup} + t_{execute}$ ——————1

Time for setting up dependant libraries, $t_{setup} = t_{download} + t_{install}$ ——————2

$\therefore\ t_{total} = t_{download} + t_{install} + t_{execute}$ ————3

Therefore, the forecast feature will start the Docker containers always t_{setup} seconds before the forecast time. However for Apache OpenWhisk, the total execution time of a function (denoting as w_{total}) has different components. OpenWhisk depends on "action", which needs to be created and invoked every time a function is to be executed OpenWhisk [n.d.b] Sciabarrà [2019]. Once "action" is invoked, the logs can be checked under the activation logs OpenWhisk [n.d.b]. However, the execution time shown in the activation logs is only the function execution time ($w_{execute}$). The time taken to create and invoke the action (w_{action}) needs to be taken into consideration, to make a fair comparison with the DLS framework. Hence, the total time taken for function execution in OpenWhisk involves two components - w_{action} and $w_{execute}$. We however cannot measure the time taken to install the dependent libraries in case of OpenWhisk.

$\therefore$ Total time for executing a function in OpenWhisk, $w_{total} = w_{action} + w_{execute}$ ————4

4 Design Specification

4.1 Architecture

The architecture of the proposed DLS framework can be seen in Figure 3. The framework and its corresponding packages could be made to work on any operating system supporting Docker. However, since this is a cloud deployment, a VM based on Linux is used. As the framework has been programmed using 'bash', at the moment it is designed to work only for Linux based virtual machines. Three main components of the framework are:

- Resource Manager
- Cache Manager
- Deep Learning Model

The resource manager is responsible for creating the Docker containers needed for execution and also for deleting them immediately after execution. It also checks whether any warm container is present from OpenWhisk or DLS and directs them to it when available. The cache manager keeps track of all the libraries used by the Python functions and stores in Redis storage the information about each library inside it and thus maintains a repository. Here, Redis storage acts as the cache broker for the DLS framework and is installed on the same VM as the framework, to avoid network latency. This makes it faster to identify the path of the cached Python packages and use the packages in "whl" format for setting up docker containers.

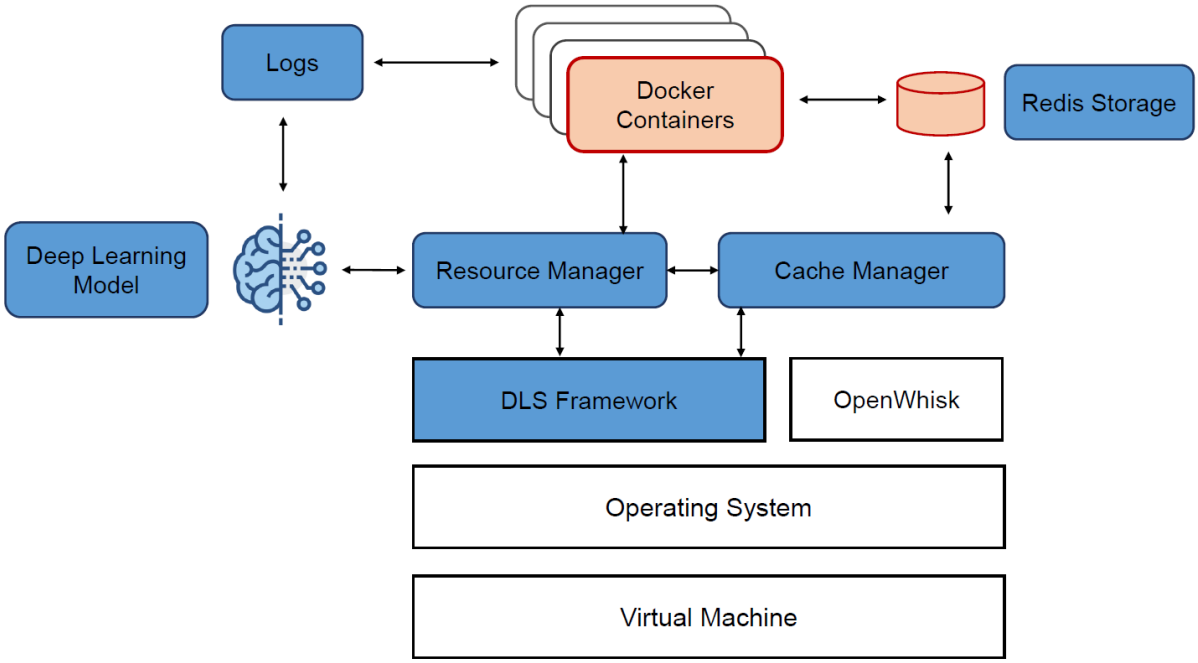

Figure 3: Architecture of the proposed DLS framework

Finally, the logs of all the executions and their runtime are logged. These logs (timestamp of request, function to be executed & duration of execution) will be the original dataset which will be cleaned and transformed to train the deep learning model. The model gets trained and predicts the function arrival times for the next day. This predicted times will be fed again to the resource manager and it schedules the Docker containers needed for the next 24 hours accordingly. This way, the containers are scheduled beforehand along with libraries needed by functions. Thus the three components work hand-in-hand and form the significant features of this framework. The execution flow of the framework could be seen in the figure below.

4.2 Algorithms

In the algorithm 1, the resource manager is responsible to identify the container in which the function could be executed. If a warm container is present, the resource manager checks if it is running any function already. If the pulse is 0, then the warm container free and can be used to execute a new function and then after execution, the container is destroyed to save resources. If the warm container is already executing a function (the pulse being 1) and if no other warm containers are present, then a new container is started up trying to make use of the cached libraries wherever possible. Each time a new function execution request arrives, it is the resource manager's responsibility to identify the list of libraries used in the function and then inform the cache manager accordingly.

In the algorithm 2, we could see the logic behind the cache manager component of the DLS framework. The name of all the libraries used by the user's code will be recorded and stored in Redis by the cache manager. In this repository, the details like how frequent a library is called, the path to the cache of this library, its last used date and the number of times a given library is used are stored. By storing the frequency with

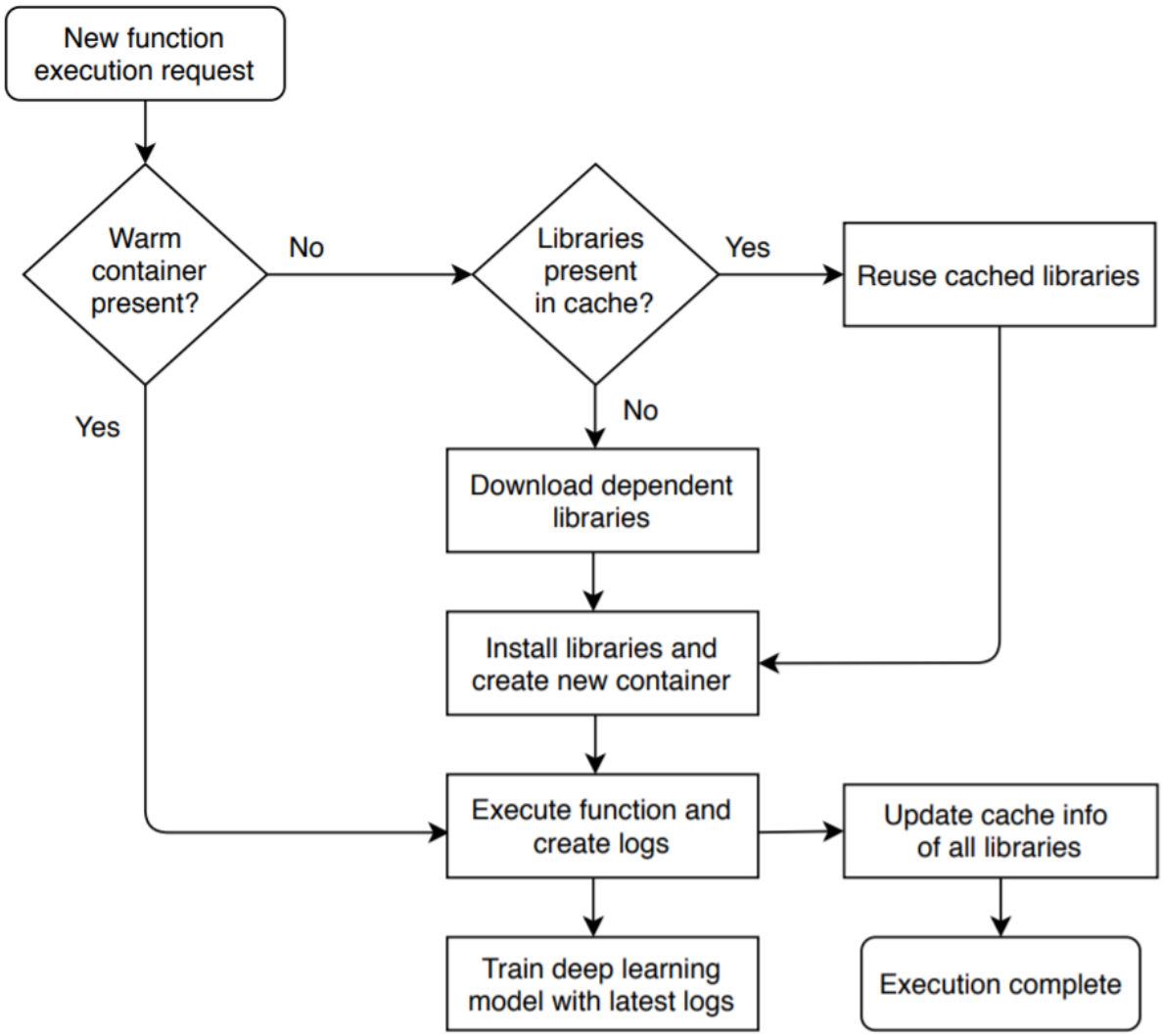

Figure 4: Flow of execution of DLS framework

which a library is used, the cache manager decides whether it can be stored in the cache or not. If it's a frequently used library, the cache manager retains it in the cache so it can be used for installation faster. However, in order to become frequent, the number of times a library is used should reach the threshold value which is 5, in our case. Besides, the cache manager also evicts the library from the cache by checking the last used date. If the number of days since the library was last used is more than 10 days, then the cache manager evicts it from the cache and saves space. Thus, the cache manager ensures that the DLS framework has faster access to the library it needs to install and also ensures the cache storage is not wasted by unused libraries and is efficiently managed.

Algorithm 1: Resource manager

```
Input: New function execution request
Result: Executes function in a warm container if it exists. Else creates a new
        container and executes
pulse;                                  // defines if a warm container is under use
function;                                  // function or code to be executed
containerList;                          // list of warm containers identified
if containerList >= 1 then
    for eachContainer do
        if pulse = 0 then
                                                // warm container identified is free
            Execute the function in the warm container;
            Destroy the container after execution;
        end
        else if pulse = 1 then
                                              // identified warm container is in use
            Identify the next warm container if available;
            Execute the function in the warm container;
            Destroy the container after execution;
        end
    end
end
else
                                                        // no warm container identified
    Invoke a new container action;
    Use cached libraries to setup new container;
    Execute the function in the new container;
    Destroy the container after execution;
end
Inform cache manager about new function execution;
```

The time complexity of Algorithm 1 is $O(N^2)$

Algorithm 2: Cache manager

Input: Information about new function request from Resource Manager
Result: Identify the library needed by the new function execution request and update the cache depending on the frequency of usage

```
library;                    // Name of the library needed by the new function
lib_freq;                   // Frequency status of library
NEW, FREQUENT, INFREQUENT   // Statuses a library can have
lib_count;                  // Frequency of number of times the library is used
threshold;                  // Threshold frequency to decide if the library stays in cache
lib_last_used;              // The date a library was last used
while new function request do
    Identify the frequency at which the library is used;
    Store the frequency of the library in status;
    if status = NEW then
        Initiate frequency count for new library;
        Store frequency count;
        Store the last used date;
    end
    else if status = FREQUENT then
        Retain the frequency count;
        Retain the cache path in redis storage;
        Store the last used date;
    end
    else if status = INFREQUENT then
        Increase and store the frequency count;
        if frequency = threshold then
            Update redis storage with cache path of the library;
            Store the last used date;
        end
    end
    if last_used >= 10 then
        // Delete the library from cache if it's not used for more than 10 days
        Delete the library cache path in redis storage;
        Delete the library from the cache Reset frequency count and last used
         date;
    end
end
```

The time complexity of Algorithm 2 is O(N)

5 Implementation

The DLS framework depends on Docker for containerization and Python as the runtime language. Hence, Docker and Python 3 are mandatory prerequisites of the framework. Along with DLS, OpenWhisk components will also be installed so the user can choose which platform to execute their functions in. All the prerequisites installation and setup of the actual framework have been automated and the install.sh bash script takes care of the setup end-to-end to keep the user inputs to a minimum.

The three main components of the proposed DLS framework - the resource manager, cache manager and the deep learning model for time series are accommodated in the Python-based script that will be called "function" (similar to "wsk" in OpenWhisk platform). The "function" command supports only Python-based executions at the moment. Support for other languages like JS, Golang is scope for future work. When executed, "function" looks for the existence of a valid Python script with appropriate extension and then identifies the list of libraries that are mentioned in the script. This is the special feature of the DLS framework in a way that it automatically detects the libraries needed in the Python script and sets up the container accordingly. However, platforms like OpenWhisk need these libraries to be mentioned in the "requirements.txt" file before the functions can be executed. Additionally, DLS also checks with the Python packages repository Python Software Foundation [n.d.] whether the libraries given in the script are valid. When the right Python script is given as input to "function", it identifies a Python-based Docker container for it to run. The framework finds and allocates a "warm" or "cold" container based on availability and then runs the function in it. If the libraries needed are in cache already, then the cache path is found and is used for installation inside the Docker container. If not, the libraries are downloaded and then installed. Upon execution, the logs like the timestamp of the request arrival, the actual function, the duration of execution and the type of execution (cold or warm) are collected and saved, as shown in Figure 5. The execution logs can either be stored in the database or as CSV format. However, for experimentation purposes, the logs are stored in a CSV file so it is convenient to prepare the data and use it for training the deep learning model.

```
Timestamp, Function, Duration, Type
2020-08-07 14:54:50,test.py,1353,warm
2020-08-07 15:21:54,test.py,2696,cold
2020-08-07 15:35:23,test.py,2772,cold
2020-08-07 15:45:01,test.py,2622,cold
2020-08-07 16:30:01,test.py,2730,cold
2020-08-07 16:35:22,test.py,1208,warm
2020-08-07 16:35:56,test.py,1231,warm
2020-08-07 16:36:05,test.py,1219,warm
```

Figure 5: Logs created upon function executions by DLS framework

```
root@ip-172-30-0-177:~# ./function hello.py

Runtime is Python!

Installing required libraries...

Processing /cache/pip/wheels/1a/18/5a/3c918b3de538cabab699fe6a29e0361313bf5a2d7e0b82325a/sql-0.4.0-py3-none-any.whl
Installing collected packages: sql
Successfully installed sql-0.4.0


Output of the execution is:

{
Hello World!
}

Execution complete! Function execution time is - 372ms
```

Figure 6: Output of "function", reusing cached libraries

Figure 6 shows the output of running a Python script "hello.py" using the "function" command of DLS framework, showing the function execution time ($t_{execute}$). We could see that the library "sql" mentioned in the Python script is identified and the framework tries to install it automatically. Since it is available in the cache and it is directly used and installed onto the container without having to download.

Once the data about the function execution requests are collected, the timestamp data of the function requests are separated and formatted as described in section 3.3. This data now serves as the input to train the deep learning model and hence predict the possible function execution time arrival for the next day. Python libraries TensorFlow (for deep learning) and Keras (for neural network) have been used to train the LSTM model as this involves time series problem. The output of the model gives us an overall accuracy of 88.88% and the model summary is shown in Figure 7. This training step in this experiment is manual as the logs are fetched from the server and trained and tested separately. However, automating the data cleaning, preparation and formatting and training the model automatically is a part of future work. Finally, the output of the Docker containers scheduled to start at "forecasted" times or at the times predicted by the model can be seen in Figure 8.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_1 (Bidirection (None, 100)               24000
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 24,101
Trainable params: 24,101
Non-trainable params: 0
_________________________________________________________________
Train on 4749 samples, validate on 528 samples
```

Figure 7: Summary of the LSTM model training

```
root@ip-172-30-0-177:~# ./function forecast
One container(s) scheduled to start at 00:29:57
One container(s) scheduled to start at 00:44:57
One container(s) scheduled to start at 00:59:57
One container(s) scheduled to start at 01:14:57
One container(s) scheduled to start at 01:29:57
One container(s) scheduled to start at 01:44:57
```

Figure 8: Output of using the "forecast" feature of DLS

6 Evaluation

The below experiments were performed to find the efficiency of the proposed DLS framework and its comparison with Apache OpenWhisk implementation. The technical details, experiments and the results of these case studies have been compared and reviewed in the following sections.

6.1 Experiment 1

In experiment 1, two AWS t2.medium machines of the same configuration (RAM, memory, CPUs & storage) have been set up as machine 1 hosting the DLS framework and Apache OpenWhisk and machine 2 hosting Apache OpenWhisk standalone. A test Python script "helloWorld.py" was executed on machines 1 & 2. The reason for choosing this script is that it was already used to generate the dataset and the model for the DLS framework has been trained accordingly. The only dependant library for this script is "sql". The script was triggered as usual by the crontab jobs on the $(t+1)^{th}$ day (07-08-2020). For the same day, the Docker containers were scheduled to start at the times forecast by the DLS framework for this particular script, using the 'forecast' feature. To keep the comparison fair, on machine 2 as well the same Python script (packaged into a ZIP file along with 'requirements.txt' as per the procedure for OpenWhisk) was scheduled to be executed by OpenWhisk exactly at the forecast times using crontab job scheduler. We consider 10.00AM to 05.00PM as the evaluation period. The execution times t_{total} & w_{total} (as calculated in section 3.5) were measured in milliseconds for accurate measurement and the results of the same could be seen in Figure 9.

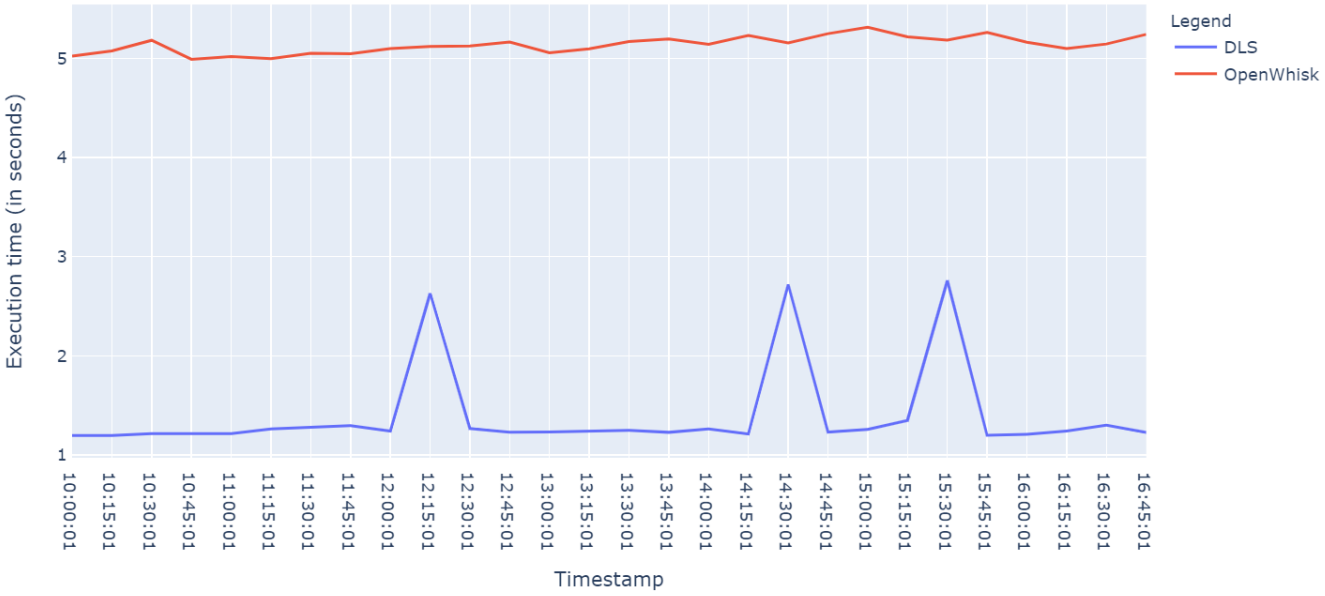

Figure 9: Function execution time of DLS vs OpenWhisk in Experiment 1

The measurements are similar to those calculated by Ghosh et al. Ghosh et al. [2020] where they calculate the total execution time right from the arrival of the request until the completion of execution. However, in the case of OpenWhisk, the total time is to be calculated including the time taken to create and invoke the action, which is often overlooked as it explains the time needed to set up and install the dependent libraries on the container for execution (as discussed in section 3.5). Figures 10 & 11 show the memory used and CPU utilization percentage.

6.2 Experiment 2

We discussed in section 3.1 that the threshold time for warm containers in OpenWhisk is 10 minutes. Hence, it becomes important to notice and compare the execution times of DLS with OpenWhisk for function executions with a frequency of less than 10 minutes.

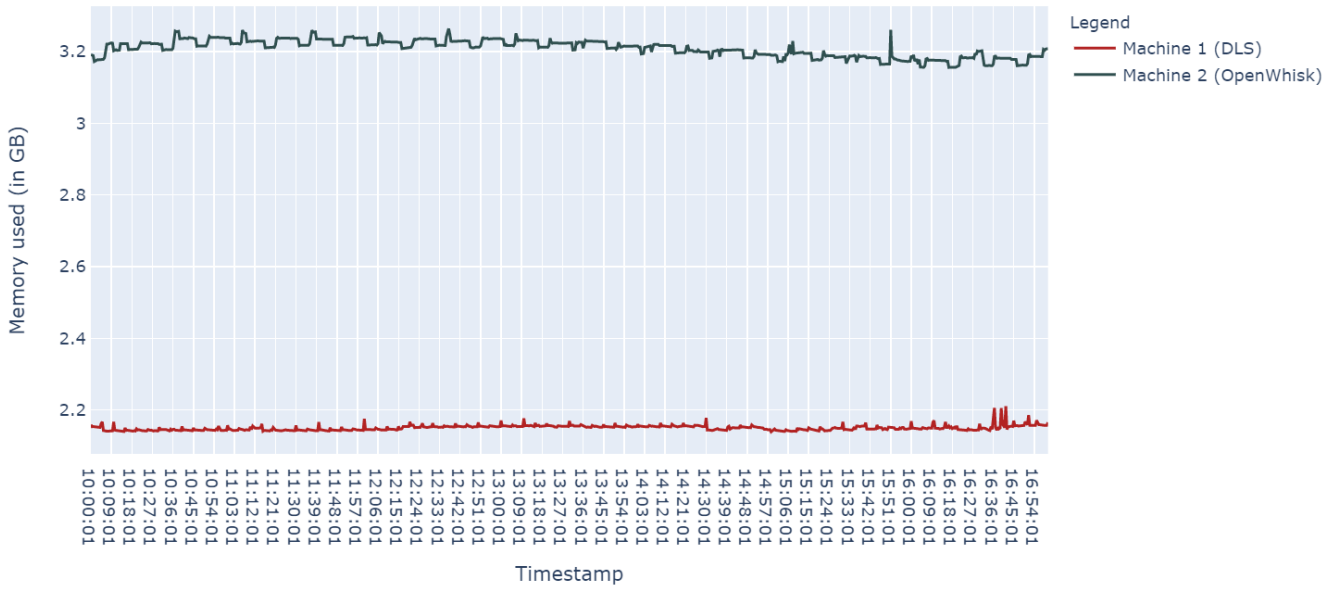

Figure 10: Memory usage on machines 1 & 2 during evaluation period

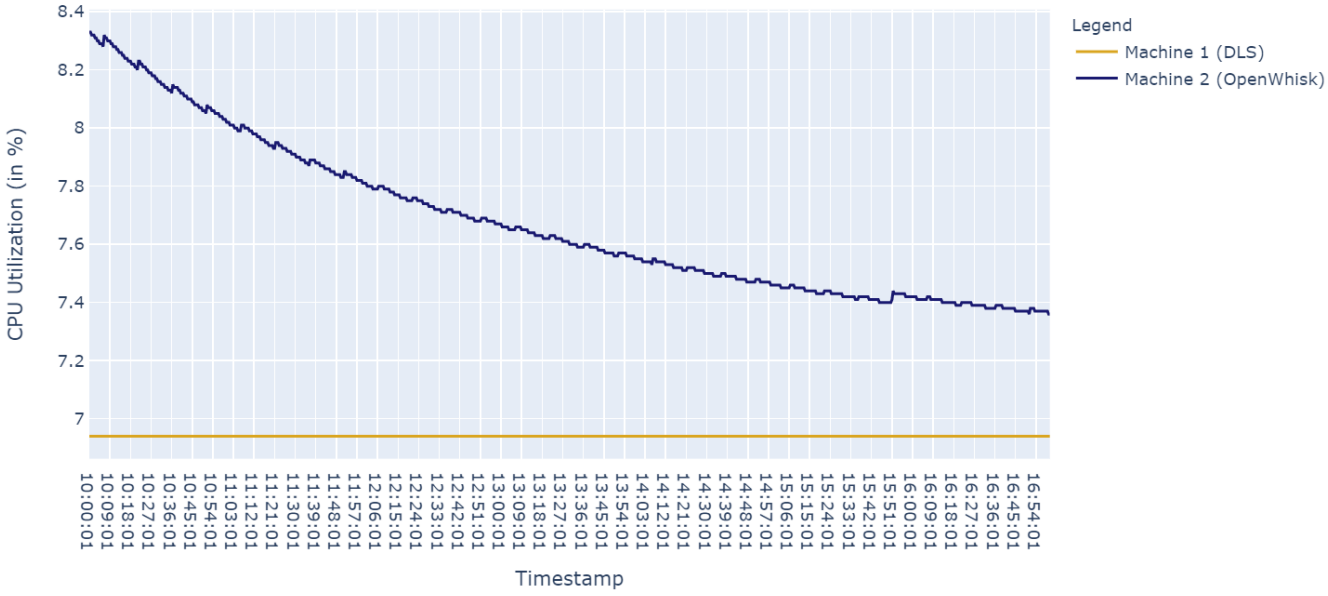

Figure 11: CPU utilization on machines 1 & 2 during evaluation period

Therefore, as a second experiment, the same script "helloWorld.py" was scheduled on both machines 1 & 2 with regular intervals of 5 minutes. On machine 1, the crontab jobs were scheduled for 5 minutes and forecast was done accordingly for this frequency to start the containers beforehand. A manual forecast was made using the "function" script on machine 1 and crontab jobs were scheduled to execute the script "helloWorld.py" at 5-minute interval on machine 2 with OpenWhisk respectively. The results of execution times of both machines for 3 hours from 08:00 AM to 11:00 AM could be seen in Figure 12.

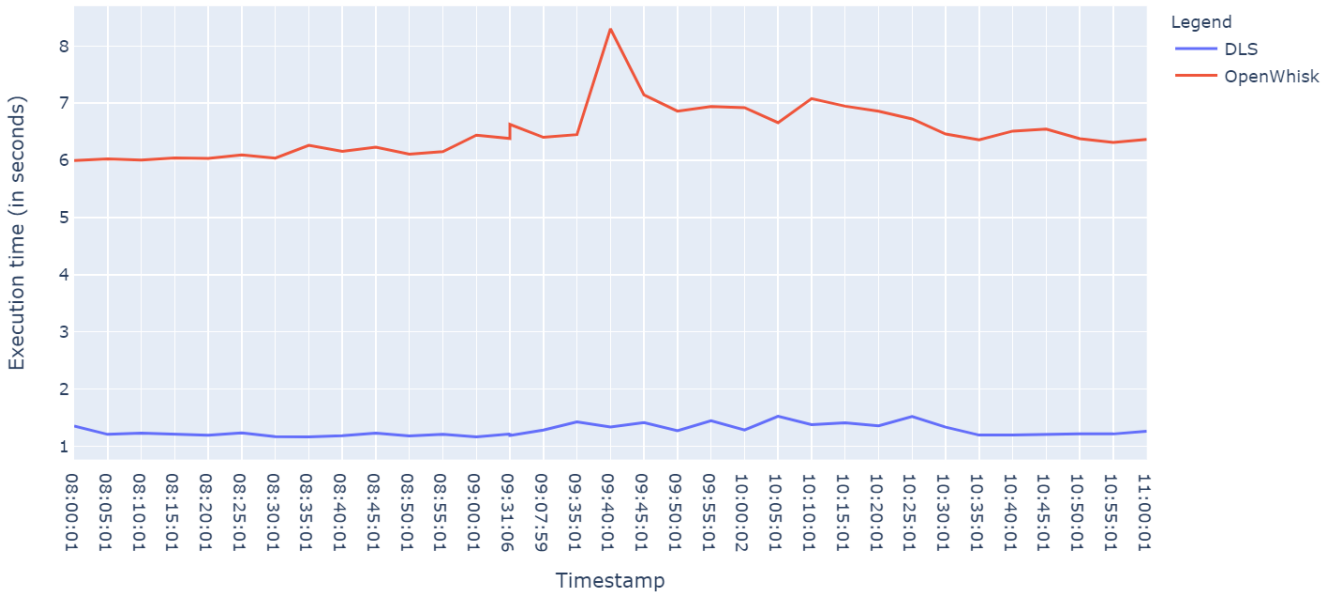

Figure 12: Function execution time of DLS vs OpenWhisk in Experiment 2

6.3 Experiment 3

As a final experiment, it is important to measure the turnaround times in DLS - t_{total} without the forecast feature. That is, to have a fair analysis and comparison we have to also test the framework in terms of what happens when the predicted times for the next day are inaccurate or if the actual execution times for $(t+1)^{th}$ day follow a new and different pattern. In other words, all executions could be "cold" executions. Hence to make a comparison of the same on OpenWhisk, we again schedule the script "helloWorld.py" at 15-minute frequency so there will be 100% cold executions on both the platforms. The results of the same could be seen in Figure 13.

6.4 Discussion

From Figure 9, the mean total execution time of the script "helloWorld.py" using DLS framework (t_{total}) was found to be 1.39 seconds, while the same for OpenWhisk (w_{total}) was 5.13 seconds. This justifies the improvement in the DLS framework that when the containers are accurately predicted and started before a function execution request arrives, the execution times could be reduced by nearly 72.9%, which is a significant improvement. The three spikes in DLS show three instances where there were cold starts, which could be explained by a longer time to install the libraries or an error in the predicted request arrival times. It is very important to note that this improvement is provided by two important factors - containers started before the request arrival and also setting up the dependant libraries beforehand. Since the library "sql" is present already in the cache, a significant improvement has been achieved. Even if the library is not present in the cache, only the first five executions will take additional time to download the library, post which the library becomes a "FREQUENT" library and then will be stored in the cache, as per the design of DLS. From figure 14, we could see the mean memory usage and CPU utilization values on machines 1 & 2 during the evaluation period. From the values, we could derive that the mean memory on machine-1 with the DLS framework was 33.12%

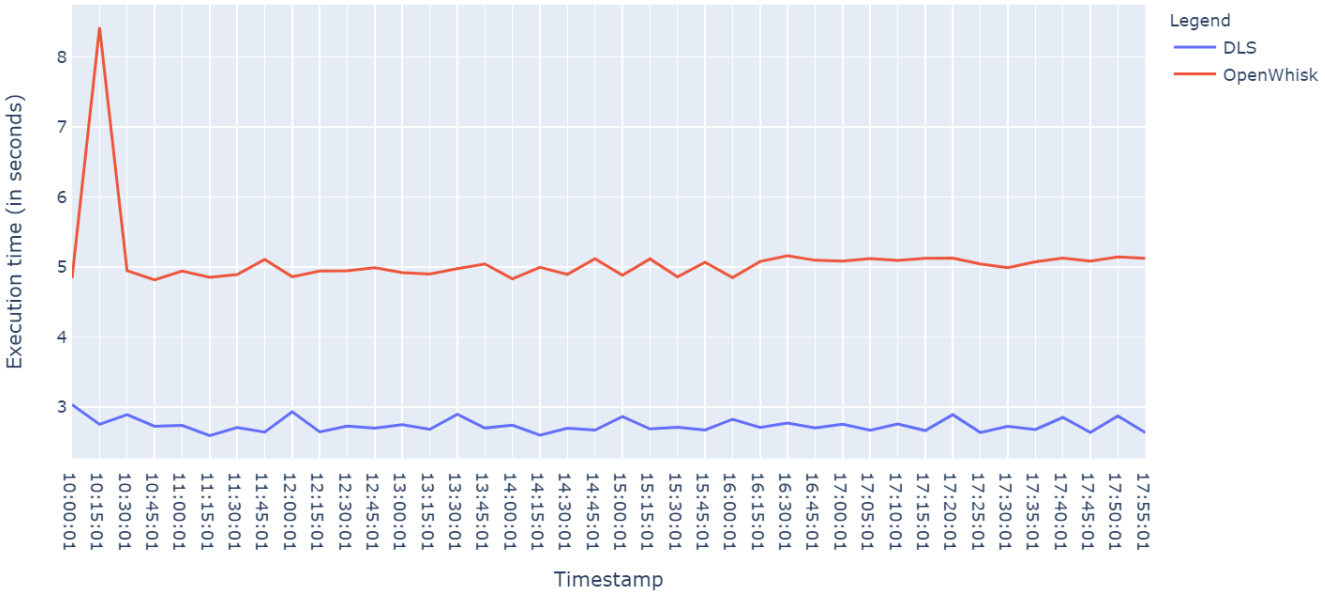

Figure 13: Function execution time of DLS vs OpenWhisk in Experiment 3

lesser than the machine 2 with OpenWhisk. The CPU utilization could also be seen to be 9.5% lesser on machine 1 during the evaluation period, as seen in figures 10 & 11. This could be because on the machine with OpenWhisk, the resources are retained for 10 minutes even after the execution is complete. But DLS tends to delete the containers immediately after execution, thus consuming lesser resource.

Metric	Machine 1	Machine 2
Mean Memory (GB)	2.14	3.2
Mean CPU utilization (%)	6.94	7.67

Figure 14: Mean memory usage and mean CPU utilization on machines 1 & 2

Figure 12 shows the execution times of the script with 5-minute frequency where all the executions in both OpenWhisk and DLS were warm. Even when the executions were warm, the mean execution time with DLS (t_{total}) was 1.27 seconds and with OpenWhisk (t_{total}) was 6.49 seconds. This explains that even when the executions are 100% warm, the creation and invoking of action with dependant Python libraries in OpenWhisk take additional time. On the other hand in DLS, reusing cached libraries and keeping the container warm before request arrival provides us up to 5x improvement in execution time. In the experiments 1 & 2, execution with DLS has mostly remained warm executions, since the forecast feature starts the containers beforehand (t_{setup} seconds before the forecast time). But experiment 3 was conducted with the forecast feature turned off and all the executions were cold executions on DLS. The results in Figure 13 show us that with all cold executions, the mean t_{total} was 2.7 seconds and mean w_{total} was 5 seconds. The final experiment also shows us that the overall execution times even without the forecast feature in DLS proved to be 1.8x faster than OpenWhisk.

7 Conclusion and Future Work

Serverless computing is undeniably one of the finest advancements to date in the field of Cloud Computing. However, performance issues like cold-start latency are yet to be addressed, so they become suitable for latency-sensitive applications as well. In the experiments performed on the DLS framework, we see that the cold-start latency could be efficiently avoided and resource utilization could be improved by initializing the containers before a function execution request arrives. This paper evaluated the usage of deep learning in serverless platforms to accurately forecast the function execution request times. We also saw that caching the runtime libraries efficiently and reusing them to set up new containers reduces setup time, as also proved by Oakes et. al Oakes et al. [2018] and Ghosh et. al Ghosh et al. [2020]. The linear time-series based deep learning helped forecast the upcoming request arrivals based on the past logs and thus helped forecast when should the containers be kept warm in future. Most serverless platforms today have a hard threshold time defined for warm containers and this timeout is the same for all customers and applications. This type of "one-size-fits-all" may not be suitable for customers with different usage patterns. Hence, it is extremely important to alter the approach to maintain warm containers based on type of application used and the usage patterns of each customer. This way, DLS framework proves to be efficient in making resources "warm" only when the model predicts that a function execution request "will" arise based on previous data and not under the assumption that it "may" arise, as compared to today's serverless platforms.

The DLS framework produced at least 1.8x times faster execution times, avoiding the need for cold-start to a great extent. Nevertheless, this framework will prove to be successful only when there are recognizable patterns in the function executions and usage. For instance, this will prove to be 100% efficient for applications where regular background jobs run at a given frequency. When there is no recognizable pattern, then the deep learning model may not prove to be accurate enough in forecasting, thus may lead to wastage of resources. This way, the proposed framework is limited to applications with regular patterns e.g data processing & ETL workloads, batch processes in ERP systems like SAP in which jobs usually run at regular intervals. Hence, this may not be suitable for typical user-interactive applications with random or irregular usage patterns. Besides usage, some libraries may not be stored by default in the VM's cache. In such cases, having to purchase additional cache storage to download and store libraries may also limit the framework's usage in terms of expenditure. Therefore, a trade-off between reduced cold-starts and cost should be made by customers to implement this framework.

Owing to cost and the framework being limited to applications with recognizable usage patterns, further research needs to be done with real-time usage data and analyse whether this framework will be suitable for user-interactive applications with irregular usage patterns. The framework did prove to be successful on the generated dataset which of course had regular jobs scheduled using crontab. Only when tested on real-time usage logs, the framework's efficiency on latency-sensitive and user-interaction based applications could be deduced. Additionally, the framework now has been implemented to work only on Python-based scripts, which could be extended for other languages and technologies like Java, Golang, Swift, etc., which also could be considered as scope for future work.

References

A. Sarah, K. L. and Kim, H. [2018]. LSTM Model to Forecast Time Series for EC2 Cloud Price, *16th International Conference on Dependable, Autonomic and Secure Computing, DASC 2018*, IEEE, Athens, pp. 1085–1088.

Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. [2018]. SAND: Towards high-performance serverless computing, *USENIX Annual Technical Conference, USENIX ATC 2018*, USENIX Association, Boston, pp. 923–935.

Amazon Web Services [n.d.a]. AWS Lambda - Predictable start-up times with Provisioned Concurrency, `https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/`.

Amazon Web Services [n.d.b]. AWS Lambda Developer Guide, `https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf`.

Amazon Web Services [n.d.c]. AWS Lambda Developer Guide, `https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf`.

Apache OpenWhisk [n.d.]. OpenWhisk Documentation, `https://openwhisk.apache.org/documentation.html#using-openwhisk`.

Aumala, G., Boza, E., Ortiz-Avilés, L., Totoy, G. and Abad, C. [2019]. Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms, *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, IEEE, Cyprus, pp. 282–291.

Esposito, C., Palmieri, F., Kim-Kwang and Choo, R. [2018]. Serverless Computing: From Planet Mars to the Cloud, *Computing in Science & Engineering* **20**(6): 73–79.

Ghosh, B. C., Addya, S. K., Somy, N. B., Nath, S. B., Chakraborty, S. and Ghosh, S. K. [2020]. Caching Techniques to Improve Latency in Serverless Architectures, *International Conference on COMmunication Systems & NETworkS, COMSNETS 2020*, IEEE, Bengaluru, pp. 666–669.

Khondokar Solaiman, M. A. A. [2020]. WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing, *8th IEEE International Conference on Cloud Engineering, IC2E 2020*, IEEE, Sydney, pp. 144–153.

Lin, P.-M. and Glikson, A. [2019]. Mitigating Cold Starts in Serverless Platforms, **abs/1903.12221**. `http://arxiv.org/pdf/1903.12221.pdf`.

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L. and Pallickara, S. [2018]. Serverless Computing: An Investigation of Factors Influencing Microservice Performance, *IEEE International Conference on Cloud Engineering, IC2E 2018*, IEEE, Orlando, p. 159–169.
URL: *https://doi.org/10.1109/IC2E.2018.00039*

McGrath, G. and Brenner, P. R. [2017]. Serverless Computing: Design, Implementation, and Performance, *IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, IEEE, Atlanta, p. 405–410.
URL: *https://doi.org/10.1109/ICDCSW.2017.36*

Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N. and Sukhomlinov, V. [2019]. Agile cold starts for scalable serverless, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019*, USENIX Association, Renton, pp. 57–69.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T. and Andrea C. Arpaci-Dusseau, R. H. A.-D. [2017]. Pipsqueak: Lean Lambdas with Large Libraries, *International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, IEEE, Atlanta, pp. 395–400.

Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. [2018]. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers, *USENIX Annual Technical Conference, USENIX ATC 2018*, USENIX Association, Boston, pp. 57–69.

OpenWhisk [n.d.a]. OpenWhisk Documentation, `https://openwhisk.apache.org/documentation.html#openwhisk_architecture`.

OpenWhisk [n.d.b]. OpenWhisk Documentation, `https://openwhisk.apache.org/documentation.html#actions-creating-and-invoking`.

Python Software Foundation [n.d.]. Python Package Index (PyPI), `https://pypi.org/`.

Sciabarrà, M. [2019]. Learning apache openwhisk, *O'Reilly Media, Inc.* **1**.
URL: *https://learning.oreilly.com/library/view/learning-apache-openwhisk/9781492046158/ch07.html*

Stenbom, O. [2019]. Refunction: Eliminating serverless cold starts through container reuse, *Department of Computing, Imperial College London* p. 7.
URL: *https://github.com/ostenbom/refunction/blob/master/report.pdf*

van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A. and Iosup, A. [2018]. Serverless is More: From PaaS to Present Cloud Computing, *IEEE Internet Computing* **22**(8481652).

Wang, L., Li, M., Zhang, Y. and Ristenpart, T. [2018]. Peeking Behind the Curtains of Serverless Platforms, *USENIX Annual Technical Conference, USENIX ATC 2018*, USENIX Association, Boston, pp. 133–145.

Xu, Z., Zhang, H., Geng, X., Wu, Q. and Ma, H. [2019]. Adaptive Function Launching Acceleration in Serverless Computing Platforms, *IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, Tianjin, pp. 9–16.

Zhang, C., Yu, M. and Yan, W. W. F. [2019]. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving, *USENIX Annual Technical Conference, USENIX ATC 2019*, USENIX Association, Renton, pp. 1049–1062.