2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00027

KneeScale: Efficient Resource Scaling for Serverless Computing at the Edge

Xue Li, Peng Kang, Jordan Molone, Wei Wang and Palden Lama
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas
Email: {xue.li, peng.kang, jordan.molone2, wei.wang, palden.lama}@utsa.edu

***Abstract*—Serverless computing is a promising paradigm for delivering services to the Internet of Things (IoT) applications at the edge of the network. Its event-triggered computation, as well as fine-grained and agile resource scaling, is well-suited for a resource-constrained edge computing environment. However, general-purpose auto-scalers that are predominant in the cloud settings perform poorly for serverless computing at the Edge. This is mainly due to the difficulty in quickly determining the optimal resource allocation under resource-budget constraints and dynamic workloads. In this paper, we present an adaptive auto-scaler, *KneeScale*, that dynamically adjusts the number of replicas for serverless functions to reach a point at which the relative cost to increase resource allocation is no longer worth the corresponding performance benefit. We have designed and implemented KneeScale as lightweight system software that utilizes Kubernetes for resource management. Experimental results with a function-as-a-service (FaaS) benchmark, *FunctionBench*, and an open-source serverless computing platform, *OpenFaaS*, demonstrate the superior performance and resource efficiency of KneeScale. It outperforms Kubernetes Horizontal Pod AutoScaler (HPA) and OpenFaaS built-in scheduler in terms of cumulative performance under a given resource budget by up to 32% and 106% respectively. KneeScale achieves higher cumulative throughput than both competing techniques, lower latencies than OpenFaaS built-in scheduler, and similar latencies compared to HPA for a variety of serverless functions.**

***Keywords*-Resource Scaling, Edge, Serverless Computing**

I. Introduction

Serverless computing is an application deployment architecture that completely hides server management from cloud customers. Cloud functions, packaged as FaaS (Function as a Service) offerings, represent the core of serverless computing. It allows customers to focus on developing functions (small code dedicated to specific tasks) that execute in a dedicated short-lived function instance (e.g., a container). A function instance is launched only when the function is invoked and is shut down when it is idle for a certain period of time. With its fine-grained resource scaling, event-triggered computation, and ephemeral functions, serverless computing provides an attractive execution model for a resource-constrained edge computing environment.

Fig. 1 illustrates the workflow of running serverless functions on edge nodes. IoT devices (e.g., wireless sensors, drones, AR/VR devices, etc.) generate events that trigger the execution of short-lived functions on edge nodes (e.g., router, cellular base station, Raspberry Pi, etc.) for data processing and analytics. Edge computing leverages computing resources located at the network edge, close to users and end-devices, to provide low-latency, location-aware and privacy-enhanced services for Internet of Things (IoT) applications [1]–[3]. The adoption of the serverless paradigm for edge computing has recently gained significant attention from both cloud providers (e.g., Lambda@Edge[1] and Fastly Compute@Edge[2]) and academia [4]–[6]. However, there are some significant resource management challenges that need to be addressed.

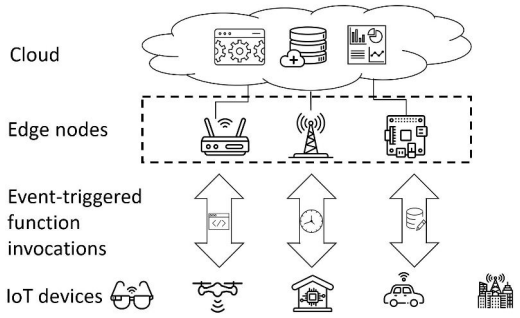


Fig. 1. Serverless Computing at the Edge

Existing serverless computing platforms rely on general-purpose auto-scalers such as Horizontal Pod Autoscaling (HPA)[3] for resource management. These auto-scalers adjust the number of function instances (aka replicas) based on a user-specified resource utilization target. However, it is indeed challenging for end-users to find the optimal resource utilization target for diverse functions and such an approach can perform poorly in a resource-constrained edge computing environment. Our case study in Section II using an open-source Function as a Service platform, OpenFaaS[4] and FunctionBench [7] benchmark shows that HPA with various resource utilization targets result in poor throughput and/or high function latency under resource budget constraints. Due to the limited resources available in edge nodes, there can be resource-budget constraints for the services running on them, including serverless computing services [8]. Determining opti-

[1]https://aws.amazon.com/lambda/edge/
[2]https://www.fastly.com/products/edge-compute/serverless
[3]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[4]https://www.openfaas.com

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00027

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.

mal resource allocation under resource-budget constraints is a challenging problem especially when the workload is dynamic and hard to predict [9], [10].

In this paper, we present an adaptive auto-scaler, KneeScale, that efficiently scales serverless functions under resource budget constraints in edge systems. KneeScale is designed to dynamically adjust the number of function instances to quickly reach a Knee point at which the relative cost to increase resource allocation is no longer worth the corresponding performance benefit. Our hypothesis is that running serverless computing systems at such a knee operating point can maximize the performance under a given resource budget. KneeScale leverages the Kneedle [11] algorithm, a general approach to detect performance knee point that is applicable to a wide range of systems. Utilizing the Kneedle algorithm for serverless computing systems at the edge is a non-trivial task. This is because the Kneedle algorithm requires prior performance data for a range of resource allocations. However, data collected offline can be unreliable in the presence of previously unseen workloads (new functions, arrival rates, etc.) and changing hardware configurations. On the other hand, a naive online approach of sampling performance data for all possible resource allocations can be prohibitively expensive. KneeScale addresses these challenges through a search algorithm that we developed for quick detection of performance knee points.

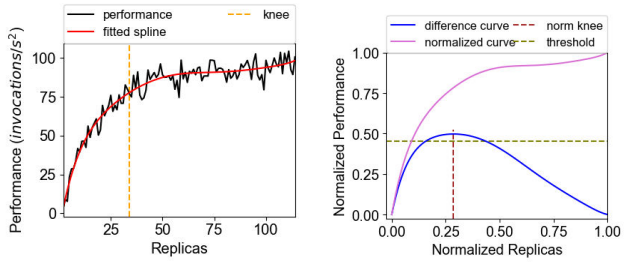
We have designed and implemented KneeScale as lightweight system software that utilizes Kubernetes for resource management and Docker containers for running serverless functions. Throughout the paper, we use the terms function instance, replica, and container interchangeably. We evaluated KneeScale using the OpenFaaS platform, deployed on a testbed of four Virtual Machines (VMs) representing the Edge nodes. Experimental results with a representative FaaS benchmark, FunctionBench [7], show that KneeScale significantly outperforms existing auto-scalers including OpenFaaS built-in scheduler and HPA in terms of cumulative performance under a given resource budget. KneeScale achieves higher cumulative throughput than both competing techniques, lower latencies than OpenFaaS built-in scheduler, and similar latencies compared to HPA for a variety of serverless functions under static as well as dynamic workloads.

The remainder of the paper is organized as follows. Section II discusses the background and motivation for running serverless computing systems at a knee operating point. Section III elaborates the key design and implementation details of KneeScale. Section IV details the testbed setup and experimental results. In Section V, we present the related work. Finally, conclusions are drawn in Section VI.

II. Background and Motivation

A. Detecting Performance Knee Point

It is common to see computer systems reach a point at which the relative cost to increase resource allocation is no longer worth the corresponding performance benefit. These "knees" typically represent the trade-off points between performance and resource cost [11]. As a case study, we analyzed the



(a) Performance and the fitted spline (b) Normalized knee

Fig. 2. Knee point detection for *float-operation* at concurrency level 20

performance of *float-operation* function from the FunctionBench [7] benchmark using the OpenFaaS platform that was deployed on a testbed of four VMs. As shown in Eqn. 1, we define a performance score as the ratio of throughput and average function latency. Here, throughput is the number of successful function invocations per second. Function latency is the sum of multiple components including (1) initialization time or cold start time spent in preparing a function instance (warm functions skip initialization), (2) wait time spent inside OpenFaaS before execution, and (3) execution time spent to run the function. Higher throughput and lower latency lead to higher performance score. We used an HTTP workload generation tool, *Hey*[5], to produce function invocations using 20 concurrent workers.

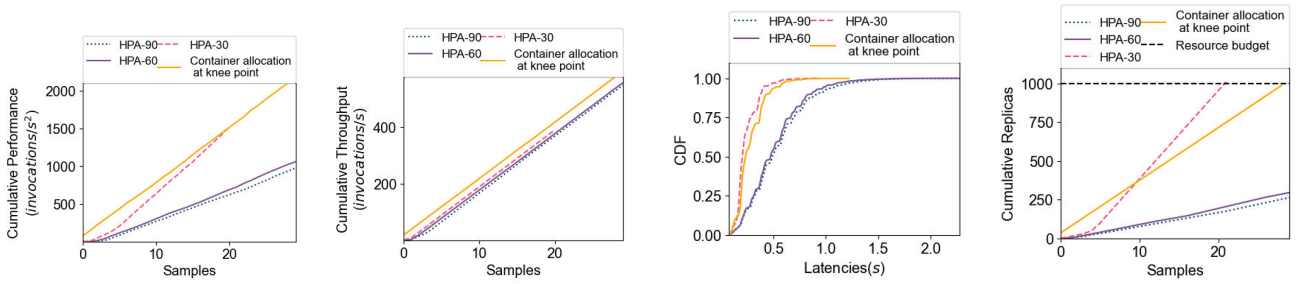
$$Performance\ Score = \frac{Throughput}{Average\ Function\ Latency} \tag{1}$$

$$T_{lmx_i} = y_{lmx_i} - S \cdot \frac{\sum_{i=1}^{n-1}(x_{i+1} - x_i)}{n-1} \tag{2}$$

Fig. 2(a) shows that initially the performance score increases sharply as more replicas (function instances) are allocated but after a certain point the gain in performance score keeps getting smaller. The curve approximately follows Amdahl's law. With the given performance data for various resource allocations, the knee point can be detected by utilizing Kneedle [11], a general knee detection algorithm. Kneedle defines the knee point as the point of maximum curvature in a continuous function. It uses six steps to find the knee point of the curve. (1) Using replica count as x- and performance score as y-values, Kneedle fits a smoothing spline to the data points, as shown in Fig. 2(a). (2) It normalizes the data to keep both x and y-values in the range $[0, 1]$. (3) It calculates the set of differences between the x and y-values, i.e., the set of points $(x_i, y_i - x_i)$, which is represented by the blue difference curve in Fig. 2(b). The goal is to find out when the difference curve changes from horizontal to sharply decreasing, since this indicates the presence of a knee in the original data set. (4) It finds the local maxima of the difference curve. If there are more than one local maximum point, each of these points are a candidate knee point in the original data curve.

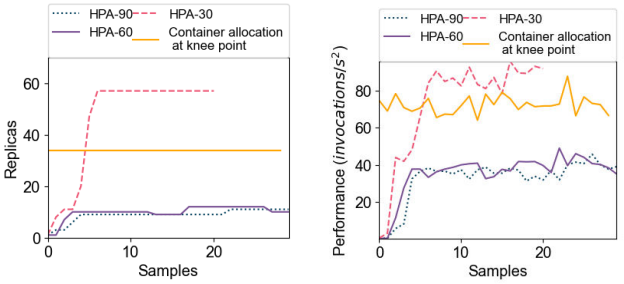
[5]https://github.com/rakyll/hey/

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.



(a) Cumulative performance score (b) Cumulative throughput (c) Function latency distribution (d) Cumulative resource allocation

Fig. 3. Comparison between HPA and knee-based resource allocation for *float-operation* at concurrency level 20. A resource-budget constraint of 1000 replicas is imposed on the cumulative resource allocation. The experiment is run for 15 minutes and data is sampled at 30-second intervals.



(a) HPA vs. knee-based resource allocation (b) Performance score at each sampling interval

Fig. 4. Impact of HPA and knee-based resource allocation on the performance score of *float-operation* at concurrency level 20

(5) For each local maximum point (x_{lmx_i}, y_{lmx_i}), Kneedle defines a threshold T_{lmx_i} based on Eqn. 2 that depends on the weighted difference between the local maximum value and a user-defined sensitivity parameter, S. Smaller values for S detect knees quicker, while larger values are more conservative. Throughout the paper, we set S = 1 for our experiments since it provides a good tradeoff between Knee detection time and accuracy in online settings [11]. (6) If any value in the difference curve drops below the threshold $y = T_{lmx_i}$ for the local maximum (x_{lmx_i}, y_{lmx_i}) before the next local maximum is reached, Kneedle declares a knee at the x-value of the corresponding local maximum $x = x_{lm_x}$. As shown in Fig. 2(b), the normalized knee is found at x = 0.29 and is detected after observing the point x = 0.45. The equivalent knee point in the original data set is at 34 replicas.

B. Inefficiency of Existing Auto-scalers

Existing cloud-based serverless computing platforms predominantly rely on general-purpose auto-scalers such as Kubernetes Horizontal Pod Autoscaling (HPA) for resource management. Kubernetes implements HPA as a control loop that periodically monitors the average resource utilization (usually CPU utilization) of currently running function instances. With the monitoring results, it then automatically scales the number of replicas based on the ratio between the current resource utilization and a user-defined target utilization value. Eqn. 3 shows how HPA calculates the desired number of replicas for a given function.

$$desiredReplicas = \lceil currentReplicas * \frac{currentMetricValue}{desiredMetricValue} \rceil \quad (3)$$

In this section, we demonstrate that HPA-based resource allocation significantly underperforms compared to Knee point-based resource allocation under resource-budget constraints. We use *Hey* to invoke the *float-operation* function with 20 concurrent workers for a duration of 15 minutes. We assume a resource-budget constraint of 1000 replicas on the cumulative resource allocation for the given function. The cumulative resource allocated to a function is the area under the curve of instantaneous resources allocated over time. When the cumulative sum of replicas allocated to the *float-operation* function exceeds the budget, all relevant replicas are stopped to free the allocated resources. To evaluate HPA performance, we set the CPU utilization target to be 30%, 60%, and 90% and denote these cases as HPA-30, HPA-60, and HPA-90 respectively. To evaluate knee-based resource allocation, we set the replica count to 34, which is the knee point detected by the Kneedle algorithm as shown in Fig. 2(a).

Fig. 3(a) shows that the cumulative performance scores of HPA-30, HPA-60, and HPA-90 are significantly lower than that of knee-based resource allocation. The cumulative performance score is the sum of performance scores accumulated so far at a given sampling interval. We observe that high CPU utilization targets of 60% and 90% will lead to high function latencies as shown in Fig. 3(c) and low function performance as shown in Fig. 4(b). Fig. 3(d) shows how each approach reaches the resource budget. A low CPU utilization target of 30% achieves lower latencies at the cost of excessive resource allocation that can exhaust the resource budget quickly. The knee-based approach finds a good balance between throughput, function latency, and efficiency in resource allocation.

C. Challenges of On-the-fly Knee Detection

Detecting the performance knee point of serverless computing systems at the edge is a non-trivial task. Not only that each function may have a different knee point, but the knee point for the same function can also vary under different workload

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.

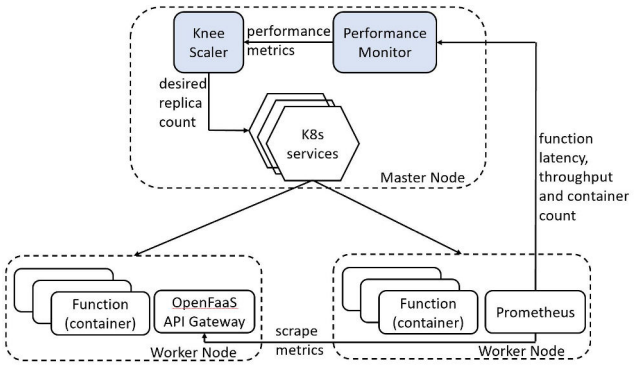


Fig. 5. KneeScale architecture

conditions. Existing knee detection algorithms such as Kneedle require prior performance data for resource allocations. However, data collected offline can be unreliable in the presence of previously unseen workloads (new functions, arrival rates, etc.) and changing hardware configurations. On the other hand, a naive online approach of sampling performance data for all possible resource allocations can be prohibitively expensive, especially under resource-budget constraints.

III. KneeScale Design

The key design goals of KneeScale are as follows:

1) Detect performance knee point for an ad-hoc function on-the-fly without prior performance data.
2) Minimize the overheads of knee detection.
3) Adapt to dynamic workload conditions.

A. Architecture

As shown in Fig. 5, the KneeScale architecture comprises two main components: Performance Monitor and Knee Scaler. The Performance Monitor periodically measures serverless function-related performance metrics from Prometheus[6], and Prometheus scrapes the metrics from OpenFaaS API Gateway. It also keeps track of the invocation error rate and CPU utilization of the function instances to decide when to trigger resource scaling. The invocation error rate is the number of failed function invocations per second. The Knee Scaler component is responsible for on-the-fly knee detection and resource scaling of the running functions. KneeScale can be implemented on top of any Kubernetes based architecture such as KubeEdge[7], K3S[8] etc. In this paper, KneeScale follows the K3S architecture where all the modules are running on the edge nodes. In contrast, KubeEdge architecture consists of both cloud and edge modules. The entire KneeScale codebase is written in Python and Bash with about 2,000 lines of code.

[6]https://prometheus.io/
[7]https://kubeedge.io/
[8]https://k3s.io/

TABLE I
Mathematical notations used in Algorithm 1

Notation	Description
b_l	The left boundary of the search space
b_r	The right boundary of the search space
r_{new}	The new replica count to be allocated
d	Search direction
C	Collection of explored replica counts
P	Collection of observed replica counts and corresponding performance scores
c_{last}	The last observed resource allocation
k	Detected Knee point
$PerfScore$	Use Eqn. 1 to calculate performance score

1) Performance Monitor: The performance monitor collects both the application-level performance metrics and resource utilization metrics at a sampling interval of 30 seconds. In particular, it measures the invocation error rate, throughput, and average function latency at the application level. In addition, it keeps track of the CPU utilization of function instances, and the cluster capacity per function. We define cluster capacity as the maximum number of replicas (function instances) that can be allocated to a function. As shown in Eqn. 4, the cluster capacity $r_{cluster}$ of a function depends on the resource requested (CPU, Memory) by a single function, and the resource available in the cluster of Edge nodes.

$$r_{cluster} = min(\frac{CPU_{avail}}{CPU\ req_f}, \frac{Memory_{avail}}{Memory\ req_f}) \quad (4)$$

2) Knee Scaler: The Knee Scaler component is invoked conservatively when the invocation error rate is greater than zero, indicating the need to scale-up, or when the rate of decrease in the CPU utilization is higher than a threshold, indicating the need to scale-down. It is designed to quickly detect the performance Knee point of a function using an efficient search algorithm (Algorithm 1). This algorithm explores a few steps of resource allocations, while iteratively applying the Kneedle algorithm after each data point until it converges to the final Knee. In order to handle dynamic workloads efficiently, the Knee Scaler reuses previously detected knee values under similar workload conditions. It uses the *kubectl scale* API in Kubernetes to allocate the required number of replicas for a given function.

B. Finding Knee Using Binary Search

Algorithm 1 shows how KneeScale detects the knee point on-the-fly without prior performance data. Table I describes the notations used in the algorithm. The function *FindKnee* takes two input parameters (b_l, b_r), which denote the left and the right boundary of the search space. The value of b_l is always set to the current replica count plus one. Whereas the value of b_r depends on the current invocation error rate, current replica count, and the cluster capacity $r_{cluster}$ given by Eqn. 4. The criteria for determining b_r is discussed in the next section. Since the Kneedle algorithm requires at least three data points to find a knee point, we collect the performance data for three resource allocations given by b_l, b_r, and $(b_l + b_r)/2$.

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Finding knee using binary search

```
1: function FindKnee (b_l, b_r) {
2: Initialization:
3: d ← LEFT ;
4: C ← b_l, b_r, (b_l+b_r)/2 ;
5: P ← (c, PerfScore(c)), ∀c ∈ C ;
6: k ← Kneedle(P);
7:
8: while b_l < b_r do
9:    c_last ← C[len(C) − 1];
10:   if d == LEFT then
11:      if k == None or k == c_last then
12:         r_new ← (c_last + b_l)/2;
13:      else
14:         d ← RIGHT ;
15:         b_l ← c_last ;
16:         r_new ← (b_l + b_r)/2 ;
17:      end if
18:   else
19:      if k == None or k == c_last then
20:         r_new ← (c_last + b_r)/2;
21:      else
22:         d ← LEFT ;
23:         b_r ← c_last ;
24:         r_new ← (b_l + b_r)/2 ;
25:      end if
26:   end if
27:   C.append(r_new) ;
28:   P.append(r_new, PerfScore(r_new)) ;
29:   k ← Kneedle(P);
30: end while
31: return knee;
    }
```

TABLE II
MATHEMATICAL NOTATIONS USED IN ALGORITHM 2

Notation	Description
$t_{current}$	Throughput measured at current sampling interval
K	Collection of observed throughput and knee values.
$t_{closest}$	Throughput in K that is closest to $t_{current}$
e	Invocation error rate
r	Current replica count
$r_{cluster}$	Cluster capacity calculated by Eqn. 4
θ_{thr}	Threshold for throughput change
θ_{CPU}	Threshold for CPU utilization change
θ_{err}	Threshold for error rate
$CPU_{previous}$	CPU utilization of previous sampling interval
$CPU_{current}$	CPU utilization of current sampling interval

List C is initialized with these resource allocations and list P is initialized with the corresponding performance scores, given by Eqn. 1. We apply the Kneedle algorithm to detect the initial knee point k on the collected data. We set the initial search direction to the left.[9] In each iteration, the search direction remains unchanged if the most recent data point c_{last} is detected to be the knee (line 11, 19) or knee is not found. Otherwise, we start exploring resource allocations in the opposite direction after resetting b_l to c_{last} if the new search direction is right (lines 13-15), and resetting b_r to c_{last} if the new search direction is left (lines 21-23). We sample the next resource allocation at the middle point between b_l and b_r. The search stops when b_l is greater than b_r.

[9]The initial search direction does not affect the time complexity and accuracy of the algorithm.

Algorithm 2: Handling dynamic workload

```
1: Initialization: diff_min ← +∞;
2:
3: function FindClosestThroughput(t_current, K) {
4: for t ∈ K do
5:    if |t_current − t| < diff_min then
6:       diff_min ← |t_current − t|;
7:       t_closest ← t;
8:    end if
9: end for
10: return t_closest;
    }
11:
12: function ScaleUp(t_current, e, r) {
13: t_closest ← FindClosestThroughput(t_current, K);
14: if |t_current − t_closest| < θ_thr then
15:    r ← K[t_closest];
16: else
17:    if e < θ_err then
18:       k ← FindKnee(r + 1, min(r + r_cluster/2, r_cluster));
19:    else
20:       k ← FindKnee(r + 1, r_cluster);
21:    end if
22:    K.append(t_current, k);
23:    r ← k;
24: end if
    }
25:
26: function ScaleDown(t_current) {
27: t_closest ← FindClosestThroughput(t_current, K);
28: r ← K[t_closest];
    }
29:
30: while t_current > 0 do
31:    if e > 0 then
32:       ScaleUp(t_current, e, r);
33:    end if
34:
35:    if (CPU_previous − CPU_current)/CPU_previous > θ_CPU then
36:       ScaleDown(t_current);
37:    end if
38: end while
```

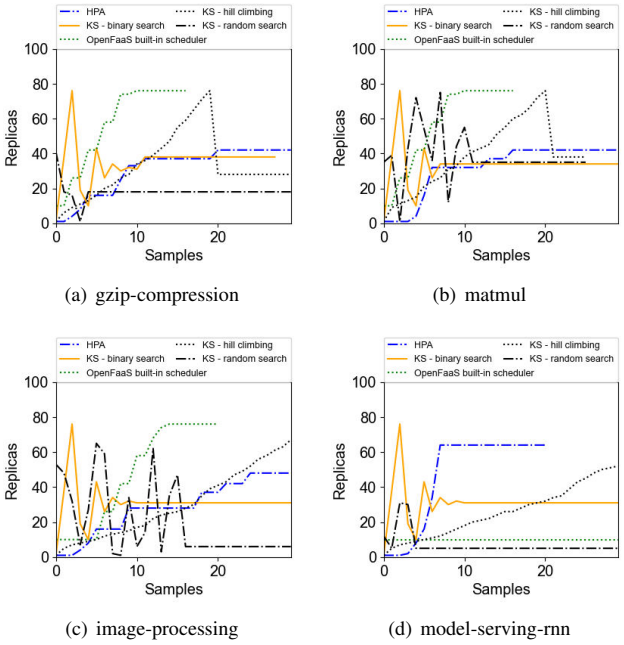
C. Handling Dynamic Workload

Algorithm 2 shows how KneeScale handles dynamic workloads. Without assuming any offline data collection, KneeScale keeps track of knee points detected for various workload conditions in the live system. Table II describes the notations used in this algorithm. In lines 31-33, the $ScaleUp$ action is conservatively triggered only if the invocation error rate is greater than 0. The $ScaleUp$ function, in lines 13-15, sets the replica count to be equal to a previously detected knee value corresponding to the closest throughput $t_{closest}$ observed in the past. This is done only if the difference between the current throughput $t_{current}$ and $t_{closest}$ is smaller than a threshold θ_{thr}. Otherwise, the search algorithm (Algorithm 1) is invoked

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.

TABLE III
WORKLOAD BENCHMARK

Name	Resource Demands	Description	Pod Requests	Cluster Capacity
float-operation	CPU, Memory	floating point arithmetic operations	200m, 400MiB	114
matmul	CPU, Memory	two N-dimensional square matrix	300m, 1000MiB	76
gzip-compression	Disk I/O	file compression using gzip	300m, 1000MiB	76
image-processing	CPU, Memory, Disk I/O, Network	image transformation tasks	300m, 1000MiB	76
model-serving-rnn	CPU, Memory, Disk I/O, Network	words generation model using a RNN	300m, 1000MiB	76



(a) gzip-compression (b) matmul

(c) image-processing (d) model-serving-rnn

Fig. 6. Resource allocation under a stationary workload of 20 concurrent clients. A resource-budget constraint of 1000 replicas is imposed on the cumulative resource allocation.

to find the new knee point. In lines 17-18, $FindKnee(b_l, b_r)$ function is invoked by extending the right boundary of the search space, b_r, to the smaller of two values $r + \frac{r_{cluster}}{2}$ and $r_{cluster}$. This conservative extension of the search space is done only when the invocation error rate is smaller than θ_{err}. Otherwise, b_r is extended all the way to the cluster capacity, $r_{cluster}$. In both cases, the left boundary of the search space, b_l, is set to $r + 1$. In lines 35-36, if the average CPU utilization of the function instances decreases by a rate greater than θ_{CPU}, the $ScaleDown$ action is triggered. Since majority of serverless functions are considered CPU intensive [7], so we choose CPU utilization as the appropriate metric to $ScaleDown$. The $ScaleDown$ function sets the replica count to be equal to the knee value corresponding to the closest throughput, $t_{closest}$. In our implementation, the threshold parameters θ_{thr}, and θ_{err} were set to be 2 and 1 respectively. θ_{CPU} was set to be 5%. These values provided a good tradeoff between performance and resource efficiency.

IV. EVALUATION

A. Experimental Testbed

We set up a prototype testbed running the KVM hypervisor to host four Ubuntu (v16.04) Virtual Machines (VMs). Each VM, representing an Edge node, was equipped with 8 CPU cores and 16 GB RAM. We deployed the OpenFaaS platform on these VMs, using the Docker container engine (v18.06.2-ce) and Kubernetes (v1.20.4) container orchestration system. The VMs were hosted on a bare-metal Intel Xeon E5-2630 v3 system equipped with 16 CPU cores, 252 GB RAM, 1.8 TB HDD, and 1000Mb/s Ethernet.

B. Workload Benchmark

We used FunctionBench [7], a suite of practical function workloads for FaaS platforms. As shown in Table III, we ran our experiments with five representative functions from different application scenarios. *Float-operation* and *matmul* represent CPU and memory bound workloads. *Gzip-compression* is a disk I/O-intensive workload. The *image-processing* function, performing image transformation tasks, represents a real-world application. *Model-serving-rnn* represents a workload associated with deep neural network (DNN) inference. Pod requests specify the minimum resource that will be reserved for the function instance to use. We used an HTTP workload generation tool, Hey[10], to produce function invocations.

C. Alternative Approaches

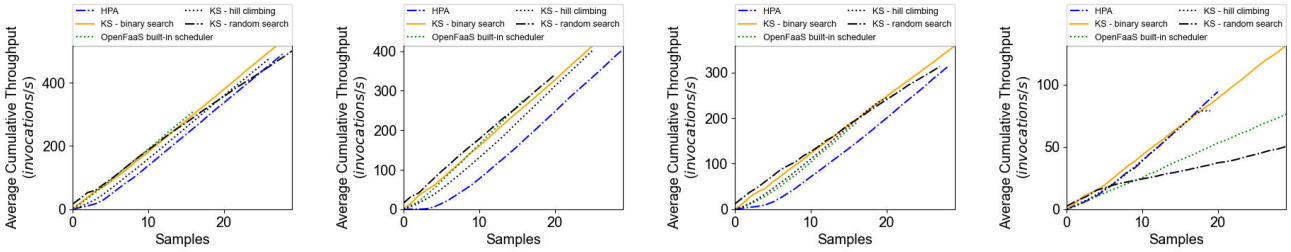
For performance comparison, we consider various alternative approaches for scaling serverless functions as follows:

OpenFaaS built-in scheduler: The default auto-scaling mechanism in OpenFaaS is triggered when the rate of successful invocations is greater than a threshold of 5 $invocations/sec$ for 10 seconds. We set OpenFaaS built-in scheduler's minimum container count, *com.openfaas.scale.min*, to be 10 so that it could achieve the best performance in our testbed. We observed that with the default value of *com.openfaas.scale.min=1*, the function throughput never exceeds 5 $invocations/sec$, and auto-scaling is not triggered at all. Another important parameter, *com.openfaas.scale.max*, was set to be equal to the cluster capacity ($r_{cluster}$) to facilitate fair comparison with competing techniques.

HPA: HPA scales the number of replicas based on the ratio between the current resource utilization and a user-defined

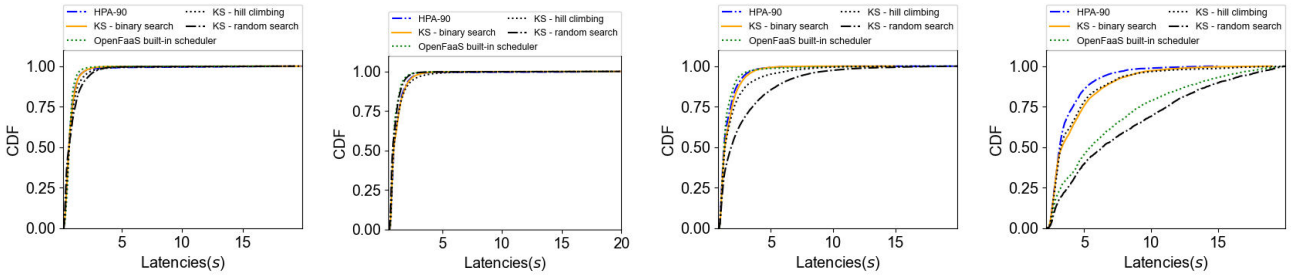
[10]https://github.com/rakyll/hey/

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.



(a) gzip-compression (b) matmul (c) image-processing (d) model-serving-rnn

Fig. 7. Average cumulative throughput under a stationary workload of 20 concurrent clients.



(a) gzip-compression (b) matmul (c) image-processing (d) model-serving-rnn

Fig. 8. Function latency distribution under a stationary workload of 20 concurrent clients.

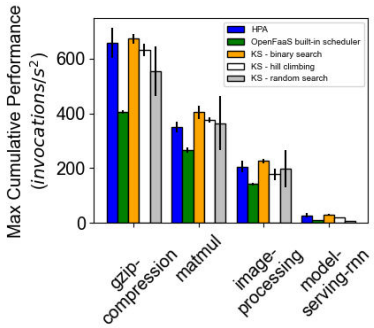


Fig. 9. Max cumulative performance under a stationary workload.

target utilization value according to Eqn. 3. We set the CPU utilization target to be 90% to represent a resource budget-friendly policy. A high CPU utilization target means that the resource scaling is done conservatively.

KneeScale - hill climbing: As an alternative to KneeScale's binary search algorithm, we implemented a hill climbing algorithm. Initially, we allocate one replica to a given function and incrementally increase the replica count until the cluster capacity is reached. Finally, the Kneedle algorithm is applied to detect the knee point. The number of replicas to be added at each step is inversely proportional to the degree of slope, θ, between the previous and the current sampling points. We calculate θ as the inverse tangent of the ratio of change in performance score and the change in replica counts between the sampling points. Since the knee is more likely to appear in the steep slope area, the increment in replica count is set to a small value when θ is large.

KneeScale - random search: In this approach, the replica count is randomly selected and the Kneedle algorithm is applied at each exploration step. The search stops when the knee values remain unchanged for three iterations.

D. Effectiveness of KneeScale

First, we evaluate KneeScale under a stationary workload of 20 concurrent clients and compare its performance with that of alternative approaches. A resource-budget constraint of 1000 replicas was imposed on the cumulative resource allocation. Each experiment was run for 15 minutes unless the cumulative replica budget was exceeded, in which case, function instances were killed to free the allocated resources.

KneeScale - binary search can find knee points accurately and efficiently. Fig. 6 shows that KneeScale takes 10 sampling intervals to find the knee point for the various workloads. Whereas KneeScale - hill climbing takes the longest time to find the Knee point due to its incremental search process. KneeScale - random search does not find a reliable knee point in most cases. OpenFaaS built-in scheduler shows extreme behaviors for different workloads. In three out of four workload cases, it rapidly scales the replica count and exceeds the resource budget on cumulative resource allocation before the experiment completes. Whereas in the case of *model-serving-rnn*, it does not trigger any resource scaling since the instantaneous throughput is less than 5 $invocations/sec$.

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.

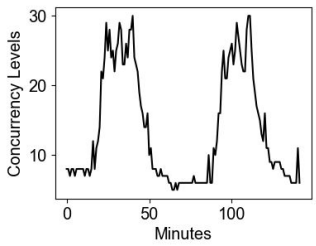
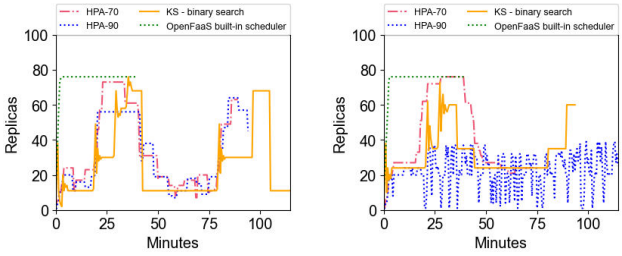
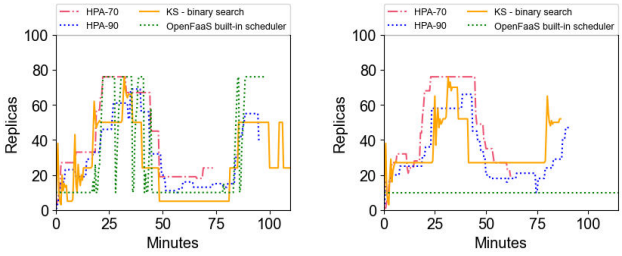


Fig. 10. Dynamic workload.



(a) gzip-compression (b) matmul



(c) image-processing (d) model-serving-rnn

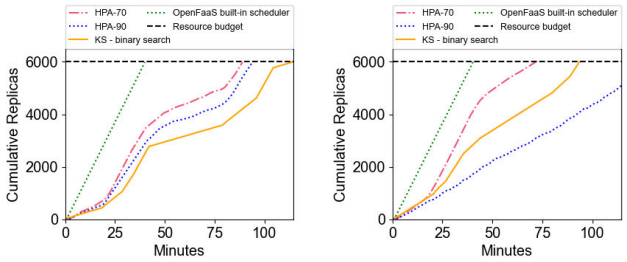
Fig. 11. Resource allocation under the dynamic workload.

Fig. 7 shows that KneeScale - binary search achieves higher cumulative throughput than competing approaches for each workload. As shown in Fig. 8, it achieves similar latencies compared with HPA and much lower latencies than OpenFaaS built-in scheduler and KneeScale - random search. OpenFaaS built-in scheduler performs very poorly in the case of *model-serving-rnn* since its auto-scaling is never triggered.

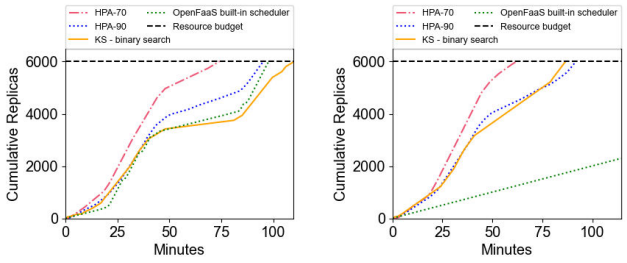
KneeScale - binary search achieves the highest cumulative performance score under resource budget. Fig. 9 shows that KneeScale - binary outperforms HPA, OpenFaaS built-in scheduler, KneeScale - hill climbing, and KneeScale - random search by up to 12%, 93%, 35%, and 56% respectively. The max cumulative performance is the cumulative performance measured at the end of experiments. Results shown are an average of three runs.

E. Performance under a Dynamic Workload

Fig. 10 shows the dynamic workload used in our evaluation. We randomly selected two days' workload pattern from the Azure Functions Trace 2019 [12] and sampled every 20 minutes to get about 2 hours of dynamic workload. The



(a) gzip-compression (b) matmul



(c) image-processing (d) model-serving-rnn

Fig. 12. Cumulative replicas under dynamic workload. A resource-budget constraint of 6000 replicas is imposed on the cumulative resource allocation.

workload data is normalized to have a maximum concurrency level of 30. A resource-budget constraint of 6000 replicas was imposed on the cumulative resource allocation. We compare the performance of KneeScale - binary search with that of OpenFaaS built-in scheduler, HPA with CPU utilization target 70% (HPA-70), and HPA with target 90% (HPA-90).

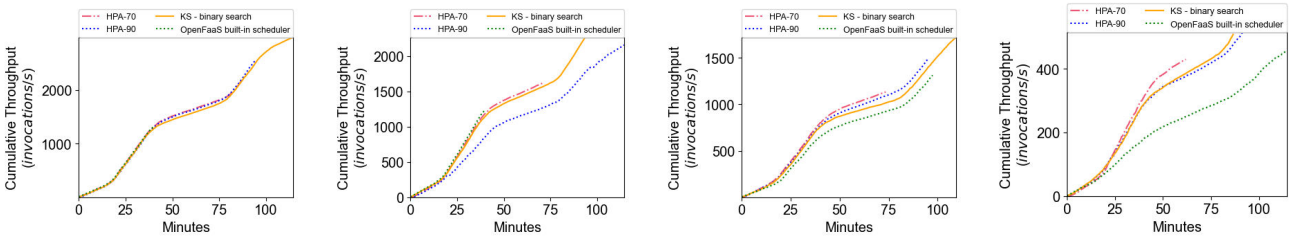
KneeScale can adapt quickly to workload changes. Fig. 11 shows the instantaneous replica count allocated to various functions under the dynamic workload. In the case of *model-serving-rnn* function, the replica count stays at the minimum value of 10 since the instantaneous throughput is below the OpenFaaS built-in scheduler's threshold of 5 $invocations/sec$. Fig. 12 illustrates how each approach reaches the resource budget. In the cases of *gzip-compression* and *matmul* functions, the OpenFaaS built-in scheduler aggressively scales the replica count and quickly exceeds the cumulative resource budget. And as expected, HPA-70 exceeds the cumulative resource budget more quickly than HPA-90.

Fig. 13 shows that KneeScale provides excellent cumulative throughput even under a dynamic workload. Fig. 14 shows that KneeScale provides similar function latencies compared with HPA-70 and HPA-90, and much lower latencies than OpenFaaS built-in scheduler. As shown in Fig. 15, KneeScale outperforms HPA-70, HPA-90, and OpenFaaS built-in scheduler in terms of cumulative performance score by up to 32%, 20%, and 106% respectively.

F. Cold Start Latency

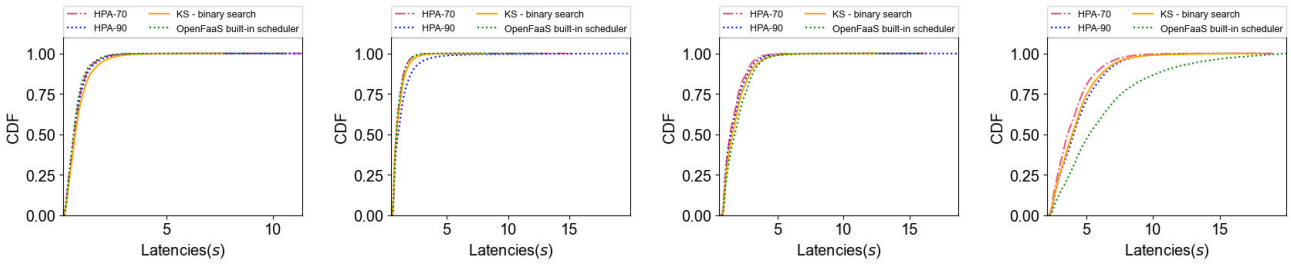
Cold start latency is the time spent on preparing a function instance. Its impact is most significant when a function is scaled from zero. Fig. 16 shows each function's first 200 invo-

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.



(a) gzip-compression (b) matmul (c) image-processing (d) model-serving-rnn

Fig. 13. Cumulative throughput under dynamic workload



(a) gzip-compression (b) matmul (c) image-processing (d) model-serving-rnn

Fig. 14. Latency distribution under dynamic workload

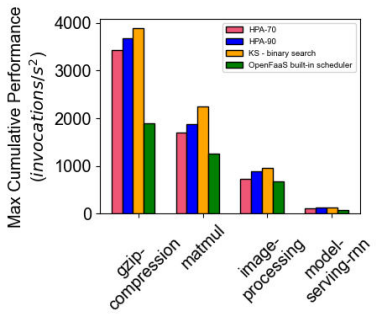


Fig. 15. Max cumulative performance under a dynamic workload

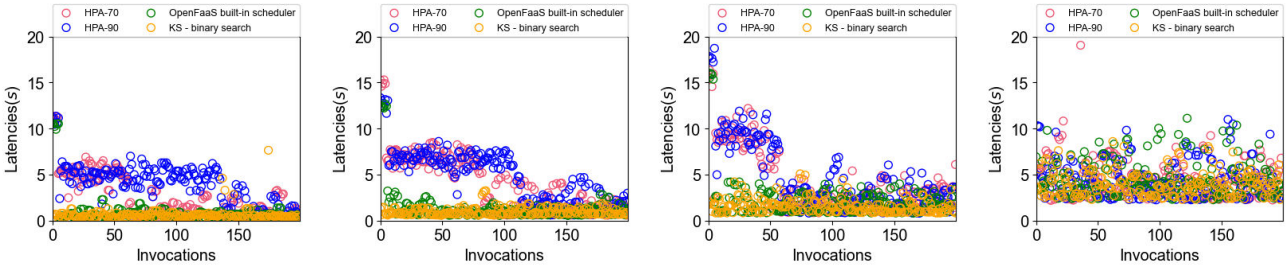
cation latencies under the dynamic workload. We observe that HPA and OpenFaaS built-in scheduler are heavily influenced by cold start. On the other hand, KneeScale is less affected. Since KneeScale quickly and aggressively scales the replica count during the initial exploration of the search space, the variance in the cold start latency associated with different replicas amortizes its overall influence. We also observe that the *image-processing* function experiences more severe cold-start latency than other functions. This is because this function has dependencies on more software libraries than others.

V. Related Work

Automatic resource scaling of containers and microservices has been studied in the context of traditional cloud computing platforms. Bella *et al.* [13] implemented an adaptive auto-scaler by combining Kubernetes vertical and horizontal auto-scaler. Rattihalli et al. [14] introduced an auto-scaler that can dynamically perform non-disruptive vertical scaling. Calheiros *et al.* [15] dynamically allocate the required resource in advance by feeding the latest observed loads' feedback to the autoregressive integrated moving average (ARIMA) model for prediction. Bhattacharjee *et al.* [16] presented a dynamic resource management framework for providing horizontal and vertical autoscaling of containers based on time series to forecast service workloads. Unlike these works, our paper focuses on quickly determining optimal resource allocation for serverless functions under resource-budget constraints of edge computing environment. Furthermore, our approach does not require extensive workload profiling and modeling.

HoseinyFarahabady *et al.* [17] utilized a model predictive control (MPC) framework to predict the upcoming workload and guarantee the QoS requirements of FaaS platforms. Shahrad *et al.* [12] used time series analysis to predict function invocations, and pre-warm the function instances to significantly reduce cold start latencies. SOCK [18] implemented a three-layer caching system for shortening package install and import time of serverless functions. SAND [19] reduced serverless function startup time by utilizing lightweight sandboxing mechanism for functions within the same application. FaaSNet [20] decentralized serverless container provisioning process across host VMs that are organized in function-based tree structures. Cicconetti *et al.* [21] extended serverless computing and developed an architecture to offload stateless tasks from user terminals to edge nodes. Cicconetti *et al.* [22] devise a resource-efficient edge computing scheme such that an intelligent IoT device user can well support its computation-

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.



(a) gzip-compression (b) matmul (c) image-processing (d) model-serving-rnn

Fig. 16. Invocation latencies including cold starts under a dynamic workload.

ally intensive task by proper task offloading across the local device, nearby helper device, and the edge cloud in proximity. These approaches are complementary to our work.

VI. Conclusion and Future Work

In this paper, we presented an adaptive auto-scaler, KneeScale, that efficiently scales serverless functions under resource budget constraints in edge systems. We developed a novel approach that dynamically adjusts the number of function instances to quickly reach a Knee point at which the relative cost to increase resource allocation is no longer worth the corresponding performance benefit. We implemented KneeScale on a Kubernetes-based OpenFaaS platform and evaluated its effectiveness using a representative FaaS benchmark, FunctionBench. Experimental results show that KneeScale outperforms existing auto-scaling techniques under resource budget constraints for both stationary and dynamic workloads. In our future work, we will evaluate and further enhance KneeScale for large scale and heterogeneous settings. We will also study the impact of resource contention and performance interference between serverless functions running on a shared FaaS platform.

VII. Acknowledgement

The research is supported by NSF CNS 1911012 grant.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, pp. 637–646, 2016.

[2] M. Satyanarayanan, "The emergence of edge computing," *Computer*, pp. 30–39, 2017.

[3] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[4] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, and S. Dustdar, "Towards a serverless platform for edge ai," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[5] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, pp. 64–71, 2017.

[6] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *ACM International Conference on Internet of Things Design and Implementation*, 2019, p. 225–236.

[7] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 502–504.

[8] D. T. Nguyen, L. B. Le, and V. Bhargava, "Price-based resource allocation for edge computing: A market equilibrium approach," *IEEE Transactions on Cloud Computing*, vol. 9, no. 1, pp. 302–317, 2021.

[9] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 261–2615.

[10] A. H. Mahmud, Y. He, and S. Ren, "Bats: Budget-constrained autoscaling for cloud performance optimization," in *IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 232–241.

[11] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "kneedle" in a haystack: Detecting knee points in system behavior," in *31st International Conference on Distributed Computing Systems Workshops*, 2011, pp. 166–171.

[12] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX Annual Technical Conference (ATC 20)*, 2020, pp. 205–218.

[13] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[14] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 33–40.

[15] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' qos," *IEEE Transactions on Cloud Computing*, 2015.

[16] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2019, pp. 23–33.

[17] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari, "A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1442–1455, 2018.

[18] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference (ATC 18)*, 2018, pp. 57–70.

[19] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *USENIX Annual Technical Conference (ATC 18)*, 2018, pp. 923–935.

[20] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *USENIX Annual Technical Conference (ATC 21)*, 2021, pp. 443–457.

[21] C. Cicconetti, M. Conti, and A. Passarella, "Architecture and performance evaluation of distributed computation offloading in edge computing," *Simulation Modelling Practice and Theory*, 2020.

[22] X. Chen, Q. Shi, L. Yang, and J. Xu, "Thriftyedge: Resource-efficient edge computing for intelligent iot applications," *IEEE Network*, pp. 61–65, 2018.

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on August 06,2022 at 20:38:41 UTC from IEEE Xplore. Restrictions apply.