# Locality-aware Load-Balancing For Serverless Clusters

Alexander Fuerst
Indiana University
alfuerst@iu.edu

Prateek Sharma
Indiana University
prateeks@iu.edu

## ABSTRACT

While serverless computing provides more convenient abstractions for developing and deploying applications, the Function-as-a-Service (FaaS) programming model presents new resource management challenges for the FaaS provider. In this paper, we investigate load-balancing policies for serverless clusters. Locality, i.e., running repeated invocations of a function on the same server, is a key determinant of performance because it increases warm-starts and reduces cold-start overheads. We find that the locality vs. load tradeoff is crucial and presents a large design space.

We enhance consistent hashing for FaaS, and develop CH-RLU: Consistent Hashing with Random Load Updates, a simple practical load-balancing policy which provides more than 2× reduction in function latency. Our policy deals with highly heterogeneous, skewed, and bursty function workloads, and is a drop-in replacement for OpenWhisk's existing load-balancer. We leverage techniques from caching such as SHARDS for popularity detection, and develop a new approach that places functions based on a tradeoff between locality, load, and randomness.

## CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

## KEYWORDS

Functions as a Service, Serverless Computing, Cloud Computing, Load Balancing

**ACM Reference Format:**
Alexander Fuerst and Prateek Sharma. 2022. Locality-aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27–30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531459

## 1 INTRODUCTION

An ever-increasing range of applications and workflows are now using Functions as a Service (FaaS). 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 provided by FaaS platforms.

For cloud providers, FaaS has emerged as an important and challenging workload. While it provides more convenient abstractions for developing and deploying applications, the FaaS programming model presents new fundamental performance and efficiency tradeoffs. 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.

A direct consequence of the FaaS programming model and a fundamental performance attribute is the high startup latency of functions. Before the user-provided function code can be executed, several initialization steps must first be performed by the FaaS platform. 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), can take a significant amount of time. These initialization overheads can be significant, and increase the function latency by orders of magnitude. These startup overheads can be ameliorated and amortized by keeping the initialized function state "warm" in server memory. However, this function "keep-alive" presents an important tradeoff between memory usage and effective function latency.

In this paper, we present the design and implementation of an 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. When deciding which server a function should be run on, our key design principle is to use *locality* to reduce function cold-starts. Running a function on the same subset of servers increases the probability that a "warm" in-memory invocation of the same function can be reused, which reduces function latency. 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 key finding is that the tradeoff between function-locality and server-load is critical to achieve good function latency and cluster utilization. This tradeoff results in a large design space of load-balancing policies and algorithms, and presents three main challenges. First, functions are highly **heterogeneous** in terms of their initialization overheads, resource footprints, and invocation frequencies. However, most classic load-balancing approaches for clusters of web-servers and data storage tend to assume homogeneous requests, and thus cannot directly be used. Second, production FaaS workloads tend to have extremely high **skew**: a tiny fraction of functions tend to account for a vast majority of invocations. This has severe ramifications for hashing-based load-balancing (such as consistent hashing), which assumes homogeneous object popularities to provide guarantees about server


*HPDC '22, June 27–30, 2022, Minneapolis, MN, USA.*

ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531459

loads. And finally, large FaaS clusters cannot assume that exact and timely information about server loads will always be available: and thus load-balancing policies must deal with stochastic and **stale server loads**.

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 [20] 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. In particular, we extend Consistent Hashing with Bounded Loads [26], 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. Our load-balancing policy is cognizant of the different utility of locality for functions based on their cold and warm running times: functions that have a high benefit from keep-alive are more likely to be run on their home servers. To deal with stale server load information and bursty function invocations, we use stochastic random loads, such that very popular functions can be spread out among more servers and minimize the herd-effect of overloading servers running bursty popular functions.

We make use of the recent equivalence between function keep-alive and caching [15], and develop a new simplified version of SHARDS [36] for our load-balancing policy for detecting and handling popular functions. Our policies are designed to be simple and practical, have a small number of user-controlled parameters, which allows them to be a drop-in replacement for the default load-balancing implementation in OpenWhisk [2].

Prior work in serverless computing has largely focused on optimizing performance on a *single* server using various cold-start mitigating mechanisms and policies [15, 35]. 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 extreme skew in function popularity.

In summary, 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. Our resultant load-balancing policy, Consistent Hashing with Random Load Updates (CH-RLU), is simple, and tackles practical challenges of highly heterogeneous functions, bursty workloads, and stale/imprecise load information on a large cluster of servers.
(2) We implement and evaluate our CH-RLU policy in OpenWhisk. It provides more than 2.2× reduction in average function latency.
(3) We conduct an empirical investigation into OpenWhisk's performance at various load levels, and find extremely high jitter due to resource contention (latencies can increase by more than 10×). With the help of our optimizations, OpenWhisk can serve 5× more traffic.

| 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 |

**Table 1: FunctionBench [22] 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.**

(4) We evaluate various load and locality-sensitive policies in a discrete-event simulator using the Azure function traces [30], and find that CH-RLU can outperform even a omniscient, impractical, online greedy approach by more than 20% under a wide range of conditions.

## 2 BACKGROUND

### 2.1 FaaS Function Execution

**Function Initialization Overheads.** The Function-as-a-Service computing paradigm sees providers running user code on-demand when a request comes in, and importantly deciding *where* it should run. Each invocation must be run in isolation from other concurrent and co-located invocations, and thus 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 [3]) created on the server that runs the invocation. Creation time for both choices can be significant, adding latency to the in-flight request.

Many techniques have been proposed to reduce the initialization overhead from such *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. Keeping the function "warm" thus allows a provider to amortize the startup cost across future invocations.

The warm and cold time for different functions from the FunctionBench [22] workload suite are shown in Table 1. The table shows the total execution latency when these functions are run on OpenWhisk. The cold times, because of the large initialization overheads, are $1 - 10\times$ larger than the warm running time, which occurs when the function is run in an already initialized container which is cached in memory.

**Keep-alive For Reducing Cold-starts.** Servers cannot keep all the functions routed to them in memory indefinitely, and 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 here, use a time-to-live approach that *evicts* a function from memory if it isn't used within a certain time window. More recent advanced techniques based on LRU and GreedyDual [8] caching algorithms make the decision based on the functions

startup time, memory footprint, and invocation frequency when making evictions [15].

However, the benefits of locality have been investigated primarily at the *single-server* level. Most real-world FaaS deployments use a large cluster of servers sitting behind a load-balancer. Tailoring the load balancing algorithm to compliment 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 and routing a function to the same server preserves locality, but at the risk of a highly overloaded server. Since function latencies also depend on the load on the server, this naïve policy would be sub-optimal, especially if the workload consists of functions with highly variable popularities and running times. We shall elaborate on this further in the next section, and show that the tradeoffs imposed by FaaS requires a new class of load-balancing approaches.

## 2.2 Load Balancing

Managing the load of a cluster of servers is a common problem in distributed computing systems. Load-balancing policies typically rely on some notion of "load" of a server, such as the number of concurrently executing tasks, length of the task-queue, cpu-utilization, etc. The first broad class is *compute-oriented* load-balancers, typically used for short-running tasks and queries. Load-balancing for computational tasks is common in scenarios like web-clusters [21]. In these systems, the tasks can be executed on any server, servers in a cluster are largely fungible, and the task performance largely depends on the server-specific cpu-utilization at the time.

Load-balancing techniques have received significant theoretical attention (especially using queueing theory), as well as many practical systems [11]. 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) [27], and distributed key-value stores (such as Amazon Dynamo [11]) 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 to ensure locality is Consistent Hashing [20, 21]. Objects are mapped to servers based on some object id or key. Consistent hashing preserves object-server mapping even in the face of server additions and removals, which improves locality. 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.

OpenWhisk uses a modified consistent hashing algorithm for its default load balancer. As functions are sent to servers, their expected memory footprint is added to a server-specific running

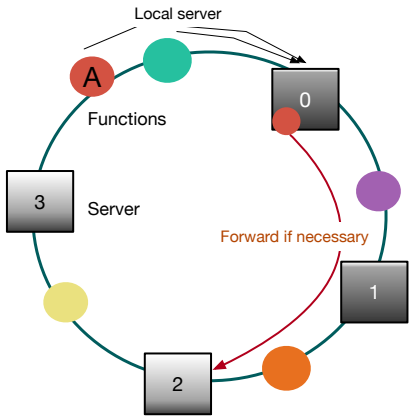



**Figure 1: Consistent hashing runs functions on the nearest clockwise server. Functions are forwarded along the ring if the server is overloaded.**

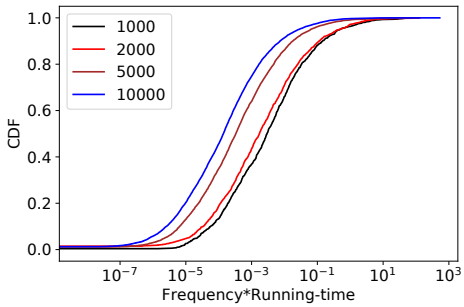



**Figure 2: Function load is very heavy tailed (note the log X axis). Each line represents a different random subset and associated subset size from the Azure function trace.**

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 the performance can be 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 shall demonstrate in the next section.

# 3 CHALLENGES IN FAAS LOAD-BALANCING

Load balancing in FaaS clusters represents a unique set of challenges which we describe in this section. We motivate our observations using the Azure function trace [30], as well as empirical performance measurements conducted using OpenWhisk.

## 3.1 Function Heterogeneity and Skew

For locality-sensitive load-balancing techniques to be effective, it is important for each function to impose roughly similar load on the system. However, functions vary widely in their frequency of

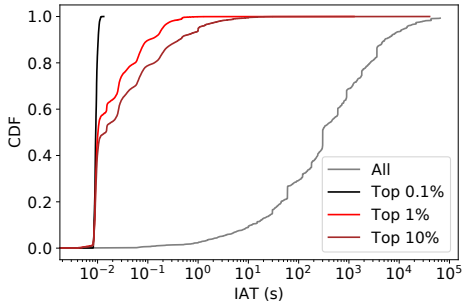



**Figure 3: Inter arrival times of popular functions can be extremely low, and show a very wide variance (note the log-scale of the X axis).**

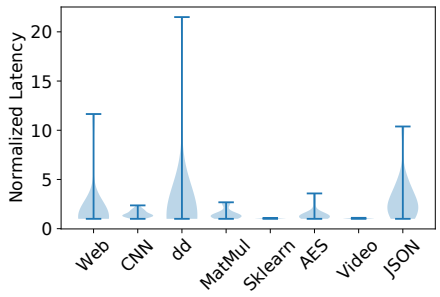



**Figure 4: Functions' warm latencies vary widely even under no system load, due to OpenWhisk jitter.**

invocation as well as their running time. The running-time heterogeneity of functions can be seen in Table 1, which shows that the times can 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 × 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 2 orders of magnitude more resources than the average. Thus, with classic consistent hashing the servers handling the heavy functions will be extremely overloaded, which will contribute to severe function slow-down due to resource contention on the servers.

## 3.2 Bursty Invocations

The second challenge is that the function arrivals can be very bursty and can vary widely by function. Figure 3 shows the inter-arrival-time (IAT) distribution computed from the Azure functions dataset. We see 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. This heavily skewed workload also has significant ramifications for Load Balancing, since we must be able to handle highly bursty functions, as well as the long tail of infrequently invoked functions. This fairness in function handling is thus an important challenge in FaaS load balancing.

## 3.3 Function Performance and Server Load

Our goal is to minimize the total end-to-end function execution latency. Unlike classic data-oriented load-balancing, function execution latency can be highly sensitive to the server load. That is, running on an overloaded server (even if it is a warm-start), can result in significant latency increase and performance degradation.

Function performance can be affected by many factors such as the number of concurrently running functions, the CPU utilization, the load-average, interference due to other co-located functions, etc. The slowdown in a processor-sharing system due to system load has been well modeled. Queueing theory approaches for G/G/PS systems approximate the running time of a task to be proportional to $1/1-\rho$, where $\rho$ is the system utilization/load.

However, function heterogeneity and their execution characteristics presents many challenges in modeling and understanding their performance. We have found that the container initialization and other OpenWhisk overheads, *even for warm starts*, can be a significant source of latency, slowdown, and jitter.

**Function Jitter.** To understand how system conditions can affect runtime of our functions, we first track their end-to-end latencies while the system is empty. In Figure 4, we run each function repeatedly until we have 15 warm runs, then normalize those warm times by the minimum run and plot the results as a violin. Surprisingly most functions have wildly inconsistent runtimes, ranging from 2x to 20x! OpenWhisk traverses a complicated code path with several network hops in order to run user code, even on a warm start. We recorded the minimum time for all this OpenWhisk overhead to be 0.015 seconds, the *average* was a shockingly high 0.5 seconds, and the $99^{th}$ percentile reached 5 seconds. These high system overheads are largely caused by write-contention on the shared databases OpenWhisk maintains for tracking functions and their results. This high-variance system overhead most strongly affects short running functions, but negatively affects everything in the system. Such significant jitter motivates our stale-load aware load balancing policy, which we develop in the next section.

**Load-sensitivity.** Furthermore, we have found that *different functions are affected by server load differently.* Figure 5 shows the correlation between function latency and the Linux load-average of the server for different function types. The load-average is normalized to the number of CPU cores: thus a load-average of 2 in the figure for the 16-CPU VMs corresponds to a Linux load-average of 32. The load on the servers was increased by increasing the number of concurrently executing functions of the same type.

We see that in general, as the server load increases, so does the latency of the function invocations. Each point in the scatter-plots of Figure 5 represents a unique invocation, with the latencies normalized to the lowest execution latency observed for that function.

The AES encryption function (Figure 5a) shows a gradual increase in latency as the load increases. Surprisingly, the effect of load is minimal: the latency increases by "only" 2x even at a 10x

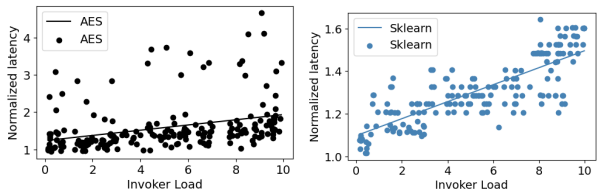



**Figure 5: Latency increases due to system load, but is function-dependent.**

load. The longer-running ML training function (Figure 5b) also sees a correlation between server load and its end-to-end latency. However, it's latency variance is lower because the longer running time (50 seconds) hides the variable OpenWhisk overhead. Both functions presented here have the highest correlation between system load and latency, yet themselves do not have a high correlation. Thus scheduling on an overloaded server can degrade function performance, it must be weighed against the performance penalty of a cold start.

# 4 LOAD-AWARE CONSISTENT-HASHING

In this section, we describe the load-balancing algorithm which is locality, stale-load, and burst aware. We assume a cluster homogeneous servers, and 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.

## 4.1 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. In the rest of this section, we describe our approach.

## 4.2 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 [26] (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 overloads, but also improves locality, *even in overload scenarios*. Forwarding along the ring has the advantage that even if function is not run on its "home" server, subsequent invocations that "overflow" still have a high warm-start probability on the servers on the overflow chain. The warm-start probability is highest on the home server, and decays the farther the function is from its home server. This is more beneficial than alternative techniques such as Consistent Hashing with Random Jumps [7], which do not preserve locality and instead forwards to randomly chosen least loaded servers.

## 4.3 Server Load Information

Server load is a key metric in load-balancing policies. We need to be able to determine the *relative* suitability of one server over another, and thus many existing metrics can be used to provide information about server loads. Simple metrics such as number of running functions are insufficient, since functions can have highly variable execution times. OpenWhisk currently uses occupied-memory used by active/running invocations as a proxy for load, and is unsuitable for the same reason. Both these metrics fail to capture CPU loads and lead to scalability issues when used by the load-balancer.

Instead, we primarily rely on *system-level* load metrics, such as the standard Linux 1-minute load-average. In addition to CPU utilization, this also captures the I/O wait due to cold-starts, and provides a more realistic measure of load. Traditional Linux load-average estimates the total number of processes running and ready-to-run, and we normalize the load-average by the number of CPUs. Thus, a load-average of 8 on an 8 core server (discounting hyper-threading) is normalized to 1.

An important practical consideration is that load information is often *stale*, with the degree of staleness ranging from a few seconds to several minutes. For instance, because the Linux load average is an exponential moving average, it is slow to change. Furthermore, load monitoring and reporting has delays due to how frequently the metrics are gathered at the local server, and how often they are made available to the load-balancer. We use a simple publish-subscribe-like system, where individual servers periodically (every

5 seconds) push their load information, and the load-balancer uses these published loads to make all scheduling decisions.

## 4.4 Why CH-BL Is Insufficient

The high computing load of functions, their bursty nature, and the staleness of loads, are the three major challenges to Consistent Hashing with Bounded Loads [26] that the original algorithm is not designed to meet. There are a few practical considerations and key differences between simple object/storage caching and function execution: 1. CH-BL does not take into account the heterogeneity in running times and memory size of the objects (i.e., functions). 2. The implicit CH-BL performance model is binary: running-time is assumed to be uniform as long as servers are under the load-bound. 3. The server loads evolve as a result of the actual function execution and are not just uniformly incremented as in the original algorithm. Object deletions are also not handled explicitly: we let the lazily computed load average determine whether a server meets the load-bound or not.

*Importantly, we do not assume complete and consistent state information about the servers.* Omniscient knowledge of the execution state of all functions running all servers can certainly be leveraged effectively to run functions on the most suitable server. However, such maintaining such global knowledge is expensive and impractical as far as storage consistency and latency are concerned. Thus, we are striving for load-balancing policies which are robust to stale, incomplete, and coarse-grained information about server states. In the rest of this section, we shall show how the above three limitations of CH-BL can be overcome in FaaS load-balancing settings.

## 4.5 Incorporating Function Performance Characteristics

Different running time and performance characteristics of functions can be incorporated into consistent hashing. *The key problem is to determine when and which function to forward.* The forwarding policies need to be cognizant of the warm and cold running times, and the sensitivity to load of different functions.

Assume a load-bound of $b$, the warm time of a function is $w$, and the cold time is $c$ (slow-start). The current or the home server will be "0", and the next server in the ring that the function may be forwarded-to will be denoted by "1". Running it on the "home"/local server will result in expected time $E[T_0] = (p_0 w + (1 - p_0 c)S(L_0)$, where $p_0$ is the cache-hit/keep-alive probability, and $S(L_0)$ is the slowdown in function if the load on the server is $L_0$. When a function in invoked the load balancer has the choice to either run in on the home server or forward it to the next server, where it is less likely to be found in the keep-alive cache, because the reuse-distance is much larger for the servers down the chain. Therefore we can compute the forwarding regret, $E[T_0]/E[T_1]$.

The properties of bounded-loads allows us to easily compute this value. The probability of being forwarded is small, and is $1/b$ based on Lemma 4 of [26]. The reuse-distance of the function, and hence the hit-rate on the original/home server will be larger: $p_0 > p_1 * b$. Based on our empirical observation of sub-linear performance decrease due to load (Section 3.3), in the worst case, the home server will be overloaded and alternative server will not be, and hence the ratio of slowdowns, $S(L_0)/S(L_1) > b$. Minimizing the regret, we get that the function should be forwarded if $L > cb/w$. Thus, the effective load upper-bound is *increased* by a factor of cold/warm time, allowing us to run more functions per server. In our empirical evaluation, we will show that this can significantly improve performance over plain CH-BL with a function-agnostic constant load-bound. If the cold and warm times of a function are not available, then they are assumed to be equal, thus this degrades to classic function-agnostic bounded-loads.

## 4.6 Handling Bursts

Functions come in a variety of frequency classes and are also prone to unpredictable burstiness (i.e., very low inter-arrival-times for a short duration). Identifying these bursts and both keeping latency for such "popular" functions low and preventing them from negatively impacting co-located functions is critical. We have found that handling overload conditions is a key requirement and can significantly affect the tail latency.

Bursty function invocations result in two main problems. First, they cause an increase in server load beyond the actual load-bound, because load is only lazily tracked. The delayed load information can result in a popular function completely overwhelming a server, causing load "hotspots" in the cluster. The second problem is that in extreme cases, the inter-arrival-time is less than the function latency, causing concurrent invocations. Even if these concurrent invocations are run on a "local" server with the function present in the keep-alive cache, there will still be cold-starts, since each invocation must run in its own container.

Our solution to these two problems caused by bursty invocations is to detect popular function bursts, "spread" these invocations around multiple servers to prevent cluster hot-spots, and use stochastic/random load updates to introduce randomness into the load-balancing.

*4.6.1 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 first 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. Recall that reuse distance is the number of *unique* objects accessed, whereas inter-arrival-time is simply the difference in wall-clock times.

Our solution to identifying popular functions and function bursts is inspired by the popular SHARDS [36] 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. Because 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)
```

*4.6.2 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 many servers without causing cold starts. A fair load balancing algorithm must spread popular functions to ensure QoS for less frequent functions. Because load information is stale, adhering to locality and load can result in servers facing a herd-effect. Randomization is a powerful strategy to ameliorate such effects, however, we must use it judiciously because of the strong effects of locality in FaaS load-balancing.

Our solution is to introduce random forwarding (along the ring) which is 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 function arrival rate on the server ($\lambda$). Specifically, the $L_{\text{noisy}} = L + \mathcal{N}(\mu = \lambda, \sigma = 0.1)$, where $\mathcal{N}$ is a Gaussian random variable. For popular functions, we compare the $L_{\text{noisy}}$ 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.

## 4.7 Putting it all together: CH-RLU

Our overall policy, Consistent Hashing with Random Load Updates (CH-RLU), combines all the previously described techniques and insights. When a new invocation arrives, we query the popular IAT threshold to determine what class of function it is. Functions are distributed via Algorithm 2, which combines the use of SHARDS for popularity detection, cold and warm times for increasing the effective load-bound, and noisy loads. We bound the cold/warm ratio

**Algorithm 2** Random Load Update Forwarding Function

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

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.

Forwarding along the chain has diminishing returns of locality, and if the function gets forwarded more than max_chain_len times, we simply run it on the least-loaded server. If the least loaded server is also overloaded, we drop the function. We have also implemented a simple PID controller with hysteresis for horizontal scaling, by using server load averages as the input control signal. This horizontal scaling is conservative, with a large dead-band of 5 minutes, and scaling is triggered only if the at least 50% of the servers are overloaded. As we shall show in the empirical evaluation, CH-RLU significantly reduces the variance in the loads among servers, and thus is more amenable to this horizontal scaling policy.

# 5 IMPLEMENTATION

We have implemented our consistent hashing with random load update (RLU) policy and other load-balancing policies in OpenWhisk, a popular FaaS system. Our changes amount to more than 1,700 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 which significantly improve OpenWhisk performance and scalability by more than 4×.

Our policies are implemented by modifying the load-balancer module of OpenWhisk (see Figure 6). CH-RLU is implemented by modifying the existing OpenWhisk "container sharding" policy, which also uses consistent hashing, and forwards functions using only available memory as the load metric. We use OpenWhisk's existing consistent hashing implementation, permiting an "apples to apples" comparison, and also making 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 [15].

The 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 (every 5 seconds) captured and stored

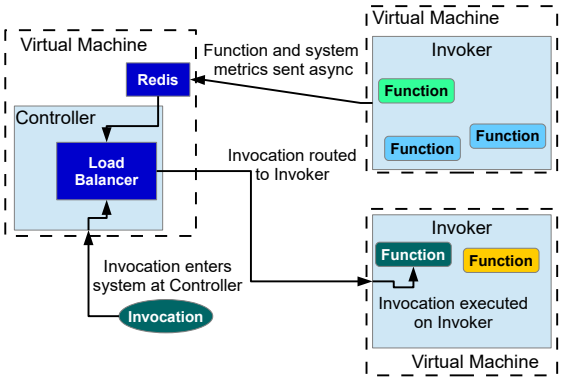



**Figure 6: System diagram of relevant OpenWhisk components and communication used to schedule and run function invocations.**

in a 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, and they are suitable as user-inputs.

## 5.1 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 their details and our actions to overcome them, hoping that the fast-growing serverless computing research field can benefit from our lessons.

In our experience, the primary source of scalability bottlenecks is running Docker containers concurrently. We found significant contention in `dockerd`, Docker's control daemon which handles all the container lifecycle events. Even at moderate loads (normalized server load average close to 1), high dockerd contention can increase tail latencies by *several minutes!*

Currently, OpenWhisk **pauses** each container after function execution, which prevents it from being scheduled by the CPU. It then resumes the container before running the next invocation of the same function (assuming a warm start). Thus each invocation requires these two additional (pause/resume) events to be handled by dockerd, which results in 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. Therefore, we remove these redundant pause/resume operations to reduce dockerd contention. This reduces the OpenWhisk overhead by 0.2 seconds *per-invocation* on average. More importantly, by reducing dockerd contention, we were able to run a much 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. We found this to be a very expensive operation because of Linux network stack overheads (several 100 ms), and because of dockerd's userspace lock (futex) contention for its networking database. We found that 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/servers 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 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 to communicate with the controller/load-balancer directly.

**CPU limits.** OpenWhisk uses the *--cpu-shares* option to set container CPU priority. This has an unintended consequence of allowing functions to use more than one CPU core while running. Major FaaS providers constrain functions to a single core unless they have extremely high memory allocations (<1 GB). In order to stay in line with providers and prevent outsized impact on system load from some functions, we use the *--cpus* flag instead to assign each function no more than one CPU.

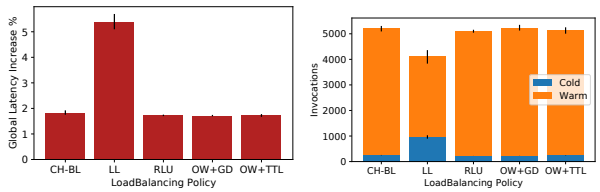
Together, these performance optimizations have allowed us to run OpenWhisk on invokers that are 4× larger, and serve more than 6× the load, without dropping functions due to timeouts. We plan to upstream all these performance optimizations in OpenWhisk to provide a higher-performance and lower-jitter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

# 6 EVALUATION

In our evaluation we present the effectiveness of our load-balancing policy (RLU) using our 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.

## 6.1 Evaluation Environment

**HW and SW Config.** We run OpenWhisk in a distributed mode across 9 VMs. 8 invokers are each in their own VM with 16 vCPUs and assigned to use 32 GB RAM for hosting functions. The final VM hosts the controller, load-balancer, and remaining services, with 12 vCPUs and 50 GB RAM to ensure it is not a bottleneck. Metrics about system load were captured every 5 seconds by calling *uptime* on each invokers 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




**(a) Global Latency Impact** **(b) Invocation Throughput**

**Figure 7: Latency and throughput under low-load. Locality-agnostic least-loaded policy has more cold starts and a higher impact on latency.**

policy changes to the invoker eviction policy, but use the changes from FaasCache [15] for eviction decisions on the invoker.

**Contenders.** In addition to our proposed load balancing policy, we compare 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 implement two other load balancing polices for comparison: least loaded (LL), and consistent hashing with bounded loads using stale load-averages (CH-BL). 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 $[2-8]$. This is also shown earlier in our latency vs. load analysis in Section 3.3.

**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., slow-down). 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 [22] 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 mimicing the arrival frequencies of the Azure trace [30]. Our load is generated using the closed-loop load generation tool Locust [24] to invoke functions, running 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.

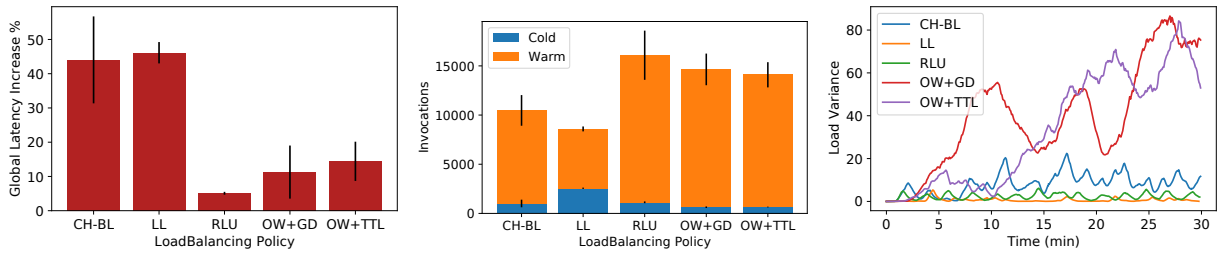
## 6.2 Load-balancing Performance

When we run them under **light load** in Figure 7, 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 excess cold starts, giving us a "lower bound" on load balancing. The low 1-2% latencies in Figure 7a we see here 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 it's lack of locality causes excessive cold starts as evidenced by the high number of cold starts in its invocation results detailed in Figure 7b.

Next we run the policies under our **heavy load** scenario, and get a clear distinction between how each of them performs. The two versions of OpenWhisk in Figure 8a only increase latency by 11% and 14% respectively which is rather good. They cannot complete with RLU whos 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.

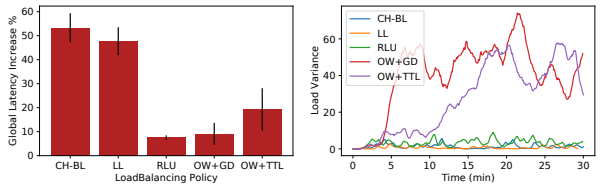
The wide gap between policies can be understood by comparing the load variance between their workers (Figure 8c). OpenWhisk's default policy is to only move a function to another server if the "home" one does not have available memory to run it. While very good for locality (getting fewer cold starts than RLU in Fig 8b), it creates severe imbalance on the worker loads. A few 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 actually does the best at keeping equal load amongst workers, but at the cost of poor locality.

*6.2.1 Handling Bursty Traffic.* Next we take two different bursty workloads to see how the polices handle changes in invocation patterns. The first uses the same closed-loop load generation but adjust the weights by which functions are invoked. Every 30 seconds two of the top weighted functions are chosen to become bursty, and have their weights set much higher. At the end of a burst their weights are returned to normal and another two functions are chosen. As can be seen in Figure 9a our policy acheives a 17% lower impact on global latency than OpenWhisk with GreedyDual. RLU represents a 60% reduction to 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 in our workload have a larger effect on system load and the load balancer must be aware of this impact and either spread that 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 9b. Policies




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

**Figure 8: 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

**Figure 9: RLU improves latency by 10% compared to OpenWhisk under bursty load conditions, while keeping a low worker load variance.**

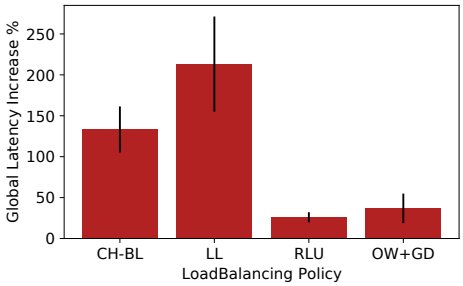



**Figure 10: Global latency impact under a 30-minute long rising burst load from an open-loop generator. RLU reduces latency by 17% compared to OpenWhisk.**

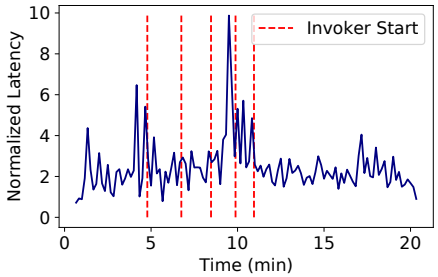



**Figure 11: The average normalized function latency over time for a dynamic workload. New invokers are launched at the dashed lines, keeping the latency in check.**

that monitor load, our RLU, CH-BL, and least loaded keep tighter control on load variance.

The second busrty load is a 30 minute long-rising burst, starting with just a few invocations per second and reaching a sustained peak of roughly 18 invocations per second at roughly 25 minutes. We generated this load with a custom open-loop load tool that fires invocations but does not block waiting for completion. New invocations are continually fired in a preset pattern of function types and times. The global latency impact of this final scenario can be seen in Figure 10. 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. RLU's 25% increase in global latency is still significantly better, 30% lower, than OpenWhisk. Our policy is able to make ideal choices for function placement under a varient of realistic workload scenarios.

*6.2.2 Scaling.* Lastly we want to demonstrate how our policy reacts to scaling the number of workers as demand increases. We start our cluster with only 3 invokers and increase applied load up to the heavy load scenario above. Rather than starting with the 120 threads of the heavy load with this smaller cluster, we adjust the scenario to start with a single thread and add a new one every 6 seconds, reaching the final thread count at about minute 12.

As the average invoker load increases, the controller activates a new worker and starts directing work towards it. New workers are kept under the load bound of 6 and see load similar to our previous experiments that had a constant load. Figure 11 shows the function latencies (normalized to respective min. warm times). Preceeding each worker being started is a rise in overall latency, which then falls after the invoker has come online and starts taking additional load. Thus, our horizontal scaling is able to dynamically keep the function latency in check, even though it only uses coarse-grained server load metrics.

*6.2.3 Load-balancer Overhead.* More complicated routing decisions naturally mean they 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 5. Even so,

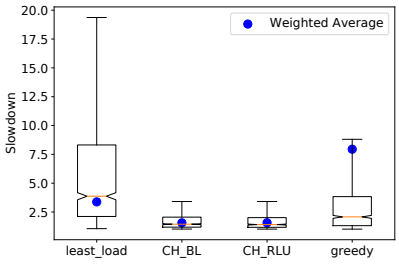



**Figure 12: [Simulated] Function latency distribution.**

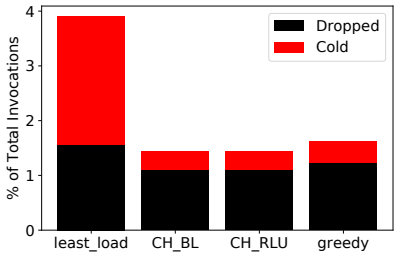



**Figure 13: [Simulated] Cold and dropped functions.**

RLU is on significantly slower making individual routing decisions, taking on average 1242.6$\mu$s to OpenWhisks' 472.3$\mu$s. Such times represent a fraction of the time spent per-request by the system and is made up for by our more optimal placements.

## 6.3 Simulation Evaluation

To investigate the performance of various load-balancing policies at larger scales, we use a simulation approach. We have developed a discrete-event simulator, which plays a function workload trace, and emulates the various aspects of function execution and slowdown: slow/warm starts, slowdown due to concurrent processing by emulating a G/G/k queueing system on each server, and various load metrics (emulating Linux exponentially decaying load averages, stale loads, real-time loads, etc.). The simulator allows us to implement different policies using information that would not otherwise be available on a real system: accurate function cold and warm times, instant load information, etc. The function running times are computed by adding the actual provider-captured running times to the OpenWhisk and Docker startup overheads that are empirically measured and modeled. This, when combined with queueing delays, captures the overall slowdown due to concurrent processing. The simulator is implemented in Python in about 3,000 lines of code.

We run the Azure trace with 1000 randomly chosen functions spanning almost a million invocations, and compare different load-balancing policies. This workload is highly bursty and is characterized by the Figures in Section 3. We have implemented a "online greedy optimal" policy that considers *all* servers when running functions, by picking the server with the lowest expected running time (based on Section 4.5). This would be impractical and unrealistic to implement, since it needs accurate information about every function's keep-alive status on every server, and an accurate

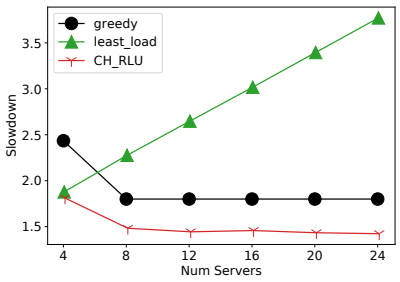



**Figure 14: [Simulated] Function latencies as cluster size is increased. Least-loaded performs *worse* because its locality and cold-starts become worse as more servers are added.**

model of function performance at various loads. Nevertheless, it provides an optimistic baseline: our consistent-hashing based policy ($CH - RLU$) is significantly simpler, only considers a small subset of servers, and does not require omniscient cluster state.

Figure 12 compares this greedy policy with a locality-agnostic "least-loaded" policy that is popular in web-clusters, and the two consistent-hashing based policies. The figure shows the function slowdown factor for each function, as well as the global average slowdown. Slowdown is defined as the ratio of function's execution time to its base warm time without any system overhead, resource contention, or queueing delays. Most functions do poorly with the least-loaded policy: the median function slowdown is almost 4×, primarily because of high cold-starts. Figure 13 compares the cold and dropped statistics for the various policies.

Returning to Figure 12, both CH-BL and CH-RLU have comparable performance, with a median slowdown of 2.4. Finally, and surprisingly, the omniscient greedy policy performs poorly: with the global slowdown approaching 7. The primary reason for this is because of the bursty nature of the workload: the greedy policy tends to pick the server with the least-loaded server that can run the warm function. However because of stale load information, we see a *herd effect* on the server, and this causes extremely high resource contention and latency, even though the number of cold-starts is small (compared in Figure 13).

Finally, we investigate performance when the cluster size changes. Figure 14 shows the slowdown for the three policies when the number of servers is increased. Importantly, the total number of computing and memory in the cluster is kept constant at 256 CPUs and 512 GB, and the size of the individual servers is changed. Thus 4 servers with 64 CPUs are compared with 8 servers with 32 CPUs, etc. This experiment is intended to capture the effects of locality: smaller number of servers may have a higher hit-rate, and a more even load-spread.

Figure 14 shows how the function slowdown for both CH-RLU and greedy decays as the number of servers is increased. This is because larger servers see heavier lock and other resource contention, and thus while they may exhibit better locality, the load-induced slowdown dominates. This has important ramifications for large FaaS providers, since they can continue using smaller servers for running functions, and expands the utility of small deflatable/harvestable VMs for running colocated functions [39].

CH-RLU reduces the function slowdown by 20% compared to the greedy approach, across a wide range of cluster configurations in Figure 14. Interestingly, least-loaded's performance worsens with increasing number of servers and fragmentation. The main culprit is worsening locality. With a small number of servers, least-loaded can get "lucky" and score a keep-alive cache-hit. These fortuitous warm-starts get less probable with an increasing number of servers.

## 7 RELATED WORK

**FaaS Resource Management.** The initialization overheads of serverless functions and their repeated invocations have spawned a great deal of research into optimizing their resource management. Recent surveys [14, 18, 23, 25, 29] provide an overview of the challenges and solutions in this very active research area.

Reducing the overhead of serverless functions through various systems and virtualization-level mechanisms and optimizations [3, 4, 6, 12, 13, 35]. Locality for FaaS resource management has been explored in the form of function keep-alive policies [30]. Our work builds on and uses the caching-based Greedy-Dual policy from FaasCache [15]. Single-server environments have been the focus of these mechanisms and policies: we have made an initial attempt to understand their interactions in a distributed cluster context. Inter-function dependencies can also be used for predictive resource management and reducing function communication and startup costs [10, 16, 31]: incorporating these policies into our load-balancer is part of future work.

**Function Load Balancing:** Package-aware load balancing [5] identifies and uses function code dependencies (software packages) as an important source of data locality. While this is an important factor, we focus on in-memory locality of kept-alive functions, since memory capacity is much smaller than permanent storage and caching functions in memory has a very large performance impact. CPU contention and interference is a major source of performance bottlenecks for co-located functions, and adjusting CPU-shares using cgroups can provide significant benefits [32–34]. The load-locality tradeoff we explore is complementary to these CPU scheduling optimizations. The repetitive nature of functions and their workflows can also be used to improve resource utilization and latency [9, 19, 28, 37]: our load-balancer is stateless for the sake of simplicity and can be enhanced with these techniques if necessary.

The tradeoff between locality and performance has also been explored in the context of delay scheduling [38] for data-parallel applications like MapReduce. Load-balancing is seen as a "dispatch" problem in queueing theory, and the FaaS cluster system most closely approximates G/G/PS, since the arrivals and service times are not markovian. Techniques such as "join the shortest queue", and "least work left" [17] have been shown to be effective. The online-greedy policy evaluated in the previous section closely approximates least-work-left. However, it is difficult to implement in practice since the running times of functions is hard to predict due to their volatile arrival distribution mixtures and high variances in running time due to various system interference effects.

## 8 CONCLUSION

In this paper we have explored the tradeoff between locality and load for serverless functions. Our CH-RLU policy can tackle the challenges of function heterogeneity, workload skew, bursty workloads, and stale loads. We enhance consistent hashing to yield a simple and practical load-balancing policy. Empirical evaluation shows substantial improvements in function latency (by more than 2×) compared to current OpenWhisk.

## REFERENCES


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

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

[3] 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.

[4] 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.

[5] Aumala, G., Boza, E., Ortiz-Avilés, 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.

[6] 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.

[7] 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.

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

[9] Daw, N., Bellur, U., and Kulkarni, P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, pp. 356––370.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L., and Iosup, A. Serverless applications: Why, when, and how? *IEEE Software 38*, 1 (2020), 32–39.

[15] 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.

[16] 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.

[17] Gupta, V., Balter, M. H., Sigman, K., and Whitt, W. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation 64*, 9-12 (2007), 1062–1081.

[18] Hassan, H. B., Barakat, S. A., and Sarhan, Q. I. Survey on serverless computing. *Journal of Cloud Computing 10*, 1 (2021), 1–29.

[19] Hunhoff, E., Irshad, S., Thurimella, V., Tariq, A., and Rozner, E. Proactive serverless function resource management. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing* (2020), pp. 61–66.

[20] 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.

[21] 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.

[22] 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.

[23] Li, Z., Guo, L., Cheng, J., Chen, Q., He, B., and Guo, M. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys* (Jan 2022).

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

[25] Mampage, A., Karunasekera, S., and Buyya, R. A holistic view on resource management in serverless computing environments: Taxonomy, and future directions. *arXiv preprint arXiv:2105.11592* (2021).

[26] 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.

[27] 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.

[28] Przybylski, B., Żuk, P., and Rzadca, K. Data-driven scheduling in serverless computing to reduce response time. *arXiv preprint arXiv:2105.03217* (2021).

[29] Raza, A., Matta, I., Akhtar, N., Kalavri, V., and Isahagian, V. Sok: Function-as-a-service: From an application developer's perspective. *Journal of Systems Research 1*, 1 (2021).

[30] Shahrad, M., Fonseca, R., Goiri, , 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. *arXiv:2003.03423 [cs]* (June 2020). arXiv: 2003.03423.

[31] 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.

[32] Suresh, A., and Gandhi, A. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International Workshop on Serverless Computing* (2019), pp. 19–24.

[33] Suresh, A., and Gandhi, A. Servermore: Opportunistic execution of serverless functions in the cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 570–584.

[34] Suresh, A., Somashekar, G., Varadarajan, A., Kakarla, V. R., Upadhyay, H., and Gandhi, A. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (2020), pp. 1–10.

[35] 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.

[36] 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.

[37] Yu, H., Irissappane, A. A., Wang, H., and Lloyd, W. J. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (2021), IEEE, pp. 31–40.

[38] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (2010), pp. 265–278.

[39] Zhang, Y., Goiri, I. n., Chaudhry, G. I., Fonseca, R., Elnikety, S., Delimitrou, C., and Bianchini, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 724–739.