A Deep Reinforcement Learning approach for Optimizing Function Allocation in Serverless with a Distributed Image Registry

Submitted for completion of the research project requirement of CS798-001 and CS798-002 W22 in the University of Waterloo

JUSTIN SAN JUAN (20887688), University of Waterloo

RIZWAN SHAHID (20957854), University of Waterloo

SUPERVISED BY: RAOUF BOUTABA, University of Waterloo

SUPERVISED BY: BERNARD WONG, University of Waterloo

The emergence of the Function-as-a-Service (FaaS) paradigm enables software developers to focus on business logic all the while entrusting the service provider in a serverless environment to elastically scale up during a burst of requests or scale down when request rates are low. One of the key enablers of FaaS is the widespread adoption of containerization technology such as Docker, which packages software into images that can be loaded and executed in any other computer running the Docker engine. An encumbering limitation of FaaS is significant overhead when downloading these images to newly instantiated workers from a centralized image repository. Large bursts of requests cause new workers to be instantiated and lead to so-called "cold-start" latencies. Recent works have investigated distributed image registries in the flavor of downloading images from existing workers and the scheduling of functions with knowledge of the server resource availability and function request information separately. However, the scheduling of functions in servers distant from existing images can lead to sub-optimal function completion times, especially in the context of a bursty workload. FaaSFabric addresses these issues by incorporating the information of image locations into the scheduling decisions. In this paper, we present fine-grained overhead breakdowns of workload traces, a Mixed Integer Linear Program (MILP) formulation of incorporating distributed image registry information in request allocation decisions, a justification of the potential of Deep Reinforcement Learning (DRL), and a proposed DRL formulation. We also demonstrate that in a simple workload, image pulling can take up to 80% of the total function completion time on average, which shows the potential for a distributed image registry-aware scheduler to outperform baseline and state-of-the-art schedulers.

1 INTRODUCTION

In recent years, the Function-as-a-Service (FaaS) paradigm has become increasingly popular in the software development community. It is a trend away from monolithic on-site-hosted servers to modularized cloud-hosted functions. It has

Authors' addresses: Justin San Juan (20887688), University of Waterloo, justin.sanjuan@uwaterloo.ca; Rizwan Shahid (20957854), University of Waterloo, rizwan.shahid@uwaterloo.ca; Supervised by: Raouf Boutaba, University of Waterloo, rboutaba@uwaterloo.ca; Supervised by: Bernard Wong, University of Waterloo, bernard@uwaterloo.ca.

gained widespread adoption especially in event-driven contexts such as video encoding in the case of Netflix [4]. FaaS is typically implemented in a "serverless" environment where service providers execute the developer's functions on a shared cluster of servers. This allows service providers to elastically provision resources among tenants, reduce costs from idle servers, as well as reduce marginal infrastructure costs, thus achieving economies of scale. This reduces costs for both the service provider and the tenants. AWS Lambda [5], Microsoft Azure Functions [19], Google Cloud Function [10], IBM Cloud Functions [12], and Alibaba Cloud Function Compute [3] are a few of the major cloud providers that offer FaaS platforms.

Two key drivers of FaaS adoption include the development of light-weight containerization technology and the dynamic and bursty nature of compute workloads. Using containers to deploy cloud functions has been gaining popularity[28]. Containers are more suitable for heavy workloads like machine learning, data analytics, and video processing which often require large dependencies. Such workloads are difficult to support otherwise with limited function package support. Containerization technology, such as Docker, packages application code, written in any programming language such as Python, Java, NodeJS, Go, etc., along with system libraries and tools into a file called a container image and then deploys it to any computer running the Docker Engine. Containers isolate the computation environment by running as an isolated process in the user space and including the file system in the image. This prevents dependency issues when running on a different computer and allows flexibility on what computer performs the execution. On the other hand, virtualization technology such as virtual machines perform heavyweight isolation and replication by making a complete copy of OS and necessary system libraries [2]. The lightweight nature of containerization technology compared to virtualization enables function code to be loaded onto servers and removed relatively quickly, avoiding unnecessary costs incurred from idly waiting. Meanwhile, the dynamic and bursty nature of compute workloads causes the static allocation of servers for any service to be sub-optimal in resource utilization efficiency and other costs from uncompleted requests due to resource unavailability timeouts [6]. Instead, a variable allocation of resources based on the demand performs better resource utilization efficiency as well as service performance. This is commonly referred to as elastic resource allocation due to the nature of increasing and decreasing the amount of resources provisioned on-the-fly.

This paper focuses on the service provider side of the paradigm. The major high-level objective is to maximize resource utilization efficiency while minimizing average function completion time. Maximizing resource utilization implies that idle resources are automatically and swiftly reallocated for use by other functions. Meanwhile, in order to minimize average function completion time, the service provider's infrastructure must provide minimal overhead while keeping function code replicated and loaded in servers in a warm state, ready to be executed. While essentially all commercial cloud FaaS providers provide Service-Level Agreements (SLAs) in terms of service availability, practically all of them do not provide performance guarantees and serve functions on a best-effort basis [20]. The aim for the service provider is then to balance these objectives to provide the fastest service with the lowest costs.

As a new paradigm, FaaS has its limitations. In particular, platform overhead produced in applications modularized into many small functions tend to be significant relative to the execution time of the functions. This is because the function images may need to be loaded and executed on a server separately for each invocation. As observed in production serverless systems such as Alibaba Cloud Function Compute [28], a large portion of the total function completion time is attributed to pulling the image into an execution node. This delay is also known as a cold start. This type of issue occurs when the system uses a centralized image registry and receives a burst of requests. As a large number of requests arrive in the system, a proportionally large number of new servers request the function image from the centralized image registry, inciting a network bottleneck at the image registry.

In order to address the Docker image distribution problem, recent works have proposed the usage of distributed image registries. An example of such registries is Kraken from Uber [27], which uses a peer-to-peer (P2P) mode of communication. The nature of a P2P network works best when needing to distribute large images to multiple nodes. It does so by defining two types of agents: a seeder and a leecher. First, the data of interest is broken down into chunks. Then a seeder distributes this to another leecher node. Afterwards, as the leecher node receives the chunk, it can then become a seeder and send that chunk to another leecher node. By distributing different chunks of the data to different leechers based on which chunks they are missing, a greater portion of the overall network is utilized in comparison to all clients downloading data from a centralized server cluster.

Another distributed image registry uses a binary tree structure [28]. This structure allows newly instantiated servers to download the image while limiting the likelihood of network bottlenecks. However, such works have not considered the optimization of function allocation in the context of a dedicated distributed image registry with bursty workloads. In the study of optimizing function allocation, several works have implemented linear programs as well as deep reinforcement learning (DRL) solutions to minimize average function completion time. These models generally incorporate information about the availability of resources in running servers as well as function metadata such as requested memory and average initialization time into the allocation decisions[31]. FaaSFabric improves on this by further incorporating the replication information of images in distributed nodes to minimize the delays incurred from pulling images, thus optimizing the average function completion time.

Modern FaaS scheduling algorithms treat the function allocation problem as a bin-packing problem[29]. Using the counts of function requests and their corresponding memory requirements, the scheduling algorithm calculates the minimum number of servers to serve these requests. While the co-location of function requests due to the bin-packing algorithm reduces cold starts by attempting to minimize the changes in number of servers, it also leads to performance degradation due to many function requests sharing the same resources in a server and thus receiving a smaller slice of CPU time and virtual memory [29, 31]. The typical workaround for this issue in commercial services is to limit the number of in-flight requests at each server, which also causes fragmentation of server resources that lead to sub-optimal resource utilization. It is shown that commercial FaaS platforms such as AWS Lambda still use a bin-packing type of scheduling algorithm [29].

The bin-packing algorithm is often implemented as an integer linear program, which incurs temporally expensive calculations. This is often impractical in the context of FaaS where thousands of requests must be served in the order of milliseconds [29, 32]. Heuristic bin-packing algorithms have also been proposed [7, 9] to address the prohibitively long calculations while maintaining a reasonable level of optimization in terms of average function completion time. However, the order of magnitude of scheduling overhead incurred by such algorithms are still considerable. Furthermore, these heuristics often do not incorporate the bursty nature of FaaS workloads, and would thus fail to adequately provision servers and resources in the time scale of a burst of requests [21]. Works such as [33] propose a model that incorporates the time-series nature of workloads and optimizes the allocation of servers using predictions of the workload; however, the work does not provide a practical implementation beyond a quadratic programming solution in the context of FaaS. To validate that the incorporation of image replication information improves function allocation decisions, a Mixed-Integer Linear Programming model could be designed and implemented.

Deep Reinforcement learning is a strong candidate to address the limitations of bin-packing and linear program solutions. In FaaSRank, a score function is learned by the machine learning model to address the issues of incorporating thousands of server information, enabling cluster scalability, as well as the huge action space of scheduling thousands of functions to any of thousands of servers [31]. When an invocation is received, the score function is applied to each server to calculate a rating and the server with the highest rating is selected for the allocation. The strength of this type of machine learning formulation is that the model's inference can be completed in the order of magnitude of function execution times. Meanwhile, it can also be scalably implemented by executing the scoring of each server in parallel. This is also a benefit of this model in comparison to a standard machine learning output formats which have a fixed-size output length representing the maximum number of servers in the service provider's infrastructure.

In order to tackle these problems as well as verify that the pulling of images to new servers is a significant bottleneck on the critical path, a fine-grained breakdown of the different overheads incurred in a function invocation must be measured. The open-source serverless platform OpenWhisk is used in this project due to its popularity in the software development community and other works. In OpenWhisk, the path of a request can be divided into five invariable stages: authentication, scheduling, queuing, image pulling, code initialization, and execution. However, the current system only reports metrics in terms of: "wait time", "init time", and execution duration. In OpenWhisk, a executor node is called an Invoker. The "wait time" is time spent between the controller receiving the activation request and when the invoker provisioned a container for the action. The "init time" is the time spent initializing the function after the content of the source code has been downloaded to the invoker. More work, as presented in this paper, is needed to achieve fine-grained measurements to pin-point areas of improvement in the system as well as to validate the overall improvement achieved by the proposed model.

Due to limitations in time and resources, the implementation of the DRL model will not be completed. Instead, a formal justification of the potential for reinforcement learning to provide improvements in the system and supporting measurements will be provided alongside a proposed formulation of the DRL solution.

The main contributions of this paper are:

(1) Fine-Grained Breakdown of Platform Overheads
(2) MILP formulation for incorporating distributed image registry information
(3) Justification of Potential for Reinforcement Learning
(4) Proposed Deep Reinforcement Learning Formulation

2 BACKGROUND AND MOTIVATION

In this section, we describe FaaS platforms and existing function scheduling algorithms. To demonstrate the necessity for an intelligent scheduler, we offer an example considering a scenario in a realistic FaaS infrastructure. We also show how reinforcement learning is used to learn scheduling policies.

2.1 Distributed Image Registry

A number of solutions exist on distributed docker image registries. Among the notable ones is Kraken[27]. It supports image management, replication, and distribution in rapid cloud environments. It is highly scalable and provides availability guarantees. The Kraken architecture consists of a tracker that orchestrates participating nodes to form a pseudo-random, regular, highly connected graph. These nodes then share data in a similar fashion to BitTorrent. One limitation of Kraken is that it requires dedicated hosts to seed the content to the network. Another solution is Dragonfly[1] which is also a P2P-based image distribution system specifically designed to speed up large scale distribution. It comprises of four components: a manager, scheduler, df-daemon and CDN. The manager is responsible

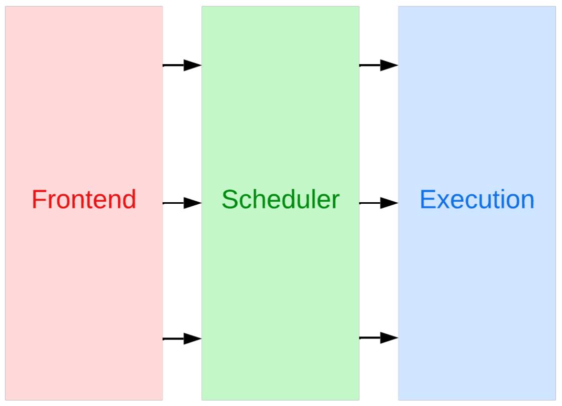

Fig. 1. Components of typical FaaS Architecture

for handling the relationship between each P2P cluster. The scheduler selects the optimal download peer while the df-daemon provides multiple command download tools that support download capabilities. The CDN is responsible for downloading tasks, caching data, and minimizing traffic. It is a root node in the P2P network. Dragonfly maintains a few nodes that coordinate every chunk of data in the cluster. Therefore, throughput is limited by performance of these few nodes. Finally, DADI[15] is another work that supports managing and distributing images to reduce the container startup time. It relies on P2P architecture to balance network traffic among participating nodes. It specifically divides images into block-based layers such that each layer corresponds to a set of file changes in the given file system. It facilitates fine-grained on demand data transfer of images, efficient online decompression, and efficient modification of large files. We believe that these P2P image distribution approaches are complementary to our work. We can use the replication information from these systems and feed it into our DRL model to optimally decide function placement.

2.2 FaaS and Function Scheduling

FaaS platforms generally take on the architecture shown in 1. The architecture includes components of an API gateway, also called a frontend, a scheduler with load balancers, and multiple execution servers. In FaaS, numerous clients are expected to make several requests to the frontend. These requests are then allocated to the servers by the scheduler, and the result will either be synchronously returned, or asynchronously saved to a database for later retrieval. Both commercial and open-source platforms generally use traditional algorithms for function scheduling. For example, a bin-packing strategy is employed by AWS Lambda, which tends to introduce performance problems [29]. On the other hand, greedy packing strategies also introduce resource contention, which introduces further performance degradation.

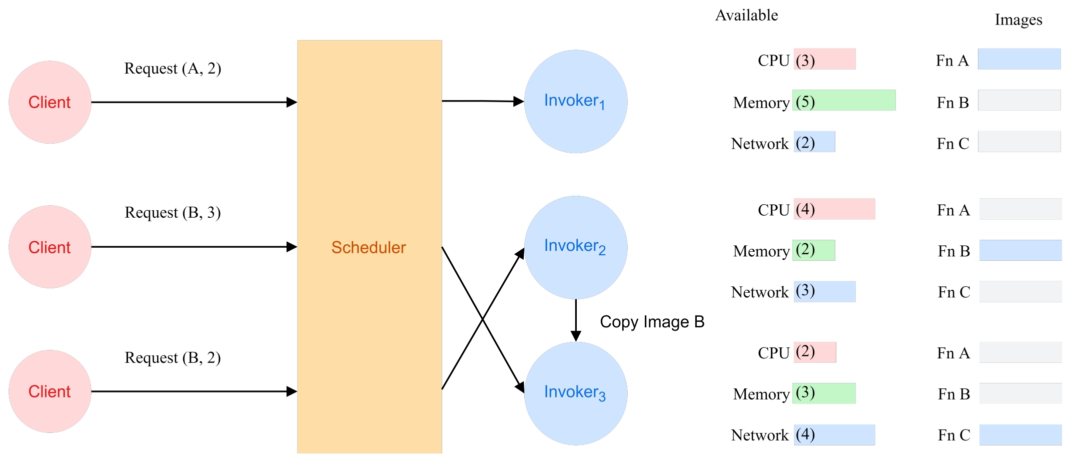

Fig. 2. An illustrative example of non-trivial optimization opportunities in a FaaS platform with a distributed image registry

Meanwhile, previous works have shown that human-designed schedulers cannot handle complex scenarios in the FaaS environment showing that there is a need for intelligent schedulers [31].

Other works have also shown the need to optimize image pulling delays as up to 60% of functions incur image pulling delays that consume at least 72% of the total function completion time (FCT) in production systems [28]. These works have suggested the use of a distributed image registry to reduce download durations. This increases the maximum throughput for pulling each image and increases availability by image chunk / layer sharding [28]. However, these distributed image registries add costs to the provider in terms of memory in order to achieve better latencies. In order to maximize the benefits of these added costs, intelligent and coordinated decisions must be made by the distributed image registry as well as the function scheduler. These intelligent decisions are elaborated on in Section 2.3.

Docker images used in production tend to be built on top of "slim" or "alpine" images, which are slimmed down images of popular runtimes such as Python, Java, NodeJS, etc. For reference, python images can be in the range of 300MB to 800MB [26, 28], meanwhile OpenWhisk's base runtime images are in the order of 600MB to 900MB.

In order to address the large image pulling delay problem, FaaSFabric proposes a DRL model that incorporates distributed registry information into scheduling decisions.

2.3 An Illustrative Example

Figure 2 shows the motivation of the project. Modern schedulers incorporate the information of available resources in each server [31]. This type of scheduler will tend to prioritize servers with available resources, but not necessarily currently hosting the correct image. Meanwhile, other classic algorithms such as bin-packing aim reduce the total number of servers and thus will tend to prioritize the same servers which already have the correct image loaded, but may not properly utilize the available resources. In Figure 2, client requests include the required image as well as the required memory units. It is shown that a balance of server resources and image availability has to be struck and adapted to by the scheduler. This can be done by incorporating more information with regards to image pulling. For example, Request(B, 3) can be allocated in either $Invoker_1$ or the $Invoker_3$, which both have adequate memory availability. If both invokers hosted the image for function B, any of the first or third invoker can be selected. However, since the image must be replicated from $Invoker_2$, the invoker with higher network capacity or lower latency to the nodes with the required image should be selected, which is $Invoker_3$. In addition, it is important that an adaptive algorithm is used to handle the variability in burstiness of workloads just as static allocation of resources is never optimal in a dynamic setting [6].

2.4 Deep Reinforcement Learning

In reinforcement learning, two main types of entities are defined: $state$ and $action$. A reinforcement learning algorithm then generates predictions of the future state $Q(state, action)$ and selects the best course of action. This is typically done using a neural network (NN) that contains weights and biases for a few layers. The algorithm then *"learns"* by adapting the weights and biases of its neurons based on the actual future state obtained in the system for the selected action. A *"deep"* reinforcement learning (DRL) algorithm simply means that the NN contains hidden layers that are capable of generating its own features based on the inputs [30].

FaaSFabric learns to schedule functions using DRL as inspired by FaaSRank [31]. Through multitudes of sequential decisions, and DRL agent learns how to achieve the most benefit from the decisions through repeated interaction with an environment and accumulating information from previous iterations. It does this by adjusting the internal weights and biases of neurons in its neural network based on the output state after performing an action.

In order to learn the impact of long-term actions such as increasing or decreasing the number of servers in the cluster, the DRL agent is also able to learn the correlation through neural layers that act as memory, such as in an LSTM neural net [24]. The DRL agent is then capable of learning the correlation of certain decisions in its action space with rewards.

Real-world applications of DRL include data processing clusters [17], network optimization [8], and mobile edge computing [16].

The benefits of DRL compared to classic optimization algorithms lie in its generalizability, better speed and accuracy due to lack of erroneous human-designed assumptions, as well as ability to incorporate a global view of the problem and historical data. While problems such as over-fitting exist in machine learning models, most well-trained models are able to outperform classic algorithms and human decisions [31] in problem sets and contexts it has never seen before.

Meanwhile, due to the fixed neural network architecture, the time complexity of calculations can be considered constant except in the case of sequential inputs, which are at most in the linear order of the number of inputs. This hints at great potential for scalability in the application of machine learning in real-time decisions. In addition, classical algorithms as well as heuristics and feature selection all suffer from erroneous human-designed assumptions that negatively affect the performance of the algorithm [31]. Through training, the DRL model automatically learns which features are more important and hold precedent over others. It is also able to incorporate a global view of the problem. For example, maximizing rewards using the context of all servers and functions at the same time, and also optimizing decisions based on arbitrarily distant decisions in the past through memory layers, as compared to having to select a fixed, error-prone window of inputs. This is important due to the well-recognized relation that a local optimizations for individual stages in a series of actions is guaranteed to under-perform compared to globally-targeted optimizations, which incorporates information from all stages in the series of actions [14]. Due to these benefits and potential for scalability as well as greater global performance, DRL is chosen as the neural network that implements the scheduler.

3 OVERVIEW

FaaSFabric is a serverless function scheduler that makes intelligent decisions using neural networks. Scheduling events on FaaS systems entails orchestrating the execution of a function across a distributed cluster. FaaSFabric uses long-running stateful controllers to collect information about the clusters state. The scheduler is activated by an event-driven function request. It uses the cluster state, function request, and image replication information as inputs and produces a scheduling action, i.e., select the server to schedule the incoming function invocation on.

3.1 Challenges

FaaSFabric aims to tackle four major problems in scheduler design:

(1) **Global View Assessment**. The assessment of server relevance with regards to a particular function request is non-trivial. Several factors including CPU, Memory, Disk I/O, and Network bandwidth availability, as well as function image availability all impact with various trade-offs as described in Section 2.3. In order to address this, FaaSFabric incorporates all of the variables and learns the optimal weighting for each variable, whereas including all of this information in classic algorithms becomes overcomplicated.

(2) **Scalability**. In order to achieve horizontal scalability, as more nodes are added into the system, a proportionally higher throughput of requests should be manageable while introducing minimal additional overhead. This is an issue that many traditional schedulers face that reinforcement learning has potential to overcome. In particular,

learning a score function enables parallel evaluation of servers, which cuts down the scheduler overhead in both an increasing and decreasing number of servers in the system. This is significantly faster in comparison to traditional machine learning models or classical algorithms that have a fixed-length output to map thousands of actions.

(3) **Elasticity**. As a burst of requests arrive, the system should minimize the impact of cold start latencies as new servers have to be provisioned to host specific function request images. In order to do so, a non-trivial number of warm servers must be kept. A static allocation of such buffer is sub-optimal due to changing request rates in a dynamic workload. While some works have attempted to incorporate the time-series nature of FaaS workloads using predictions given a fixed window, it is noted that such methods are limited by the quality of predictions as well as the arbitrarily set window length k, which can be optimal using different values for different workloads [33]. A reinforcement learning model is capable of addressing this issue by learning the optimal buffer for different granularities of function types and at different stages of a workload in the case of online learning.

(4) **Adaptive Resource Utilization**. In order to reduce costs for the service provider, which in the end also impacts the users, server resources must be utilized efficiently. Related to the elasticity challenge, the system must avoid over-provisioning resources to a non-trivial extent. Classical algorithms and other works fail to address the utilization of these servers in conjunction to minimizing the average function completion time of bursty workloads.

3.2 Objectives

FaaSFabric optimizes two major components in a FaaS context, minimizing the *Average Function Completion Time* (Average FCT) and maximizing *Resource Utilization* (RU).

Average Function Completion Time is defined as the summation of all function completion times divided by the total number of invocations N as shown in equation (1). Function completion time for request i is then described in equation (2), where it is the sum of the following delays: authentication, scheduling, queue, image pulling, and execution. A typical strategy to achieve a minimum for average FCT, is to keep servers warm and minimize the image pulling overhead.

$$\overline{C} = \frac{\sum_{i=1}^{N} C(i)}{N} \tag{1}$$

where:

N is the total number of function invocations

$C(i)$ is the completion time of request i

$$C(i) = A(i) + S(i) + Q(i) + P(i) + E(i) \tag{2}$$

where:

$A(i)$ is the authentication delay of request i

$S(i)$ is the scheduling delay of request i

$Q(i)$ is the queuing delay of request i

$P(i)$ is the image pulling delay of request i

$E(i)$ is the execution time of request i

Average Resource Utilization Efficiency is defined in equation (3) as the sum of CPU, Memory, Disk I/O, and Network I/O resources efficiencies weighted by lambda variables λ_c, λ_m, λ_d, and λ_n respectively for time slices k from 1 to K, all divided by K. Resource utilization efficiency for any resource r at time slice k, $E_r(k)$ is defined in equation (4) as the sum of usage $u_r(j,k)$ for every server j from 1 to M_k, the total number of servers at time k divided by the amount of provisioned resources $a_r(j,k)$ for every server j from 1 to M_k. In order to maximize resource utilization efficiency rates, functions of different types which utilize CPU, Memory, Disk I/O, and Network differently should be co-located in a server depending on the amount of each resource provisioned to that server and idle servers should be switched off. These lambdas can be reasonably set as the ratio of costs for each resource type. For example, lambda disk than memory since disk resources are often cheaper than memory.

$$\overline{E} = \frac{1}{K} \sum_{k=1}^{K} \left(\lambda_c \cdot E_c(k) + \lambda_m \cdot E_m(k) + \lambda_d \cdot E_d(k) + \lambda_n \cdot E_n(k)\right) \tag{3}$$

where:

K is the total number of time slices

$E_c(k)$ is the CPU resource utilization efficiency at time slice k

$E_m(k)$ is the memory resource utilization efficiency at time slice k

$E_d(k)$ is the disk I/O resource utilization efficiency at time slice k

$E_n(k)$ is the network I/O resource utilization efficiency at time slice k

$$E_r(k) = \frac{\sum_{j=1}^{M_k} u_r(j,k)}{\sum_{j=1}^{M_k} a_r(j,k)} \tag{4}$$

where:

M_k is the total number of servers at time slice k

$u_r\,(j,k)$ is the total usage of r of server j at time slice k

$a_r\,(j,k)$ is the total provisioned resource r of server j at time slice k

$r \in \{c := CPU, m := Memory, d := Disk, n := Network\}$

3.3 Hypothesis

The hypothesis of the project is that if distributed image registry information is incorporated into scheduling decisions, then image pulling delays can be reduced, reducing average function completion time across all function requests while maintaining high resource utilization by not simply replicating images in all nodes. An implementation using MILP can show that the image pulling delay is optimized when distributed image registry information is considered. However, MILP algorithms are notoriously slow in actual calculations, especially in the context of short-lived functions in FaaS, and are thus not practical in production environments.

4 DESIGN

4.1 Distributed Image Registry

In FaaSFabric, image registry nodes reside in the same machines that host invoker nodes, but are virtually separate to enforce performance isolation. The image replication information is in the format of a graph with nodes as the servers, node information as a one-hot encoding of base and function image replication as well as their warmth statuses, edges as virtual network connectivity and edge information as a running average of latency between nodes. By having the image replicated across multiple nodes, a new server can be instantiated faster by downloading different chunks of the required image from different servers in a peer-to-peer fashion than downloading the whole image from a centralized image registry. This is critical to minimize the duration cold-start latencies.

4.2 Server Scheduling Assessment

FaaSFabric assesses the relevance of a server to a function call based on three types of information: server, function, and distributed image placement. Beyond the image placement information, the data collected from servers are inspired by FaaSRank [31]. FaaSFabric gathers resource consumption information from each server, as shown in Table 1. In the table, Δ represents the change in resource utilization given the sampling interval. Within the server, the different metrics are categorized as: CPU, memory, disk I/O, network I/O, and the set of images stored in the server. When a function is received the requested function image, requested memory, and average function initialization time is collected. In order to consolidate the image dependency for the function, the set of images is converted into a boolean that indicates the

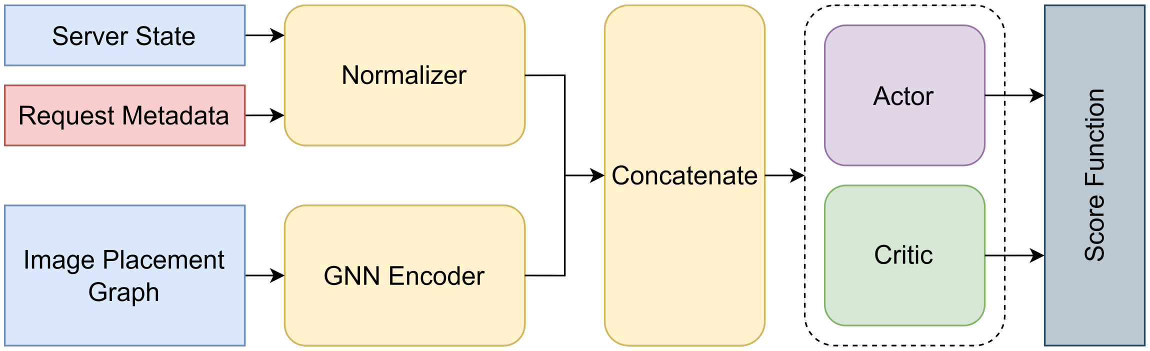

Fig. 3. Components of the deep reinforcement learning neural network.

presence of the required image. Finally, the information from the distributed image registry is integrated by injecting the nodes and edges state into the model via a Graph Neural Network (GNN) encoder [34]. A flat feature vector is then created from the state information as input to the DRL model. An Actor-Critic model is used to generate the score function due to its effectiveness in reducing training variance and delivering faster convergence[13]. Figure 3 shows the design of the DRL neural network.

4.3 Score Function

A score function is learned by the DRL agent in order to overcome the fixed-sized output limitations of typical machine learning models. In the case of a score function, each server can be scored in parallel and thus limits the action space of decisions by the DRL agent to allocations of individual functions to any server, rather than deciding the allocations of all functions to all servers at the same time. The use of a score function is inspired by [18]

At any time t, FaaSFabric will receive the neural-network-encoded state information as described in Section 4.2. The final concatenated vector is then used as input to the Actor and Critic networks to learn the appropriate score function. Through the composition described in Section 4.2, FaaSFabric's overall network is end-to-end differentiable. FaaSFabric's scoring function does not need manual feature engineering; that is, nothing is hard-coded in the score function. FaaSRank learns what is important for computing a priority score given a state vector through training. FaaSFabric's architecture is also lightweight because it employs the same scoring mechanism for all servers and function invocations.

Training. The training of the DRL agent would be done in epochs. Using workload generators from ServerlessBench and traces from the Microsoft Azure dataset [23], the dataset would be divided into 1000 episodes. Of these episodes, 20% would be randomly selected as the test set, 60% as the training set, and 20% as the validation set. The datasets would be analyzed to ensure a fairly even distribution of burst scenarios in each dataset. In order to perform the training, the

Metric	Description
1. Server Information	
Δcpu_user	CPU time in user mode
Δcpu_nice	CPU time executing prioritized processes
Δcpu_kernel	CPU time in kernel mode
Δcpu_idle	CPU idle time
Δcpu_iowait	CPU time waiting for I/O to complete
Δcpu_irq	CPU time handling HW interrupts
Δcpu_softirq	CPU time handling software interrupts
Δcpu_steal	CPU time spent by other operating systems
Δcpu_ctx_switches	Number of context switches
cpu_freq	Frequency of the CPU
cpu_load_avg	Average system load over the last minute
memory_free	Physical RAM left unused by the system
memory_buffers	Temporary storage for raw disk blocks
memory_cached	Physical RAM used as cache memory
memory_percent	Percent of memory used
Δdisk_read	Number of disk reads completed
Δdisk_read_merged	Number of disk reads merged together
Δdisk_read_time	Time spent reading from the disk
Δdisk_write	Number of disk reads completed
Δdisk_write_merged	Number of disk writes merged together
Δdisk_write_time	Time spent writing to the disk
Δnet_byte_recv	Network Bytes received
Δnet_byte_sent	Network Bytes sent
inflight_invocations	Number of inflight requests in the server
2. Image Placement Information	
base_images	Indices of base images in the server
function_images	Indices of function images in the server
warm_images	Indices of warm function images in the server
3. Function Information	
requested_image	Image index requested by the function
requested_memory	Memory requested by the function
init_time	Measured function cold initialization time

Table 1. Resource utilization metrics incorporated into the deep reinforcement learning model

OpenWhisk environment and DRL agent are cleanly set up before each episode and trained online for the training set. The online training indicates that the DRL agent makes decisions only based on the state information available at the time and does not know about future states. A single DRL agent is used for the training. For optimization, an AdamW optimizer could be used following [31].

Reward Function. The training of the DRL agent requires a reward function that incorporates both objectives described in Section 3.2. The goal of the DRL agent is to maximize the expected reward in equation 5. The expected reward $R(t)$ consists of the resource utilization efficiency $\bar{E_t}$ for efficiency of requests up to time t while penalizing the average function completion time $\bar{C_t}$ for average FCT of requests up to time t. The weight of the two objectives are balanced by the hyperparameter λ_f as the weight of function completion time. By increasing the hyperparameter λ_f, a higher priority is given to reducing the average function completion time at the potential cost of having more idle resources.

$$R_t = \mathbb{E}\left[\left(1 - \lambda_f\right) \cdot \bar{E_t} - \left(\lambda_f\right) \cdot \bar{C_t}\right] \tag{5}$$

4.4 Bin-Packing

The bin-packing model described as a MILP formulation in this section incorporates predicted image pulling delays based on network connection metrics and image replication information.

$$minimize\, K = c_1 \cdot \sum_{j=1}^{M} y_i + c_2 \cdot \sum_{i=1}^{N} \sum_{j=1}^{M} p(i,j) x_{ij} \tag{6}$$

Where:

$p(i,j)$:= the predicted image pulling latency of allocating request i to server j. The details of this is described in 5.4

Subject to constraints:

$K >= 1$ (at least one server is provisioned)

$\sum_{i=1}^{N} s(i) x_{ij} \le B y_j,\ \forall j \in \{1, ..., M\}$ (server memory capacity constraint)

$\sum_{j=1}^{M} x_{ij} = 1,\ \forall i \in \{1, ..., N\}$ (each function is allocated exactly once)

$y_j \in \{0, 1\}, \forall j \in \{1, ..., M\}$ (boolean constraint)

$x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., N\}, \forall j \in \{1, ..., M\}$ (boolean constraint)

5 IMPLEMENTATION

5.1 OpenWhisk Architecture with Distributed Image Registry

Figure 4 visualizes the architecture of FaaSFabric integrated with OpenWhisk. The path of a request is described in more details as follows. The request is first received by the NGINX component, which is a reverse proxy and load balancer responsible for SSL termination [25]. Then, the controller receives the request and performs authentication by contacting the system's database (CouchDB) instance and validating that the requested function exists in CouchDB. Once authenticated, the controller designates an invoker to be the executor of the request. It does so by sending the request message to the system's Kafka message queue with the topic as the unique invoker name. The message queue present in the system to persist messages and avoid failures from invoker crashes. This also limits the number of

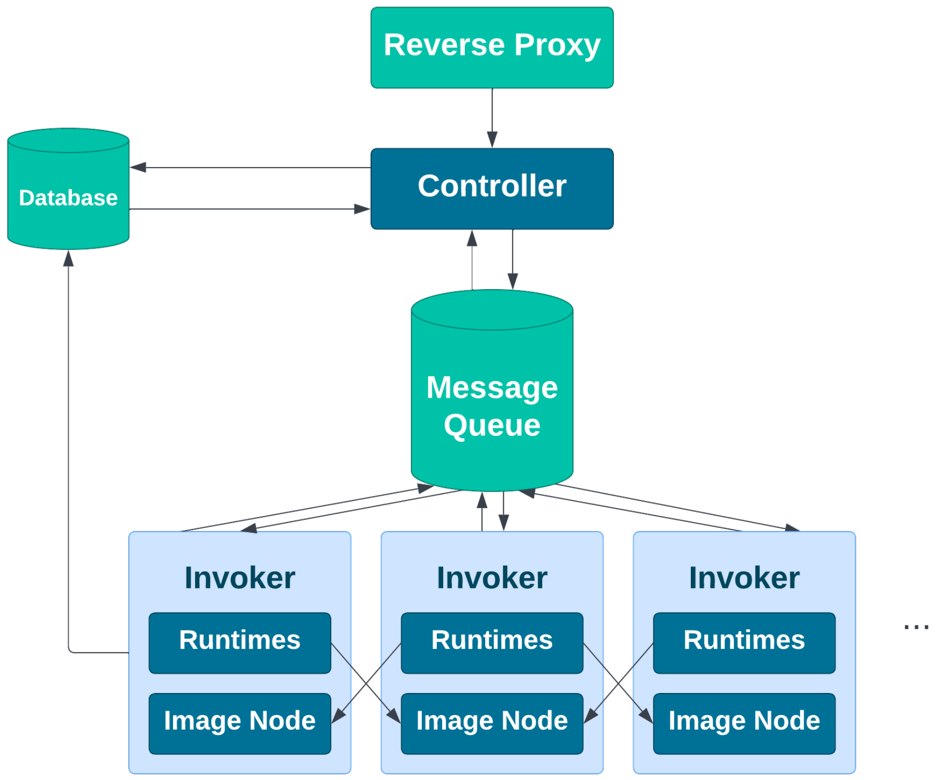

Fig. 4. Components of the OpenWhisk architecture.

connections that controllers and invokers have to maintain. If the invoker does not have the required image, it is instructed by the controller to download it from the distributed image registry. The invoker then pulls the message from the queue and executes the request. The components of the OpenWhisk architecture (NGINX, controller, CouchDB, Kafka, Invokers), are all containerized and deployed as Docker images in dedicated physical or virtual machines using Kubernetes. The state of the image can be divided into three stages: cold, pre-warmed, and warm. In the cold stage, the invoker container has not downloaded the base environment image for the function. This base image is a programming language environment such as Python, Java, NodeJS, etc. In the pre-warmed state, the invoker has downloaded the base image but not the function source code, which can be Docker images or executables. In the warm state, the function source code has been downloaded, initialized, and is running in the invoker, ready to receive requests via the handler function required in the OpenWhisk schema. In the default OpenWhisk deployment, base images and function source code are stored in CouchDB, while in FaaSFabric, these are stored in the distributed image registry.

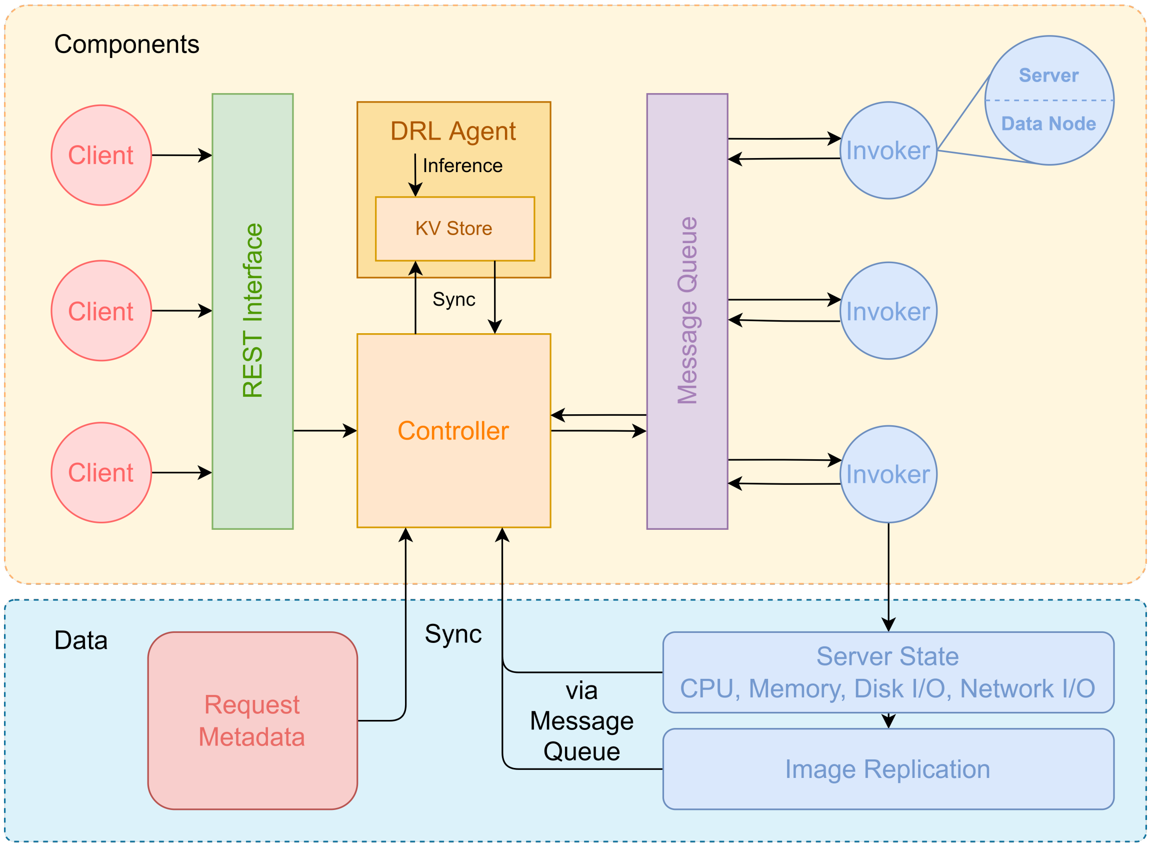

Fig. 5. Components of the FaaSFabric architecture.

5.2 FaaSFabric

The design of FaaSFabric is described in Figure 5. A REST interface is provided as the entrypoint for requests. This is directly implemented by the NGINX layer in OpenWhisk. After the REST interface, the request arrives at the controller. Within this controller a key-value (KV store) is used following inspiration from FaaSRank [31]. This KV store is synchronized with a DRL agent that can be hosted on a separate machine. This DRL agent receives the request and state information from the KV store and performs the inference using information from all the servers and current requests and scores each server. Upon synchronization in the controller, the controller then selects the server with the highest score for the request and forwards the request to the corresponding topic in the message queue, which is Kafka in the case of OpenWhisk. From here, the standard operation of OpenWhisk continues where the invokers pull messages from the message queue, execute the request, then store results in the database. In order to collect state

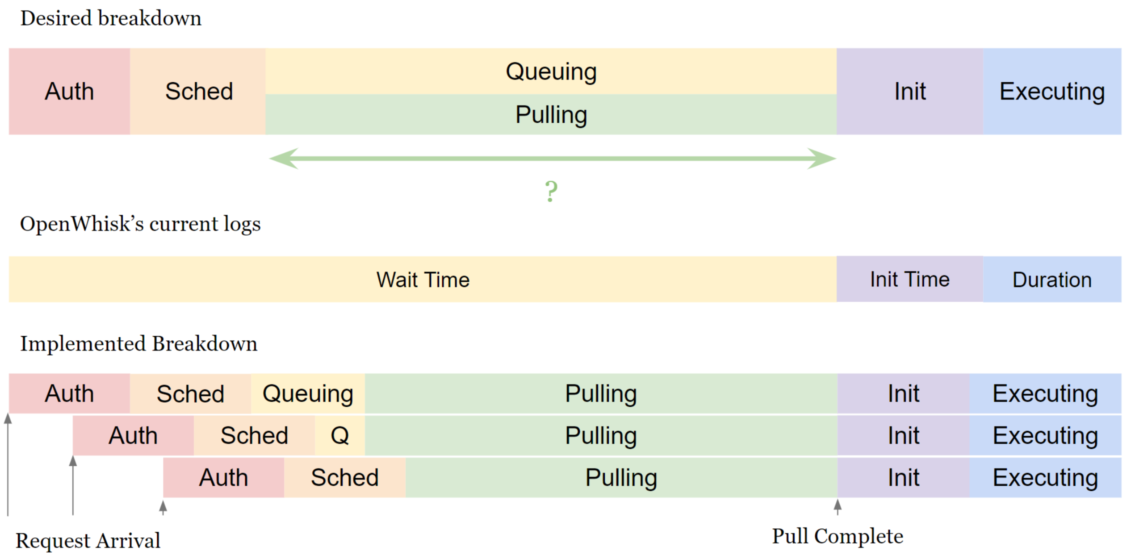

Fig. 6. Desired breakdown, OpenWhisk breakdown, and implemented breakdown of function request stage durations. The relative scale of different stages is only for reference and are not precise.

information in the controller, invokers periodically send their state information including image replication information to the message queue with a topic destined for the controller.

5.3 Fine-grained measurements

The default OpenWhisk deployment does not collect fine-grained measurements needed to inform meaningful improvements to the system. As shown in Figure 6, the target breakdown of the duration of stages in a function invocation includes authentication, scheduling, queue, image pulling, initialization, and execution delays. Meanwhile, OpenWhisk only provides "waitTime", "initTime", and "duration" which coarsely covers multiple stages. "waitTime" includes authentication, scheduling, queue, and image pulling delays, meanwhile, initTime includes initialization, and duration corresponds to execution delay.

To achieve fine-grained measurements that are important to inform design decisions, the logs of controllers and invokers are used. The ideal method to measure and report these delays is to record the durations in controllers and invokers, and include them as annotations in the final request message. However, due to time and resource limitations in the project, the logs are gathered after the workload has completed. To calculate the delays for each request from the different sources, the transaction id (tid) recorded in the logs is used to group logs for each request. From there, the transition points of each stage are marked and the delays are calculated. Then the average for each stage across all requests is calculated. It is noted that there can be gaps in clock synchronization across nodes, however, from preliminary results, such gaps have been shown to be negligible.

One non-trivial measurement is the image pulling delay because while an invoker fetches an image, the request sits in the queue. This waiting time can be added to the queuing delay or the pull delay, and is shown in the question mark in the "Desired breakdown" in Figure 6. In order to meaningfully calculate the image pull delay, which is typically shared by multiple functions jointly allocated to a new server, the starting marker of the pull delay is either the start of the invoker pulling the image, or the end of the scheduling decision, which ever occurs later. Both cases immediately mark the end of the queuing delay as shown in6.

5.4 Bin-Packing Mixed-Integer Linear Programming Model

The Bin-Packing algorithm with consideration of image pulling latency is implemented by caching the calculations for $p(i,j)$ in the system based changes in image placements and network latencies. In this manner, the results of $p(i,j)$ can be used frequently in the LP optimization. This caching requires regularly updating three pieces of information: average network throughput between invoker nodes, and image size, and presence of image in nodes. The average network throughput between nodes can be calculated using sample downloads regularly performed across the system similar to the method in speedtest.net [22].

6 EVALUATION

6.1 Baseline Schedulers

The baseline schedulers used in our evaluation are as follows:

(1) **OpenWhisk default Hashing scheduler.** The Open-Whisk default scheduler uses a hashing algorithm to schedule function invocations. It calculates a hash value for each function, and always schedules invocations of the same function to the same invoker with the aim of maximizing warm starts.
(2) **Bin-packing scheduler.** A scheduler that calculates the fewest identical servers needed to host all of the heterogeneous requests.
(3) **FaaSRank scheduler.** A DRL score-function-based scheduler that is shown to outperform a round-robin, least-connections, greedy, and static-rank schedulers [31].

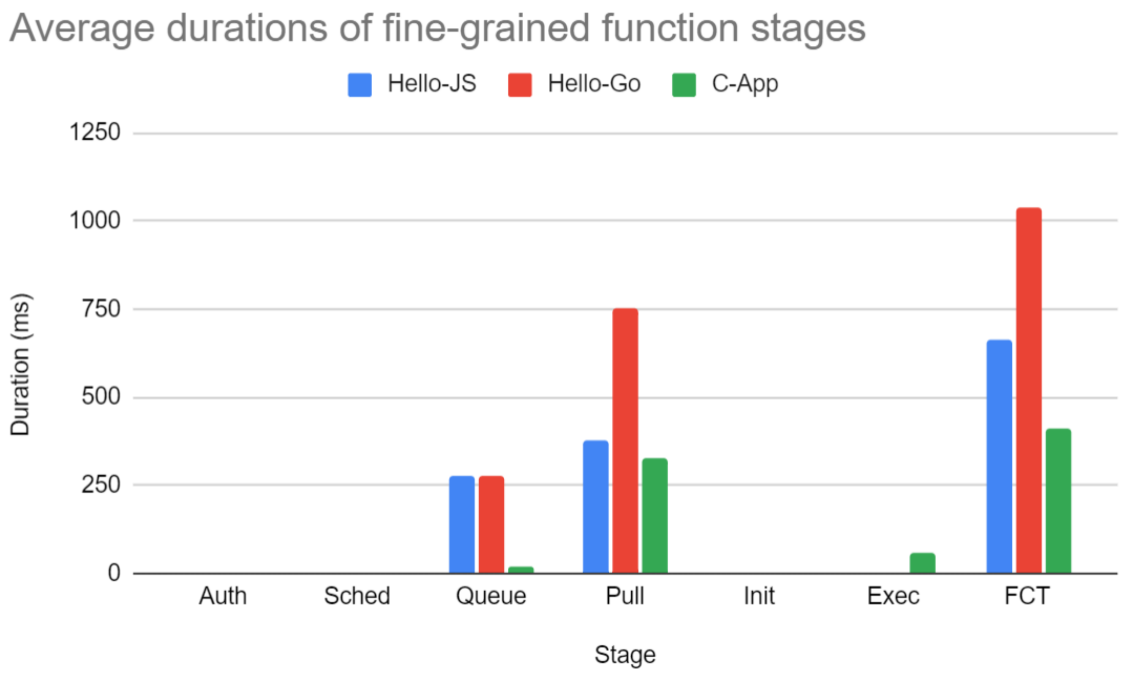

Fig. 7. Graphical results for average durations of function stages in different workloads and schedulers

6.2 Experimental Setup

This section describes the infrastructure and workload for the experiments. The OpenWhisk cluster contains 1 Controller and 5 Invoker machines. Each machine, including the controller has 12 physical CPU cores, 64GB of RAM, and 1Gb/s links to other machines. A separate machine with the same specifications hosts a Kubernetes master that monitors and deploys the core system images to the machines.

The workloads tested on the system are the hello-world and simple applications from the startup breakdown workloads in ServerlessBench [32].

6.3 Workload Evaluation

Figure 7 and Table 2 show the results of the experiments with header columns of the average authentication, scheduling, queuing, image pulling, initialization, and execution delays, as well as the average function completion time and average resource utilization efficiency for all workloads.

Each workload is run with 15 seconds warm up, 60 seconds testing time, and 15 seconds cooldown. The workload is run using 3 separate client machines with 5 threads each. The C-App workload is an RSA key pair generator. Each lambda is set to 0.25 for the calculation of RUE.

Workload	Scheduler	Avg. Auth Delay (ms)	Avg. Sched Delay (ms)	Avg. Queue Delay (ms)	Avg. Pull Delay (ms)	Avg. Init Delay (ms)	Avg. Exec Delay (ms)	# Reqs	Avg. FCT (ms)	Avg. RUE (%)
Hello-JS	OpenWhisk default	0.91	1.10	277.44	377.29	0.88	4.34	903	**661.96**	**13.31**
Hello-Go	OpenWhisk default	0.96	1.39	276.65	752.22	3.05	1.98	594	**1036.25**	**14.49**
C-App	OpenWhisk default	0.82	1.12	20.26	326.45	0.48	60.94	723	**410.06**	**16.49**

Table 2. Average delay for each stage and average FCT for each workload

The cold starts during the measurement phase is observed to be about 1 to 4% of invocations, with around 9 to 26 seconds in image pull time. The percentage of cold starts would be higher in the warm up phase. The average overhead of the system relative to the execution time ranges from 85% to 99% and has significant room for improvement. It is shown that the image pulling delay comprises of 80% of the total function completion time on average for the C-App workload.

Due to time and resource limitations, the bin-packing and DRL implementation of the scheduler is not implemented and is left as future work.

6.4 Justification of Deep Reinforcement Learning Approach

As observed in Section 6.3, the image pulling delays are large relative to the execution time of the functions. In order to improve this delay while limiting the increase in scheduling delay, a deep reinforcement learning model can be used to perform a fast inference in the same order of magnitude as the execution time while utilizing greater breadth and depth of information. This indicates potential for a deep reinforcement learning model to outperform all baseline schedulers based on principles.

7 RELATED WORK

7.1 Deep Reinforcement Learning in Placement Decisions

A few works have used deep reinforcement learning for generating placement decisions. Two notable works are FaaSRank in the context of FaaS [31], and DSP-DRL in the context of mobile edge computing [16]. FaaSRank uses a DRL agent that utilizes server resource utilization information as well as function request metadata to perform intelligent allocation decisions. Meanwhile, DSP-DRL uses a distributed DRL model that makes service placement decisions with considerations of demand fluctuations and limited storage in mobile edge computing. Neither works have considered the context of a distributed image registry and optimized decisions using relevant information.

7.2 Distributed Image Registries

FaasNet addresses the challenge of rapid container provisioning in server-less under dynamic workloads. It proposes a scalable middle-ware system for accelerating container provisioning. It creates a function tree structure and uses on demand fetching mechanism to reduce function provisioning cost. Function tree is a logical network overlay that consists of multiple host VMs. Our work differs with FaasNet in a number of ways. First, we focus on function placement and use machine learning to make an optimal placement decision. On the other hand, FaasNet uses a bin-packing heuristic for function placement and creates a tree like overlay network to provision data in a distributed fashion. Second, our model take as input an arbitrary data distribution policy and will make a best possible decision while FaasNet only focuses on a tree like network for storing and provisioning data in a distributed fashion. Third, although we focused more on docker images, our approach is more generalized and can work with any kind of data e.g ML training data, video data.

7.3 Other Distributed Image Registry Techniques

Kraken[27] and DADI [28] are some other P2P approaches to accelerate container provisioning at scale. However, they require powerful server for data seeding and management. We ,on the other hand, don't require any specific server requirements. Moreover, these systems generally assume static P2P topology which is generally not true in server-less environment. Besides this, some work has been done on optimizing storage and retrieval of container images. Slacker[11] is one such example. It reduces container startup time by utilizing lazy cloning and lazy propagation. We note that such techniques are specific to docker container and can be used in complement to our work.

8 CONCLUSION

In this paper, we present FaaSFabric, a deep reinforcement learning (DRL) scheduler that optimizes function completion time and resource utilization efficiency by minimizing image pull delays using distributed image registry-aware function placements decisions. We have developed code to measure fine-grained function stage duration and resource utilization efficiency and demonstrated that image pulling delays are a significant factor to improve function completion times. We have also formulated a MILP design to incorporate image replication information into decisions, justified the potential for DRL to optimize scheduling decisions, and designed a formulation using DRL.

9 FUTURE WORK

Due to time and resource limitations, the authors leave the following as future work:

The distributed image registry implementation has to be incorporated into the system to optimize image pull delays. In conjunction, more realistic workloads with a larger variety of function images and larger burst peaks should be tested. In particular, testing the model on other data intensive workloads like video processing, ML training etc. will likely emphasize the benefit of optimized scheduling decisions. The MILP and DRL formulation should be implemented and refined based in findings from the implementation. Finally, regarding the challenge of scalability, a horizontal scaling assessment should be performed wherein the effect of adding invoker nodes to the system should result in increased throughput capacity.

10 ACKNOWLEDGEMENTS

We would like to acknowledge support for this project from the Systems and Networking (SyN) Research Group from the University of Waterloo for the access to computing resources for model training and evaluation.

REFERENCES

[1] [n. d.]. Dragonfly. ([n. d.]). https://github.com/dragonflyoss/Dragonfly2

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Alibaba Cloud. n.d.. Function Compute. (n.d.). https://www.alibabacloud.com/product/function-compute

[4] Amazon Web Services. 2014. Netflix & AWS Lambda Case Study. (2014). https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/

[5] Amazon Web Services. n.d.. AWS Lambda. (n.d.). https://aws.amazon.com/lambda/

[6] Michael Armbrust, Armando Fox, Griffith Rean, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2009. Above the Clouds: A Berkeley View of Cloud Computing. (01 2009).

[7] Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. 2015. Bin Packing with Linear Usage Costs. (2015). arXiv:1509.06712 http://arxiv.org/abs/1509.06712

[8] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 191–205. https://doi.org/10.1145/3230543.3230551

[9] Maxime Cohen, Philipp Keller, and Vahab Mirrokni. 2016. Overcommitment in Cloud Services - Bin Packing with Chance Constraints. *SSRN Electronic Journal* (01 2016). https://doi.org/10.2139/ssrn.2822188

[10] Google. n.d.. Cloud Functions. (n.d.). https://cloud.google.com/functions

[11] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 181–195.

[12] IBM. n.d.. IBM Cloud Functions. (n.d.). https://www.ibm.com/cloud/functions

[13] Vijay Konda and John Tsitsiklis. 1999. Actor-Critic Algorithms. In *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller (Eds.), Vol. 12. MIT Press. https://proceedings.neurips.cc/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[15] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. {DADI}:{Block-Level} Image Service for Agile and Elastic Application Deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 727–740.

[16] Shuaibing Lu, Jie Wu, Jiamei Shi, Pengfan Lu, Juan Fang, and Haiming Liu. 2022. A Dynamic Service Placement Based on Deep Reinforcement Learning in Mobile Edge Computing. *Network* 2, 1 (2022), 106–122. https://doi.org/10.3390/network2010008

[17] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[18] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[19] Microsoft. n.d.. Azure Functions. (n.d.). https://azure.microsoft.com/en-us/services/functions/#features

[20] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing (WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3366623.3368133

[21] Hai Duc Nguyen, Chaojie Zhang, Zhujun Xiao, and Andrew A. Chien. 2019. Real-Time Serverless: Enabling Application Performance Guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing (WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3366623.3368133

[22] Ookla. 2014. How does the test itself work? How is the result calculated? (2014). https://sandboxsupport.speedtest.net/hc/en-us/articles/202972350-How-does-the-test-itself-work-How-is-the-result-calculated-

[23] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[24] Ralf C. Staudemeyer and Eric Rothstein Morris. 2019. Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks. (2019). https://doi.org/10.48550/ARXIV.1909.09586

[25] Markus Thömmes. 2016. Uncovering the magic: How serverless platforms really work! *Apache OpenWhisk* (08 2016).

[26] Itamar Turner-Trauring. 2020. Using Alpine can make Python Docker builds 50× slower. (2020). https://pythonspeed.com/articles/alpine-docker-python/

[27] Uber. 2018. Kraken. (2018). https://github.com/uber/kraken

[28] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[29] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[30] Terri Williams. 2020. Reinforcement Learning Vs. Deep Reinforcement Learning: What's the Difference? (10 2020). https://www.techopedia.com/reinforcement-learning-vs-deep-reinforcement-learning-whats-the-difference/2/34039

[31] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Washington, DC, USA, September 27 - Oct. 1, 2021*,

Esam El-Araby, Vana Kalogeraki, Danilo Pianini, Frédéric Lassabe, Barry Porter, Sona Ghahremani, Ingrid Nunes, Mohamed Bakhouya, and Sven Tomforde (Eds.). IEEE, 31–40. https://doi.org/10.1109/ACSOS52086.2021.00023

[32] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

[33] Qi Zhang, Quanyan Zhu, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L. Hellerstein. 2013. Dynamic Service Placement in Geographically Distributed Clouds. *IEEE Journal on Selected Areas in Communications (JSAC)* 31, 12 (Dec. 2013), 762–772.

[34] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. (2018). https://doi.org/10.48550/ARXIV.1812.08434