Priority- and Locality-aware Load Balancing For Serverless Clusters

Alexander Fuerst, Prateek Sharnma, Cristina L. Abad

Abstract—

An ever-increasing range of applications and workflows now rely on Functions as a Service (FaaS). As the adoption of serverless computing keeps growing, cloud providers must increase the efficiency in managing the resources of the clusters where these services run. However, the heterogeneity, scale, and different latency tolerances of functions makes scheduling and load-balancing challenging.

In this paper, we show it is feasible and effective to prioritize functions by their latency tolerances, to provide differentiated quality-of-service (QoS). We extend Consistent Hashing with priority-, locality-, and server load-awareness as key principles, and introduce a new load-balancing algorithm, k-CH-RLU. k-CH-RLU supports k priority levels, dynamically assigns more resources to higher priority functions, and mitigates workload bursts. We implement and evaluate k-CH-RLU in OpenWhisk; experiments show that our algorithm can improve latency of all functions by up to $5\times$ compared to OpenWhisk. Our function prioritization can also reduce the total resources required: We can reduce latency of high-priority functions by $3\times$ even with 25% fewer servers.

1 Introduction

Serverless computing, or Function-as-a-Service (FaaS), has emerged as an important and challenging abstraction for cloud providers, with an ever-increasing range of applications and workflows using this new abstraction. By handling all aspects of function execution–including resource allocation–cloud platforms can provide a "serverless" computing model where users do not have to explicitly provision and manage cloud resources (i.e., virtualized servers). Applications such as web services, machine learning, data analytics, and even high-performance computing can benefit greatly from the resource elasticity, lower pricing, auto-scaling, and development convenience of FaaS platforms.

While it provides more convenient abstractions for developing and deploying applications, the FaaS programming model presents new fundamental performance and efficiency tradeoffs for *cloud providers*. When coupled with the extreme scale (public clouds execute millions of functions a day) and heterogeneity (applications have widely varying function invocation rates and resource footprints), FaaS presents cloud providers with several performance challenges. The growing diversity and popularity of serverless applications introduces new challenges and opportunities in load-balancing and cluster-level scheduling.

Recent work has highlighted the importance of *locality* in serverless function execution. A direct consequence of the FaaS programming model and a fundamental performance attribute is the high startup latency of functions, which is referred to as a "cold start". Cold starts are caused by operations such as initializing a virtual execution environment (such as a lightweight VM or a container), and installing data and code dependencies (packages and libraries), and can add 100s of milliseconds to function latency. Reducing cold-starts is a key challenge in serverless computing [44]. In the FaaS context, locality entails running future invocations on the same server, which increases the "warm starts", since the initialized function sandbox can be retained in server memory (also known as keep-alive).

In this paper, we present the design and implementation of a priority-aware auto-scaling load balancer for FaaS platforms (such as OpenWhisk, OpenFaaS, and others). We address the challenges of routing and scheduling functions on a cluster of servers, and how such clusters can be horizontally scaled. Our work extends locality as the primary design principle for FaaS cluster-level resource management. However, we find that locality alone is insufficient: server load is also an important parameter which influences the slowdown of functions due to queueing delays and resource contention on overloaded servers.

Our second key insight is that functions have different *latency tolerances* that can be used to prioritize their execution and provide quality-of-service (QoS) differentiation. Function prioritization introduces additional resource-management and scheduling challenges and opportunities. For instance, we can delay lower-priority functions during workload bursts and periods of high server loads. Our empirical analysis into production FaaS workload traces highlights the pervasiveness and importance of highly bursty function invocations.

The triumvirate of locality, load, and priorities has many intricate tradeoffs resulting in a large design space of load-balancing policies and algorithms, and presents many challenges. Our goal is to design simple load-balancing and scaling policies that address these challenges in a rigorous and practical manner. Because of the importance of locality in improving function latency, we use consistent hashing [29] as the building block, which preserves locality even when the cluster is scaled by adding or removing servers, which is critical since function workloads are highly bursty.

We tackle the load-balancing challenges in a modular manner. We first address the problem of locality and load-aware load balancing with a solution that can handle function heterogeneity, skew, and burstiness. For this, we extend

Consistent Hashing with Bounded Loads [34], where the key idea is to run a function on its "home" server as long as the server is not overloaded. This preserves locality and allows for functions to be "forwarded" to other servers in case of overload. We compensate for stale server loads by introducing the notion of stochastic random loads. Our resulting algorithm, Consistent Hashing with Random Load Updates (CH-RLU), is cognizant of function locality, server loads, and different function cold and warm times.

We then extend CH-RLU to be priority aware, by introducing the notion of *cluster priority pools.* Functions of the same priority run within the same cluster partition or pool. This prevents lower-priority functions from interfering with latency-sensitive high-priority functions, and allows overcommitting and shrinking low-priority pools to reduce the resource requirements of serverless clusters. Our overall load-balancing algorithm is called k-CH-RLU, since it can support k priority classes. Our policies are designed to be simple and practical, with a small number of user-controlled parameters, allowing them to be a drop-in replacement for OpenWhisk's default load balancing implementation [2].

Prior work in serverless computing has largely focused on optimizing performance on a *single* server using various cold-start mitigating mechanisms and policies [43], [21]. We build on past insights on the importance of function locality, and extend them to a large cluster of servers instead of a single server. While load balancing has a long history of rigorous solutions, we find that the heterogeneity, skew, and stale loads of the FaaS environment present unique challenges. Classic load-aware techniques that use randomization such as power-of-2 random choices do not capture locality and lead to high cold-starts, and hashing-based techniques cannot deal with the heavy skew in function popularity. In sum, we make the following contributions:

1) We find that the locality vs. load tradeoff is central to function performance, and show how it can be combined with consistent hashing.
2) We conduct an empirical study of FaaS workloads and applications, and introduce the notion of function prioritization and QoS. We find prioritization can ameliorate bursty invocations and resulting load-spikes. Prioritization improves function latencies and cluster utilization.
3) We develop a priority-aware load-balancing policy, k-CH-RLU: Consistent Hashing with Random Load Updates with k priorities. k-CH-RLU partitions a cluster into k pools to avoid interference and increase locality, and tackles practical challenges of highly heterogeneous functions, bursty workloads, and stale/imprecise load information on a large cluster of servers.
4) We implement and evaluate k-CH-RLU in OpenWhisk. Compared to OpenWhisk, it provides more than $3\times$ reduction in latency for high-priority functions, and reduces the required number of servers by 25%.

2 Background

2.1 FaaS Function Execution

Function Initialization Overheads. With the Function-as-a-Service computing paradigm, providers run user code on-demand when a request comes in, and–importantly–decides *where* it should run. Each invocation is run in isolation from other concurrent and co-located invocations; security isolation is provided by running each invocation in a fresh sandboxed environment. Sandboxes are generally implemented using containers (such as Docker [1]) or lightweight VMs (such as Firecracker [4]) created on the server that runs the invocation. Creation time for both choices can be significant, adding latency to the in-flight request.

TABLE 1: FunctionBench [31] functions run times' are significantly longer on cold starts. Ideally we want all of our functions to run warm to lower user latency. Cold starts also increase system load by creating runtime overhead.

Application	Warm Time (s)	Cold Time (s)
Web-serving	0.179	1.153
ML Inference (CNN)	2.211	7.833
Disk-bench (dd)	1.068	2.944
Matrix Multiply	0.117	1.067
Sklearn Regression	53.57	54.45
AES Encryption	0.587	2.064
Video Encoding	10.28	11.51
JSON Parsing	0.414	1.962

Many solutions seek to reduce the initialization overhead from *cold starts*. Cold starts can be mitigated by skipping initialization entirely, by saving in memory and reusing the execution environment for subsequent invocations of the same function. By keeping the function "warm," a provider can amortize the startup cost across future invocations.

The warm and cold running times for functions from the FunctionBench [31] workload suite are shown in Table 1; running time is the total execution latency of these functions on OpenWhisk. Due to the large initialization overheads, the cold times are $1-10\times$ larger than the warm running time; the latter is faster because the function runs in an already initialized container, cached in memory.

Keep-alive For Reducing Cold-starts. A server cannot keep all the functions routed to it in memory indefinitely; when memory is needed it must choose which function to retain, aka a *keep-alive* policy. Many FaaS offerings—including OpenWhisk (which we build on in this work)—use a time-to-live approach that *evicts* a function from memory if it isn't re-used within a certain period. Recent advanced techniques based on the LRU and GreedyDual [11] caching algorithms decide to evict based on the function's startup time, memory footprint, and invocation frequency [21].

The benefits of locality have been investigated primarily at the *single-server* level. However, most real-world FaaS deployments use a large cluster of servers sitting behind a load balancer. Tailoring the load balancing algorithm to complement the choices made by the cache eviction algorithm, especially locality reuse, is vital in FaaS, and is the focus of this paper. For example, "sticky" load balancing to route a function to the same server preserves locality, at the risk of server overload. Since function latencies also depend on the load on the server, this naïve policy is sub-optimal, especially if the workload consists of functions with skewed popularities and running times. We elaborate on this further in the next section, and show that the tradeoffs imposed by FaaS requires a new class of load balancing approaches.

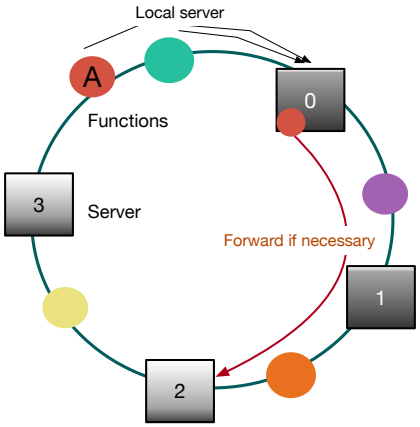


Fig. 1: Consistent hashing runs functions on the nearest clockwise server. Functions are forwarded along the ring if the server is overloaded.

2.2 Load Balancing

Managing the load of a cluster of servers is a common problem in distributed computing systems. Load-balancing policies rely on some notion of server "load," such as the number of concurrent tasks, length of the task queue and cpu utilization. We can classify load balancers into compute-oriented or data-oriented, as described herein.

Compute-oriented load balancers are typically used for short-running tasks and queries. A common example is that of web clusters [30] where: the tasks can be executed on any server, servers in a cluster are largely fungible, and the task performance largely depends on the current server-specific cpu utilization, so simple approaches like round robin and least connected are commonly used. From a queueing theory perspective, policies such as least-work-left (LWL) and join-shortest-queue (JSQ), have studied near-optimal load balancing for computing load-dependent workloads under a processor-sharing (PS) setting.

Interestingly, load balancing for *data-oriented* systems, such as Content Delivery Networks (CDNs) [36] and distributed key-value stores like Amazon Dynamo [13], must also balance the load on servers, but with data locality as a key requirement. In this context, locality refers to requests for the same object being handled by the server, or the same subset of servers if the object is replicated.

We find that FaaS load balancing requires and benefits from *both* these objectives: minimizing computing load *and* maximizing locality to reduce cold-starts.

2.3 Consistent Hashing

For data-oriented systems, a common technique for routing with locality is Consistent Hashing [30], [29], where objects are mapped to servers based on some key. Consistent hashing preserves object-server mapping even in the face of server additions/removals. Figure 1 provides an overview of consistent hashing. Both objects and servers are hashed to points on a "ring," and objects are assigned to the next server (in the clockwise direction) in the ring. Addition or removal of servers only affects the nearby objects by remapping them to the new next server in the ring.

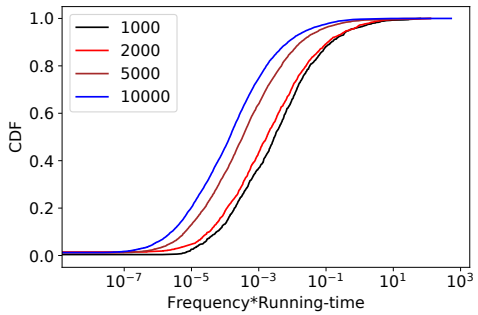


Fig. 2: Function load is very heavy tailed (note the log X axis). Each line represents a different random subset and associated subset size from the Azure function trace.

OpenWhisk uses a modified consistent hashing algorithm for routing functions: As functions are sent to servers, their expected memory footprint is added to a server-specific running counter of outstanding requests. Upon completion, the memory size of a function is decremented from that server's counter. If the counter for a function's "home" server would exceed the assigned memory on the server, it is forwarded along the ring. The drawback of this policy, and consistent hashing as a whole, is that performance is affected by the relative popularities of the different objects. A highly popular object can result in its associated server getting overloaded. This problem is exacerbated in the case of FaaS functions, as we show in the next section.

3 Challenges In FaaS Load-Balancing

Load balancing in FaaS clusters represents a unique set of challenges described in this section. We motivate our observations using the Azure function traces [39] and empirical performance measurements with OpenWhisk.

3.1 Function Heterogeneity and Skew

For locality-sensitive load-balancing techniques to be effective, it is important for each function to impose a roughly similar load on the system. In reality, the the frequency of invocation and running time of functions vary widely. Table 1 shows that the running times of functions is highly heterogeneous and range from 100ms to almost one minute. Thus, the computing requirements (in terms of running time) of functions are highly heterogeneous.

The popularities of the functions (i.e., their invocation frequency) is also highly skewed. Figure 2 shows the distribution of the frequency $\times$ running-time, for four randomly sampled subsets of functions from the Azure trace. This metric is effectively the "induced-load" of a function. We see that the functions are extremely heavy-tailed in their induced-load: the "heavy" top 20% functions consume two orders of magnitude more resources than the average. As a result, with classic consistent hashing, the servers handling the heavy functions would be extremely overloaded, leading to severe function slowdown due to resource contention.

3.2 Bursty Invocations

The second challenge is that the function arrivals can be very bursty and vary widely by function. Figure 3 shows the

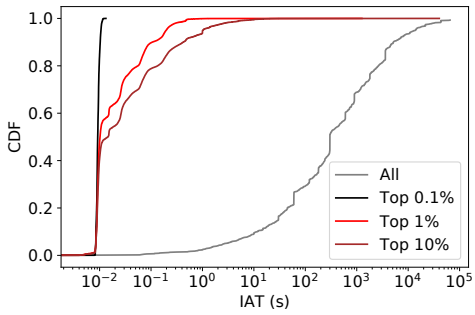


Fig. 3: Inter-arrival times of popular functions can be extremely low, with a wide variance (note log-scale of X axis).

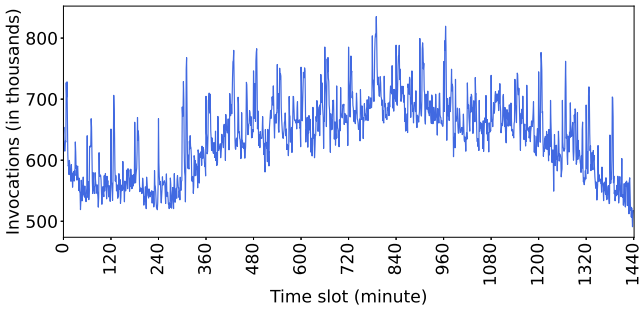


Fig. 4: Bursty request arrivals (invocations per minute) in day 5 of the Azure Traces.

average inter-arrival time (IAT) distribution computed from the Azure functions dataset; the average IAT of functions varies widely (the "All" line in the figure): by more than seven orders of magnitude. Importantly, the IAT of the popular functions (ordered by number of invocations) can be significantly lower and different. For instance, for the top 10% of the popular functions, their 90^{th} percentile IAT is less than 1 second. In contrast, the 90^{th} percentile IAT for all functions is 2,000 seconds.

We also characterize burstiness using the index of dispersion for counts (IDC) [24], [26], defined as the ratio of the variance to the mean: $IDC = \sigma_W^2/\mu_W$, over time window W. As the IDC is dimensionless, we use the standard deviation, σ to describe the variability of the peaks in the unit of the workload (requests per minute). Figure 4 shows that invocations are very bursty ($IDC = 6092$, $\sigma = 62316$), with peaks that are up to 31% higher than the mean.

This heavily skewed, bursty workload has significant implications for load balancing, since we must be able to handle highly bursty functions, as well as the long tail of infrequently invoked functions. The fairness in handling functions is an important challenge in FaaS load balancing.

4 Function Prioritization Motivation

In this section, we show that—in opposition to what others have argued in the past [47]—(some) real serverless workloads *are* delay tolerant (§ 4.1), we discuss the current status of differentiated serverless functions (§ 4.2) and provide a workload characterization based on dividing functions into high- and low-priority classes (§ 4.3). Our study of the delay tolerance of serverless functions is based on a recent dataset of serverless applications [18], [20]. For the workload characterization, we analyze traces with real serverless workloads from Microsoft Azure [39] and present the results of one representative day (day 5 of the 2-week trace).

4.1 Can Serverless Functions be Delayed?

In this subsection, we analyze if serverless functions can tolerate some level of delay in their execution. This delay could come from not executing the function immediately (e.g. through queueing), or via reduced priority in scheduling (e.g. giving less resources to lower-priority functions). While these two mechanisms differ, the result of both is increasing the turnaround time of the functions; this increase is not appropriate for some functions—like those supporting interactive or real-time applications—but is tolerable for multiple other applications, as discussed in this subsection.

Serverless functions can be triggered through several mechanisms like http invocations, cloud-native events, queue messages and timers. Functions triggered by http are time sensitive as these are synchronous requests that can timeout and are often used in interactive applications; all other triggers invoke functions that can tolerate delays to varying degrees. Thus, functions not triggered by http are (potentially) delayable; in the Azure trace, these constitute 59.06% of the functions and 69.07% of the invocations.

Furthermore, a recent survey of serverless use cases [20] found that 66% of applications in a large dataset have at least one delay-tolerant function. To further illustrate our point, we analyzed the applications in the dataset and selected one delay-tolerant scenario for each of the triggers (except http, which is not delayable as already explained); these scenarios are described in Table 2 and include examples such as removing unutilized EBS volumes, a web analytics (clickstream) application, a messenger chatbot, a batch image manipulation application for a serverless galleria, a subscription fulfillment application for The Guardian, and an application for sequence alignment of protein sequences.

A note on http-triggered functions: Some web applications need end users to invoke asynchronous behavior via http requests. The Microsoft Azure Cloud Design Patterns documentation [17] provides a solution to this problem with the "Asynchronous Request-Reply" pattern which disaggregates such behavior into three functions: two http-triggered ones (to enqueue request and to check status of job), and a queue-triggered backend function. In this scenario, the http-triggered functions are not delay tolerant, but the queue-triggered function is.

4.2 Current Status of Differentiated FaaS Invocations

A recent empirical study [42] found that current cloud providers (AWS, Azure, GCP and IBM) treat http functions differently from those triggered by other means, frequently via undocumented behavior, including different concurrency limits and prioritization versus background functions. In addition, while asynchronous functions are queued and thus the user understands that they may not execute immediately, providers don't typically enqueue synchronous functions but rather return with an error if peak exceeds

TABLE 2: Delay-tolerant scenarios. The ID of the use case is taken from a public dataset [19]. We studied each case and estimate a likely delay tolerance for that scenario.

Trigger group	ID	Delay tolerance	Description
timer	6	<10min	NetApp's cloud function to periodically find unutilized EBS volumes to delete.
event	28	<60min	Google Analytics clone to track website visitors.
queue	59	<60min	Messenger Chatbot for a media company; sends weekly news updates to user.
storage	30	<10min	Serverless Galleria, for batch manipulation and publishing of images.
orchestration	12	<24h	Subscription fulfillment for online and print subscriptions of The Guardian.
others	101	<10min	Scientific task, batch, on demand, for sequence alignment of protein sequences.

concurrency limits or provider capacity.[1] Thus, the major public cloud providers are already implicitly defining two classes of functions: high priority (synchronous) and lower priority (asynchronous), with the latter being delay-tolerant.

In OpenWhisk asynchronous functions are defined with the `async` keyword and follow a different execution path than synchronous ones. However, the platform does not use any mechanism to delay or execute them at a lower priority. A solution that treats `async` functions as delay-tolerant, scheduled with a lower priority than synchronous functions, can be added to OpenWhisk without API modifications.

4.3 High- and Low-Priority Workloads

Considering the two function classes described in the prior subsection—high and low priority—we analyze the Azure traces applying this division in the workload. For the hig priority class, we consider http-triggered functions. For th low priority class, we consider all other functions.

High-priority functions constitute a significant portion (the workload, though the low-priority functions are great in number and account for more than 2/3 of the invocation Specifically, high-priority functions constitute 41% of th functions and 31% of the invocations, versus 59% of th functions and 69% of the invocations for the low-priority.

Figure 5 (Left) shows the cumulative distribution fun tion (CDF) of the average function durations. The functic durations are similar for both classes, with high-priori functions being slightly shorter (median 0.2 vs 0.8 minutes Figure 5 (Right) shows the CDF of the per-function average inter-arrivals; the IATs for both classes are similar.

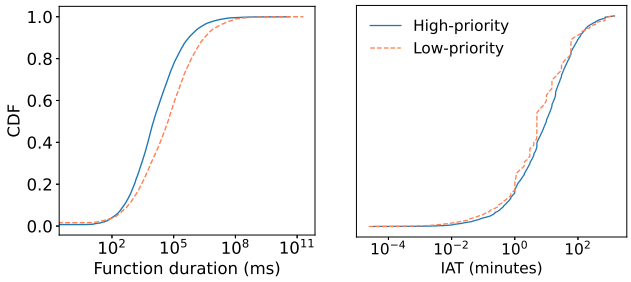


Fig. 5: Left: CDF of the function durations. Right: CDF of the average function inter-arrivals. Functions are divided into two classes: high and low priority.

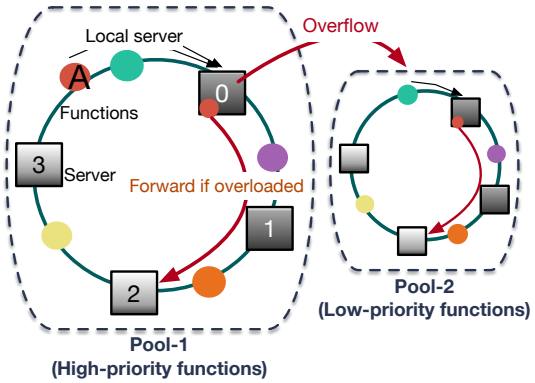


Fig. 6: k-CH-RLU partitions a cluster into multiple server pools and runs server-load-aware consistent hashing in each pool. Functions are forwarded if servers are overloaded, or to a lower-priority pool if the entire pool is.

5 Load and Priority Based Consistent Hashing

In this section, we describe the load-balancing algorithm that takes into consideration function locality and priority, bursty workloads, and stale server load information. We assume a cluster of homogeneous servers, and that a new function invocation can be sent to any of the servers. Each server implements keep-alive for functions: after successful execution, the function's container is stored in server memory, and evicted based on some eviction policy.

Architecture. To support QoS for functions of different priority levels, we use a two-stage load-balancing architecture (see Figure 6). The cluster is partitioned into multiple pools, one for each priority level. For ease of exposition and without loss of generality, we consider two levels: high- and low-priority functions, and thus two corresponding pools. Within a single pool, functions are load-balanced among servers using a load-aware consistent hashing technique. Our approach is modular: each pool runs an independent load-balancing policy, which is the focus of most of this section. This approach also preserves function locality, which is important for reducing cold-starts, as we describe next.

1. An exception is GCP which handles synchronous calls in best effort fashion, performing queuing but not ensuring zero drops [42].

5.1 Tradeoff between Locality and Load

We use consistent hashing as the fundamental principle to ensure high locality: repeated invocations of the same function occur on the same server. However, popular functions,

i.e., which are invoked very frequently, can result in overloaded servers. Because function performance is affected by server load and resource availability, focusing on locality alone can result in slow function execution.

Function popularities are also highly skewed: a small percentage account for a vast majority of invocations. With pure locality-based load-balancing, the servers of these popular functions would be severely overloaded. Functions also can run for significantly longer than simple web requests, and thus they impose more load on servers, and the cost of a wrong placement decision is higher. This, combined with bursty invocations, can significantly increase the tail latency of functions. Thus, pure-locality policies such as classical consistent hashing are not sufficient, and our research question is: *Can consistent hashing be used to reduce latency due to overloaded servers?* Or put another way, can we balance the tradeoff between function locality and server loads with consistent hashing? Our key idea is to extend consistent hashing to take also into account server loads, the cold-start overheads of different functions, and the bursty traffic that is a key characteristic of FaaS workloads.

5.2 Key Principle: Load-based Forwarding

To balance the locality vs. server load tradeoff, we build on a new variant of consistent hashing called Consistent Hashing with Bounded Loads [34] (abbreviated as CH-BL in the rest of the paper). The key idea behind CH-BL is to use consistent hashing to locate servers for objects, and if the servers are "full", then "forward" the objects to the next server in the consistent hashing ring.

For example, in Figure 1, function A is originally assigned to server 0, but this "home" server is overloaded (already running many functions), and thus the function is forwarded along the ring until a suitable non-overloaded server (2) is found. Any 5-independent hashing function can be used for determining the "home" server of a function. Users can specify the load upperbound or the capacity of the server (b), which determines the max load the server can sustain. Consistent hashing with bounded loads provides many strong theoretical guarantees on the length of the forwarding chain until the object is safely placed on a server.

Interestingly, forwarding along the ring not only avoids server overload, but also improves locality, *even in overload scenarios*. Forwarding along the ring has the advantage that even if a function is not run on its "home" server, subsequent invocations that "overflow" still have a high warm-start probability on the servers in the overflow chain. The warm-start probability is highest on the home server, and decays the farther the function is from it. This works better than alternative techniques such as Consistent Hashing with Random Jumps [10], which do not preserve locality and forwards to a randomly chosen least loaded server.

Server Load is a key metric in load-balancing policies to determine the *relative* suitability of one server over another. OpenWhisk currently uses occupied-memory used by active/running invocations as a proxy for load, and is unsuitable because functions have highly variable CPU utilization. Instead, we primarily rely on *system-level* load metrics, such as the standard Linux 1-minute load-average, which captures CPU utilization and I/O wait due to cold-starts, and provides a more realistic measure of load. We normalize the load-average by the number of CPUs. Thus, a load-average of 8 on an 8 core server (discounting hyperthreading) is normalized to 1. An important practical consideration is that load information is often *stale* due to delays in monitoring and collection, with the degree of staleness ranging from a few seconds to several minutes. Techniques for ameliorating the staleness are presented in Section 5.2.2.

5.2.1 Detecting Popular Functions with Spatial Sampling

Our goal is to detect "popular" functions with low inter-arrival times, in an online low-overhead manner. Popularity detection must take into account the changing invocation frequencies of different functions over time, and be low-overhead. We identify the top p percentile of functions by their inter-arrival times (IAT), or below some explicit IAT threshold, to reduce unnecessary hyperparamaters.

Our approach is general: we build a histogram of inter-arrival times using sampling and then query it. We note similarities with computing reuse distance histograms, which are the building block of miss-ratio curves. Reuse time histograms are a simpler version of reuse distances: Reuse distance is the number of *unique* objects accessed, whereas inter-arrival time is the difference in wall-clock times.

Our solution to identifying popular functions and function bursts is inspired by the popular SHARDS [46] algorithm for building reuse distance histograms. Following SHARDS, we randomly sample invocations to track individual function IATs. This tracking is simplified by only recording the most recent access time, and then computing the IAT as an estimated moving average of the current IAT and $now - last\_access$. These values are tracked for every function, and functions in the top p^{th} percentile of IATs are considered **popular**. For the sampled functions using spatial hashing, we update their IAT. Note that this approach keeps only a small number of last-accessed IAT entries in memory: "have-been" popular functions are naturally evicted from the tracking list. Since we do not care about reuse distances, we avoid keeping a tree of reuse distances, resulting in a simplified SHARDS-like algorithm (see Algorithm 1).

Algorithm 1 SHARDS-inspired popular function detection. Functions with the top p percentile of IATs are 'popular'.

```
1:  procedure UPDATE_SHARDS_POPULAR(func, time)
2:      P ← 100.0
3:      T ← 20.0                          ▷ Effective sampling rate
4:      R ← T/P
5:      Ti ← abs(hash(func.name))
6:      if Ti ≤ T then
7:          if last_access_times.contains(func) then      ▷
    Already in our sample set
8:              iat ← (t − last_access_times[func])/R
9:              last_access_times[func] = t
10:             iat_heap.push((iat, func))
11:         else                          ▷ First access... iat=='inf'
12:             last_access_times[func] = t
13:             iat_heap.push((t/R, func))
14:     iats_only ← iat_heap.values()
15:     pop_thresh ← percentile(iats_only, p)
```

5.2.2 Randomly Updating Stale Loads

Popular functions represent such a large percentage of invocations yet a small number of functions that they can be safely spread across multiple servers without causing cold starts. A fair load balancing algorithm must spread popular functions to ensure QoS for less frequent functions. As load information is stale, enforcing locality and load can result in servers facing a herd effect. Randomization is a powerful strategy to ameliorate this; but, we must use it judiciously due to the strong effects on locality in FaaS load balancing.

Our solution is to introduce random forwarding (along the ring) proportional to the load of the server, such that popular functions are forwarded with a higher probability. If the (stale) load of the server is L, we update its load by adding gaussian noise with a mean of the *extra anticipated load* on the server, based on the staleness and server-level function arrival rate (λ). Specifically, $L_{\text{noisy}} = L + \mathcal{N}(\mu = \lambda, \sigma = 0.1)$, where $\mathcal{N}$ is a Gaussian random variable. For popular functions, we compare the L_{noisy} to the load bound. For remaining functions, we continue to use the stale load L. Thus, for highly loaded servers "near" the upper bound, the extra random noise will result in the popular bursty functions being forwarded more to avoid the herd effect.

5.3 Putting it all Together: k-CH-RLU

Our overall policy, Consistent Hashing with Random Load and Updates with k pools (k-CH-RLU), combines all the previously described techniques and insights.

Upon a new function invocation, it runs in its pool using the `forward` procedure in Algorithm 2, which combines the use of SHARDS for popularity detection, cold and warm times for increasing the effective load bound, and noisy loads. The initial server is determined using a consistent hashing function. We bound the cold/warm ratio with a final load upper bound, b_max. The load-bound parameters determine the locality sensitivity: higher values of b and b_max increase locality at the risk of resource-contention delays. Similarly, higher values of p results in more aggressive random forwarding and reduces locality.

Our two-level architecture is modular and allows us to parameterize different load-balancing policies for different pools. Lower-priority pools are run with a higher load bound b_max, and thus tolerate more overloaded servers, at the risk of lower function performance. Forwarding along the chain has diminishing returns on locality, and if the function gets forwarded more than max_chain_len times, it triggers the overflow condition.

Function prioritization and QoS are controlled via the server pools: Each function has a default pool based on their priority level, with higher-priority functions having lower pool numbers. The high-priority functions recursively overflow to the next lower-priority pool, and thus have good locality because all pools use our consistent hashing approach. The high-priority functions can thus overflow and potentially use the entire cluster in case of workload spikes, preserving their QoS. The lower-priority pools thus also serve as a "burst buffer," crucial considering the bursty nature of functions. Low-priority functions can't make use of higher-priority servers even if they are available, since we want to be able to handle bursty invocations of higher-priority functions. For the lowest-priority pool, we run the function on the least-loaded server in their pool. If the least loaded server is also overloaded, we drop the function.

Algorithm 2 k-CH-RLU

```
1:  procedure FORWARD(pool, func, server, chain_len)
2:      b, b_max, max_chain_len ← system_params
3:      if chain_len > max_chain_len then                ▷ Overflow
4:          if pool == k then  ▷ Lowest-priority pool return least-loaded-server
5:              else
6:                  forward(pool+1, func, server=CH(func, pool), 0)   ▷ Try in next pool
7:      λ ← 1.0/avg_iat                                  ▷ Computed from Algo 1
8:      L = Load(server)
9:      if popular(func) then                            ▷ Computed from Algo 1
10:         L = Load(server) + N(μ = λ σ = 0.1)
11:     if L < min(cb/w, b_max) then
12:         server
13:     else
14:         forward(pool, func, next(server), chain_len+1)
```

Pool Sizing. The priority pools are sized proportional to the number of functions registered at different priority levels. Thus, if 25% of all functions are low-priority, then 25% of the servers are in the low-priority pool and the rest are in the high-priority pool. Periodically, we recompute this ratio based on the currently registered functions, which may lead to resizing the pools. Importantly, locality is preserved even after pool resizing because of consistent hashing.

6 Implementation

We have implemented our priority-aware consistent hashing with random load update (k-CH-RLU) policy and other load-balancing policies in OpenWhisk, a popular FaaS system. Our changes amount to more than 2,000 lines of code across many OpenWhisk components, but are primarily in the load-balancer class. In this section, we describe major implementation details, as well as key performance optimizations that improve OpenWhisk's performance and scalability by more than $4\times$.

Our policies are implemented by modifying the load-balancer module of OpenWhisk (see Figure 7). k-CH-RLU is implemented by modifying the existing OpenWhisk "container sharding" policy, which also uses consistent hashing, and forwards functions using available memory as the load metric. We use OpenWhisk's existing consistent hashing implementation, permitting an "apples to apples" comparison, and also making k-CH-RLU a drop-in replacement for the OpenWhisk default load balancing. At the invoker level, we adapt FaasCache's GreedyDual keep-alive policy, which increases the keep-alive effectiveness compared to OpenWhisk's default non-resource-conserving TTL eviction [21].

The k-CH-RLU algorithm described in the previous section requires two main additional pieces of information from each invoker/server: the load averages, and the cold/warm running times of functions. Both of these are periodically captured (every 5 seconds) and stored in a

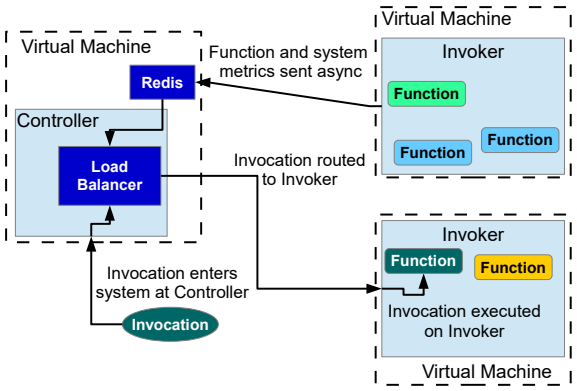
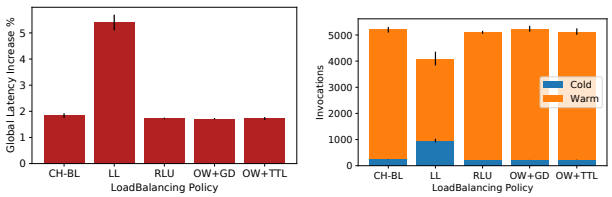


Fig. 7: Relevant OpenWhisk components and communication used to schedule and run function invocations.



(a) Global Latency Impact (b) Invocation Throughput

Fig. 8: Latency and throughput under low load. The locality-agnostic least-loaded policy has more cold starts and a higher impact on latency.

centralized Redis key-value store. The load balancer in the controller reads these asynchronously: working with stale and inconsistent metrics is our key design goal. The default load bound, b, is 1.2, and the max load, b_max is 6. Popularity threshold is set to 20%. We did not observe performance to be very sensitive to these parameters, and thus do not need to auto-tune them; they are suitable as user-inputs.

6.1 Performance Optimizations for OpenWhisk

Since our goal is to run functions under high load, we ran into a large number of OpenWhisk performance and scalability bottlenecks. We found default OpenWhisk to be almost unusably slow and unstable even under reasonable load. We present its details and our actions to overcome them, hoping that the fast-growing serverless computing research field can benefit from our lessons.

We found the primary source of scalability bottlenecks to be running Docker containers concurrently, with significant contention in `dockerd`, the control daemon that handles the container lifecycle events. Even at moderate loads (normalized server load average close to 1), high `dockerd` contention increase tail latencies by *several minutes!*

Currently, OpenWhisk **pauses** a container after function execution, preventing it from being scheduled by the CPU; it later resumes the container before running the next invocation of the same function (assuming a warm start). Thus, each invocation requires two additional (pause/resume) events to be handled by `dockerd`, leading to significant lock contention. Because of the FaaS programming model, the pausing is not necessary, since nothing in the container can run after a function has returned. We remove these redundant pause/resume operations to reduce `dockerd` contention, cutting down the OpenWhisk overhead by 0.2 seconds *per-invocation*, on average. More importantly, by reducing `dockerd` contention, we are able to run a larger number of concurrent functions.

An even larger source of scalability bottleneck is **network** namespace creation time. Using the default bridge networking requires each invocation to create a new TUN/-TAP network interface. This is a very expensive operation due to Linux network stack overheads (several 100 ms), and to `dockerd`'s userspace lock (futex) contention for its networking database. As the *historical* total number of containers launched grows, so does the size of the network-interface database. `dockerd` reads and updates this database under the critical section, and the larger database results in higher lock contention. As a result, we were unable to use VMs with more than 4 CPUs after 20 minutes of sustained load, since the `dockerd` contention resulted in many functions timing out (timeout was 5 minutes)! We sidestep this problem by not using bridge networking, but instead using Docker's *host* network option and assigning each container a unique port on the host. Implementing the network change required updating the OpenWhisk runtimes used to wrap functions to monitor their specified port. This change allowed us to run functions on larger invokers and under more sustained load, and eliminated most timeouts.

Finally, after a certain request rate threshold, we found that the default `nginx` OpenWhisk frontend would crash and return *502 BAD GATEWAY* for all URLs. We did not discover the cause of this problem and simply bypassed it by letting function invocations communicate with the controller/load-balancer directly.

CPU limits. To prevent functions from using more than their CPU allocation, we use `cgroups` to set a hard limit on CPU cores. OpenWhisk currently uses the *--cpu-shares* flag to set container CPU priority, with the unintended consequence of allowing functions to use more than one CPU core.

Together, these optimizations have allowed us to run OpenWhisk on invokers that are $4\times$ larger, and serve more than $6\times$ the load, without dropping functions due to timeouts. We plan to upstream these optimizations to OpenWhisk, to provide a higher-performance and lower-jitter platform for FaaS research and production deployments.

7 Evaluation

We evaluate our load-balancing policy (RLU) using the OpenWhisk implementation and a simulation implementation of the same policies. Our primary goal is to quantify the impact of different load-balancing policies on function latencies under varying load conditions.

7.1 Evaluation Environment

HW and SW Configuration. We run OpenWhisk across 9 VMs, with 8 invokers each in their own VM and a final VM that hosts the controller, load balancer, and remaining services. The invoker VMs have 16 vCPUs and 32 GB RAM

for hosting functions; the controller VM has 12 vCPUs and 50 GB RAM to ensure it is not a bottleneck. System load metrics were captured every 5 seconds by calling *uptime* on each invoker's VM, and normalized by the number of CPUs on that system. All latency information was recorded by the client, timing the http request until the request completed. We make no policy changes to the invoker eviction policy, but use the changes from FaasCache [21] for eviction decisions on the invoker.

Contenders. We compare our proposed load balancing policy against the default OpenWhisk load balancing policy (described in Section 2.3) with GreedyDual (OW+GD) and 10 minute Time-To-Live (OW+TTL) eviction policies, and two other load balancing polices: least loaded (LL) and consistent hashing with bounded loads using stale load-averages (CH-BL). Our load-balancing policy, k-CH-RLU will be abbreviated as k-RLU. When showing single-pool performance ($k = 1$), for ease of exposition, we shall simply refer it as RLU (or CH-RLU).

For CH-RLU and CH-BL, we set the $max\_chain\_len = 3$, a high max load bound, $b\_max = 6$, and a popularity threshold, $p = 20\%$. We did not find performance to be particularly sensitive to the load bound: the function latencies showed little changes across load upper bounds of $[2 - 8]$.

Metrics. We examine three main metrics: cold starts, the global average latency across all invocations, and the evenness with which load is spread amongst workers. The first two directly and obviously relate to end-user service quality but the third is more intricate. Providers pay for servers to run functions on and don't want those resources going unused and therefore wasted. Equally, a server that is overloaded (not enough CPU or memory resources) will cause a spike in end-user latency due to contention of queueing. To quantify the global impact on latency from placement decisions, we normalize each invocation's latency by the ideal (minimum) latency, take the per-function mean of these, multiply each mean by the percentage of invocations that function had in the whole trace, and finally take the mean of those function latency means. This is essentially a weighted average of latency-increase (i.e., slowdown). It gives some balance between outcomes; for example, a rare function may get several bad placement decisions and thus increase the global latency, or a very common function generally has warm hits and does not impact latency.

Workload. We convert 12 functions from FunctionBench [31] to run on OpenWhisk. To create a more realistic variety of functions, we create ten copies of each function with unique names, giving us 120 unique functions. Each function clone is invoked at different frequencies mimicking the arrival frequencies of the Azure trace [39]. Our load is generated using the closed-loop load generation tool Locust [33] to invoke functions, with 20 threads for low load and 120 for heavy load stressing. Locust cannot easily have dedicated threads to invoke each function, so we convert the "frequencies" into weights and use those to randomly choose what function will be invoked next. Each thread will iteratively invoke a random function, and after its completion wait 0-1 seconds before invoking another function. Unless stated otherwise all experiments are run with the above settings, under heavy load, for 30 minutes, and results are the average of 4 runs.

7.2 Single-pool Load-balancing Performance

We start our empirical evaluation with an analysis of *single-pool* load balancing, i.e., when all functions have same priority. When we run them under **light load** in Figure 8, the policies that use a locality mechanism are essentially identical. The load on any one server is never high enough to impact co-located functions and we never have to forward invocations and incur in excess cold starts, giving us a "lower bound" on load balancing. The low 1-2% latencies in Figure 8a are due to initial cold starts for functions and the varied overhead imparted by the system analyzed earlier. The least loaded policy is significantly worse as its lack of locality causes excessive cold starts as shown in Figure 8b.

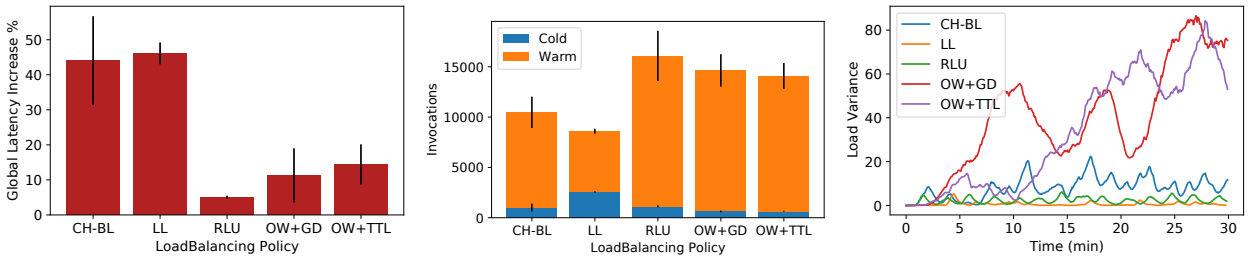
Next we run the policies under our **heavy load** scenario, and get a clear distinction in performance. The two versions of OpenWhisk in Figure 9a only increase latency by 11% and 14% respectively, which is rather good. They cannot complete with RLU whose increase is less than half of that, a tiny 5% impact on global latency. CH-BL and least loaded increase global latency by over 40%, showing terrible performance in that metric and on invocation throughput.

The wide gap between policies can be understood by comparing the load variance between the workers (Figure 9c). OpenWhisk's default policy is to only move a function to another server if the "home" one does not have enough available memory. While very good for locality—getting fewer cold starts than RLU in Figure 9b—it creates severe worker load imbalance: Some workers grow to extremely high load and their functions suffer, while others are mostly empty. RLU intelligently forwards invocations when a worker is near overload, keeping load variance low while protecting locality. Least loaded does the best at keeping equal load amongst workers, but at the cost of poor locality.

7.2.1 Handling Bursty Traffic

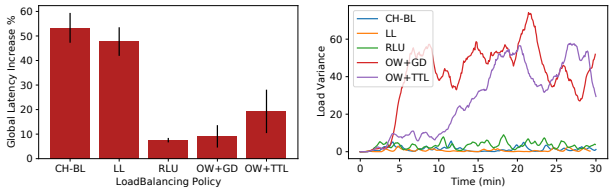
Next, we use two bursty workloads to see how the policies handle changes in invocation patterns. The first workload uses closed-loop load generation but adjusts the weights by which functions are invoked: Every 30 seconds two of the top weighted functions are chosen to become bursty and their weights are set much higher; at the end of a burst their weights are returned to normal and another two functions are chosen. As seen in Figure 10a our policy achieves a 17% lower impact on global latency than OpenWhisk with GreedyDual and RLU has a 60% reduction in latency over OpenWhisk with its default TTL backend. The more advanced eviction decision choices have a clear effect on improving the system, even when the load balancer does not optimize for it. The longer running functions have a larger effect on system load, and the load balancer must be aware of this impact and either spread the heavy popular function around or move other functions off of that server. Again, OpenWhisk does not take load into account and severely overloads some servers while languishing others. We see more sky-high load variances from this bursty workload in Figure 10b. Policies that monitor load—our RLU, CH-BL, and least loaded—keep tighter control on load variance.

The second bursty workload has a 30 minute long-rising burst, starting with a few invocations per second and reaching a sustained peak of 18 invocations per second at $\sim$25



(a) Latency (b) Throughput (c) Server Load variance

Fig. 9: At high server loads, our RLU policy reduces average latency by 2.2x at higher throughput, compared to OpenWhisk's default policy. It does so by keeping cold-starts and load-variances low.



(a) Global Latency Impact (b) Worker Load Variance

Fig. 10: RLU improves latency by 10% compared to OpenWhisk under bursty load conditions, while keeping a low worker load variance.

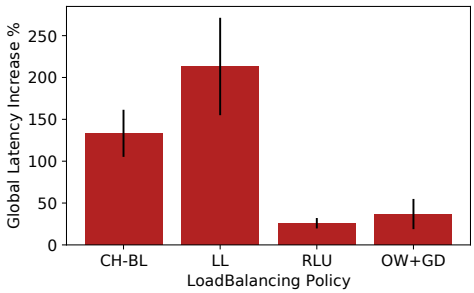


Fig. 11: Global latency impact under a 30-minute long rising burst load from an open-loop generator. RLU reduces latency by 17% compared to OpenWhisk.

minutes. We generate this load with a custom open-loop load tool that fires invocations but does not block waiting for completion, with new invocations fired continually in a preset pattern of function types and times. The global latency impact can be seen in Figure 11. Only the final 10 minutes of the workload place the system under extreme load, and the differences between policies reflect this. CH-BL and least loaded cannot keep up with the suddenly changing load, causing a latency increase of over 100% and 200% respectively, while RLU's 25% increase in global latency is significantly better (30% lower) than OpenWhisk. Our policy is able to make ideal choices for function placement under a variant of realistic workload scenarios.

7.2.2 Load-balancer Overhead

More complicated routing decisions are more computationally expensive to perform. We have been able to keep balancing decisions to roughly 1 ms thanks to the optimizations described in Section 6. Even so, RLU is on significantly slower making individual routing decisions, taking on average 1242.6μs to OpenWhisks' 472.3μs. Such times represent a fraction of the time spent per-request by the system and is made up for by our more optimal placements.

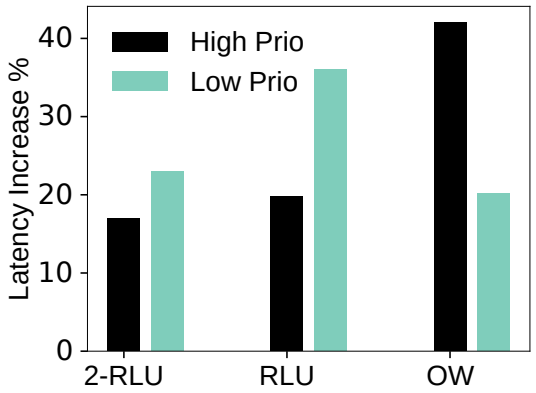


Fig. 12: Function prioritization improves latency for both high and low priority functions, and provides significant service differentiation and improvement over OpenWhisk.

7.3 Multi-pool Load-balancing Performance

So far we have seen how our load-balancing policy performs within a single pool when all functions have the same priority level. In this subsection, we evaluate the performance with *multiple* priority levels and corresponding pools. We are interested in the relative difference in performance between the pools, as well as how effectively we can provide service differentiation (by comparing to a single cluster).

7.3.1 Two-pool performance

For ease of exposition, we use two priority levels: high and low. Functions are evenly assigned priority levels (i.e., half are high priority). We use a cluster of size 8, with 4 servers in the high-priority pool.

Figure 12 shows the increase in latency for both high and low priority functions, compared to their best-case warm start performance under no system load. The "OW" and "RLU" categories are the baseline performance in a single unpartitioned cluster without any function priorities. With 2

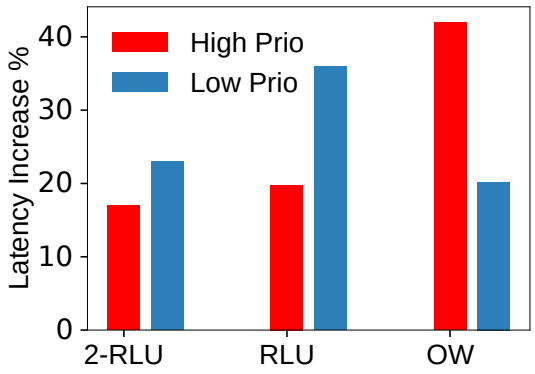


Fig. 13: Latencies on a 25% smaller cluster. High-priority functions see a 2× decrease vs. OpenWhisk.

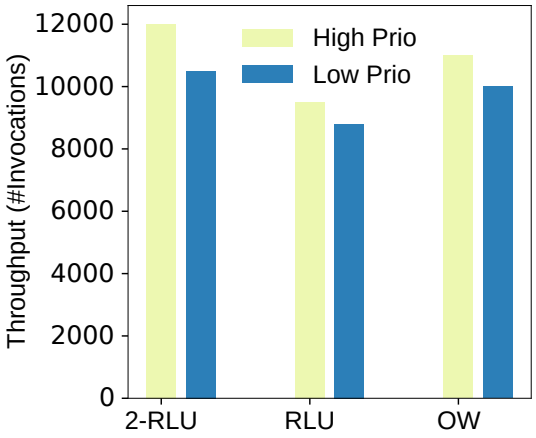


Fig. 14: Throughput with deflatable cluster.

pools (2-RLU), we can see a that the latency of high-priority functions decreases compared to low-priority functions. By splitting the cluster in two, the locality *increases*, and compared to the single-pool RLU, we see a decrease of 8% in latency for high-priority and 10% for low-priority functions. Compared to OpenWhisk, high-priority latency decreases by 5×, and low-priority latency *increases* by 20%, as OpenWhisk is not priority-aware. Thus, k-RLU can provide significant service differentiation over OpenWhisk, improving performance for *both* low and high priority functions compared to our single-pool RLU.

7.3.2 Deflated lower-priority pool

Service differentiation also allows us to *reduce* the size of the low priority pool and thus the overall cluster size—improving utilization and efficiency. We now evaluate function performance in such a differentiated setup. Half the functions are low-priority, but the size of the low-priority pool is *half* of that in the previous experiment. That is, we shrink the initial cluster of 8 by 25% to 6 servers, with 4 in the high-priority pool and only 2 in the low-priority pool.

Figure 13 shows the latency in such a setup. The single-pool RLU and OpenWhisk configurations are using the full original cluster size of 8 servers, while the 2-RLU has the 6 server configuration described above. Because of the reduction in total resources, the difference in performance between the 2-RLU and the single-pool methods like RLU and OpenWhisk is starker. Compared to single-pool RLU, high-priority functions see a 10% decrease in latency, and the low-priority functions see a decrease of 20%. Compared to OpenWhisk, we see a large 3x reduction in high-priority latency and an increase of 10% in low-priority latency. Thus, we can reduce the overall function latency by almost 2x *even while reducing total cluster size by 25%.*

A similar analysis of throughput is presented in Figure 14. The 2-pool small cluster configuration achieves 12% average higher throughput compared to OpenWhisk and 27% compared to single-pool RLU. Interestingly, RLU drops a larger number of functions owing to its strict load-bound, and thus achieves lower throughput.

8 Related Work

FaaS Resource Management: Multiple recent projects have looked into ways for optimizing the resource management of serverless frameworks, including solutions that reduce the overhead of the cloud functions [15], [4], [16], [5], [43], [8], improve locality through keep-alive policies [39], better caching algorithms for worker nodes [21], reducing function communication and startup costs [23], [12], [40], among others. Our work leverages the caching-based Greedy-Dual policy [21], in a cluster with pools that support differentiated services for high- versuls low-priority functions.

Latency-sensitive scheduling and load balancing in serverless: Nightcore [27] provides low end-to-end latency and variability using fast paths for internal calls, low-latency message channels, efficient threading and concurrency. Atoll [41] uses deadline-aware two-level scheduling as part of a low-latency serverless platform. Our proposal can be added to those solutions to further implement differentiated services on top of more performant serverless platforms. In general, there is a rich previous body of work in performance improvement methods [37], [28], [25] that are complementary to our approach. In addition, we extend the notion of locality-aware function routing [4], [6], [22], [3] and use it within cluster pools that ensure differentiated services when functions can be divided into priority classes.

Differentiated services in serverless: Sequoia [42] is a serverless framework with a QoS scheduler based on a simple priority-based queue; however, the issue of starvation in the presence of a continuous arrival of high-priority functions is not considered. Furthermore, the framework is based on a completely new design that does not support synchronous function calls. In contrast, our solution was implemented on top of OpenWhisk and considers both synchronous and asynchronous functions. Bilal et al. [7] analyzed the trade-off space between performance and cost that arises from different CPU/RAM configurations and the resulting function performance. This approach is orthogonal to ours and can be leveraged by the provider to offer differentiated services that span this configuration space. Qiu et al. [38] suggested that providers could implement resource over-commitment for FaaS workloads with loose latency objectives; our approach ties over-commitment with current

demand, with a dynamic mechanism that supports handling of bursts in high priority workloads at the expense of low priority ones. *Real-time serverless* [35] is a work-in-progress system that describes an interface for specifying invocation rate guarantees and proposes delivering them via admission control and predictive container management.

Workload shifting in the cloud: Delaying of tasks has been proposed to make better use of renewable excess energy [47], [48], to reduce energy consumption for workflow execution [45], among others. While we have not implemented policies with energy management goals, our solution could be extended to consider energy information (peaks, variable costs, green energy availability).

Multi-pool cluster scheduling: Virtual cluster pools have been used for dynamic resource management in datacenters [9], to handle increased workloads in data analytics clusters [32], for QoS multi-class admission control [14], and to decrease the delays of scheduling decisions [41]. We use this technique for service differentiation, and complement it with a novel control-based dynamic resizing mechanism to support burst absorption in serverless workloads.

9 Conclusion

We described and evaluated k-CH-RLU, a novel priority- and locality-aware load balancing algorithm for Function-as-a-Service (FaaS) clusters. k-CH-RLU supports k priority levels, dynamically assigns more resources to higher priority functions, and mitigates workload bursts. Empirical evaluation shows significant performance improvements (by up to $5\times$) compared to OpenWhisk. Moreover, k-CH-RLU can be used to reduce the resources required by the provider: We can reduce the cluster size by 25%, while achieving a $3\times$ latency reduction for high-priority functions.

Acknowledgments

The authors would like to thank Jose Viteri and Robinson Flores from ESPOL, who helped analyze the Azure trace data with respect to burstiness and workload characterization of high- versus low-priority functions (Figure 4 and Section 4.3).

References

[1] Docker. https://www.docker.com/, June 2015.

[2] Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/, 2020.

[3] Abdi, M., Ginzburg, S., Lin, C., Faleiro, J. M., Goiri, I. n., Chaudhry, G. I., Bianchini, R., Berger, D. S., and Fonseca, R. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)* (2023).

[4] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.

[5] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards High-Performance Serverless Computing. *USENIX ATC* (2018), 14.

[6] Aumala, G., Boza, E., Ortiz-Avils, L., Totoy, G., and Abad, C. Beyond load balancing: Package-aware scheduling for serverless platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2019), pp. 282–291.

[7] Bilal, M., Canini, M., Fonseca, R., and Rodrigues, R. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. *CoRR abs/2105.14845* (2021).

[8] Carreira, J., Kohli, S., Bruno, R., and Fonseca, P. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2021), pp. 58–64.

[9] Chase, J., Irwin, D., Grit, L., Moore, J., and Sprenkle, S. Dynamic virtual clusters in a grid site manager. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)* (2003).

[10] Chen, J., Coleman, B., and Shrivastava, A. Revisiting consistent hashing with bounded loads. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), vol. 35, pp. 3976–3983.

[11] Cherkasova, L. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Tech. rep., HP Labs Technical Report 98-69 (R.1), 1998.

[12] Daw, N., Bellur, U., and Kulkarni, P. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 585–599.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

[14] Delimitrou, C., Bambos, N., and Kozyrakis, C. QoS-Aware admission control in heterogeneous datacenters. In *International Conference on Autonomic Computing (ICAC)* (2013).

[15] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 467–481.

[16] Dukic, V., Bruno, R., Singla, A., and Alonso, G. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 45–59.

[17] Eastbury, W., et al. Cloud design patterns: Asynchronous request-reply pattern, 2022.

[18] Eismann, S., Scheuner, J., Eyk, E. V., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L., and Iosup, A. Serverless applications: Why, when, and how? *IEEE Softw. 38*, 1 (2021).

[19] Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., and Abad, C. The State of Serverless Applications: Collection, Characterization, and Community Consensus - Replication Package, Aug. 2021.

[20] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C., and Iosup, A. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* (2021).

[21] Fuerst, A., and Sharma, P. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, pp. 386–400.

[22] Fuerst, A., and Sharma, P. Locality-aware load-balancing for serverless clusters. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2022).

[23] Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., and Das, C. R. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 280–295.

[24] Gusella, R. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE Journal on Selected Areas in Communications 9*, 2 (1991).

[25] Hunhoff, E., Irshad, S., Thurimella, V., Tariq, A., and Rozner, E. Proactive serverless function resource management. In *International Workshop on Serverless Computing (WoSC)* (2020).

[26] Jagerman, D. L., and Melamed, B. Burstiness descriptors of traffic streams: Indices of dispersion and peakedness. In *Proceedings of the Conference on Information Sciences and Systems* (1994).

[27] Jia, Z., and Witchel, E. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).

[28] KAFFES, K., YADWADKAR, N. J., AND KOZYRAKIS, C. Centralized core-granular scheduling for serverless functions. In *ACM Symposium on Cloud Computing (SoCC)* (2019).

[29] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), pp. 654–663.

[30] KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. Web caching with consistent hashing. *Computer Networks 31*, 11-16 (1999), 1203–1213.

[31] KIM, J., AND LEE, K. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (July 2019), pp. 502–504. ISSN: 2159-6182.

[32] LEE, G., AND KATZ, R. Heterogeneity-Aware resource allocation and scheduling in the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2011).

[33] LOCUST. Locust: A modern load testing framework. https://locust.io/.

[34] MIRROKNI, V., THORUP, M., AND ZADIMOGHADDAM, M. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (2018), SIAM, pp. 587–604.

[35] NGUYEN, H. D., ZHANG, C., XIAO, Z., AND CHIEN, A. Real-time serverless: Enabling application performance guarantees. In *International Workshop on Serverless Computing (WoSC)* (2019).

[36] NYGREN, E., SITARAMAN, R. K., AND SUN, J. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 2–19.

[37] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *USENIX Annual Technical Conference (ATC)* (July 2018).

[38] QIU, H., JHA, S., BANERJEE, S. S., PATKE, A., WANG, C., HUBERTUS, F., KALBARCZYK, Z. T., AND IYER, R. K. Is Function-as-a-Service a good fit for latency-critical services? In *International Workshop on Serverless Computing (WoSC)* (2021).

[39] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX Annual Technical Conference (ATC)* (2020).

[40] SHEN, J., YANG, T., SU, Y., ZHOU, Y., AND LYU, M. R. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)* (2021), IEEE, pp. 194–204.

[41] SINGHVI, A., BALASUBRAMANIAN, A., HOUCK, K., SHAIKH, M. D., VENKATARAMAN, S., AND AKELLA, A. Atoll: A scalable low-latency serverless platform. In *ACM Symposium on Cloud Computing (SoCC)* (2021).

[42] TARIQ, A., PAHL, A., NIMMAGADDA, S., ROZNER, E., AND LANKA, S. Sequoia: Enabling quality-of-service in serverless computing. In *ACM Symposium on Cloud Computing (SoCC)* (2020).

[43] USTIUGOV, D., PETROV, P., KOGIAS, M., BUGNION, E., AND GROT, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 559–572.

[44] VAN EYK, E., IOSUP, A., ABAD, C. L., GROHMANN, J., AND EISMANN, S. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (2018).

[45] VERSLUIS, L., AND IOSUP, A. TaskFlow: An energy- and makespan-aware task placement policy for workflow scheduling through delay management. In *ACM/SPEC International Conference on Performance Engineering (ICPE), Companion Volume* (2022).

[46] WALDSPURGER, C. A., PARK, N., GARTHWAITE, A., AND AHMAD, I. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 95–110.

[47] WIESNER, P., BEHNKE, I., SCHEINERT, D., GONTARSKA, K. K., AND THAMSEN, L. Let's wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud. In *ACM/IFIP International Middleware Conference* (2021).

[48] WIESNER, P., SCHEINERT, D., WITTKOPP, T., THAMSEN, L., AND KAO, O. Cucumber: Renewable-aware admission control for delay-tolerant cloud and edge workloads. In *International European Conference on Parallel and Distributed Computing (Euro-Par)* (2022).