FaaSRank: Learning to Schedule Functions in Serverless Platforms

Hanfei Yu
University of Washington
hanfeiyu@uw.edu

Athirai A. Irissappane
University of Washington
athirai@uw.edu

Hao Wang
Louisiana State University
haowang@lsu.edu

Wes J. Lloyd
University of Washington
wlloyd@uw.edu

***Abstract*—Current serverless Function-as-a-Service (FaaS) platforms generally use simple, classic scheduling algorithms for distributing function invocations while ignoring FaaS characteristics such as rapid changes in resource utilization and the freeze-thaw life cycle. In this paper, we present FaaSRank, a function scheduler for serverless FaaS platforms based on information monitored from servers and functions. FaaSRank automatically learns scheduling policies through experience using reinforcement learning (RL) and neural networks supported by our novel Score-Rank-Select architecture. We implemented FaaSRank in Apache OpenWhisk, an open source FaaS platform, and evaluated performance against other baseline schedulers including OpenWhisk's default scheduler on two 13-node OpenWhisk clusters. For training and evaluation, we adapted real-world serverless workload traces provided by Microsoft Azure. For the duration of test workloads, FaaSRank sustained on average a lower number of inflight invocations 59.62% and 70.43% as measured on two clusters respectively. We also demonstrate the generalizability of FaaSRank for any workload. When trained using a composite of 50 episodes each for 10 distinct random workloads, FaaSRank reduced average function completion time by 23.05% compared to OpenWhisk's default scheduler.**

I. Introduction

Serverless computing is a new cloud computing paradigm that has growing prosperously in recent years. Function-as-a-Service (FaaS), the most commonly used service delivery model of serverless computing, has become increasingly popular [1]. Serverless FaaS platforms free users from low-level tasks while automating resource provisioning, scaling, and isolation. Users are only responsible for deploying source code and configuring memory limits for applications. FaaS platforms feature pay-as-you-go pricing models while enabling simplified deployment of applications. Consequently, FaaS platforms provide an enticing option for developers to consider for hosting computational workloads that simplify management with potential to reduce costs incurred from renting idle servers. Major cloud providers offer FaaS platforms such as AWS Lambda [2], Microsoft Azure Functions [3], Google Cloud Function [4], and IBM Cloud Functions [5].

Similar to traditional web service load balancing, FaaS platforms schedule function invocations by distributing requests across available servers known as workers for execution. Though sharing some characteristics with traditional web service scheduling, FaaS scheduling introduces new challenges. First, FaaS exposes unique characteristics that traditional scheduling strategies don't consider. On a traditional web service cluster, service deployments are typically fixed and do not change. On a FaaS cluster, function invocations are stateless and much more bursty than traditional web requests while also experiencing the infrastructure freeze-thaw life cycle [6]. Several minutes after finishing execution, the temporary infrastructure used to host FaaS functions known as function instances [7] are removed from a server. When function instances are recreated, they may not be provisioned on the same server. This inconvenience can result in additional initialization overhead as necessary source code and libraries may not be cached on the new host, a phenomenon contributing to cold start latency [8]. FaaS function deployments constantly change the location of free capacity across the cluster, whereas traditional web service hosting features largely static deployment and resource management. Second, FaaS clusters enable all resources for unused functions to be reclaimed and reprovisioned: *e.g.*, processes, memory, disk space, and CPU capacity. It's non-trivial to provide an intelligent scheduling approach to simultaneously address new challenges unique to FaaS scheduling. New scheduling approaches are needed to address these challenges that expand upon traditional load balancing algorithms for web services.

However, existing FaaS platforms have generally adopted simple, classic scheduling algorithms while ignoring the unique challenges associated with FaaS characteristics (*e.g.*, AWS Lambda [2] and Apache OpenWhisk [9]). In this paper, we present FaaSRank, a function scheduler designed for serverless FaaS platforms. FaaSRank learns dynamic scheduling policies through experience using reinforcement learning (RL) and neural networks. Given a high-level goal (*e.g.*, minimize average function completion time in our context), FaaSRank leverages function and worker metrics to automatically make decisions on scheduling function invocations. FaaSRank optimizes the average function completion time (FCT) of FaaS workloads while maintaining scheduling fairness for scheduling individual functions from different clients. FaaSRank strives to improve upon the use of traditional neural networks for training smart schedulers for scalable clusters, by developing a novel Score-Rank-Select architecture based on reinforcement learning.

Here are the main contributions of the paper:

- We describe the design of FaaSRank, a general function scheduler for serverless platforms.

- We implement a prototype of FaaSRank and four other baseline schedulers in the Apache OpenWhisk [9].
- We evaluate FaaSRank using realistic serverless functions collected from [10] [11] and [12].
- We adapt real-world serverless traces from Microsoft Azure Functions [13] to conduct extensive evaluations of FaaSRank performance.

II. Background and Motivation

In this section, we introduce general FaaS platforms and existing function scheduling strategies. We use an example evaluated on a realistic FaaS platform to motivate the need for an intelligent scheduler. We also briefly illustrate how to learn scheduling policies using reinforcement learning.

A. *FaaS and Function Scheduling*

FaaS platforms generally consist of a frontend, a controller, and multiple servers hosting backend services. The frontend receives function invocations and forwards them to a scheduler. The scheduler maintains a list of available servers for function execution. Once informed of an incoming invocation, the scheduler selects a server on which to execute the function.

Function scheduling algorithms vary for existing FaaS platforms. Open-source FaaS platforms generally use classic algorithms for function scheduling. For example, Apache OpenWhisk [9] adopts a hashing method to schedule functions within a distributed cluster. Details regarding scheduling algorithms implemented by commercial FaaS platforms are not available publicly, nevertheless researchers reveal some facts through reverse engineering. [7] and [14] identify that AWS Lambda [2] greedily packs containers running function invocations on Virtual Machines (VMs) to improve resource utilization.

Existing approaches for function scheduling, *e.g.*, classic algorithms or the packing strategy employed by AWS Lambda, tend to introduce performance problems. For classic algorithms, in addition to the cold start delay when launching a new function instance with its dependencies [8], performance can suffer further from resource contention if function instances are co-located on busy servers with too many other functions. Round-robin and least-connections algorithms are unaware of function dependencies, which can result in more cold starts. Greedy packing strategies also introduce resource contention, and can produce performance degradation [7]. Given available server metrics, it's non-trivial to select a suitable server to schedule a function invocation. Section II-B illustrates that human-designed strategies are conservative and can hardly handle complicated FaaS workloads.

B. *An Illustrating Example*

To motivate the need of an intelligent scheduler for FaaS platforms, we conduct an evaluation of function scheduling on Apache OpenWhisk [9], an open-source FaaS platform used to implement IBM Cloud Functions [5], to demonstrate performance of adopting different schedulers. We deployed an OpenWhisk cluster with 10 servers for executing functions.

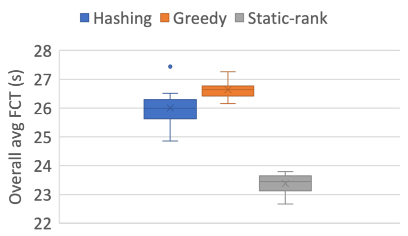


Fig. 1: Average Function Completion Time (FCT) of three schedulers

Each server had 8 vCPU cores and 16 GBs of memory, with 2 GBs available for function runtimes. The details of our OpenWhisk configuration are described in Section VI-B.

Our motivating experiment examines three schedulers:

- **Hashing scheduler.** The default scheduler employed by OpenWhisk. The hashing scheduler calculates a hash value for each function, and always schedules the same function invocation to the same server with the aim of minimizing cold starts.
- **Greedy scheduler.** We use a simple greedy algorithm to mimic the scheduling strategy adopted by AWS Lambda, which always schedules functions to the same server until the server no longer has available resources. The greedy scheduler maximizes the resource utilization of servers in-use while minimizing the total number of servers to shrink the platform's footprint.
- **Static-rank scheduler.** A fine-tuned heuristic scheduler we developed that calculates an overall score for each server using a fixed fitness function. The overall score is calculated as

$$\begin{aligned} score &= 2 \times cpu + 1.5 \times memory + disk \\ &+ network + load\_avg + infra \\ &+ 3 \times avail\_mem\_slots, \end{aligned} \tag{1}$$

  where 2, 1.5 and 3 are weights assigned to CPU, memory, and available slots based on server free memory.

We evaluated these schedulers using a testing workload adapted from real-world serverless function traces provided by Microsoft Azure [13] that describes function invocations over a 60-second period. Further details of our experimental workload traces are described in Section VI-B.

Figure 1 reports the average function completion time (FCT) for our three schedulers for 10 repetitions of the workload. The Static-rank scheduler reduced the average FCT by 10.10% and 14.88% respectively, compared to OpenWhisk's default Hashing scheduler and the Greedy scheduler. The Static-rank scheduler considers FaaS characteristics as well as conventional factors (load average and available resource) when scheduling functions. However, given a variety of serverless workloads, manually devising a reasonable fitness function for each may require countless testing and carefully fine-tuning.

In address the challenge of manually tuning a heuristic-based scheduler like Static-rank, we present FaaSRank, a self-learning algorithm to schedule functions for FaaS platforms based on deep reinforcement learning (DRL). FaaSRank uses neural networks to automatically learn the optimal function scheduling policies based on high-level objectives such as average FCT. FaaSRank evaluates resource utilization metrics of individual servers and information of the incoming function invocation, ranks all the servers, and selects the best server to schedule the function.

C. Deep Reinforcement Learning

FaaSRank uses RL and neural networks to learn function scheduling algorithms for FaaS. In a general RL setting, an agent learns how to benefit most from making sequential decisions by iteratively interacting with the environment and accumulating knowledge from previous experience. RL is well-suited to learning policies for computer systems, because RL agents are able to learn from real-world workloads and operating conditions without human-designed inaccurate assumptions. RL has been applied to various scheduling problems, such as resource management [15], network optimization [16], and device placement [17].

Specifically in RL, at every time t, the agent first observes a state s_t of the environment, and then makes a decision on taking an action a_t. Following the action, the environment changes its state to s_{t+1} and the agent perceives a reward r_t as feedback. The interactions are stochastic and assumed to be a Markov process, *i.e.*, the next state s_{t+1} and reward r_t solely depend on the previous state-action pair (s_t, a_t). Thus the agent learns to maximize its expected cumulative rewards

$$\mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t * r_t\Big], \tag{2}$$

where $\gamma \in (0, 1]$ is the discount factor to discount the sum of rewards by how far off in the future they're obtained [18].

The agent takes actions based on a *policy*, defined as a mapping between states and actions. A policy π outputs an action a_t when given a state s_t, *i.e.*, $a_t \sim \pi(\cdot|s_t)$. *Function approximators* are commonly used to represent parameterized policies. A function approximator outputs computable functions that depend on a set of adjustable parameters, θ, which we can adjust to affect the behavior of a policy via optimization algorithms. We refer to θ as *policy parameters* and represent the policy as $a_t \sim \pi_\theta(\cdot|s_t)$. In DRL, neural networks are used as function approximators to solve stochastic RL tasks, as neural networks are end-to-end differentiable for training and self-adaptive without hand-crafted features [19]. Therefore we use neural networks to represent FaaSRank's scheduling policy.

III. Overview

FaaSRank is an intelligent scheduler that uses neural networks to make function scheduling decisions for serverless platforms. On FaaS platforms, scheduling events involves orchestrating where a function executes on a distributed cluster. Activated by an event, FaaSRank takes as an input the current state information of the cluster, and the function request, and outputs a scheduling action, *i.e.*, a server to schedule the incoming function invocation on.

A. Challenges

We tackle three key challenges by designing FaaSRank:

1) **Server Assessment**. Given a function invocation request, the scheduler must select the best server from a FaaS cluster for function scheduling. It's non-trivial to compose together available metrics to assess individual servers to make reasonable trade-offs between cold starts and resource contention in real-time.
2) **Cluster Scalability**. It's necessary to define a fixed output size of a neural network, *i.e.*, a fixed number of servers within a cluster. However, it's common for multiple servers to join or leave the cluster, a problem that can force neural networks to be retrained.
3) **Huge action space.** Providers host commercial FaaS platforms on clusters consisting of thousands of servers. Selecting a server from a huge cluster requires the scheduler agent be trained over a huge action space, *i.e.*, the output size of a neural network must be linear to the cluster size. Mapping conditions to thousands of actions poses a challenge for training the scheduler, which has to explore the action space to learn a good policy.

To address Challenge 1, we adopt a combination of *resource utilization metrics* and function information as features to characterize the state of individual servers for function execution. To address Challenge 2 and 3, we propose a *score function* inspired by [19], which is implemented using neural networks to make scheduling decisions across clusters having an arbitrary number of servers. We describe our proposed solutions in detail in Section IV.

B. Objective

FaaSRank optimizes the average *Function Completion Time (FCT)* of a workload. The FCT of a function invocation is defined as the time from its arrival until completion. This includes initialization overhead, waiting time in any platform queues, and execution time.

We consider a FaaS platform that handles a multiple function concurrent workload. Let S denote the set of functions invoked within the workload, f denotes a function invocation in S. The platform captures the FCT c_f of an invocation after it completes execution. The total FCT C of a workload is denoted by $C = \sum_{f \in S} c_f$, which we want to minimize. Hence, we aim to minimize the average FCT given by:

$$\bar{C} = \frac{\sum_{f \in S} c_f}{|S|}. \tag{3}$$

Note that this objective does not guarantee scheduling fairness of individual functions, *i.e.*, FaaSRank treats a workload as an entity to optimize the average FCT, but does not guarantee a performance improvement for every function. We assessed the scheduling fairness provided by FaaSRank and report our results in Section VI. We compared the average FCT of individual functions processed by FaaSRank and with other baseline scheduling approaches. Our results show that in additional to the average FCT of the workload, FaaSRank is able to achieve the best average FCT for most of the individual functions while maintaining good performance for others.

Metric	Description
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 servicing HW interrupts
Δcpu_softirq	CPU time servicing soft interrupts
Δcpu_steal	CPU time spent by other operating systems
Δcpu_ctx_switches	Number of context switches
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
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
Δnet_byte_recv	Network Bytes received
Δnet_byte_sent	Network Bytes written

(a) Resource utilization metrics

Metric	Description
avail_memory	Memory available in the server
inflight_invocations	Number of inflight requests in the server
warm_infrastructures	Warm infrastructures in the server
requested_memory	Memory requested by the function
init_time	Measured function cold initialization time

(b) Server and function metrics

TABLE I: Metrics that comprise state observed by FaaSRank.

IV. Design

In this section, we present the design of FaaSRank. We present our approaches to address the challenges identified in Section III: server assessment (Section IV-A), cluster scalability, and huge action space (Section IV-B). We also describe the algorithm used to train FaaSRank (Section IV-C).

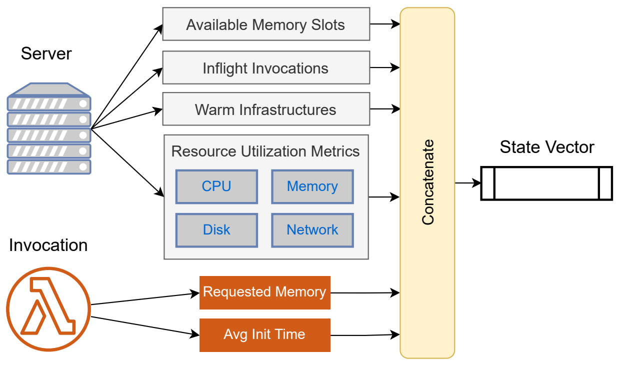


Fig. 2: The embedding of a state observed by FaaSRank

A. Server Assessment

FaaSRank uses *resource utilization metrics* to assess the resource condition of individual servers. Table I(a) shows the resource utilization metrics that FaaSRank collects from each server, where Δ indicates the change in resource utilization for the sampling interval. We characterize four dominant types of resource utilization when assessing a server: CPU, memory, disk, and network I/O. Previous research has shown that resource utilization metrics are powerful in identifying resource contention of Infrastructure-as-a-Service (IaaS) cloud [20] [21] [22], and predicting performance and costs of cloud workloads [23]. In addition to resource utilization metrics, FaaSRank also leverages metrics to assess cluster load and infrastructure as shown in Table I(b).

To observe state, FaaSRank collects the resource utilization metrics from each server, and encapsulates them with other metrics shown in Table I(b). Figure 2 describes the embedding of a state observed by FaaSRank, which contains information of a server and the incoming function invocation. FaaSRank concatenates the information into a flat feature vector as input to the score function in Section IV-B.

B. Score Function

FaaSRank calculates a *score function* to rank each server. The server with the highest score is selected for function scheduling. FaaSRank learns how to score individual servers using a common score function to select a server with the highest score, rather than training a neural network to choose a specific server, which would require a fixed size cluster resulting in a huge action space. Figure 3 visualizes the policy network and shows how FaaSRank selects the best server given a batch of state information. At time t, the FaaS cluster has in total N available servers to schedule an invocation event. FaaSRank collects a batch of the latest state vectors $s_t = (s_t^1, \dots, s_t^n, \dots, s_t^N)$ from the cluster, where n represents the n-th available server. After collecting state vectors, FaaSRank normalizes the state batch to ${s'}_t = ({s'}_t^1, \dots, {s'}_t^n, \dots, {s'}_t^N)$ as inputs to the score function. The score function is implemented using two neural networks, an actor and a critic network. Actor-Critic methods are effective in reducing training variance and delivering faster convergence [24].

- **Actor network** computes a score q_t^n, which is a scalar value mapped from the normalized state vector ${s'}_t^n$ representing a priority to select the server n. Then FaaSRank applies a Softmax operation [25] to the scores $(q_t^1, \dots, q_t^n, \dots, q_t^N)$ to compute the probability of selecting server n based on the priority scores, given by

$$P_t(server = n) = \frac{\exp(q_t^n)}{\sum_{n=1}^{N} \exp(q_t^n)}, \tag{4}$$

  at time t.
- **Critic network** outputs a baseline value b_t^n for server n, the averaged baseline value $\bar{b}_t$ is calculated as

$$\bar{b}_t = \frac{\sum_{n=1}^{N} b_t^n}{N}, \tag{5}$$

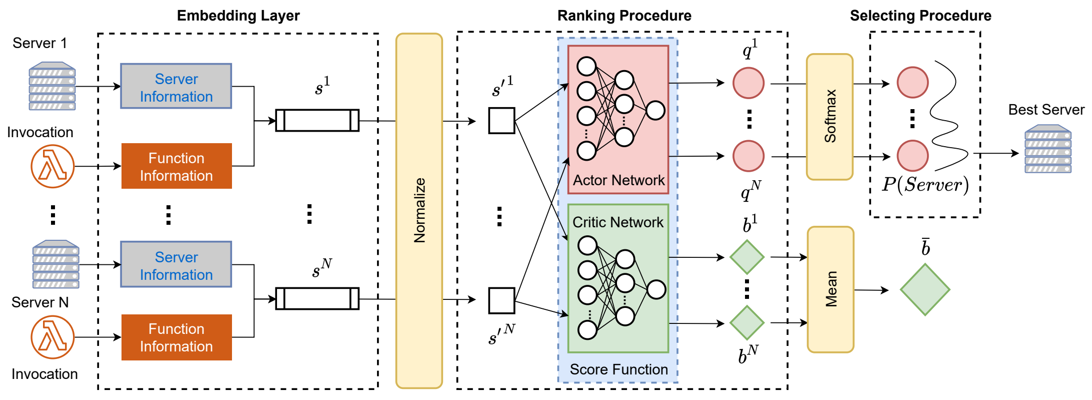


Fig. 3: The policy network in FaaSRank

which is used to reduce variance when training FaaSRank.

The whole operation of our policy network is end-to-end differentiable.

FaaSRank requires no manual feature engineering for its score function, *i.e.*, nothing is hard-coded in the score function. Through training, FaaSRank automatically learns what is important for computing a priority score given a state vector. More importantly, the design of FaaSRank is lightweight as it reuses the same score function for all servers and all function invocations. We further describe the details of training FaaSRank in Section IV-C.

Algorithm 1 FaaSRank Training Algorithm.

```
1: Initial policy (actor network) parameters $\theta_0$ and value function (critic network) parameters $\phi_0$
2: for episode k = 0, 1, 2, ... do
3:     Run policy $\pi_k = \pi(\theta_k)$ in the environment until terminating at time $T$
4:     Collect set of trajectories $\mathbb{D}_k = \{\tau_i\}$, where $\tau_i = (s_i, a_i), i \in [0, T]$
5:     Compute reward $\hat{r}_t$ via Equation 2
6:     Compute baseline value $\bar{b}_t$ via Equation 5
7:     Compute advantage $\hat{\mathbb{A}}_t = \hat{r}_t - \bar{b}_t$
8:     Update actor network by maximizing objective using stochastic gradient ascent:
           $$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathbb{D}_k|T} \sum_{\tau\in\mathbb{D}_k} \sum_{t=0}^{T} \mathbb{L}(s_t, a_t, \theta_k, \theta) \quad (6)$$
9:     Update critic network by regression on mean-squared error using stochastic gradient descent:
           $$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathbb{D}_k|T} \sum_{\tau\in\mathbb{D}_k} \sum_{t=0}^{T} (\bar{b}_t - \hat{r}_t)^2 \quad (7)$$
10: end for
```

C. Algorithm for Training FaaSRank

FaaSRank training proceeds in *episodes*. In each episode, client function invocations arrive at the FaaS platform where each requires a scheduling action to select a server. When all function invocations finish, we consider the episode complete. Let T denote the total number of actions in an episode, and t_i denote the wall clock time of the i-th action. Similar to [19], we continuously feed FaaSRank with a reward r after FaaSRank takes an action based on the objective (average FCT) mentioned in Section III-B. Concretely, we penalize FaaSRank with $r_i = -(t_i - t_{i-1}) * F_i$ after the i^{th} action, where F_i is the number of inflight function invocations in the FaaS system during the interval $[t_{i-1}, t_i)$. By setting the discount factor γ to be 1 in Equation 2, the goal of the algorithm is to maximize the expected cumulative rewards given by

$$\mathbb{E}\Big[\sum_{i=1}^{T} -(t_i - t_{i-1}) * F_i\Big], \quad (8)$$

which is aligned with Equation 2. Notice that this cumulative objective approximates the total FCT of a workload, and hence FaaSRank learns to minimize the average FCT in Equation 3 for a given workload.

FaaSRank uses a policy gradient algorithm for training. Policy gradient methods [26] are a class of RL algorithms that learn policies by performing gradient ascent directly on the parameters of neural networks, denoted by θ, using the rewards received during training. When updating policies, a large number of steps may deteriorate the performance, while a small number of steps may worsen the sampling efficiency. We use the Proximal Policy Optimization (PPO) algorithms to ensure FaaSRank takes appropriate steps when updating its policies. Recall in Section II-C, π_θ denotes a policy with parameters θ, a is the action taken when observing state s, the PPO algorithm updates policies via

$$\theta_{k+1} = \arg\max_\theta \mathop{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} \Big[\mathbb{L}(s, a, \theta_k, \theta)\Big], \quad (9)$$

where $\mathbb{L}$ is the *surrogate advantage* [27], a measure of how policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy. Specifically we use the PPO-clip version of a PPO algorithm, where $\mathbb{L}$ is given by

$$\mathbb{L}(s, a, \theta_k, \theta) = \min\Big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a),\ g(\epsilon, A^{\pi_{\theta_k}}(s, a))\Big), \quad (10)$$

and $g(\epsilon, A)$ is a clip operation defined as

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0, \end{cases} \tag{11}$$

where A is the advantage calculated as rewards r subtracted by baseline values b.

In Equation 11, ϵ is a hyperparameter which restricts how far the new policy is allowed to go from the old. Intuitively, the PPO algorithm sets a cap for the range of policy updates, to prevent the new policy from going too far (either positive or negative) from the old, thus ensuring an appropriate range of update steps.

Algorithm 1 presents the training of FaaSRank. During training for each server, the actor network outputs a score, and the critic network outputs a baseline value. For each episode, we record the whole set of trajectories including the states, actions, rewards, baseline values predicted by the critic network, and the logarithm probability of the actions for all events. After each training episode finishes, we use the collected information to update the actor and critic networks.

V. Implementation

FaaSRank provides a broadly applicable function scheduling algorithm for use in serverless platforms. For concreteness, we describe its implementation in the context of Apache OpenWhisk, an open source, distributed serverless platform. OpenWhisk executes functions in response to events at any scale, while managing infrastructure, servers, scaling, and execution of functions using Docker containers [9]. This section briefly describes the architecture of OpenWhisk, and illustrates the workflow of FaaSRank, *i.e.*, how FaaSRank interacts with and makes scheduling decisions for OpenWhisk.

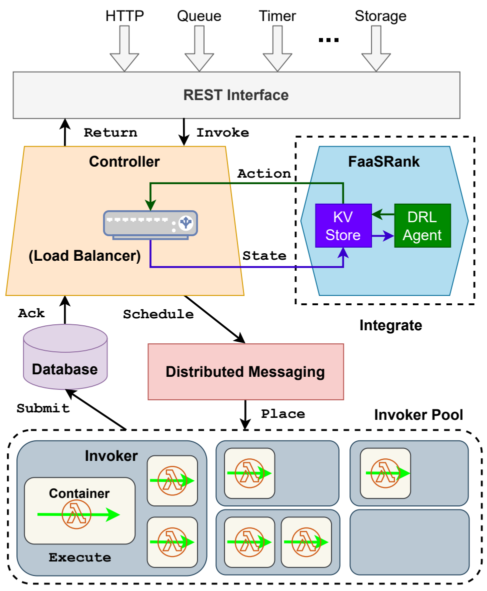


Fig. 4: FaaSRank scheduling architecture for OpenWhisk

A. OpenWhisk Architecture

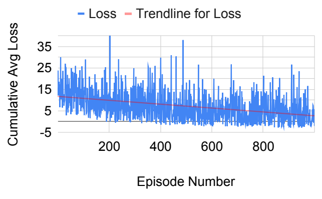
Figure 4 describes the architecture of our FaaSRank scheduler integrated with OpenWhisk. OpenWhisk exposes an NGINX-based [28] REST interface for creating new functions, invoking functions, and querying results of invocations. Invocations are triggered by users via an interface, and then forwarded to the Controller. The Controller selects an Invoker (typically hosted using VMs) to schedule the function invocation. The Load Balancer inside the Controller makes scheduling decisions for function invocations based on (1) a hashing algorithm, and (2) information from the Invokers, such as health, available capacity, and infrastructure state. Once an Invoker is chosen, the Controller sends the function invocation request to the selected Invoker via a Kafka-based [29] distributed message broker. The Invoker receives the request and executes the function using a Docker container. After the function execution is finished, the Invoker submits the results to a CouchDB-based [30] Database and informs the Controller. Then the Controller returns the results of the function executions to users synchronously or asynchronously.

B. FaaSRank

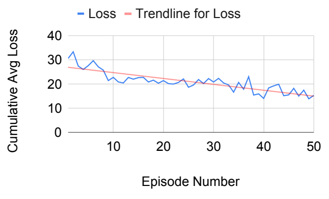
Workflow. FaaSRank communicates with OpenWhisk Controller via a Key-Value (KV) Store, which is implemented using Redis [31]. When receiving a function invocation, the Load Balancer in the Controller first sends the current state information to the KV Store. The DRL Agent in FaaSRank then fetches the state and sends an action to the KV Store, where the Controller picks up the action. Finally, the Load Balancer schedules the function invocation on the selected Invoker based on the action provided by FaaSRank.

Implementation. We implement the policy network of our FaaSRank prototype using two neural networks, each with two fully-connected hidden layers. The first hidden layer has 32 neurons, and the second layer has 16 neurons, each neuron uses `Tanh` as its activation function. The agent of FaaSRank is implemented in Python using PyTorch [32]. The implementation of FaaSRank is lightweight as the policy network consists of 2,818 parameters (16 KBs in total) because FaaSRank reuses the score function. Mapping a state to a scheduling action takes less than 70 ms.

Training. We use the algorithm presented in Section IV-C to train FaaSRank with 5 epochs per surrogate optimization and a 0.2 clip threshold [33]. We update the policy network parameters using the AdamW optimizer [34] with a learning rate of 0.0002. For single-workload evaluation, we train FaaSRank over 1000 episodes using the same workload. For multi-workload evaluation, we retrained FaaSRank over 500 episodes, where each workload had 50 training episodes. The total time for single-workload and multi-workload training took about 120 and 96 hours, respectively. We restarted the OpenWhisk platform before each training episode. Figure 5 shows the learning curve of FaaSRank for single-workload and multi-workload training. The descending loss trendlines indicate that FaaSRank gradually learns to make good decisions on scheduling functions through training.



(a) Single-workload training



(b) Multi-workload training

Fig. 5: Learning curve of FaaSRank through single-workload and multi-workload training

VI. Evaluation

We conducted extensive evaluations of FaaSRank in the OpenWhisk platform, using realistic serverless functions and real-world serverless traces realised by Microsoft Azure [13]. In this section, we first introduce baseline schedulers including the default OpenWhisk scheduler used in the evaluation (Section VI-A). We then describe our experimental setup including clusters, workloads, and invocation traces (Section VI-B). We present a comprehensive performance evaluation of FaaSRank in a single-workload experiment (Section VI-C), and the generalizability evaluation of FaaSRank in a multi-workload experiment (Section VI-D).

A. Baseline Schedulers

For our evaluation, we compare FaaSRank with five other schedulers serving as baselines:

1) **OpenWhisk default Hashing scheduler**. The OpenWhisk 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) **Round-robin scheduler**. A scheduler that distributes the load by sending successive requests to different invokers in a cyclical manner.
3) **Least-connections scheduler**. A scheduler that always sends the incoming invocation to the invoker with least in-flight requests.
4) **Greedy scheduler**. A scheduler that greedily packs function invocations onto the same invoker until the invoker reaches its capacity. This scheduler mimics the scheduling strategy of AWS Lambda, with the aim of improving resource utilization.
5) **Static-rank scheduler**. A fine-tuned scheduler that schedules function invocations based on the state information that FaaSRank receives, including resource utilization metrics, invoker, and function information. However, in contrast to FaaSRank's trained RL network, the static-rank scheduler employs a fixed, human-designed fitness function (Equation 1) to select the best invoker for function invocation.

B. Experimental Setup

OpenWhisk clusters. We deployed and tested FaaSRank using two independent OpenWhisk deployments on different public clouds, where each OpenWhisk cluster consisted of 13 VMs. One VM hosted the REST front-end, API gateway, and Redis services; one backend VM hosted the Controller, Distributed Messaging, and Database services; one VM hosted our FaaSRank agent; and the remaining 10 VMs were configured as invokers for scheduling functions. We present the details of two OpenWhisk clusters:

1) **Compute Canada Cloud cluster.** We deployed FaaSRank under OpenWhisk on the Compute Canada Cloud [35] using 13 VMs, each with 8 Intel Xeon Skylake vCPU cores and 32 GBs memory. Each invoker provided 2 GBs RAM for individual function executions while allocating CPU power to functions proportionally based on function memory requirements.
2) **AWS EC2 cluster.** We deployed FaaSRank under OpenWhisk on an AWS EC2 cluster consisting of 13 `c5d.2xlarge` VMs launched as spot instances, each with 8 Intel Xeon Platinum vCPU cores, and 16 GBs memory. Each invoker provided 2 GBs RAM for individual function executions while allocating CPU power to functions proportionally based on function memory requirements.

For single-workload evaluation, we trained FaaSRank on our Compute Canada Cloud and AWS OpenWhisk clusters independently. We conducted a comprehensive performance evaluation for each cluster independently. For multi-workload evaluation, we retrained FaaSRank using an AWS cluster to evaluate the generalizability of FaaSRank. To assess scheduler performance, we repeated our performance experiments 10 times using FaaSRank comparing performance against our five baseline schedulers in both evaluations.

Functions. For our experiments, we leveraged ten real-world serverless functions from SeBS [10], ServerlessBench [11], and ENSURE workloads [12]. Table II characterizes the memory consumption, cold *vs.* warm runtimes, and initialization time for cold starts. All ten functions are implemented in Python3 using CouchDB to store input and output objects. To accurately profile the functions, we deployed a mini OpenWhisk cluster hosted on an AWS EC2 dedicated host [36], an isolated private server to characterize the average runtime and overhead in a setting without resource contention from other users. The test cluster consisted of 4 `c5.2xlarge` VMs (user, frontend, backend, and one invoker), each with 8 vCPU cores and 16 GBs memory. We executed each function 10 times to characterize average values for our performance metrics as shown in Table II.

Adapting Azure Functions Traces. We leveraged public function invocation traces from Microsoft Azure provided in [13] for our evaluations. Our objective was to derive realistic testing workloads that would leverage the full capacity of our OpenWhisk clusters. We adapted the Azure traces by reinterpreting the platform sampling interval from minutes

Function	Type	Dependency	Memory (MBs)	Cold (s)	Warm (s)	Init (s)
Dynamic Html (DH)	Web App	Jinja2, CouchDB	512	4.45	2.34	1.55
Email Generation (EG)	Web App	CouchDB	256	2.20	0.21	1.55
Image Processing (IP)	Multimedia	Pillow, CouchDB	256	5.88	3.52	1.69
Video Processing (VP)	Multimedia	FFmpeg, CouchDB	512	6.86	1.19	4.77
Image Recognition (IR)	Machine Learning	Pillow, torch, torchvision, CouchDB	512	4.28	0.09	1.33
K Nearest Neighbors (KNN)	Machine Learning	scikit-learn, CouchDB	512	4.99	1.11	3.45
Gradient Descent (GD)	Machine Learning	NumPy, CouchDB	512	4.15	0.60	2.59
Arithmetic Logic Unit (ALU)	Scientific	CouchDB	256	5.72	3.45	1.50
Merge Sorting (MS)	Scientific	CouchDB	256	3.87	1.94	1.54
DNA Visualisation (DV)	Scientific	Squiggle, CouchDB	512	8.57	3.11	4.13

TABLE II: Characterizations of Experimental Serverless Function Workloads

to seconds. This was necessary because the traces captured the total number of function invocations per minute. This re-scaling increases the intensity of our workloads while speeding up our training by reducing the total workload duration. To derive a workload, we randomly selected 10 individual Azure traces, and mapped each to a FaaS function from Table II.

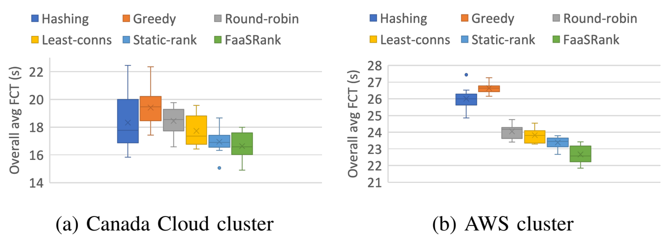
WL	Load	Agg CPU Time	Num calls	Avg IAT	Len
SC	93.75 %	4368.71 s	292	0.262 s	60 s
SA	132.9 %	6196.89 s	408	0.184 s	60 s
M1	56.67 %	2640.94 s	209	0.219 s	37 s
M2	57.00 %	2656.32 s	178	0.242 s	36 s
M3	57.01 %	2656.69 s	201	0.255 s	44 s
M4	59.05 %	2751.87 s	201	0.217 s	35 s
M5	71.83 %	3347.33 s	226	0.236 s	44 s
M6	74.95 %	3492.49 s	253	0.251 s	53 s
M7	80.22 %	3738.29 s	256	0.244 s	52 s
M8	82.54 %	3846.15 s	276	0.215 s	48 s
M9	86.40 %	4026.24 s	318	0.210 s	54 s
M10	100.00 %	4659.86 s	295	0.242 s	59 s

TABLE III: Characterization of workloads used in evaluation. Metrics include: CPU load (%) relative to workload 10, aggregated CPU time estimate, total function invocations, average inter-arrival time (IAT), and workload duration. Workload types include: (SC) single-workload on Canada Cloud cluster, (SA) single-workload on AWS EC2 cluster, and (M) 10 multi-workloads on AWS EC2 cluster.

C. Single-workload Evaluation

Workloads. For our single-workload evaluation we created workloads for the Canada Cloud and AWS by randomly selecting function invocation traces from the Azure Functions traces. Each *workload* contains 10 invocation traces, *i.e.*, one trace per function. Two workloads (SC and SA) shown in Table III were designed to intensively occupy the full capacity of two clusters. To fully occupy available cluster capacity on AWS, we had to increase the intensity of workload SA relative to workload SC. The AWS EC2 `c5` instances were found to be more powerful than the Canada Cloud VMs. As a result we compensated by increasing workload SA intensity by approximately 40%.

We evaluate the following characteristics of the FaaS schedulers from Section VI-A: (1) Performance. Average FCT is the primary objective that FaaSRank tries to optimize; (2) Fairness. We examined how FaaSRank improved the average FCT of individual functions. Improving FCT for one function should not degrade that of another; (3) Cluster load. The time series load condition for the OpenWhisk cluster, which helps to evaluate the utility of scheduling decisions.

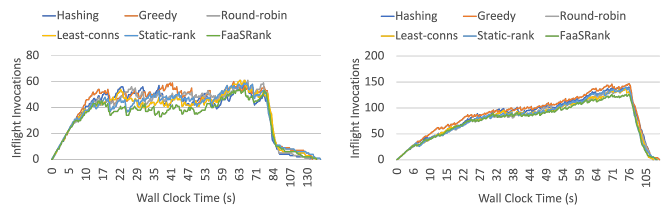


(a) Canada Cloud cluster (b) AWS cluster

Fig. 6: Avg. FCT of schedulers measured on two OpenWhisk clusters

Average FCT. Figure 6 shows the average FCT of individual schedulers evaluated on two clusters when scheduling an experimental workload. FaaSRank outperforms the five baseline schedulers. Compared to OpenWhisk's default Hashing scheduler, FaaSRank reduced average FCT by 9.25% and 12.82%, on the Canada Cloud and AWS clusters respectively. Compared to the baseline schedulers, FaaSRank learns how to reduce average FCT while improving overall performance.

Fairness. To evaluate the fairness of FaaSRank's scheduling, we recorded the average FCT for each function during the experiment. Table IV presents the average FCT of individual functions from 10 repeated runs. FaaSRank achieves the minimum average FCT for 7 and 8 functions on the Canada Cloud and AWS clusters respectively. While reducing the average FCT for most functions, FaaSRank also maintains acceptable performance for other functions. FaaSRank provided the minimum average FCT of workloads of the tested schedulers.



(a) Canada Cloud cluster (b) AWS cluster

Fig. 7: Time series observation of inflight invocations

Cluster load. We recorded the cluster load conditions of each scheduler during the experiments. Figure 7 displays the time series of inflight invocations for each scheduler on our

Function	H-C	G-C	RR-C	LC-C	SR-C	FaaSRank-C	H-A	G-A	RR-A	LC-A	SR-A	FaaSRank-A
DH	100.00	116.18	102.79	108.19	102.57	**98.55**	100.00	105.67	97.83	96.39	93.39	**89.86**
EG	**100.00**	116.56	125.33	114.93	114.37	106.89	100.00	107.10	96.32	92.20	87.98	**82.65**
IP	100.00	127.88	123.07	112.79	102.37	**94.12**	100.00	98.10	87.80	86.62	84.62	**83.17**
VP	100.00	97.73	97.56	98.23	92.26	**81.88**	100.00	104.51	98.68	92.48	**91.93**	93.91
IR	100.00	117.53	113.68	**97.21**	100.75	103.85	100.00	110.82	97.24	91.23	87.76	**83.65**
KNN	100.00	112.31	101.15	91.66	92.84	**82.71**	100.00	100.19	90.63	91.84	88.60	**86.86**
ALU	100.00	92.33	88.36	79.70	76.86	**75.46**	100.00	98.08	88.91	88.21	88.20	**84.84**
MS	100.00	94.71	90.50	85.29	83.07	**82.76**	100.00	105.44	95.61	93.76	90.45	**88.76**
GD	100.00	103.78	102.24	104.42	101.69	**97.69**	100.00	102.12	89.44	90.90	88.56	**86.17**
DV	**100.00**	115.60	106.81	109.56	107.59	102.86	100.00	103.91	93.85	93.67	**90.05**	93.71

TABLE IV: Fairness evaluation depicting average FCT of individual functions normalized as a percentage (%) relative to the Hashing scheduler on the (**C**)anada Cloud, and (**A**)WS clusters. Schedulers include: (**H**)ashing, (**G**)reedy, (**RR**) Round-Robin, (**LC**) Least-Connections, (**SR**) Static-Rank. FaaSRank improved performance of 7 out of 10 (**C**), and all functions (**A**).

two OpenWhisk clusters. For the duration of test workloads, 59.62% and 70.43% of the time on average FaaSRank sustained a lower number of inflight invocations for the Canada Cloud and AWS EC2 cluster respectively. Workloads scheduled by FaaSRank were completed faster overall than on the baseline schedulers.

D. Multi-workload Evaluation

Workloads. We randomly selected 100 Azure Functions invocation traces to create 10 distinct workloads to evaluate the generalizability of FaaSRank when trained over multiple workloads. Table III characterizes our randomly generated workloads. The intensity of the gradually increases from workload M1 to M10. We sought to evaluate FaaSRank's scheduling ability when trained over multiple distinct workloads. We retrained FaaSRank on our AWS EC2 cluster with 500 episodes, 50 for each workload. No other parameters were changed from our single-workload training. We evaluated the average FCT of 10 workloads from Table III. We repeated each workload 10 times and report the average results.

Average FCT. Table V shows the average FCT normalized as a percentage relative to the Hashing scheduler for each workload on the AWS EC2 cluster. FaaSRank achieved minimum average FCT for 4 of 10 workloads while outperforming the default Hashing scheduler for all workloads. Compared to the Hashing scheduler, FaaSRank achieved an improvement of 23.05% on average FCTs for 10 workloads.

WL	H	G	RR	LC	SR	FaaSRank
1	100.00	121.55	81.17	**72.13**	77.31	80.82
2	100.00	116.19	79.94	80.52	**77.12**	85.23
3	100.00	101.86	70.20	66.36	**64.68**	70.40
4	100.00	108.00	92.13	85.39	84.99	**81.14**
5	100.00	109.43	77.88	**74.32**	76.91	75.98
6	100.00	106.41	63.57	60.00	60.10	**57.75**
7	100.00	123.53	82.81	70.31	73.83	**70.04**
8	100.00	107.53	88.76	**88.13**	88.19	90.84
9	100.00	104.77	80.48	92.53	**78.86**	80.11
10	100.00	99.99	78.41	83.79	79.59	**77.20**

TABLE V: Average FCT normalized as a percentage (%) relative to the hashing scheduler for 10 workloads used in multi-workload evaluation. Schedulers: (**H**)ashing, (**G**)reedy, (**RR**) Round-Robin, (**LC**) Least-Connections, (**SR**) Static-Rank.

VII. Related Work

Schedulers for traditional web services. Classic load balancing algorithms have been working well for traditional request dispatching on clusters hosting web services. Some commonly used load balancing algorithms include Round-robin, Least-connections, Greedy, and Hashing. Providers generally employ one or a combination of classic algorithms to balance load for web services. However we demonstrate in our paper that classic algorithms are not FaaS aware and can degrade performance of serverless workloads.

Schedulers for serverless computing. Among all the FaaS schedulers, Smart Spread is the most similar to FaaSRank [37]. Smart Spread leverages supervised ML to provide a function placement algorithm to schedule functions in serverless environments while incorporating resource utilization metrics of servers as input data. However, in contrast to our RL-based FaaSRank, one critical drawback of smart spread is the requirement to perform function profiling before deployment, which must occur on a separate VM before the function can be deployed. This approach limits the immediate availability of function deployment, which is counter to the goals of FaaS. FaaSRank gradually incorporates training data as RL training proceeds, with no requirement to profile functions before deployment. Another limitation of smart spread is the lack of a performance evaluation on a real serverless platform. We evaluate FaaSRank with extensive experiments on OpenWhisk clusters. In [38], a centralized scheduler for serverless platforms is proposed that assigns each CPU core of worker servers to CPU cores of scheduler servers for fine-grained core-to-core management. While the proposed approach seems promising, it doesn't consider resource conditions of worker servers when scheduling functions, and hasn't been evaluated in a real serverless platform. In [12], the ENSURE scheduler for managing resources in serverless platforms is presented which is also implemented and evaluated on OpenWhisk. While ENSURE focuses on CPU power allocation, it only adopts a simple greedy algorithm to schedule functions. In [39], a scheduler for optimizing distributed machine learning using a serverless computing service is presented. In contrast, FaaSRank is designed to schedule any serverless workloads.

RL-based schedulers. Recent RL-based scheduling approaches share similar objectives with FaaSRank. DeepRM is an RL-based scheduler in a simulation system [15], which achieves scheduling fairness by assuming jobs execute identically regardless of the state of resource contention in the system. The Decima scheduler also leverages RL to schedule DAG jobs for data processing clusters [19]. Decima is implemented in Apache Spark framework to assign executors to optimize job completion time of workloads. FaaSRank optimizes a similar objective (average FCT). Decima optimizes the average job completion time for a workload without addressing fairness [19]. However, Decima is not designed for serverless environments, as it ignores serverless computing features such as rapid changes in resource utilization, and warm *vs.* cold starts. Directly replacing FaaSRank with Decima in a serverless environment is not feasible. To our knowledge, there are no existing solutions for function scheduling on serverless platforms that employ RL.

VIII. Conclusion

In this paper, we present FaaSRank, an intelligent scheduler driven by deep reinforcement learning tailored to schedule functions in serverless platforms. FaaSRank employs neural networks to automatically learn function scheduling decisions. We implement FaaSRank in the Apache OpenWhisk platform, and evaluate performance against other baseline schedulers using realistic serverless functions and workload traces on two clusters deployed to the Compute Canada Cloud and AWS EC2. For single-workload evaluation, our results show that FaaSRank outperforms baselines by achieving the minimal average function completion time of workloads while maintaining good performance for individual functions. FaaSRank reduced the average function completion time by 9.25% and 12.82% over the OpenWhisk's default scheduler. For the duration of test workloads, 59.62% and 70.43% of the time on average FaaSRank sustained a lower number of inflight invocations for the Canada Cloud and AWS EC2 cluster respectively. For multi-workload evaluation, our results demonstrate the generalizability of FaaSRank to any workload. FaaSRank reduced average function completion time by 23.05% compared to OpenWhisk's default scheduler.

Acknowledgments

This research is supported by the NSF Advanced Cyberinfrastructure Research Program (OAC-1849970), NIH grant R01GM126019, and AWS Cloud Credits for Research.

References

[1] P. Castro *et al.*, "The rise of serverless computing," *Comm. of the ACM*, 2019.

[2] "AWS Lambda: Serverless Compute," https://aws.amazon.com/lambda/, [Online].

[3] "Microsoft Azure Functions," https://azure.microsoft.com/en-us/services/functions/s, [Online].

[4] "Google Cloud Function:Event-Driven Serverless Compute Platform," https://cloud.google.com/functions, [Online].

[5] "IBM Cloud Functions," https://www.ibm.com/cloud/functions, [Online].

[6] W. Lloyd *et al.*, "Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads," in *Proc. IEEE/ACM Int. Conf. on Utility and Cloud Comp.*, 2018.

[7] L. Wang *et al.*, "Peeking behind the Curtains of Serverless Platforms," in *Proc. the Usenix Annual Technical Conf. (USENIX ATC)*, 2018.

[8] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[9] "Apache OpenWhisk," https://openwhisk.apache.org, [Online].

[10] M. Copik *et al.*, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," *arXiv preprint arXiv:2012.14132*, 2020.

[11] T. Yu *et al.*, "Characterizing Serverless Platforms with ServerlessBench," in *Proc. ACM Sym. on Cloud Comp. (SoCC)*, 2020.

[12] A. Suresh *et al.*, "ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments," in *Proc. the Int. Conf. on Autonomic Comp. and Self-Organizing Systems (ACSOS)*, 2020.

[13] M. Shahrad *et al.*, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. the USENIX Annual Technical Conf. (USENIX ATC)*, 2020.

[14] W. Lloyd *et al.*, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *Proc. IEEE Int. Conf. on Cloud Engineering (IC2E)*, 2018.

[15] H. Mao *et al.*, "Resource Management with Deep Reinforcement Learning," in *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2016.

[16] L. Chen *et al.*, "AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization," in *Proc. 2018 Conf. of the ACM Spec. Int. Group on Data Communication (SIGCOMM)*, 2018.

[17] A. Mirhoseini *et al.*, "Device Placement Optimization with Reinforcement Learning," in *Proc. the Int. Conf. on ML (ICML)*, 2017.

[18] "OpenAI Spinning Up," https://spinningup.openai.com/en/latest/, [Online].

[19] H. Mao *et al.*, "Learning Scheduling Algorithms for Data Processing Clusters," in *Proc. ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[20] W. Lloyd *et al.*, "Performance Modeling to Support Multi-tier Application Deployment to Infrastructure-as-a-Service Clouds," in *Proc. IEEE Int. Conf. on Utility and Cloud Comp. (UCC)*, 2012.

[21] W. Lloyd *et al.*, "Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services," in *Proc. IEEE Int. Conf. on Cloud Engineering (IC2E)*, 2017.

[22] X. Han *et al.*, "Characterizing Public Cloud Resource Contention to Support Virtual Machine Co-residency Prediction," in *Proc. IEEE Int. Conf. on Cloud Engineering (IC2E)*, 2020.

[23] C. Robert *et al.*, "Predicting Performance and Cost of Serverless Computing Functions with SAAF," in *Proc. IEEE Int. Conf. on Cloud and Big Data Comp. (CBDCom)*, 2020.

[24] V. Konda *et al.*, "Actor-Critic Algorithms," in *Proc. the Int. Conf. on Neural Information Processing Systems (NIPS)*, 2000.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[26] R. S. Sutton *et al.*, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Proc. the Int. Conf. on Neural Information Processing Systems (NIPS)*, 1999.

[27] J. Schulman *et al.*, "Trust region policy optimization," in *Int. conf. on ML*. PMLR, 2015, pp. 1889–1897.

[28] "NGINX: High Performance Load Balancer, Web Server, Reverse Proxy," https://www.nginx.com/, [Online].

[29] "Apache Kafka," https://kafka.apache.org, [Online].

[30] "Apache CouchDB," http://couchdb.apache.org, [Online].

[31] "Redis," http://redis.io/, [Online].

[32] "PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration," https://pytorch.org, [Online].

[33] J. Schulman *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[34] L. Ilya *et al.*, "Decoupled Weight Decay Regularization," in *Proc. IEEE Int. Conf. on Learning Representations (ICLR)*, 2019.

[35] "Compute Canada Cloud," https://www.computecanada.ca/, [Online].

[36] "Amazon EC2 Dedicated Hosts," https://aws.amazon.com/ec2/dedicated-hosts/, [Online].

[37] N. Mahmoudi *et al.*, "Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm," in *Proc. the 29th Int. Conf. on Comp. Sci. and Software Engineering (CASCON)*, 2019.

[38] K. Kaffes *et al.*, "Centralized Core-Granular Scheduling for Serverless Functions," in *Proc. ACM Syp. on Cloud Comp. (SoCC)*, 2019.

[39] H. Wang *et al.*, "Distributed Machine Learning with a Serverless Architecture," in *Proc. IEEE Conf. on Comp. Comm. (INFOCOM)*, 2019.