Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds

Viyom Mittal*, Shixiong Qi*, Ratnadeep Bhattacharya+, Xiaosu Lyu+, Junfeng Li§, Sameer G Kulkarni#, Dan Li§, Jinho Hwang⋆, K. K. Ramakrishnan*, Timothy Wood+
*University of California, Riverside, +George Washington University, §Tsinghua University, #Indian Institute of Technology, Gandhinagar, ⋆Facebook Inc.

Abstract

Serverless computing platforms simplify development, deployment, and automated management of modular software functions. However, existing serverless platforms typically assume an over-provisioned cloud, making them a poor fit for Edge Computing environments where resources are scarce. In this paper we propose a redesigned serverless platform that comprehensively tackles the key challenges for serverless functions in a resource constrained Edge Cloud.

Our Mu platform cleanly integrates the core resource management components of a serverless platform: autoscaling, load balancing, and placement. Each worker node in Mu transparently propagates metrics such as service rate and queue length in response headers, feeding this information to the load balancing system so that it can better route requests, and to our autoscaler to anticipate workload fluctuations and proactively meet SLOs. Data from the Autoscaler is then used by the placement engine to account for heterogeneity and fairness across competing functions, ensuring overall resource efficiency, and minimizing resource fragmentation. We implement our design as a set of extensions to the Knative serverless platform and demonstrate its improvements in terms of resource efficiency, fairness, and response time.

Evaluating Mu, shows that it improves fairness by more than 2× over the default Kubernetes placement engine, improves 99th percentile response times by 62% through better load balancing, reduces SLO violations and resource consumption by pro-active and precise autoscaling. Mu reduces the average number of pods required by more than ∼15% for a set of real Azure workloads.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3487014

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Networks** → **Network resources allocation**.

Keywords

Edge clouds, serverless, resource management

ACM Reference Format:
V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G Kulkarni, D. Li, J. Hwang, K.K. Ramakrishnan, T. Wood. 2021. Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds . In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3487014

1 Introduction

Serverless platforms have gained popularity because they allow easy deployment of services in a highly scalable and cost-effective manner [26]. This should make serverless a perfect fit for Edge Computing, where tiny data centers are distributed throughout a geographic area, allowing users to access low latency services rather than relying on a distant, centralized cloud. Each edge data center will be highly resource-constrained. Thus, the autoscaling "from zero" capabilities that allow serverless platforms to use no resources if there are no requests arriving, are highly desirable. Similarly, the fast instantiation of new functions ought to be a boon for Edge deployments with high user movement in and out of the area, as in a mobile edge cloud.

Unfortunately, current serverless platforms assume access to a more-or-less infinitely scalable cloud and pay little attention to resource wastage. When deployed at the edge, these characteristics lead to unacceptable performance such as high tail latency and unfair resource allocations under multi-tenancy because of the limited resources. As a result, current designs of serverless platforms are not yet a viable option for Edge environments.

In this work, we propose Mu, a resource management framework for serverless at the Edge that extends the open-source Knative platform. Mu is composed of the following components which tackle a number of key challenges:

Autoscaler (§3.3): Mu leverages machine learning models to forecast incoming workloads and proactively allocate function containers based on the combination of demand and service level objectives (SLO). Our evaluation shows that by closely tracking the requirements of each function, Mu reduces resource use by more than 15% and better avoids underprovisioning and high response times compared to existing autoscaling algorithms.

Load Balancer (§3.4): Mu's load balancer carefully assigns requests based on up-to-date statistics of backend load. This information is efficiently propagated through the system using 'piggybacking' of key measures to reduce monitoring overheads. We demonstrate that Mu's precise load balancing improves the 99th percentile response time by up to 62%, when nodes are heterogeneous or workloads are bursty—both common occurrences at the Edge.

Placement Engine (§3.5): Mu carefully decides where to place each function container to avoid fragmentation and ensure fairness in a multi-tenant environment. Unlike centralized clouds where resources are seemingly endless, Edge clouds may frequently run at close to capacity. We show our placement heuristic achieves comparable performance to optimization techniques at a much lower computation cost and provides up to a 2× improvement in fairness among tenants.

2 Background and Related Work

Edge Clouds: The rise of 5G has led to a network provider-centric Edge vision where cellular base stations or central offices provide a (relatively) small number of servers or racks of servers which can provide services for nearby users [27]. In this work, we consider an Edge cloud environment where limited compute resources–likely on the scale of a single rack or less–are being made available to service requests for function execution from nearby network users. In this scenario, the Edge cloud needs to support a variety of different functions (since different users may have different needs), and it must manage its resources efficiently and fairly to simultaneously support all users.

Serverless Platforms: Cloud platforms provide compute and storage services at large scale and low cost through economies of scale and effective multiplexing. Serverless computing takes this multiplexing and scalability to the next level by allowing providers to commit just the required amount of resources to a particular application (as many instances as necessary, but only when needed) and utilize the resources for just the time needed to execute an invoked function [25].

In this work, we focus on Knative[4] and how it can be deployed in an Edge environment. In a Knative cluster, developers can write functions in a variety of languages, which are then deployed into backend worker pods. Each worker pod consists of two containers namely the 'queue proxy' and the 'function' itself. The 'queue proxy' is responsible for queuing incoming requests and forwarding them to the 'function' container for execution. Requests enter the system via an 'Ingress Gateway' that maintains metrics about active backend pods and routes requests to them. The platform is managed by an Autoscaler that dynamically adjusts the number of worker pods, a placement engine that places new pods, and a load balancer in the gateway that directs requests. In this work, we comprehensively consider all three of these aspects to enhance Knative's architecture and better adapt it to an Edge cloud environment.

Serverless Autoscaling: Knative enables auto-scaling by using the Knative Pod Autoscaler (KPA) [5]. The Autoscaler monitors the traffic flow to the function, and scales replicas up or down based on user-configured targets for metrics, such as concurrency and request per second (RPS). For Kubernetes, the Horizontal Pod Autoscaler (HPA) [3] periodically adjusts the number of replicas to match the observed average resource (*i.e.,* CPU, memory) utilization to the user-specified target. However, this dependency on user inputs severely limits these autoscaling methods since: i) the user is unaware of actual resource usage and other runtime features of functions: it is hard for the user to choose the proper autoscaling metric and set the right target to meet their demands, which makes it prone to misconfiguration; ii) the single metric-based autoscaling approaches by themselves are insufficient and not comprehensive enough to properly satisfy SLOs and reduce the cloud usage cost. The number of instances provisioned is often either too large and wastes resources or is too small and violates the SLO.

Load-balancing: There is a wide range of work on load balancing for web [11] and cloud applications [15]. Our load balancing algorithm is inspired by the "join the shortest queue" (JSQ) approach [13], which has been shown to be nearly optimal, but only in an environment with homogeneous servers and workloads. JSQ also requires accurate information on queue length, and we show how we can efficiently acquire this through piggybacked metrics. We also draw inspiration from HALO [9], which focuses on heterogeneous environments, and we further show that serverless load balancers need to take special care when new pods are frequently added or removed.

Placement and Scheduling: Borg [2] and Kubernetes [6] typically employ some form of heuristic-driven bin packing for scheduling pods in a data center. It first prioritizes all the eligible nodes by using several alternate scoring policies. One is to score the nodes by the amount of remaining resources, thus favoring the least loaded node. Another is to balance the allocation of CPU and memory resources, by looking at the difference between the available CPU and memory capacity fractions available and placing the pod in a server that has the best balance. Their primary focus is on reducing stranded

resources (i.e., fragmentation), but do not explore the issues of fairness among multiple contending functions demanding resources in a resource-constrained environment.

When considering fairness, max-min fairness [7] seeks to provide allocations fairly among contending sources of demand (functions in this context), and is relevant when the demand exceeds capacity. However, max-min fairness focuses on a single dimension for the resource demand (e.g., CPU requirement). When considering multiple dimensions (CPU, memory, network), approaches such as Dominant Resource Fairness (DRF) [10] help in fairly allocating resources, considering max-min allocations for each resource. We seek to adapt this approach in the serverless context. DRF performs its resource allocation based on the aggregated resource capacity in the cluster, but does not take into account resource fragmentation on nodes. The consequence is potential inefficiencies in using the resources available in the Edge cloud. Our placement engine seeks to improve the allocation beyond that achieved by DRF, balancing fairness, efficiency, and resource fragmentation.

Measurement of Serverless Platforms: There have been a number of measurement-driven efforts to understand the behavior of serverless platforms. Measurements on commercial serverless cloud platforms (AWS Lambda, Microsoft Azure and Google Cloud) [18, 28] while others [17, 29] show that it is important to consider throughput, scalability, memory footprint, etc. There have also been a number of measurement-based evaluations of open-source serverless frameworks such as Knative, OpenFaaS, OpenWhisk, Kubeless, etc. [16, 20, 21], which provide some preliminary understanding of the performance characteristics and sensitivity to configuration parameters of these platforms. We use these efforts to enhance our understanding of these open-source serverless frameworks, as we develop Mu.

3 System Design

Fig. 1 shows the architecture of Mu, which builds on the Knative, Kubernetes, and Istio tools. Mu extends the Istio Ingress Gateway to efficiently collect metrics that are "piggybacked" onto response headers by the Queue Proxy containers (§3.1) for timely feedback of critical information without resorting to periodic sampling. Mu's Autoscaler predicts upcoming load changes (§3.2) and scales function replicas up or down to meet the service level agreement (SLA) of users (§3.3). All incoming traffic goes through the Ingress Gateway's Load Balancer, which factors in the gathered metrics to evenly load the function containers (§3.4). The Placement Engine must pack pods to suitable nodes to reduce resource fragmentation, improve the efficiency and ensure fairness between functions when Edge resources are constrained (§3.5).

Table 1: Summary of main notations

Notations	Definitions
T_c	time interval of capacity estimation
C_d	response count during T_c in pod
C_{dc}	count of responses whose confidence flag is 1 during T_c in pod
C_r	ongoing request count in user container
C_{cur}	current request count in cluster
C_{new}	new request count during scaling epoch
C_{pro}	processed request count during scaling epoch
$Queue$	queue size of the queue proxy
IR_{cur}	current incoming rate
IR_{pre}	predicted incoming rate
N_{cur}	current pod number
N_{des}	desired pod number to meet SLO
R_d	departure rate of pod
R_{sd}	Smoothed departure rate of pod
Cap_e	estimated pod capacity
$Ratio_c$	confidence ratio
RT_{avg}	average responding time
QT_{avg}	average queuing time
ET_{avg}	average execution time

3.1 Metrics

A serverless platform relies on metrics such as the load on function containers to guide resource management. Some of these metrics, such as the load on each User Container are maintained by Knative in the Queue Proxy containers. The queue proxy is a sidecar container allocated for each user function container that buffers incoming requests. The queue proxy maintains a queue to throttle requests to the function container based on the container concurrency configuration parameter set by the administrator. To avoid high overhead when the number of function pods is large, Knative's Autoscaler periodically samples a subset of queue proxies to gather metrics, but we have found that this can lead to having an inaccurate view of important data.

To accurately monitor the status of function pods with low overhead, Mu extends the queue proxy at each pod to collect metrics about function processing and 'piggyback' those metrics in the response header to the ingress gateway to provide timely information. This allows the ingress gateway to maintain detailed per-pod statistics to guide its load balancing algorithm, while exporting aggregated information to the Autoscaler via its Internal Metric Server. The queue proxy gathers the following metrics:

Queue Length: The queue length metric shows the instantaneous size of the queue in the queue proxy, measured when the request is removed from the head of the queue to be executed. The load balancer uses this metric to determine the relative load across a group of worker pods and the Autoscaler uses aggregated queue length information to

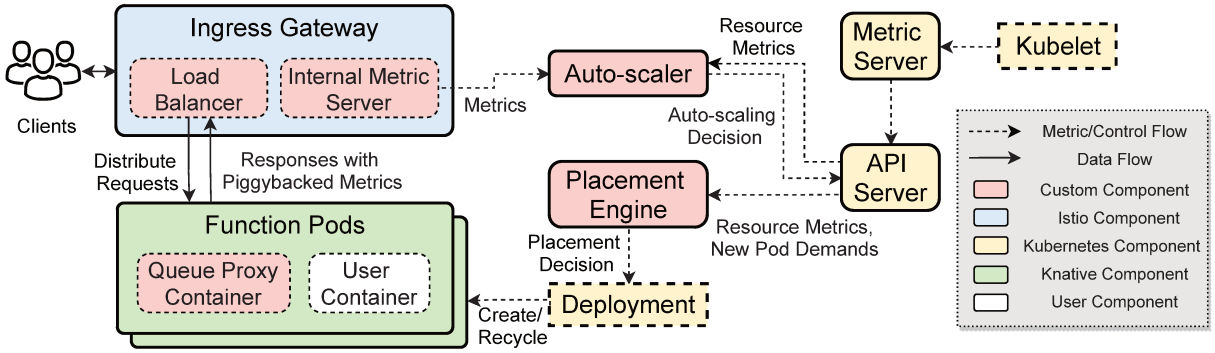


Figure 1: Mu Overview.

modulate the scaling decision and avoid potential Service Level Objective (SLO) misses.

Average Execution Time: The queue proxy measures the execution time of each request, which is the time between forwarding the request to the user container and receiving its response back. The average execution time ET_{avg} is the Exponentially Weighted Moving Average (EWMA) of the measured execution time. The function pod piggybacks this metric to the ingress gateway, which passes on the average execution time across all function pods to the Autoscaler.

Departure Rate and Confidence Ratio: Ideally, the queue proxy would report the pod's maximum service capacity, but this metric can be difficult to estimate, particularly if the incoming rate is low. Instead, Mu has the queue proxy report its departure rate as well as a "confidence ratio" that indicates how fully loaded the server is. The calculation of these metrics is detailed in Algorithm 1. The queue proxy maintains a confidence flag for each request, revealing whether the user container is fully utilized (i.e., continuously has a queue of waiting requests) when processing this request. The default value of the confidence flag is 0. When a request arrives at the queue proxy, it sets the confidence flag to 1 if the queue size is larger than 0 (line 1-6). During the processing of a particular request in the user container, the queue proxy resets the confidence flag of that request to 0 if the queue size drops to 0 (line 12-16), implying that the user container is underloaded (departure rate is smaller than capacity).

Rather than choosing a fixed time interval for measuring estimated capacity, we adapt it based on the time scale of the request execution. The time interval for updating the estimated capacity is T_c. If the average execution time ET_{avg} increases, the time interval T_c increases accordingly (line 17-19), so as to collect sufficient responses in T_c for a more accurate departure rate estimate. When the average execution time ET_{avg} reduces, the time interval T_c drops (line 20-22), so as to update the departure rate quickly. When there are no requests in time interval T_c, then T_c will be reduced by half to react quickly for future requests, until T_c is back to its default value of 1 sec. (line 24-28).

Every time interval T_c, the queue proxy computes the departure rate and confidence ratio. The departure rate is then smoothed using EWMA (line 30-35). The confidence ratio is the ratio of the requests whose confidence flag is 1 to the total requests in the time interval T_c (line 36-40). If the user container is fully utilized in T_c, the confidence ratio is 1 and the actual capacity will be close to the departure rate. Both of these values are propagated to the load balancer, enabling it to make an estimate of a pod's maximum service capacity, *i.e.*, Cap_e and share with the Autoscaler.

3.2 Incoming Rate Prediction

Serverless platforms based on the Kubernetes architecture can take ~2-5 seconds to instantiate a pod. To avoid queuing and potentially missing the SLO for a request while waiting for a pod to startup, it is desirable to predict the incoming rate of requests. Thus, the Autoscaler can make proactive pod provisioning decisions. However, the prediction mechanism must be efficient and robust, since there may be a wide range of functions being deployed. Moreover, with many workloads having vastly different request rates, model parameters cannot be hand-tuned.

In Mu, we propose a lightweight regression-based incremental learning mechanism (Algorithm 2). The model uses linear regression to train and predict the workload in an online manner, eliminating the need to profile each function in advance. For incremental or online training of the models, we use Stochastic Gradient Descent. We propose a best-fit search prediction algorithm, where we simultaneously run many lightweight instances of the regression model with different hyperparameters, and dynamically select the model with minimum running error. Two hyperparameters are crucial in determining model performance:

Input window size: Each model takes a window of the previous n incoming rates as input and predicts the incoming rate for the next epoch. Different values of n are required to

Algorithm 1 Capacity Estimation

On receiving a request in queue proxy:
if $Queue > 0$ **then**
    $request.confidence = 1$ ▷ $request.confidence$ is the confidence flag of this request
else
    $request.confidence = 0$

On arrival of a response from user container:
$C_d = C_d + 1$ ▷ update the response count
if $request.confidence == 1$ **then**
    $C_{dc} = C_{dc} + 1$
if $Queue == 0$ **then**
    for every request in the user container **do**
        $request.confidence = 0$
if $10 \cdot ET_{avg} > 2 \cdot T_c$ **then** ▷ increase the time interval
    $T_c = max\{10 \cdot ET_{avg}, 10\}$
if $10 \cdot ET_{avg} < T_c/2$ **then** ▷ decrease the time interval
    $T_c = min\{10 \cdot ET_{avg}, 0.1\}$

At every time interval T_c:
if $C_d == 0$ and $C_r == 0$ **then** ▷ the pod is idle
    $R_{sd} = 0, ET_{avg} = 0$
    if $T_c > 1$ **then**
        $T_c = max\{T_c/2, 1\}$ ▷ decrease the time interval
else
    $R_d = C_d/T_c$ ▷ the departure rate of this time interval
    if $R_{sd} == 0$ **then** ▷ update smoothed departure rate
        $R_{sd} = R_d$
    else
        $R_{sd} = \alpha \cdot R_{sd} + (1 - \alpha) \cdot R_d$ ▷ EWMA
    if $C_d > 0$ **then** ▷ update the confidence ratio
        $Ratio_c = C_{dc}/C_d$
    else
        $Ratio_c = 0$
$C_d = 0, C_{dc} = 0$ ▷ reset the counters

capture the invocation pattern of heterogeneous workloads.

Learning rate: In the gradient descent approach, the learning rate determines the magnitude by which the weights of the model are changed on each update. Again, this can be different for different workloads.

To select the best window size and learning rate, the predictor runs different instances of the regression model while varying their values. We maintain an EWMA of the error for each model. On each invocation, Mu chooses the model with the least running error for recent predictions. But, the best model may still not provide a good prediction due to random incoming rates with a new pattern not seen in the recent past. For this, we include a naive predictor that assumes the predicted value of the incoming rate is the same as the current value. If its error is less than the best-selected model, we use the current incoming rate.

Algorithm 2 Prediction logic

On every autoscaling algorithm invocation:
$best\_model = naive$
$min\_error_{running} = naive.error_{running}$
for m in models **do**
    $m.error_{running} = \alpha * m.error_{running} + (1 - \alpha) * (IncomingRate - m.predictedIR)$
    UpdateWeights(m.weights)
    if $m.error_{running} < min\_error_{running}$ **then**
        $best\_model = m$
        $min\_error_{running} = m.error_{running}$
return $best\_model.predictIR$

Predictor Accuracy: For validating the predictor we select all the workloads with more than 100K invocations for the first day, from the Azure Functions dataset [26]. The traces in the Azure dataset contained invocations per minute for each function. We select the top 555 workloads with at least 100K total invocations, and predict the number of invocations for the next minute. We took 50 prediction models with a combination of 5 different window sizes (10, 50, 100, 500, 1000) and 10 different learning rates (10^{-1} to 10^{-10}) for each function. Based on experiments across these functions, the value of EWMA coefficient was selected as 0.99 as it yielded better results than other values ranging from 0.5 to 0.999. For each workload, the average absolute error was calculated for the predictor and for the naive approach (which takes the current requests per minute as the prediction, like the default Knative which includes no prediction logic). The %age reduction in error for all the workloads is shown in Fig. 2. For 64 out of 555 workloads, the predictor performs slightly worse (-1.53% average degradation) than the naive approach, due to the random invocation pattern. Similarly, for 22 workloads, we see no improvement. For the remaining 469 workloads, the incoming rate is predicted fairly accurately, with the absolute error reduced 19.01% on average. The predictor executes ~15.6K instructions for each prediction, taking ~100 μ secs. For 200 workload streams, the predictor takes ~20 ms every 2 seconds, an acceptably small 1% overhead.

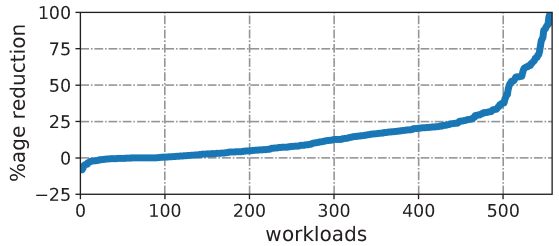


Figure 2: %age Reduction in absolute error for workloads with > 100K invocations

3.3 Autoscaler

A critical aspect in managing overall resources is to have an efficient autoscaling component to allocate and deallocate resources for functions on demand in a timely manner. Knative provides two Autoscalers: *RPS*, where the scaling is based on incoming request rate; and *Concurrency*, where the Autoscaler decides based on the number of simultaneous requests being processed. The existing Knative approaches are agnostic of the SLO for the function and require hand-tuning parameters for best results. Further, we have found that resources need to be provisioned as a function of both the incoming request rate and the queue length to ensure that SLOs are met by factoring in the average request execution time. In addition to these, the Autoscaler we design for Mu seeks to proactively scale up or down function pods based on the upcoming arrival rate of requests, to accommodate the delay involved in instantiating a function pod.

SLO Aware Autoscaling: The Mu Autoscaler computes the desired pod number for a function based on the request arrival rate, the execution time, and the current queue of requests, while ensuring we meet the SLO. This number *i.e.,* N_{des}, is determined every epoch. We choose an epoch size of 2 seconds, matching the typical period in between Knative autoscaling decisions. We first calculate the number of pods based on the incoming rate:

$$N_{IR} = \lceil Max\{IR_{Cur}, IR_{Pred}\}/Cap_e \rceil \quad (1)$$

The maximum of IR_{Cur} and IR_{Pred} ensures that we provision pods according to the predicted arrival rate only if its value is higher than the current incoming rate. Otherwise, we use the actual incoming rate to ensure that the system is not under-provisioned. We then factor the existing queue of requests built up in the queue proxy, ensuring they are served within the SLO as:

$$N_{Queue} = \lceil QT_{avg}/(Cap_e * (SLO - ET_{avg})) \rceil \quad (2)$$

Combining Eq. 1 and 2, the desired pod count is:

$$N_{des} = N_{IR} + N_{Queue} \quad (3)$$

To ensure system stability, we set limits on the number of pods provisioned in a single epoch, so as to minimize over-correction during transients. For $N_{des} > N_{cur}$, we provision at most twice the current N_{cur}:

$$N_{des} = Min\{N_{des}, 2 * N_{cur}\} \quad (4)$$

When $N_{des} < N_{cur}$, *i.e.,* for scaling down the number of pods, we introduce a hold-down time (*i.e.,* Grace flag) for the autoscaler to scale down the requested pods to smooth the scaling operation. The Grace flag specifies the number of 2-second epochs for which N_{des} should be less than N_{cur} before Mu implements the downscaling. When the Knative autoscaler, *i.e.,* Concurrency or RPS, operates normally (in its "stable mode"), there is no smoothing. But, under overload (load is twice what the currently active pods can handle), the Knative autoscaler switches to "panic mode", under which the downscaling is not performed. The panic mode is meant to avoid rapid changes by the autoscaler under overload, and lasts for 6 seconds by default, as long as a request for a larger number of pods is made during this 6 second interval.

Table 2: Autoscaling system configuration

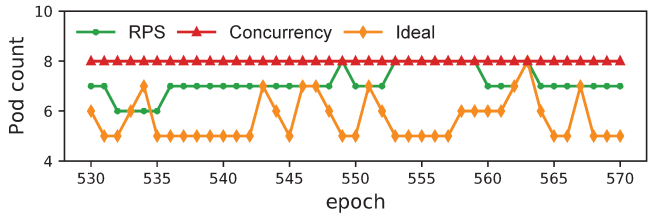
Parameter/ System Configuration	Values
Number of nodes	2 (1 Master, 1 Worker)
CPU	Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz
Memory	160GB ECC Memory
Container Concurrency	10
Average Function Execution Time	100 milliseconds
Target SLO (for Mu Autoscaler)	1 second

Autoscaler Evaluation: We compare Mu's custom autoscaling algorithm against the Concurrency and RPS based algorithms provided by Knative. For all the autoscaling experiments, we use Azure function traces [26]. To select a representative workload, we pick the function with the median value of the %age reduction in absolute prediction error from Figure 2 ensuring that we do not select workloads favorable to our predictor. The median value also helps to obtain an estimate of the performance improvement to be expected on an average by using our predictor. To emphasize system dynamics, we scaled the workload to a second time scale (i.e., each minute from the original trace lasts 1 second). The average request rate is 339 requests per second. The system configuration and parameter values used for the experiment are given in Table 2. We set the container concurrency value to 10 (10 requests can be simultaneously handled by a single container), and the average request execution time is 100 ms. Each function pod's average service capacity is then 100 responses/sec. The maximum size of the queue at the Queue Proxy is set to 100.

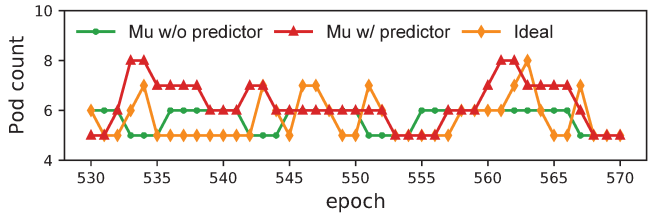
Table 3 shows the results comparing Knative's existing autoscaling algorithms with that of Mu with and without the use of the arrival rate predictor. The number of requests completed within the SLO of 1 sec for Mu's Autoscaler is better than Concurrency-based autoscaling but slightly worse than RPS (2%). However, Mu's autoscaling uses fewer pods on average and dramatically fewer maximum number of pods (less than 10 compared to 30 to 70 with Knative's default autoscaling algorithms). The maximum number of pods to be provisioned is a concern because that eventually will limit a

Table 3: Comparison of autoscaling algorithms

	RPS	Concurrency	Mu w/o predictor	Mu
Total requests	488,602	488,602	488,602	488,602
Request completed within SLO	475,884	453,867	463,367	466,885
Average pod count	5.77	6.83	4.01	4.18
Max pod count	30	73	8	9



(a) RPS vs. Concurrency vs. Ideal



(b) Mu w/ predictor vs. Mu w/o predictor vs. Ideal

Figure 3: Pod count, different Autoscaling algorithms

cloud site's overall utility. We see further improvement in the number of requests completed within SLO Mu when helped by the predictor, as it helps in anticipating the arrival rate, enabling the Autoscaler to provision the pods in advance, to meet the incoming load. Nonetheless, the maximum number of pods still remains less than 10.

In Fig. 3, we show the actual pod count for the different autoscaling algorithms and compare it against the ideal pod count, observed for a representative period of the experiment (from epoch 530 to 570, each epoch is 2-seconds). We calculate the ideal pod count based on the known incoming and serving rates. For RPS and Concurrency-based scaling, the system is always over-provisioned. The pod count for Mu's Autoscaler remains close to the ideal pod count, and is helped by the predictor to anticipate the incoming requests and provision pods earlier when there is an increase in the request rate, thereby reducing the SLO misses. The predictor helps Mu provision additional pods, but still is significantly lower than the overprovisioning of the default Autoscalers.

3.4 Load Balancer

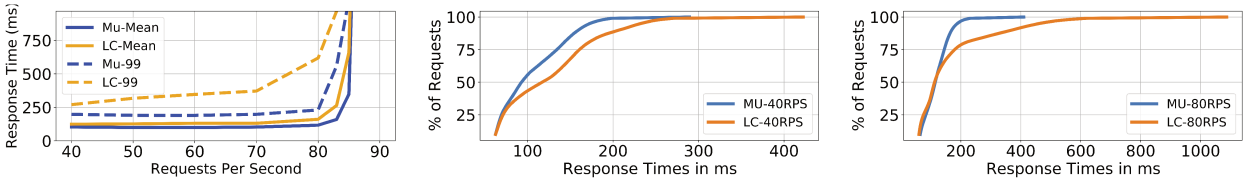
The load balancer resides in the ingress gateway and routes client requests across all pods to maximize utilization and ensure that no pod is overloaded. Load balancing requests in an Edge cloud serverless platform faces two primary challenges: resource heterogeneity and system dynamics. Unfortunately, the load balancers employed in existing serverless platforms fail to accurately account for either of these issues.

The first issue arises because an Edge cloud may be composed of a variety of hardware types, especially in "fog computing" environments where the cloud is composed of a mix of infrastructure nodes and resources pooled from mobile devices [8, 24]. Even if an Edge cloud is located in a more standardized environment such as a 5G base station, it is increasingly common for resource-constrained environments to use heterogeneity (e.g., ARM's big.LITTLE architecture which combines high and low performance CPU cores on a single chip or accelerators like programmable NICs, GPUs, etc) to provide flexible trade-offs between performance, power utilization, and overall cost. Further, even if all hardware is identical, the dense consolidation of an Edge cloud may result in interference and resource contention which may cause some pods to execute functions more slowly than others, especially in the face of diverse workloads (IoT, ML, CDN, cellular functions, etc.). This heterogeneity can impact Knative's "Least Connection" load balancer, which attempts to track the queue length at each backend pod by comparing the number of requests sent versus responses received. When deciding which pod to select for a new request, it only considers the queue length estimate, which we show can lead to poor decisions when backends have varying service capacities. Further, if the serverless platform runs multiple load balancer gateways, this queue length estimate may be inaccurate as it ignores queueing caused by other gateways.

The dynamic nature of Mu's autoscaling capabilities further complicates load balancing. The load balancer must be aware of newly added pods, and it should direct the appropriate amount of load to them – avoiding "herding" problems where too much load is shifted to a newly started pod, but also avoiding underloading it. In effect, a newly started pod represents a different type of heterogeneity since it will begin with an empty queue of requests, while other nodes may already have nearly full queues if scaling occurred due to approaching overload. The Knative Least Connection load balancer employs a power of two random choices algorithm [19] which means that it randomly selects two backends and then picks whichever has the smaller number of active connections. While this provides greater scalability as the cluster size increases, it comes at the expense of lower accuracy, which may not be the appropriate trade-off for a resource-constrained Edge cloud. As a result, a new pod in Knative has at most a $2/N$ chance of being selected in a cluster of N servers. Our evaluation shows that this limits Knative's ability to quickly shift load to new pods, leaving the system in an overloaded state despite idle resources.

3.4.1 Load Balancer Algorithm A smart load balancer should recognize both differences in service capacity and pod queue length to appropriately route requests across new and existing pods. In Mu, we implement a new load balancer that leverages the metrics gathered by function pods to make better decisions based on up-to-date information.

Estimating Pod Metrics: Most prior work on load balancing assumes access to service rate information for each backend;



(a) 99%ile latency & mean latency (b) Response time CDF with 40 RPS (c) Response time CDF with 80 RPS

Figure 4: Mu's load balancer vs. Least Connection load balancer: Mu reduces tail latency across all load levels.

further, such rates are assumed to be static. In a serverless environment, the large number of different functions makes it impractical to assume all functions have been previously profiled to determine service rates, particularly for an edge cloud with hardware heterogeneity. A backend's capacity may also change over time, particularly in a densely packed Edge environment where resource contention can occur. Thus Mu must be able to accurately and dynamically determine both the service capacity of each pod, and its current load level. As described previously, Mu's Queue Proxies piggyback key metrics as part of each response header, providing the load balancer up-to-date information about each pod. However, further processing is required in order to produce accurate estimates of pod capacity and load.

When a function is deployed for the very first time, Mu has no information about its execution cost. However, once requests start to be processed, it quickly builds a model of each pod's service capacity as follows. On each response from backend pod i, we compare the piggybacked confidence, $pigRatio_i$, and departure rate, $pigR_i$, against previously saved values for the pod, $savedRatio_i$ and $savedR_i$. If $pigRatio_i \geq savedRatio_i$ or $pigR_i \geq savedR_i$, then we update pod i's capacity estimate $Cap_i = pigR/pigRatio$ and update the saved confidence ratio and departure rate values to be equal to the piggybacked values. If the prior conditions are not met, then the saved values are not updated. A newly started pod with no data uses the maximum values seen by another pod of the same function type as a default.

The intuition behind this algorithm is that if the Confidence Ratio reported by the queue proxy is low, that indicates that the backend has had a low or empty queue, and thus it is safe to aggressively predict that the real service capacity is much higher than the departure rate. When the Confidence becomes 1, it means that the backend is consistently seeing a queue, which means its departure rate will be close to the actual maximum service capacity of the pod (otherwise the queue would have drained). Tracking a saved Confidence Ratio and Departure Rate ensures that the Load Balancer does not lose information over time, assuming that the service capacity drops simply because the arrival rate falls.

To track the load on each pod, the load balancer can use the piggybacked queue length values. Using the piggybacked

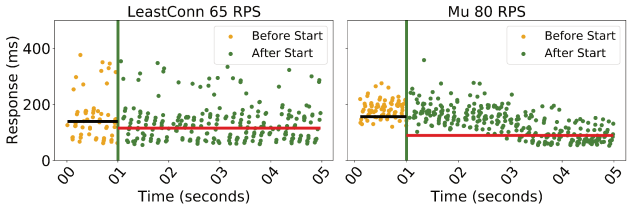


Figure 5: Mu takes advantage of a newly added pod more quickly: shifting load, improving both mean (horizontal lines) and variance in response time more

value instead of a local counter at the load balancer ensures that the metrics are accurate even if there are multiple load balancers in the cluster. These metrics are aggregated and exposed to the Autoscaler, which uses them to determine when to scale up as described in the prior section. Further, we use the service capacity information to guide downscaling, causing the system to prefer to shut down slower pods when they are no longer needed. This not only helps ensure the downscaling won't cause unexpected overload, but also naturally makes the algorithm pick a pod with fewer requests in its queue, allowing its resources to be freed sooner.

Selecting Pods: Using the above information about pod capacity and queue length, the Mu Load Balancer calculates the estimated response time, R_i, that a new request would see on each pod i in the cluster:

$$R_i = \frac{Q_i + 1}{Cap_i} \tag{5}$$

where Cap_i is the estimated service capacity and Q_i is the estimated queue length–we add one to account for the cost of processing the new request. The load balancer then selects the pod with the minimum R_i. This algorithm attempts to minimize the response times seen by all requests, and will naturally forward more requests to pods with higher service capacities or lower queue lengths (such as a newly started pod). It should be noted that since some functions may support concurrent processing of requests, this may be an inaccurate estimate of the request's actual response time; nevertheless, it represents both the service capacity and load on a function well, so we find it gives a good signal about what pod will be the best choice for the request.

Load Balancer Performance: To demonstrate the importance of using both queue length and service capacity to guide decision making, we run an experiment with two "fast" and two "slow" pods. To get a sense of what a reasonable level of heterogeneity is, we compared the service time of a CPU bound prime number calculating function on a high-performance AMD EPYC Rome 64 core Processor (3 GHz) and an Intel Xeon CPU X5650 running in a low power mode at 1.6GHz. The AMD system is roughly two times faster than the Intel one depending on the prime function parameter. Thus, in our experiments we set faster pods to be twice as fast as the slower ones; we use a function with a service time of about 100ms on a fast pod. We measure the response time when adjusting the client send rate. Fig. 4(a) shows how the mean and 99%ile latency change with a rising workload. We observe that Mu can support a higher request rate with lower response times, and that it particularly improves tail latency due to better accounting for the relative speeds of the different pods: at 80RPS, the 99%ile decreases from 618ms to 230ms, leading to a much narrower response time distribution as shown in Fig. 4(b) and 4(c). To understand why Mu provides such a benefit, we examine the queue lengths of different pod types in each algorithm. Despite attempting to pick servers that have fewer active connections, Least Connection still tends to cause a higher queue build up on slow pods compared to fast pods. In contrast, Mu correctly recognizes it can safely queue more load on the faster pods, while still maintaining a low overall execution time.

Load Balancer Agility: We next demonstrate Mu's ability to more quickly adapt by leveraging its detailed pod information. We consider a scenario where four pods (two fast, two slow) are on the verge of overload. Fig. 5 shows the response time for requests immediately before and after a new fast pod begins (marked by the vertical line and color change). While the pod addition does help reduce the mean response time of Least Connection, it still shows a wide spread of response times due to the poor balancing of the load. In contrast, Mu provides a much tighter distribution of response times, and shows a clear downward trend as new requests are directed away from the heavily loaded pods and towards the new pod. Note that in order to cause Mu to hit the same overload point as Least Connection in this experiment we need to send it a higher workload (80RPS vs 65RPS), so Mu is not only handling a larger volume of requests, but it is able to do so while significantly reducing both tail and mean latency (horizontal lines).

3.5 Placement Engine

When functions have to be instantiated, the typical approach in Kubernetes and Knative is to use a bin-packing algorithm to schedule (place) the function pods on available servers. We develop an efficient and fair algorithm for a placement engine to pack function pods to suitable nodes while reducing resource fragmentation. Since an edge cloud may have limited resources, it is important to fairly allocate resources among contending functions, while considering their demand for resources across multiple dimensions (CPU, memory, *etc*). We adapt the notion of dominant resource fairness (DRF) to arrive at a fair placement strategy [10].

3.5.1 Optimization Model and Metrics We first model the function placement task as an Integer Linear Program (ILP) formulation. Let N be a set of nodes; Let J be a set of resources; each resource $j \in J$ has its capacity $c_{n,j}$ on node n. Let F be a set of functions, each function $f \in F$ has its desired pod count p_f. Each function's pods demand $d_{f,j} \geq 0$ on resource j, and $w_{f,n}$ denotes the number of function f's pods placed at node n. We define two objective functions for two alternate models, `ILP0` and `ILP1`, both of which have the same constraints, as below:

$$\begin{aligned} \texttt{ILP0}: \max \quad & \textstyle\sum_{n\in N}\sum_{f\in F} w_{f,n} \\ \texttt{ILP1}: \max \quad & \textstyle\sum_{n\in N}\sum_{f\in F} \frac{1}{D_f} \times \sum_{i=1}^{w_{f,n}} \frac{1}{i} \\ \text{s. t.} \quad & \textstyle\sum_{f\in F} d_{f,j} \cdot w_{f,n} \leq c_{n,j}, \forall n \in N, \forall j \in J \\ & 0 \leq \textstyle\sum_{n\in N} w_{f,n} \leq p_f, \forall n \in N, \forall f \in F \end{aligned} \tag{6}$$

The goal of `ILP0` is to maximize the total number of pods and thus the overall resource efficiency among a given set of nodes, while `ILP1` maximizes both the resource efficiency and fairness across different functions. `ILP1` assigns a weight $w_{f,n}$ by decreasing the reward for placing a function's as the number of pods increases for that function. Thus, the reward for placing more pods for a single function is less than the reward of evenly placing the pods of several different functions. In addition, to ensure the function with a small resource demand will not be starved by functions with a large resource demand, `ILP1` weights the $w_{f,n}$ by the dominant resource share of function f ($D_f = \max_{j\in J} \frac{d_{f,j}\times p_f}{\sum_{n\in N} c_{n,j}}$). Placing a large function pod receives a smaller reward, which guarantees fairness between large functions and small functions. Both `ILP0` and `ILP1` are constrained by the node's resource capacity and each function's requested pod count.

Quantifying Fairness & Efficiency: We quantify fairness of the allocation of resources to each function by the placement engine based on the principle of Max-Min fairness [7]. With

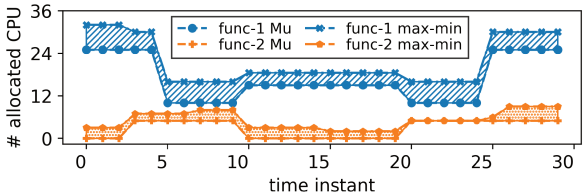


Figure 6: An example on integrating the degree of CPU unfairness over 30 intervals

Algorithm 3 Placement Algorithm

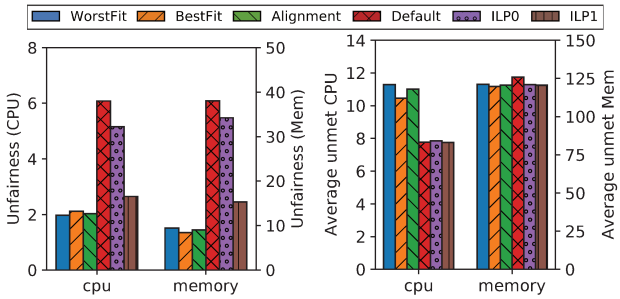
while $F \neq \emptyset$ **do**
  $D_f \leftarrow \{\max_{j \in J}(R_{f,j}/\sum_{n \in N} c_{n,j}) | \forall f \in F\}$
  Pick the function $f' \in F$ with minimum D_f
  $S_n \leftarrow \{score_{n,f'} | \forall n \in N, d_{f',j}$ fits in $a_{n,j}\}$
  if $\forall n \in N, S_n = \phi$ **then**
    $F \leftarrow F - f'$
  else
    Place f' to node $k \leftarrow \max_{n \in N} S_n$
    $R_{f',j} \leftarrow R_{f',j} + d_{f',j}$; $a_{k,j} \leftarrow a_{k,j} - d_{f',j}$; $p_{f'} \leftarrow p_{f'} - 1$
    if $p_{f'} = 0$ **then**
      $F \leftarrow F - f'$

varying demand from contending functions, it is important to evaluate fairness also as a function of time. Similarly, we evaluate the efficiency of the placement engine, by comparing its allocation with an allocation that maximizes the resource efficiency, as specified by the greedy algorithm `ILP0` above. We evaluate the degree of unfairness U_j and the inefficiency I_j of a placement algorithm on resource j integrated over a period of time, T, in Eq 7.

$$U_j = \frac{\sum_{f \in F_t} \sum_{t \in T} |R_{f,j,t} - M_{f,j,t}|}{\sum_{t \in T} |F_t|}$$
$$I_j = \frac{\sum_{t \in T} |\sum_{f \in F_t} d_{f,j,t} - \sum_{f \in F_t} R_{f,j,t}|}{\sum_{t \in T} |F_t|} \quad (7)$$

where $M_{f,j,t}$ indicates the max-min allocation on resource j to function f at time instant t, and $R_{f,j,t}$ is the the amount of resource j allocated to each function f by the placement algorithm at time t. Ideally, a placement algorithm could directly meet the max-min allocation, but in practice this is not possible because it only considers a single resource and assumes resources can be allocated without any fragmentation. Fig. 6 shows an example on quantifying the degree of unfairness, by comparing the allocations to two functions over time with regard to their ideal max-min allocation. We integrate the absolute difference between $R_{f,j,t}$ and $M_{f,j,t}$ over a period of time T, $(\sum_{t \in T} |R_{f,j,t} - M_{f,j,t}|)$. The degree of unfairness can then be calculated by averaging the cumulative area of all the functions over the entire time period T. Since the max-min allocation achieves the optimal fairness for each resource [7], a larger U_j indicates more unfair allocation on resource j. We do the same for the degree of inefficiency, integrated over time to get the overall degree of inefficiency, I_j, of the placement algorithm compared to the placement with the greedy algorithm `ILP0`.

3.5.2 Heuristic algorithms As the ILP model is NP-Hard, we also design a heuristic algorithm to solve the pod placement. We break the placement algorithm into two modules: *(i)* the pod selection module considering Dominant Resource Fairness (DRF), to decide which function pod is selected to be placed next; *(ii)* the node selection, which chooses the node



(a) Unfairness (b) Inefficiency

Figure 7: Fairness and efficiency comparison

to place the function pod at, based on a scoring function, so as to reduce the resource fragmentation, while minimizing unfairness. For a given scaling decision from Autoscaler, the placement engine invokes these two modules iteratively until function set F is empty (Algorithm 3).

Module 1: Pod selection. We calculate dominant share (D_f) of every function and pick the function f' with the minimum D_f. If multiple functions have the same minimum D_f, the function with the minimum sum of the resource demands (*i.e.*, $MIN(\sum_{j \in J} d_{f,j})$) is selected.

Module 2: Node selection. We evaluate a number of existing scoring functions for selecting the node, *e.g.* `Alignment` [12], `WorstFit` [22], and `BestFit` [22].

`Alignment` uses $score_{n,f} = \sum_{j \in J} \frac{a_{n,j}}{c_{n,j}} \times \frac{d_{f,j}}{c_{n,j}}$ to score the node n for the selected function f, $a_{n,j}$ is the remaining resources on node n. `Alignment` picks the node with the highest amount of remaining resources. `WorstFit` chooses the node with the highest value of: $\sum_{j \in J} \frac{a_{n,j} - d_{f,j}}{c_{n,j}}$. Thus, `WorstFit` seeks to pack the function into the node with the least amount of resources available that can accommodate this function's demand. `BestFit` chooses the node that has the highest value of: $\sum_{j \in J} \frac{a_{n,j} - d_{f,j}}{a_{n,j}}$. `BestFit` seeks to pack the function into the node with the most amount of resources left after accommodating this function's demand. All nodes $n \in N$ that have enough resources to fit the selected function f', are then scored using a scoring function. If f' has no node with a valid score, it is removed from F. Else, the node with maximum score is picked for f'. After placing f', we update the resource allocation and capacity. If the total pods demanded for function f' is met, it is removed from the set F.

3.5.3 Placement engine evaluation We simulate and compare our different DRF-based heuristic approaches (`WorstFit`, `BestFit`, and `Alignment`), the default Kubernetes scheduling heuristic (`Default`), and the two ILP models, by setting up 500 randomly generated placement test cases. We consider a simulated cluster of 40 nodes and 300 functions. A workload generator is used to randomize functions and nodes in each test. In the configuration used, 90% of the functions

Table 4: Experiment configuration

Parameter/Specification	Values	
Invocation Range	W-1	41-230 rps
	W-2	69-182 rps
Average invocations	W-1	154 rps
	W-2	146 rps
Container Concurrency	4	
Grace Flag (Mu only)	16	
Execution time	500ms	
Maximum pod capacity	48	
CPU and Mem. per pod	7 cores, 30GB	
Target	RPS	8
	CC	40
SLO	5 seconds	

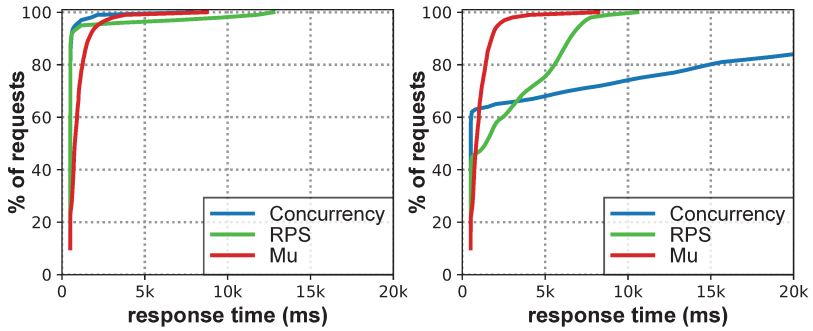


Figure 8: Response time CDF for 3 frameworks for Workload 1 (left); Workload 2 (right; only partial CDF for Concurrency)

require less than 400MB memory while 10% of functions require 500~2000MB memory. The CPU demand of functions ranges from 1~8 cores. Each function requests 1~16 pods. To ensure demand exceeds the resources in the cluster, the total CPU capacity of the cluster is set to 80% of the total CPU demand and the total memory capacity is set to 60% of the total memory demand. We use Gurobi [14] to solve the ILP models, adjusting the accuracy and termination criterion to keep computation time manageable.

Fig. 7(a) shows the fairness (as defined in Eq. 7) of the allocation decisions for the CPU and memory. The 3 DRF heuristic-based algorithms (which are all close to each other) achieve 2× better fairness than the ILP0, which does not consider fairness in its optimization. `ILP1` considers the fairness in the formulation, and achieves better fairness than the `ILP0`. However, with the accuracy and termination criteria we used with the solver, `ILP1` achieves better efficiency but poorer fairness than the DRF heuristic algorithms. The worst algorithm in terms of fairness is the Kubernetes `Default` approach. Comparing CPU efficiency (Fig. 7(b)), the DRF heuristics have an unmet CPU demand of ~10 cores on average, which is slightly worse than the ILP models. The Kubernetes `Default` is also better, with an average unmet CPU demand of ~8 cores. All the alternatives have similar memory efficiency, resulting in an average of ~ 130MB unmet memory demand. Thus, the DRF heuristic approaches strike a good balance of having very good fairness, and are close to the best case efficiency of the Kubernetes `Default` algorithm (which however ignores fairness).

In `Mu`'s deployment, the placement engine is executed once every epoch (2 seconds, driven by the autoscaler). In terms of computation time, the `Default` Kubernetes approach takes ~200 ms. The DRF heuristics are also fast, taking ~500 ms to determine the placement of 300 functions among 40 nodes. However, the ILP models, depending on the accuracy desired, take much more time (> 2 seconds on a server-class machine) and are impractical for real-time placement use. The DRF approaches, on the other hand, are feasible for deployment.

4 Overall `Mu` Implementation & Evaluation

We now integrate all the components of `Mu`, and evaluate it for a few large scale workloads. We compare `Mu` with the Knative default approaches.

Implementation Details and Testbed Setup: `Mu`'s implementation extends multiple components in the Knative ecosystem, including the Knative Queue-Proxy, Istio Gateway, Knative Autoscaler, and Kubernetes Scheduler (placement engine). We base our code on Kubernetes v1.17.0, Istio's Envoy Proxy v1.16.0, and Knative v0.13.0. Our extensions comprise ~1,000 lines of code added for the Autoscaler, ~500 lines for the load balancer and metrics server, ~200 lines for the queue-proxy, and ~800 lines for the placement engine. We evaluate the serverless platforms on the Cloudlab testbed [23] consisting of one master and ten worker nodes, each of them equipped with Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz (40 hyperthreads per host) and 160 GB ECC memory running Ubuntu 18.04.1 LTS. We do not add any extra pod heterogeneity in this experiment other than the natural fluctuations found on CloudLab.

4.1 Overall `Mu` Performance

To comprehensively evaluate `Mu`, we use the workloads received by functions in the Azure dataset [26]. We select 2 workloads with variable invocation patterns from the top 10 workloads sorted by maximum number of invocations for the first day in the dataset. We scale down these workloads by dividing the number of invocations by 100 for the experiment, treating each minute of the original trace as one second to add dynamics. The scaled down workload and the configuration of the serverless environment are in Table. 4. With the combined Autoscaler, Load Balancer, and Placement Engine, `Mu` achieves better overall performance for requests to serverless functions, even if the system is subject to a significantly heavy load, and more fairly allocates the limited edge cloud resources among the competing functions.

Latency and Fairness: The CDF of the response times for each workload and approach is shown in Fig. 8. `Mu` has good control over the response times and limits the tail latency that

Table 5: Comparing Mu with the standard Knative build

		Average response time (ms)	99% response time (ms)	# 503 errors /total requests	Requests served within SLO	Requested Pods		Active Pods	
						Max	Avg.	Max	Avg.
Mu	**Workload-1**	952	3805	6779 / 221026	213437 (96.5%)	33	20.5	24	20.0
	Workload-2	1020	4073	5211 / 209905	203622 (97.0%)	26	19.4	24	18.9
RPS	**Workload-1**	880	11757	0 / 221026	213089 (96.4%)	38	29.3	26	25.1
	Workload-2	2605	8808	0 / 209905	158511 (75.5%)	32	27.9	22	20.9
Concurrency	**Workload-1**	588	2141	0 / 221026	220144 (99.6%)	141	41.4	40	24.5
	Workload-2	7765	49526	0 / 209905	142774 (68.0%)	136	62.3	24	21.2

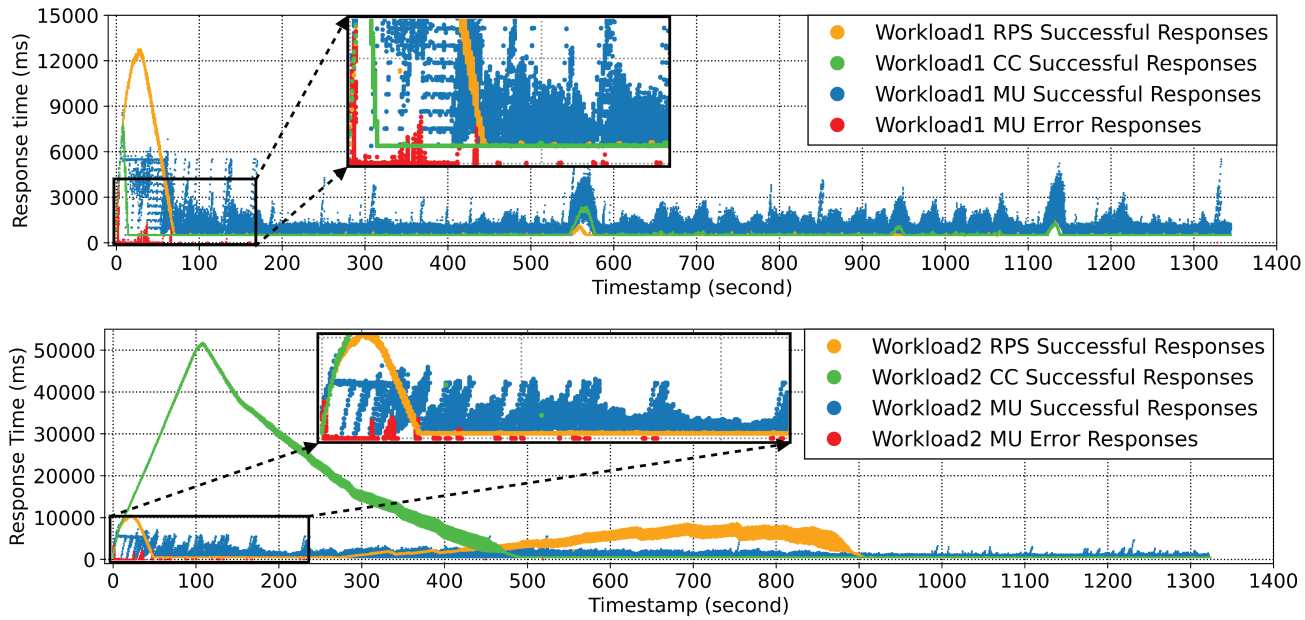


Figure 9: Time series of Response Time for Mu, RPS, and Concurrency (Top: Workload 1; Bottom: Workload 2)

exceeds the specified SLO of 5 seconds for both workloads. For Workload 2, Mu provides a substantially tighter response time distribution than RPS or Concurrency. As shown in Table 5, the 99% response time for the two workloads are both below the 5 second SLO for Mu. Examining the response time distribution (Fig. 8), and the average and 99%iles (Table 5) and the time series of the response times (Fig. 9(a), 9(b)), we see that Mu maintains fairness between the workloads for the entire length of the experiment.

In contrast, the standard Knative approaches result in much larger response time tails, and both unfairly treat one of the workloads. For Workload 1, both RPS and Concurrency (CC) achieve a lower average response time (except RPS has a relatively large number of requests experiencing high delays at the start of the workload, resulting in its 99%ile being higher). However, for Workload 2, both RPS and CC behave quite poorly at different periods of the workload execution, as seen from the time series (Fig. 9(b)), with 25-32% of requests violating the SLO. Workload 2 sees an unacceptably large 99% latency with CC as seen in Table 5. Since Mu is conservative in its pod allocation for both Workload 1 and 2, it sees a slightly higher average response time for Workload 1 than RPS and CC, but better for Workload 2 than RPS and CC. The 99%ile for Mu is clearly better than the two alternatives.

Pod Allocations: We use the term "requested pod count" for all alternatives. It comes directly from the autoscalers for RPS/CC. For Mu, the placement engine uses the pod count determined by the autoscaler and accounts for fairness and overall system capacity to determine Mu's "requested pod count". On average RPS and Concurrency use 18% and 17% more pods than Mu. Mu tends to request fewer pods since its goal is proactively provision enough pods to meet SLOs, with the predictor helping to anticipate the future workload. In Fig. 10(a), Mu is aware of both the workloads and fairly determines the requested pod count. RPS and concurrency on the other hand (Fig. 10(b) and 10(c)), run an autoscaler for each workload, without coordination between decisions for each workload. Thus, the requested pod counts may not only be unattainable, but can also be unfair. This is most evident for concurrency based autoscaling in Fig. 10(c) where the pod requests for individual workloads exceed 100, whereas the system capacity only allows provisioning 48 pods totally.

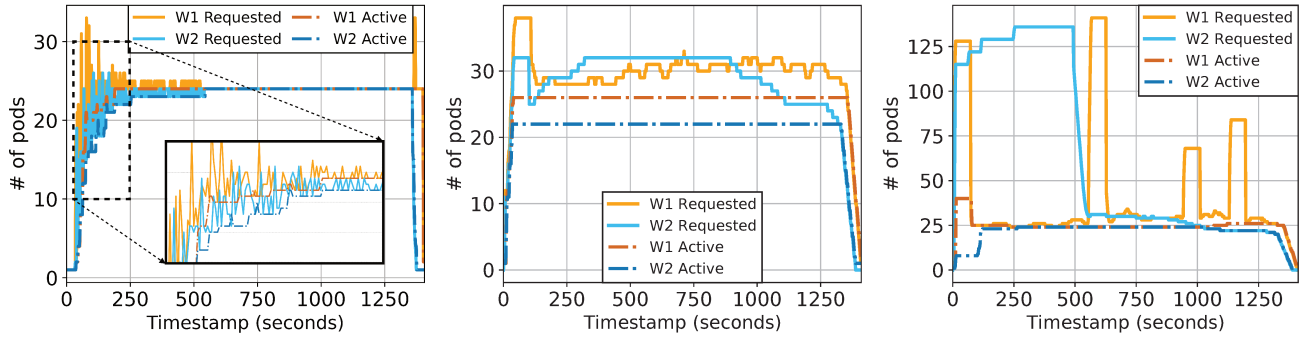


Figure 10: Time series of Pod counts for Mu (left), RPS (middle), and Concurrency (right)

SLO Performance: Overall, Mu provides a significant increase in the total number of requests served within the SLO (96.8%) compared to the RPS scaling policy (86.2%) and Concurrency scaling policy (84.2%), as shown in Table 5. Mu uses SLO-aware admission control and returns 503 errors [1] for requests which it will not be able to serve within the SLO based on current queue lengths. This avoids the build up of a large queue with the arrival of a burst of requests. RPS and concurrency do not factor SLO into account, so when bursts occur, requests are buffered in the activator, and the queueing results in a large number of SLO misses. Throughout the experiment, Mu has relatively uniform response times, increasing only during bursts, when the system is underprovisioned (e.g., first 200 seconds of the experiment when we have to scale up from zero to a large number (~ 20) of pods). On the other hand, Concurrency and RPS see persistent queuing for long periods (> 400 seconds) and the response time grows substantially more than the desired target SLO of 5 seconds. There is also significant unfairness for Workload 1 vs. Workload 2 as seen in Fig. 9(a), 9(b).

As shown in Fig. 9, Mu returns 503 errors (indicated by red dots). Our view is that by having these failures (and potentially having those requests be retransmitted) impacts a relatively small number (<5%) of requests, which is better than building up a large queue resulting in very long latencies for a large number of requests (25-30%, as seen for RPS and Concurrency) and likely to more seriously impact user Quality of Experience (QoE). These 503 errors are well correlated with the occurrence of bursts when resources are not yet provisioned by Kubernetes. This is mitigated somewhat by the predictor and proactive autoscaling. In fact, most of the 503 errors occur when the burst arrives at the beginning when the predictor has not yet learned the characteristics of the workload. Additionally, even though Mu's autoscaler requests allocation of a larger number of pods, Kubernetes can take a large amount of time to provision these pods, starting from an initial zero-scale system (as seen in the difference between pods being requested and active in the first 200 seconds for Mu (see Fig. 10(a)).

5 Conclusion

Existing platforms such as Knative suffer from their ad-hoc design that leverages existing frameworks such as Kubernetes without substantial customization for serverless use cases (*e.g.* reuse the Kubernetes scheduling algorithm and metrics subsystems). Further, today's serverless platforms are designed for large scale cloud environments with abundant resources, without meeting the strict requirements of agility and efficiency needed for Edge cloud environments.

Our work on Mu demonstrates the importance of carefully integrating the key resource management components that comprise a serverless platform: autoscaling, load balancing, and placement engine. Without the careful communication of key metrics and the predictive capabilities that Mu provides, a serverless platform lacks the information needed to make timely and accurate decisions. By accounting for SLOs, execution cost, and up-to-date load metrics across both the load balancer and Autoscaler, Mu can improve performance while making judicious use of scarce resources. When resources become overcommitted, Mu's placement engine ensures greedy functions cannot unfairly starve others. We have demonstrated that by coordinating these components and customizing them for Edge environments, Mu 1) uses resources more efficiently, reducing the average number of pods required by more than 15% for a set of real Azure workloads; 2) provides a tighter response time distribution with a 2× or more reduction in tail latency; and 3) improves fairness. Our evaluation results show that Mu uses SLOs and the placement engine to guide resource allocation, leading to more consistent performance and fairness across functions, while avoiding long tails for the response time.

Acknowledgements: We sincerely thank the US NSF for their generous support through grants CNS-1763929, CRI-1823270, CNS-1815690, CPS-1837382, and SRC Task 3046.001. We also thank our shepherd, Prof. Ramesh Govindan, and the anonymous reviewers for their valuable suggestions and comments. We thank Vivek Jain for his extraordinary support and contribution throughout the project.

References

[1] 2021. 503 Service Unavailable. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503. [online].

[2] 2021. Borg: The Predecessor to Kubernetes. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/ [online].

[3] 2021. Horizontal Pod Autoscaler (HPA). https://knative.dev/docs/serving/autoscaling/autoscaling-concepts/#horizontal-pod-autoscaler-hpa [online].

[4] 2021. Knative. https://knative.dev/ [online].

[5] 2021. Knative Pod Autoscaler (KPA). https://knative.dev/docs/serving/autoscaling/autoscaling-concepts/#knative-pod-autoscaler-kpa [online].

[6] 2021. Kubernetes. https://kubernetes.io/ [online].

[7] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. 1992. *Data networks*. Vol. 2. Prentice-Hall International New Jersey.

[8] Flavio Bonomi, Rodolfo A. Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, Mario Gerla and Dijiang Huang (Eds.). ACM, 13–16. https://doi.org/10.1145/2342509.2342513

[9] Anshul Gandhi, Xi Zhang, and Naman Mittal. 2015. HALO: Heterogeneity-Aware Load Balancing. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 242–251. https://doi.org/10.1109/MASCOTS.2015.14 ISSN: 1526-7539.

[10] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *NSDI*. 24–24.

[11] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. 2011. An up-to-date survey in web load balancing. *World Wide Web* 14, 2 (March 2011), 105–131. https://doi.org/10.1007/s11280-010-0101-5

[12] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[13] Varun Gupta, Mor Harchol Balter, Karl Sigman, and Ward Whitt. 2007. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation* 64, 9 (Oct. 2007), 1062–1081. https://doi.org/10.1016/j.peva.2007.06.012

[14] LLC Gurobi Optimization. 2021. Gurobi Optimizer Reference Manual. http://www.gurobi.com [online].

[15] Pawan Kumar and Rakesh Kumar. 2019. Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey. *Comput. Surveys* 51, 6 (Feb. 2019), 120:1–120:35. https://doi.org/10.1145/3281010

[16] Junfeng Li, Sameer G Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th International Workshop on Serverless Computing*. 37–42.

[17] Wes Lloyd and et al. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 159–169.

[18] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[19] M. Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (Oct. 2001), 1094–1104. https://doi.org/10.1109/71.963420 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[20] S. K. Mohanty, G. Premsankar, and M. di Francesco. 2018. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 115–120.

[21] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. 2019. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In *2019 IEEE World Congress on Services (SERVICES)*, Vol. 2642. IEEE, 206–211.

[22] Christos-Alexandros Psomas and Jarett Schwartz. 2013. Beyond beyond dominant resource fairness: Indivisible resource allocation in clusters. *Tech Report Berkeley, Tech. Rep.* (2013).

[23] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *The magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[24] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the Leading Edge of Mobile-Cloud Convergence. In *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services*. ICST, Austin, United States. https://doi.org/10.4108/icst.mobicase.2014.257757

[25] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM* 64, 5 (April 2021), 76–84. https://doi.org/10.1145/3406011

[26] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 205–218.

[27] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys Tutorials* 19, 3 (2017), 1657–1681. https://doi.org/10.1109/COMST.2017.2705720

[28] Liang Wang and et al. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

[29] Cui Yan. 2017. How does language, memory and package size affect cold starts of AWS Lambda? https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76. [online].