Ilúvatar: A Fast Control Plane for Serverless Computing

Alexander Fuerst
Indiana University
Bloomington, Indiana, USA
alfuerst@iu.edu

Abdul Rehman
Indiana University
Bloomington, Indiana, USA
abrehman@iu.edu

Prateek Sharma
Indiana University
Bloomington, Indiana, USA
prateeks@iu.edu

ABSTRACT

Providing efficient Functions as a Service (FaaS) is challenging due to the serverless programming model and highly heterogeneous and dynamic workloads. Great strides have been made in optimizing FaaS performance through scheduling, caching, virtualization, and other resource management techniques. The combination of these advances and growing FaaS workloads have pushed the performance bottleneck into the control plane itself. Current FaaS control planes like OpenWhisk introduce 100s of milliseconds of latency overhead, and are becoming unsuitable for high performance FaaS research and deployments.

We present the design and implementation of Ilúvatar, a fast, modular, extensible FaaS control plane which reduces the latency overhead by more than two orders of magnitude. Ilúvatar has a worker-centric architecture and introduces a new function queue technique for managing function scheduling and overcommitment. Ilúvatar is implemented in Rust in about 13,000 lines of code, and introduces only 3ms of latency overhead under a wide range of loads, which is more than 2 orders of magnitude lower than OpenWhisk.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**; Software libraries and repositories; • **Computing methodologies** → *Discrete-event simulation.*

KEYWORDS

Cloud computing; Serverless computing; Functions as a service; Open source

ACM Reference Format:
Alexander Fuerst, Abdul Rehman, and Prateek Sharma. 2023. Ilúvatar: A Fast Control Plane for Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3588195.3592995

HPDC '23, June 16–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0155-9/23/06…$15.00
https://doi.org/10.1145/3588195.3592995

1 INTRODUCTION

Serverless computing, or Functions as a Service, has emerged as a key cloud abstraction which is enabling the rapid development and cloud deployment of many applications [11, 21, 53]. Functions are small, self-contained programs, whose entire execution and scaling is managed by the FaaS provider. FaaS has emerged as a major and growing cloud workload [55], and serves as the resource abstraction for a wide range of event-driven applications (such as web and API services, IoT, and ML inference), workflows [22, 41], and even throughput-intensive parallel workloads [20, 26, 27, 67].

The FaaS programming and deployment model, along with the highly heterogeneous nature of FaaS workloads, presents many fundamental performance challenges for FaaS providers. This has motivated huge and rapid strides in many areas of systems research: advances in FaaS scheduling [62], load-balancing [30], workflow-management [50], and lightweight sandboxing [24] can all improve various facets of FaaS performance by orders of magnitude.

In most cases, these FaaS performance optimizations and resource management policies are implemented and evaluated using existing popular FaaS frameworks such as OpenWhisk [4]. These frameworks are also used in real-world deployments, and thus FaaS performance research can have a large direct impact by improving and enhancing these frameworks. These frameworks provide a "FaaS control plane", which runs on top of a large cluster of servers, and manages all facets of function execution such as scheduling, monitoring, accounting, etc.

In this paper, we focus on the performance of the FaaS control plane itself, a critical but mostly overlooked component in the FaaS ecosystem. They are an important new class of middleware, and are interesting and novel from a system design, implementation, and optimization perspective. At one end, they have to handle the extreme scale and heterogeneity of functions, where the execution and inter arrival times can vary by several orders of magnitude. At the other end, they have to work with many intricately connected software components for operating system virtualization, container runtimes (such as Docker), networking, etc.

We posit that current FaaS control planes such as OpenWhisk have become unsuitable due to the rapid growth of FaaS workloads and advances in FaaS research. Historically, functions suffered very high cold-start overheads associated with initializing their execution runtimes and dependencies in a sandboxed environment (such as a container or a VM). However, techniques such as keep-alive, prefetching, VM snapshots, and specialized lightweight sandboxing have reduced the *effective* cold-start overheads, and the spatial and temporal locality of FaaS workloads means that over 99% invocations are *warm* [29]. We find that fundamental design and implementation issues hinder good performance, adding 100s of

[0] Ilúvatar is the single omniscient and omnipotent creator in Lord of the Rings.

milliseconds of tail latency even to warm-start invocations served from fully-initialized containers in memory. Due to this *control plane overhead*, function performance is now increasingly bottlenecked by the control plane itself.

To rectify that, we present Ilúvatar, our clean-slate design of a low-latency control plane for high performance FaaS research and deployments. Ilúvatar is intended as a simple, extensible, general-purpose FaaS control plane which runs functions inside OCI-compliant containers (such as containerd, Docker, etc.), and makes minimal assumptions about the workload or function sandboxing. The combination of our design principles, performance optimizations, worker-centric architecture, and carefully optimized Rust implementation, reduces our overhead to less than 3ms, more than 2 orders of magnitude lower than OpenWhisk. Ilúvatar's design tackles these fundamental performance challenges by using simple design principles: for instance, we use resource caching heavily across the control plane for mitigating the "slow path".

For regulating worker load and improving latency, we develop new function-size-aware queueing policies. Queueing in Ilúvatar provides a new principled overcommitment knob, allowing FaaS providers to improve both utilization and latency. The worker-level queue design also helps in mitigating bursts, reducing concurrent cold starts, and prioritizing functions. Ilúvatar's queue design makes it easy to implement advanced data-driven queueing and scheduling policies, which are highly appealing because of the high temporal locality of FaaS workloads.

Ilúvatar is intended to serve as a platform for empirical FaaS research, and provides a low latency, low jitter experimentation environment. It implements and introduces state-of-the-art policies for load-balancing, function scheduling, keep-alive, and provides easy to use data and control APIs for developing advanced data and machine learning driven policies. It provides a variety of resource management and overcommitment options, and supports multiple container backends (such as containerd and Docker). We have designed Ilúvatar to be modular and compatible with the recent and anticipated advances in FaaS resource management such as snapshots, overcommit, statistical learning based scheduling, etc. We also introduce a new technique for *in-situ* simulations, where Ilúvatar can double as a full-fledged FaaS simulator for protoyping and evaluating policies. Through the design and implementation of Ilúvatar, we make the following major contributions:

(1) Ilúvatar provides fast, predictable, jitter-resistant function execution using a worker-centric architecture, resource caching, and an asynchronous implementation in a non garbage collected language. Ilúvatar is 13,000 lines of Rust code. It is open source, and available at https://github.com/cos-in/iluvatar-faas .

(2) Our research novelty lies in optimizing warm starts and queueing-based scheduling and overcommitment policies for heterogeneous and bursty function workloads (such as Azure's [55]).

(3) We reduce latency overhead by up to 100× vs. OpenWhisk.

(4) We show how our worker-level queue architecture and policies can provide new knobs for controlling overcommitment, average latency, and fairness.

(5) Ilúvatar provides a reliable and extendible FaaS platform, and our performance matches the idealized Little's law model.

2 BACKGROUND & MOTIVATION

Functions as a Service (FaaS) allows users to register small snippets of function code that get executed in response to some event or trigger (such as an HTTP request, message queue event, etc.) [5, 6, 8, 53]. These functions must be stateless: a new execution environment may be created for every invocation (and can be destroyed after the function returns). The function code also contains all the necessary code and data dependencies (such as imported libraries and packages), and thus functions may spend significant time being *initialized* before the event-handling code can execute. Functions are executed inside virtual execution environments such as hardware virtual machines, OS containers like Docker, or even language-based runtimes such as javascript WASM [58]. Function initialization, i.e., creating the execution environment and resolving code/data dependencies, can take 100s of milliseconds, and this "cold-start" can significantly increase the latency of small functions [24, 29]. Initialized function sandboxes can be retained in memory, and this *keep-alive* provides faster "warm-starts" [29]. Since functions are arbitrary user-code, they are extremely heterogeneous in their execution characteristics and resource requirements. For instance, the Azure FaaS trace [55] shows that the 50th and 95th percentile of execution time can range from 1 second to 1 minute; and the inter-arrival-time from 1 second to 15 minutes respectively.

2.1 FaaS Control Planes

All aspects of function execution are orchestrated by a FaaS *control plane*, which are implemented by frameworks like OpenWhisk [4]. For using a FaaS service, the user interacts with the control plane for registering and invoking functions, tracking their status, etc. The control plane manages the resources of a cluster of servers, and schedules functions on to them based on its load-balancing policies.

In OpenWhisk, user requests for invoking a function go through a reverse proxy (NGINX) to the central *controller*, which implements, among other things, load-balancing (a variant of consistent hashing with bounded loads by default). The controller puts the function invocation request into a shared Apache Kafka [3] queue. Inside the worker, the invoker service pulls function invocations from the Kafka queue based on that worker's own resource availability. Docker containers running a Go-based control plane agent are used to isolate functions, and each worker maintains a container pool of initialized/warm containers. OpenWhisk logs function results in a CouchDB instance. Importantly, both Kafka and CouchDB are on the critical path, and add 100s of ms to invocation latency. OpenWhisk is highly modular and distributed, with many networked services. All of these, combined with the JVM GC (it is implemented in Scala), results in large and unpredictable latency spikes [30, 56], with slowdowns of more than $10,000\times$ reported [72].

2.2 Why a new FaaS control plane?

We believe that the FaaS control plane is an important component of the modern cloud ecosystem, and presents many optimization opportunities and interesting research questions in system design.

Performance. Because of its central role in coordinating all aspects of function execution, the control plane plays a major role in determining function performance. Managing the function execution lifecycle for hundreds of concurrent invocations imposes a *control*

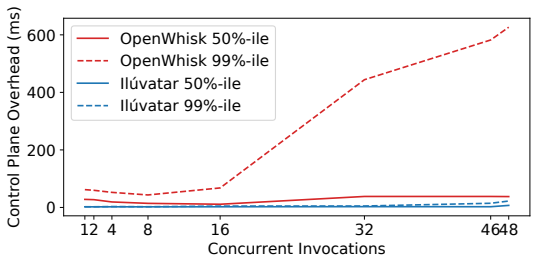


Figure 1: The latency overhead of the control plane, as the number of concurrent invocations increases. OpenWhisk overhead is significant and has high variance, resulting in high tail latency. Ilúvatar reduces this overhead by 100x.

plane overhead, and increases the end-to-end latency. This control plane overhead can be significant, and affects *all* function invocations, including and especially the "warm starts". This overhead (end-to-end latency minus the function code execution time) for the PyAES function from FunctionBench [39] is shown in Figure 1.

The figure shows the 50 and 99 percentile overheads as the number of concurrent invocations are increased. In each case, we are invoking the function repeatedly in a closed-loop, and concurrent invocations are achieved by using multiple client threads. All invocations are warm starts. The experiment is run on a 48 core server (more details in Section 6), and the figure thus shows the performance at low and medium load conditions.

From Figure 1, we can see that the OpenWhisk latency overhead is more than 10ms, which is already a significant increase in latency for small functions which dominate real-world FaaS workloads. Worryingly, the 99 percentile overhead is much higher, and rises to as much as 600ms. We also see strange inversions in the scaling behavior: the overhead reduces for certain load-levels, and then increases again. This high overhead, high variance, and uncertain scaling behavior, results in many challenges for FaaS *providers*. Due to these issues, low-latency functions see severe performance degradation, and resource provisioning and capacity planning becomes harder due to the high variance and performance unpredictability.

Some of these latency overheads are an artifact of the architecture. The shared Kafka function queue can be a major bottleneck; and there are no explicit backpressure or load regulation mechanisms, which is compounded by the CPU overcommitment. For the sake of comparison, the figure also shows the latency overhead of Ilúvatar in the same environment. We are able to achieve a per-invocation mean overhead of less than 2ms for almost all the load conditions. Importantly, the tail overhead is also small: less than 3ms for less than 32 concurrent invocations, rising to 10ms when the system is saturated.

To emphasize, for a median function in the Azure workload which runs for 500 ms, OpenWhisk can increase its latency by 100%. Thus, the control plane plays a crucial role in function performance. We note that these are the best-case warm-start latencies, when the function's containers is fully initialized and in memory. Since function cold-starts impose such a major performance penalty (increasing latency by more than 10×), mitigating them has been a major research focus. However, because of temporal and spatial locality of access, caching and prefetching techniques can be extremely effective, and the cold-start rate is often less than 1% of all invocations [29]. The majority of invocations are thus "warm", where the performance is dominated by control plane overheads.

System Design. As evidenced by the OpenWhisk architecture presented earlier, FaaS control planes are large, complex distributed systems. Due to the continually evolving needs of FaaS applications and emergence of new sandboxing techniques (such as lightweight VMs like Firecracker [12]), they are sandwiched between the scale and heterogeneity of FaaS workloads on one hand, and the deep stack of OS and virtualization components on the other.

For instance, systems for running web services or microservices do not have to deal with large and highly variable sandbox management overheads, nor with highly heterogeneous request sizes. For reducing tail latency, these systems can often rely on the OS CPU scheduler for processor sharing, can do CPU allocation at very fine granularity [35], use queueing theory techniques [47], etc. At the other extreme, for longer running containers and VMs, their control planes, like OpenStack or Kubernetes face a much lower rate of VM arrivals and departures. and can do careful and "hard" resource allocation using bin-packing [23].

Functions are highly heterogeneous, and can be seen as both latency-sensitive web requests *and* large containers requiring significant system resources for several seconds. FaaS control planes thus have to do *both* low-latency allocation *and* pack CPU and memory resources on their servers carefully to maintain high system utilization. Thus FaaS control planes are one of the more perfect microcosms of challenges in resource management and control in large scale distributed computing.

A clean-slate control plane design helps us investigate the fundamental performance tradeoffs and challenges in this fast-evolving ecosystem. Our new implementation also helps to identify the current performance bottlenecks and new avenues of OS optimizations.

Platform for Experimental Systems Research. Performance-focused FaaS research is already challenging due to the extreme scale and heterogeneity of the workloads. These challenges are compounded by existing control planes like OpenWhisk that are unfortunately highly unpredictable. The control plane jitter and the extreme bimodal cold vs. warm latencies makes it difficult to do reliable and reproducible research [42], and subtle environmental and configuration effects can mask the true effects of new research optimizations. However, it continues to be a key component in developing and evaluating FaaS research [14, 15, 29, 30, 55, 62, 70]. With OpenWhisk, function performance can be severely affected by a myriad of configuration options, such as insufficient memory for CouchDB, networking configuration, Docker configuration, etc.

Given the importance of the control plane, we want *predictable* performance to a large degree. In our experience, research in FaaS is often hindered by the large overheads and complexity of existing control planes. Thus, Ilúvatar is designed from the ground-up to be lightweight and provide predictable performance under different conditions. Our system implementation can potentially accelerate the development of new optimizations, clarify our understanding of performance characteristics of this relatively new stack, and provide a platform for robust experiments. With a robust platform, the community can share knowledge and advances, while being able to compare against a well-known and trusted baseline.

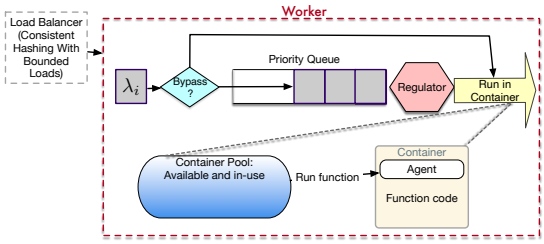


Figure 2: Ilúvatar has a worker-centric architecture. A per-worker queue helps schedule functions, and regulate load and overcommitment.

3 ILÚVATAR DESIGN

Ilúvatar's design is guided by our experience of OpenWhisk performance, and by our goals of providing predictable performance, modularity, and a platform for reliable FaaS research.

3.1 Architecture and Overview

The Ilúvatar control plane is spread out across a load balancer and the individual workers, and sits above the containerization layers. We intend for Ilúvatar to be the narrow waist [46] in the FaaS ecosystem: with optimizations for DAG scheduling [71], state handling [61], and horizontal scaling [30] implemented above it, and sandboxing and containerization below it. This architecture was motivated by the key question: *Can fast FaaS control planes be implemented with strict layering and separation of concerns?*

We have found that most of the control plane overhead is in the workers, and hence optimizing the worker performance is our major focus. Our architecture is **worker-centric**, and places more performance and load-management responsibility on the individual workers, instead of a more "top-down" centralized approach favored by prior work such as Atoll [60] and others [36, 37]. Top-down resource management requires a consistent global view of the cluster, and is complementary to our work. Predictive techniques for load-balancing, prefetching, scheduling, function-sizing can all be effective, but we want to explore the performance characteristics and limits of *reactive* control planes that work with unmodified container runtimes.

Ilúvatar's main components are shown in Figure 2. Clients/users invoke functions using an HTTP or RPC API, with the main operations being `register, invoke, async_invoke, and prewarm`. Workers also provide load and status information to the load-balancer. We use stateless load-balancing, by using variants of consistent hashing with bounded loads (CH-BL), which have been proposed for FaaS recently [30]. This is a locality-aware scheme, which runs functions on the same servers to maximize warm starts, and forwards them to other servers only when the server's load exceeds some pre-specified load-bound.

Continuing on the worker-centric theme, the worker API is a subset and almost completely identical to the overall API, and functions can be launched directly on a worker for single-worker setups and benchmarking, without going through a load-balancer and adding unnecessary latency. The workers implement various latency-hiding and burst-mitigation techniques. All functions are launched inside containers, and dealing with the container layer is a major part of the worker. Each worker maintains a container pool of initialized containers for facilitating warm starts, and has an invocation queue for handling dynamic loads. Function characteristics such as their cold and warm execution times are captured in various data-structures and are made available using APIs for developing data-driven resource management policies.

An important contribution and component of Ilúvatar is its principled support for function overcommitment based on its queueing architecture. In many environments, like public FaaS providers, function resources cannot be overcommitted. However, the actual function resource usage is often significantly less compared to their requested "size". This difference is the motivation behind recent "right sizing" work [13, 25, 31, 40, 63], and can significantly improve system utilization. Through its queue-based architecture (described in Section 4), Ilúvatar supports a wide range of overcommitment scenarios, including no overcommitment, which is absent from OpenWhisk. By default, OpenWhisk does not overcommit memory, but can overcommit CPUs, which introduces performance interference and potential SLA violations for functions.

3.2 Function Lifecycle

New functions need to be first *registered*, which entails downloading and preparing its container disk image. The container images are fetched from DockerHub or some other image repository. Container images are composed of multiple copy-on-write layers, and we prepare the images by selecting the relevant layers for the operating system and CPU architecture. The images consist of the user-provided function code and our agent, which is a simple Python HTTP server that runs in each container. Registered functions can then be directly *invoked*, which triggers launching of the function's container. The first invocation is usually a cold-start, which entails launching the container image from disk, or from a previous snapshot [16, 65] if available. Each function container starts the agent which listens for and controls the actual function code execution. The agent has two simple commands, a `GET /` endpoint for simple status checking, and a `POST /invoke` to run an invocation with some arguments. When the container is ready, the worker sends an HTTP request to the agent to start the function code execution. We detect the container's readiness using an inotify callback, which is a faster and more generic mechanism for notification compared to Docker's built-in API. Finally, when the function finishes execution, the HTTP call to the container's agent returns, and the container is marked as 'available' in the container pool, to be potentially used for future invocations of the same function.

In the spirit of a fast "baseline" control plane and for isolation, Ilúvatar does not share containers across functions. This is in contrast to SAND's application sandboxing [14], SOCK's Zygote containers [43], Nightcore [34], and even OpenFaaS [9]. Our isolation model is similar to the public cloud providers.

Additionally, Ilúvatar introduces a standard `prewarm` API call, which starts the function's container and the agent inside of it, and adds it to the container pool. This reduces most of the cold-start overhead associated with the container. Prewarming can both avoid a "thundering herd" of cold starts on worker startup, and be an optimization in which the control plane anticipates invocations and

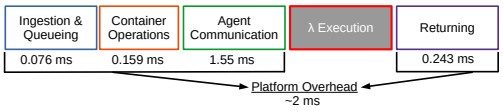


Figure 3: The main components of the Ilúvatar overheads.

Group	Function Name	Time (ms)
Ingestion & Queueing	invoke	0.026
	sync_invoke	0.013
	enqueue_invocation	0.017
	add_item_to_q	0.02
Container Operations	spawn_worker	0.029
	dequeue	0.02
	acquire_container	0.096
	try_lock_container	0.014
Agent Communication	prepare_invoke	0.154
	call_container	1.364
	download_result	0.032
Returning	return_container	0.017
	return_results	0.266

Table 1: Latency of different Ilúvatar worker components for a single warm invocation.

prepares containers for them. This allows for a systematic mechanism to implement various recently proposed predictive prewarm policies [51, 55, 59].

Function Latency Breakdown. Throughout Ilúvatar and this paper, we are interested in three main performance metrics. The first is the end-to-end latency of function execution, also called the *flow time*, shown in Figure 3. This in turn has two main components: the control plane overhead is the latency of Ilúvatar operations, which are mainly before the start of function execution. The second component is the function execution time, which is determined by the function code, and the load on the system. The function execution time is our baseline, and we compute the *normalized* end-to-end latency by dividing the full latency by the execution time (also called the *stretch*).

A more detailed latency breakdown is shown in Table 1. The majority of overhead comes from the communication with the agent which is over HTTP. This is a deliberate choice, since we wanted to be compatible with existing OpenWhisk function images. This can be reduced by using faster IPC mechanisms like in Nightcore [34]. However, these faster communication approaches would reduce compatibility, especially with functions deployed inside VMs.

For OpenWhisk, a similar latency breakdown shows that a large amount of time is spent reading/writing to couchDB (up to half a second), and the rest of the slowdown occurs in the invoker and is primarily due to its design and implementation. Interestingly, the load-balancer/controller for OpenWhisk adds less than 3ms of latency even under heavy load, indicating that the worker-level performance is relatively more important. This further motivates our worker-centric design and evaluation focus.

3.3 Worker Performance Optimizations

To achieve this low latency function execution for heterogeneous and bursty workloads, Ilúvatar uses two key underlying design principles: resource caching, and asynchronous handling of function life-cycle events.

3.3.1 Resource Caching. The cornerstone design goal of Ilúvatar is to reduce jitter, which we accomplish by removing expensive operations from the function's critical path. Instead, we cache and reuse as many function resources as possible, which minimizes the "hot path" function invocation latency significantly. This principle is applied in various worker components, which we describe below.

Container Keep-alive. The primary and exemplary application of resource caching is in the container keep-alive cache that Ilúvatar workers maintain. The containers become "warm" when their function has finished execution, and become "available" for the next invocation of the same function. We maintain a pool of all in-use and available containers for each registered function. This container cache implements classic eviction policies such as Least Recently Used (LRU), and size-aware policies like Greedy-Dual-Size-Frequency, as proposed in FaasCache [29].

Network Namespace Caching. For isolation, each container is provided with a virtual network interface and a network namespace. Through performance profiling, we've found that creating this network namespace can add significant latency to container cold starts—as much as 100ms. This is due to contention on a single global lock shared across all network namespaces [43]. To minimze this overhead, we maintain a pool of pre-created network namespaces that are assigned during container creation. The isolation is still maintained, since concurrently running containers do not share the namespace.

HTTP Clients. The worker threads communicate with the in-container agent for launching the function code. Instead of creating a new HTTP client for every invocation, we cache a client per container and use connection pooling. This affects all invocations (even warm starts), and reduces the control-plane overhead latency by up to 3*ms*.

3.3.2 Async function life-cycle handling. The second key design principle is to handle various aspects of the function's lifecycle asynchronously off the critical path. Ilúvatar achieves this through background worker threads for certain tasks, and through its Rust implementation which heavily uses asynchronous functions, futures, and callbacks wherever possible.

Keep-alive eviction. One such aspect is maintaining the function keep-alive cache, and ensuring that new functions have enough free memory to launch without waiting on existing containers to be evicted first. Traditionally, eviction decisions would be made in an online fashion, but picking victims and waiting for their removal creates high variance in function execution times. Ilúvatar performs container eviction from the keep-alive pool periodically in the background, off the critical path. This is similar to the Linux kernel page-cache implementation. We maintain a minimum free-memory buffer for dealing with invocation bursts, and periodically sort the containers list for eviction based on caching policies from [29].

Function Queueing. An important component of Ilúvatar's architecture is a per-worker function queue. New invocations are first

put into the queue, and are dispatched to the container backend by a queue monitoring thread. This allows us to tolerate bursts of invocations, and regulate the server load. Details of our queueing policies are presented in Section 4.

3.4 Container Handling

Ilúvatar uses standard Linux containers for isolating and sandboxing function execution—a "vanilla" and conventional approach. Several exciting new isolation mechanisms for cloud functions have been proposed: such as lightweight VMs [12], unikernels, WASM [58] and other language runtimes [18], etc. Importantly, the sandboxing affects the *cold start* overheads, which account for a tiny fraction of all invocations (usually less than 1%). Our control plane design and performance optimizations are independent of the sandboxing mechanism, and we address the orthogonal problem of optimizing the *warm starts*.

The basic container operations we use are: i) Create a container/sandbox with specified resource limits and disk image/snapshot, ii) launch a task inside it for the agent, and iii) destroy the container. Each container is launched with the CPU and memory resource limits. CPU limits are enforced with cgroup quotas. This limited API allows Ilúvatar to support *multiple* container backends.

By default, we use containerd [2], which is popular container library, also used by Docker. The very rich containerization ecosystem presents a large number of options, and examining their tradeoffs was a major part of Ilúvatar's design process. Importantly, the choice of containerization library impacts the cold-start times, and some library operations can take considerable time (100s of ms). High-level container frameworks like Docker are feature-rich and easy to use, but are typically used for long-running containers and are not optimized for latency. Docker uses containerd under the hood, and it provides more fine-grained control and slightly better latency. Functions require a minimal containerization, and a lot of feature-complexity in these large containerization libraries can add to latency. For instance, the crun [7] library which is written in C takes about 150ms to launch a container, whereas containerd (written in Go) needs 300ms, and Docker needs 400ms.

Using containerd allows us to use the OCI container specification [1], and makes it easier to support other container runtimes. For instance, we also support the Docker container backend, which required only a minimal programming effort. Containerd operates as a separate service, and we use it's RPC-based API, which contributes to some latency as well. We contemplated writing our own optimized container runtime in Rust to avoid the overheads due to inter process communication, extra process forks and system calls, and implement other cgroups and namespace optimizations. However, we ended up going with containerd to keep our control plane small and reusable across container runtimes. We also wanted to investigate and tackle the challenge of getting predictable performance out of higher level containerization services that are not part of the same address space.

Simulation Backend. In addition to containerd and Docker containers, we also support a "null" container backend which is useful for simulations and evaluating control plane scalability. Because of the scale and variety of FaaS workloads, using discrete event simulators for developing and evaluating resource management policies is often necessary. For instance, the recent work on FaaS load balancing [30] uses such a simulator for evaluating their policies at scale for different subsets of the Azure workload trace. Usually, the simulation is used to augment and complement the "real" empirical evaluation of the same policies which are implemented in FaaS frameworks like OpenWhisk.

However, a major methodological and practical issue is that the policy implementations, workload generation, and analysis, all need to be duplicated across the simulator and the real system. This can lead to subtle and large divergences between the simulation and real environment. Moreover, the simulator cannot capture all the real-world dynamics and jitter, and can suffer from poor fidelity.

In order to aid researchers, Ilúvatar takes a different approach to simulations, and provides *in-situ* simulations. Our "null" container backend does not run any actual function code, but instead sleeps for the function's anticipated execution time. The rest of the control plane operates exactly as with real containers, and we still handle all other aspects of the function's lifecycle. This allows us to simulate large systems and workloads. For evaluating any particular policy, researchers can use the simulator null-backend to evaluate control-plane overheads, warm-starts, etc., without requiring a large cluster. Each Ilúvatar worker can "simulate" 100s of cores, since the CPU resources are only being consumed by the control plane, and not for running actual functions. Alternatively, a large cluster can be simulated with multiple simulated workers.

With this approach, *there is minimal difference* between the simulation and the real system. Thus an experiment can be run in-situ or in-silico, following identical code paths. The main distinction is that API calls to containerd are replaced with internal dummy function calls, and function invocations are converted to sleep statements. All control plane operations, control-flow, logging, resource limits enforcement, etc., are exactly the same as with the "real" Ilúvatar. This also helps with mocking and testing new policies.

4 FUNCTION INVOCATION QUEUEING

As a way to regulate and control function execution and worker load, Ilúvatar incorporates a per-worker invocation queue architecture. Function invocations go through this queuing system before reaching the container manager, which either locates the warm container and runs the function or creates a new container. Each worker manages its own queue, differentiating our design from OpenWhisk's shared Kafka queue.

Motivation. This queuing architecture is motivated by three main factors: i) the bursty nature of the workload , and ii) Reducing cold starts due to concurrent invocations, and iii) to give workers additional mechanisms for controlling their load, implementing prioritization, etc. Note that once the function passes through the queue, it is effectively "scheduled" for execution by the OS CPU scheduler. The CPU scheduler of course has its own throttling and controling mechanisms, such as cgroups and the various scheduler tuning knobs. The invocation queue thus acts as a kind of a regulator or a filter before the CPU scheduler, and ideally, "feeds" it the right functions at the right rates for maximizing throughput and minimizing latency.

Because function workloads are so bursty and heterogeneous, running each function immediately can significantly increase the

worker load and result in severe resource contention and increase function tail latencies. The queue also helps as an explicit back-pressure mechanism for load-balancing, admission control, and elastic scaling. The queue length is used for accurately determining the true load on the worker, which is a vital input to consistent hashing with bounded loads [30]. This reduces the staleness and noise of using system load average as the load indicator, and makes load balancing more robust.

Queueing invocations also allows us to reduce cold-starts. While repeated function invocations are good and increase warm starts, *concurrent* invocations of the same function results in cold-starts for all the concurrent invocations, since each invocation needs to be run in its own container. This is also the "spawn start" [49], which causes severe latency increase of 10s of seconds in public FaaS. If there are n concurrent invocations that arrive at the same time, then the n concurrent cold-starts can significantly increase the system load and affect latency of other functions. Instead, by queueing and throttling the functions, we can wait for the invocation to finish, and then use the warm container for the next function in this "herd", and so on and so forth.

4.1 Queue Architecture

Ilúvatar's queue architecture is shown in Figure 2. We have three main components. From right to left, first, we have a concurrency regulator (or just regulator), which enforces the **concurrency limit**: the upper-bound on the number of concurrently running functions. This lets functions execute "on cpu" without timesharing, and effectively determines the overcommitment ratio. Higher concurrency limits (more than the number of CPUs) means more CPU overcommitment. Note that even with overcommitment, the cgroup quotas still provide proportional allocation (thus a 2 CPU container will still get twice the CPU cycles compared to a 1 CPU container). In addition to concurrency, other factors can also be used to regulate the queue discharge rate. The regulator can be used to run functions of only when sufficient resources (such as CPU bandwidth, warm containers, or even accelerators like GPUs) are available.

Ilúvatar can be deployed with a fixed concurrency limit based on the usage requirements, or use its dynamic concurrency limit mode. In the dynamic mode, we use a simple TCP-like AIMD [68] policy which increases the concurrency limit until we hit congestion, which in our case is hit if the system load average increases above some specified threshold. Other metrics are possible: looking at the increase in execution time (i.e., stretch) of the functions could also be used as a congestion metric. The concurrency limit affects the tail-latency, and more advanced policies can be implemented.

The second component is a **queueing discipline.** In the simplest case, we can use simple FCFS, and process functions in arrival order. However, because functions are heterogeneous, this is not always the most appropriate. Instead, we can use the past function execution characteristics such as their cold/warm running times for size-aware queueing such as shortest job first (SJF). We elaborate more on the queueing policies in the next subsection.

Finally, we note that queueing may increase the waiting time for small functions. We thus have a **queue bypass** mechanism, which allows certain functions to bypass the queue and immediately and directly run on the CPU. Bypass policies take the function running time and the current system state as input. Currently, we implement a short-function bypass, where functions smaller than a certain duration are immediately scheduled, as long as the system is under a load-average limit. More effective bypass policies can also consider reinforcement learning approaches, since the action space is simple (bypass or enqueue), and the system state is well defined (functions running and in-queue, etc.).

4.2 Queueing Policies

We implement multiple queue policies which leverage the repeated invocations of functions and use their learned execution characteristics to determining each function's priority. To accomplish this, we maintain per-function characteristics such as cold time, warm time, and inter-arrival-time (IAT). We maintain a priority queue sorted by the function priorities, which are computed using their characteristics like arrival and execution time.

FIFO is simplest and invocations are just sorted by their arrival time. For prioritizing small functions, we leverage our bypass mechanism, where the short functions can skip the queue and be scheduled directly on the CPU. Optimizing queueing policies for heterogeneous functions is challenging, and is an NP complete problem even in the offline case [17].

For improving throughput, we use shortest job first (SJF), which helps reduce the waiting time for short functions, but can lead to starvation for longer functions if the queue never drains. As a trade-off between function duration and arrival, Ilúvatar by default tries to minimize the "effective deadline" of a function, which is equal to the sum of its arrival time and (expected) execution time. This earliest effective deadline first (EEDF) approach balances both short functions and starvation. In both SJF and EEDF, an invocations' execution time is determined by its (moving window) warm time. New/unseen functions have their times set to 0, to prioritize their execution. If we expect to find available containers for a function, we use its (moving window) warm time as the execution time in both SJF and EEDF. Otherwise, we use its cold time—this also helps in reducing the concurrent cold starts, since the expected cold invocations of some functions in a burst separates them in the queue, and reduces the number of concurrently executing identical functions. This spreading of function invocations over time increases the warm starts and overall performance. Finally, the RARE policy prioritizes the most unexpected functions (i.e., functions with the highest IAT).

5 IMPLEMENTATION

Ilúvatar is implemented in Rust in about 13,000 lines of code. It will be open-sourced upon paper acceptance. Its low latency and lack of jitter are attributable to the various low-level profile-guided performance optimizations we have implemented during the course of its development and testing. Function handling and container management in the worker make up a majority of the implementation footprint and focus. Ours is a heavily asynchronous implementation using the `tokio` library in Rust, and various function lifecycle events spawn new userspace threads and trigger callbacks. The major data structure shared by the various worker threads is the container pool, which is implemented using the `dashmap` crate,

which is a concurrent associative hashmap— this provides noticeable latency improvements compared to a mutex or read-write lock. Conversely, we still use a mutex for the queue, since we found minimal performance degradation compared to a no-queue architecture during profiling. These, and many other small optimizations, keep the Ilúvatar resource consumption small: even under a heavy and sustained load that saturates a 48 CPU server, the worker process uses less than 20% of a single CPU core.

5.1 Support for FaaS research

One of our major design goals is for a reliable and extensible platform for performance-focused FaaS research. We now describe some of the Ilúvatar features and our experiences in extending it.

Performance Metrics. We keep track of all internal and external function metrics (such as their cold/warm execution time histories, inter arrival times, memory footprints, etc.) and provide them to all components of the control plane, and also to external services. One of Ilúvatar's implementation goals was to reduce the reliance on external services for system monitoring etc. We thus track key system metrics like CPU usage, load averages, and even CPU performance counters and system energy usage using RAPL and external power meters. These metrics are collected using async worker threads, and provide a single consistent view of the system performance. Additionally, we also use and provide Rust-function tracing for fine-grained performance logging and analysis. We use the `tracing` crate to instrument the passage of invocations through the control plane components, and obtain detailed function level timing information, which is used for identifying control plane and container-layer bottlenecks.

Adding New Policies and Backends. Using function and system metrics allows for easy development of data and statistical learning based resource management policies to be implemented. Our baseline policy implementations for keep-alive eviction, queueing, load-balancing, are all easily extensible using Rust traits, polymorphism, and code generation. In our experience, adding new policies is relatively straight-forward, even for new-comers. For example, all the priority-based queueing policies (SJF, EEDF, RARE, etc.) were implemented by extending the base FCFS policy. Implementing and testing these policies took less than a few dozen lines and about four hours for a graduate student unfamiliar with the code-base.

The default container runtime backend is `containerd`, but the interface is small, and supporting new backends is relatively easy. We added Docker support in about 400 lines and one person-day of development effort.

Load-generation and Testing. In the spirit of providing a single platform for FaaS experimentation, we have developed a load-generation framework. It can do closed and open loop load generation, and be parameterized by the number and mixture of functions, their IAT distributions, etc. The testing framework can use functions from FaaS suites like FunctionBench [39], or custom sized functions that run lookbusy [19] for generating specific CPU and memory load. The open-loop generation produces a timeseries of function invocations, which is helpful for repeatable experiments. The functions' IAT distributions can be exponential, or be derived from empirical FaaS traces like the Azure trace [55].

For the Azure trace, we start by randomly sampling functions and computing the CDF of their IATs. We compute the expected load level in the system using Little's law, by finding the expected number of concurrent invocations for each function and adding them for all functions. This expected load can be significantly different from the capabilities of the system under testing (for example, 100 concurrent functions will overload a 12 core system). Therefore, we can scale the individual function IAT CDFs to find a suitable load. This also allows us to change the relative popularities of individual functions, and conduct fine-grained sensitivity experimentation (like examining system performance when the popularity of one single function changes, etc.). We can generate larger traces by layering, and merging the traces from multiple smaller workloads.

For synthetic functions (using lookbusy), we use their distribution of running times and memory consumption when generating the workload. When using real functions from a benchmark-suite like FunctionBench, for each randomly sampled function, we use its average execution time (from the full trace), and assign it the closest function in the suite. For example, if the average running time of a candidate function in the Azure trace is 8 seconds, we represent it using the ML-training function, which has the closest running time of 6 seconds.

6 EXPERIMENTAL EVALUATION

We have extensively tested Ilúvatar's performance characteristics throughout its development. Here, we present a limited set of its key performance attributes and focus on new insights into FaaS performance. All our experiments are conducted on a 48 core Intel Xeon platinum 8160 CPU, and we restrict the worker to 32 GB memory, running Ubuntu 20.04 using Rust version 1.67.0 and Tokio library version 1.19.2. We are interested in evaluating latency overheads and Ilúvatar's suitability as a low-jitter research platform. This evaluation focuses exclusively on the performance of the worker, where we think most per-invocation latency improvement opportunities exist. Many effective load-balancing policies have been published, but their impact on latency is limited to balancing decision time and warm start ratio. Our stateless controller's overhead is consistent at less than 0.5ms, and we can thus ignore its latency contribution, for ease of exposition. Our CH-BL based load-balancer maximizes locality and provides 99% warm starts, and we focus on single-worker performance to remove unnecessary confounding factors.

6.1 Control Plane and Function Performance

In this subsection, we focus on the latency overheads of Ilúvatar under different workloads and configurations. For these experiments, we do not use any queueing, use a single worker, and focus on the most basic Ilúvatar configuration.

We start by examining the control plane overheads under a closed-loop load for 30 minutes generated by different number of client threads. The control plane overhead CDF for the AES function is shown in Figure 4. With 48 concurrent client threads, all the CPUs are fully utilized by function execution. Even in this saturated case, the 90 percentile overhead is less than 20ms. Just below this saturation limit, with 46 threads, the 90 percentile overhead drops to less than 10ms, and the average is less than 3ms.

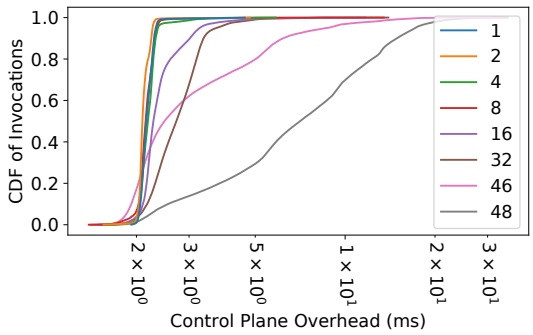


Figure 4: Ilúvatar provides low latency overhead across a range of concurrent invocations.

We now provide a more detailed breakdown of the function latency. In Figure 5, we look at the end to end (E2E) function latency (i.e., flow time) and execution time of different representative functions under different loads. The flow time is impacted by the control plane overhead and the function code execution time. Both these factors are affected by the system load, which in turn is affected by the concurrency level. The difference between the E2E and the function execution time is the control plane overhead, which is small for all functions and at all load levels.

Interestingly, a significant source of latency variance is the function execution time itself. For the small, CPU-intensive PyAES function (Figure 5a), the inter-quartile-range is 60ms, which is 20% the average execution time. Both the execution time (and hence the E2E latency) and the variance also increases with the system load. This variance is also determined by the non-determinism in the function code. For instance, the JSON function (Figure 5b) parses a random json file on every invocation, and thus has a higher natural variance in its execution time. Finally, the video processing function is long and CPU intensive: it downloads and converts a video to grayscale. This magnifies the CPU contention, and the function latency increases from 6 to 9 seconds under heavy load.

The notable increase in execution time for all three functions is a result of high CPU cache miss percentage and a reduction in the instructions per cycle (IPC). We also observed poor cache locality with an increasing number of CPU cores. When the same workload was run on half the number of CPUs (by disabling the rest of the CPU cores), the cache miss percentage significantly dropped (by more than 50%), along with a proportionate reduction in the latency variance. This highlights and emphasizes the deeper architectural challenges of FaaS, which were also shown by [54].

Result: *Ilúvatar overheads are small even under heavy load. Function code non-determinism and system load have a higher impact on the function execution times.*

Cold-starts. So far we have focused on warm-start performance which dominates function workloads. Ilúvatar also incorporates a few optimizations for cold-starts. Specifically, we are interested in quantifying the impact of the different container backends (containerd and Docker), and the network namespace caching optimizations. The end to end cold times for various functions are shown in Figure 6: this includes both container startup time and function initialization overheads. In general, smaller functions face a larger impact due to the cold starts, since it represents a higher percentage of their total flow time.

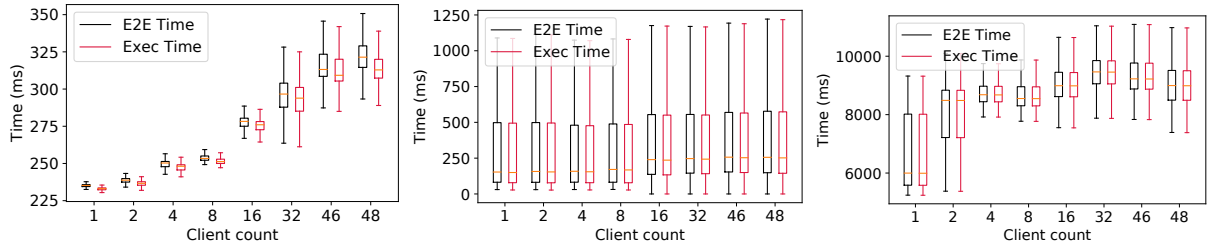
For small functions (left axis of the figure), using containerd (without network namespace caching) reduces the cold-start by more than 40%, indicating a clear advantage of using a lighter container runtime. Introducing the namespace caching further reduces the cold-start times by 15% compared to unoptimized containerd which creates a new network namespace for each new container. After using the namespace cache, each function invocation sees upwards of 100ms improvement in their cold start time. The effects also hold for larger functions (right axis of Figure 6), where Docker increases both the average and variance of the latency.

6.2 Queueing Performance

Having seen Ilúvatar performance in closed-loop micro-benchmarks, we now investigate the impact of its various queueing components and policies. We use our open-loop load-generation capabilities described in Section 5.1. Specifically, we use a random selection of 21 functions from the Azure traces, and pair them with different functions based on their closest running times. This "stationary" workload has an average 40 requests per second for 30 minutes. This represents an extremely heterogeneous workload in terms of function durations and IATs. Additionally, we also show results from a "bursty" workload generated in the same way, but with one function generating a burst of 18 requests per second for one minute. In this open-loop testing, we prewarm the function containers to prevent excessive cold-starts immediately at the start of the workload. The number of containers to prewarm for each function is determined using Little's law by using their average rates and execution times.

Metrics. We use multiple performance metrics to understand and compare different policies. Since functions can differ in execution time, we always normalize their total latency (flow time) by their execution time in an unloaded system. As shown in the previous figures 5, even with 1 closed-loop thread, the execution time has variance. For normalization, we use the *average* execution time with 1 thread for all the functions. Second, function popularities can also vary widely. We thus compute the *weighted* latency, where each function's normalized latency is weighted by the number of its invocations in the trace. Thus, the weighted latency represents the latency *per-invocation*.

Saturation Testing. We are primarily interested in how the different queueing policies impact the waiting time (which is part of the control plane overhead), and the function performance. The analysis of queueing is interesting only in saturated scenarios, where there is enough extra load on the system and not all invocations can immediately run on the CPU. We find this saturation point by weak scaling, and decreasing the number of CPU cores available to Ilúvatar (by disabling CPU cores using hot-unplug). The weighted and normalized latencies for different number of CPUs is shown in Figure 8, which shows the performance *without queuing*. We see that for our baseline trace, increasing the number of available CPU cores has diminishing returns: the per-invocation latency doesnt benefit when CPUs are increased from 18 to 48. However, we also see a sharp inflection point at 16 cores: decreasing the size to 14



(a) PyAES (b) JSON (c) Video

Figure 5: End-to-end latency and execution times for different functions as we increase the concurrency levels.

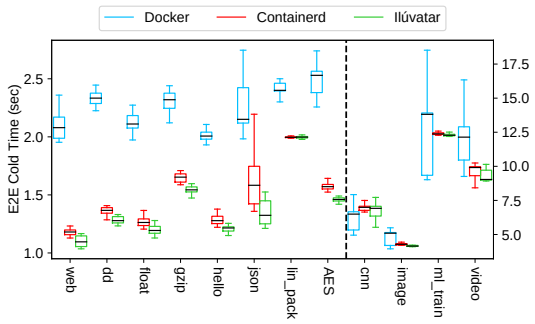


Figure 6: Most functions benefit from using a lower-level containerization and OS object caching on cold starts.

cores results in a very high, almost 6× slowdown. At 16 cores, our workload saturates the system, and we use this system configuration for all our queueing analysis. We note that the alternative is to scale the workload up and run on on all 48 cores. However, as we have shown previously through Figure 5, the poor hardware locality results in higher variance in the function execution times, and introduces more performance variance. This variance often masks the control plane jitter, which is of more interest to us.

Impact of Overcommitment. Many frameworks like OpenWhisk inadvertently overcommit CPUs by running more functions than available CPU cores. Ilúvatar can control the degree of overcommitment through its concurrency limit queue regulator. Figure 7a shows the effect of this overcommitment, when the EEDF (earliest effective deadline) queue policy is used. The worker is limited to CPU cores, so higher concurrency limits represent different degrees of overcommitment. As the concurrency limit is increased, we see a reduction in the queueing time (which is a major part of the control plane overhead). For instance, the queueing overhead is negligible when overcommitment level is 2 (i.e., 32 concurrency limit). However we can see a tradeoff: the increased concurrency risks performance interference, and the code execution time also slightly increases (by 4%). For comparison and as a baseline, we also show the "no queue" configuration which is pure processor sharing and there is no limit on CPU overcommitment. Queueing also reduces cold starts due to concurrent invocations. Without queueing, the number of cold-starts increased by more than 3×.

For the bursty workload, the impact of overcommitment is even more drastic, as shown in Figure 9a. A slight increase in concurrency limit can reduce the weighted latency by more than 3×, indicating that overcommitment is more effective for burstier workloads. Interestingly, the latency improves by 20% with queueing as compared to the "infinite overcommitment" no queueing case. This is due to the increase in function execution time due to uncontrolled CPU contention and interference, which the queue helps ameliorate.

Result: *CPU overcommitment can reduce queueing times, but come with risk of increased performance interference. Ilúvatar's queue design provides a new effective "knob" for managing this tradeoff.*

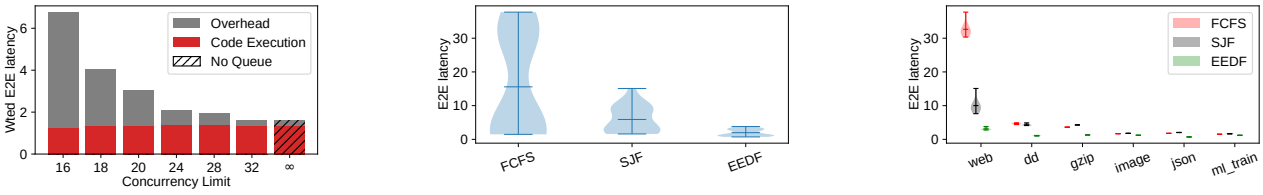
Queueing Policies and Fairness. Next, we look at the performance impact of the different queueing policies themselves. We are interested in the impact on the latencies of the different functions. Figure 7b shows the normalized latencies of different functions with the different queueing policies. This scenario has a significant amount of queueing: the concurrency limit is set to 16 (the number of CPUs). The function-size aware policies like SJF and EEDF provide much lower latency compared to the standard FCFS: the average latency is reduced by more than $2-3\times$.

A breakdown of the latency of individual functions in Figure 7c helps understand this stark performance difference. The queueing in FCFS increases the total time of the extremely small "web" function (13ms running time), which increases its latency by 30×. The small-function prioritization by SJF and EEDF reduces this significantly.

The impact of queueing for the bursty workload is even more interesting, as shown in Figure 9b. EEDF's average latency is 2× higher than simple FCFS, while SJF is 60% lower than FCFS. Investigating the per-function breakdown again in Figure 9c again points to the contribution of the small web function, which is *also the bursty function.* The bursty invocations trigger the cold-start mitigation, which deprioritizes them, and increases the queueing time, which disproportionately impacts the small functions.

Result: *Incorporating both function size and arrival times can improve function latency and fairness significantly. Very small functions see a higher % increase due to queueing.*

Ilúvatar vs. Little's law vs. Simulation. Finally, we want to show Ilúvatar's suitability for performance modeling, capacity planning, and as a research platform for developing and evaluating FaaS resource management policies. We compare the number of concurrent function invocations and queue length (EEDF) with the expected load according to Little's law, computed using average arrival rates



(a) Overcommit (b) Distribution of function latencies (c) Latency breakdown

Figure 7: Queueing performance on the stationary Azure workload. Size-based policies can provide significant latency benefits.

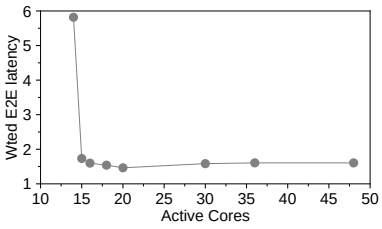


Figure 8: The per-invocation function latencies for different system sizes (# CPUs). We see a sharp inflection point at 16 CPUs, and use that in our queueing evaluation.

and execution times of all functions of our stationary trace. We see that the real system metrics, even with all the inherent burstiness in the Azure trace, and the function execution and control plane jitter, are on average very close to the Little's law estimate. This strongly indicates that our performance is indeed predictable even with highly heterogeneous workloads.

Additionally, Figure 10 also shows the output of our "simulation" container backend described in Section 3.4. This backend doesn't run actual function code, but exercises all other control plane aspects. We use constant average function execution times (without accounting for variance and stochasticity) for all invocations. Even though this simulation setup doesn't capture real-world variability and the impact of server load on function performance, we see that the simulation is also fairly closely aligned with the real experiment output. This shows that Ilúvatar's integrated simulation framework captures sufficient system dynamics and provides high-fidelity simulations. This can significantly accelerate FaaS research, especially advances in reinforcement learning based scheduling, which requires high-quality simulations for learning policies.

Discussion. Our worker-centric design allows us to focus on single-worker performance. The load balancer is stateless and uses consistent hashing with bounded loads, and has a small overhead of less than 0.5 ms. Without workers sharing state (like with OpenWhisk's shared queue), there is no/minimal performance interference, and hotspots are confined in space and time.

Finally, our performance comparison with OpenWhisk is based on end-to-end latency testing. Performance tracing of OpenWhisk is challenging due to the highly distributed nature, and the drastically different architectures prevent a clean side-by-side comparison vs. the various Ilúvatar components. The use of Rust vs. Scala provides some performance gains as well, but all our OpenWhisk evaluation was conducted with ample heap sizes to reduce extra garbage collection overheads.

7 RELATED WORK

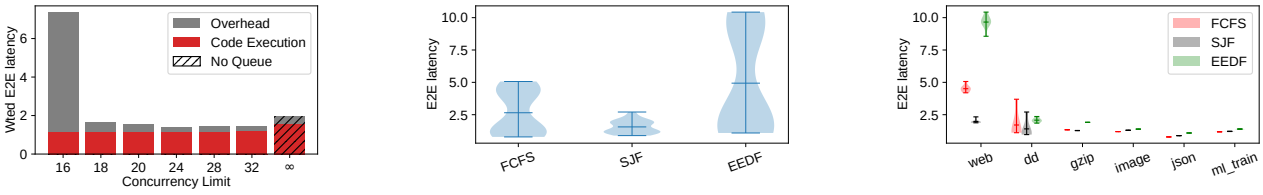
Ilúvatar occupies a somewhat unique spot in the crowded FaaS landscape because of its focus on warm starts and some key constraints in our system design. Techniques for reducing cold-start overheads, like snapshots, language isolation, unikernels, all sit "below" the control plane, and can be complemented with fast control planes. At the other extreme end, the predictable nature of serverless workloads has been used to great effect for predictive load-balancing, prefetching, sizing, etc. Ilúvatar is mostly reactive and is worker-centric, and tries to make minimal assumptions about workload predictability and focuses on more general optimizations that can work for arbitrary workload patterns.

FaaS Control Planes. SOCK [43] is closely related to Ilúvatar, and makes similar observations about network namespace overheads, and introduced storage and cgroup optimizations for serverless optimized containers. SOCK is based on OpenLambda [33] and achieves great cold-start performance with Zygotes that are cloned into new containers. These optimizations to the container runtime are also applicable to Ilúvatar and are complementary. Using the standard containerd interface allows us to use multiple current and future container backends, and is a deliberate tradeoff.

Nightcore [34] is an integrated control plane and runtime system for low-latency microsecond-scale microservices. It essentially implements containerized RPC, and uses fast message passing between the control plane and the agent. Its special container runtime precludes generic "black box" functions, and it provides a weaker isolation model by running functions concurrently within the same container. In the microservice context, container management and scheduling, dealing with heterogenenous functions, and other challenges are not relevant.

Atoll [60] is a fast and highly scalable control plane, and hugely benefits from pre-allocation and prediction. It has a two level load-balancing setup with functions scheduled to a cluster group which then places them on a worker. Ilúvatar's design and contributions are orthogonal to Atoll's more top-down and predictive approach, and we focus on the "low-level" worker problems.

Popular open-source control planes like OpenWhisk, OpenFaaS, and kNative, are an important basis for optimizing performance.



(a) Overcommit (b) Distribution of function latencies (c) Latency Breakdown

Figure 9: Small and bursty functions can get disproportionately impacted due to queueing. A little overcommitment can go a long way to reduce latency.

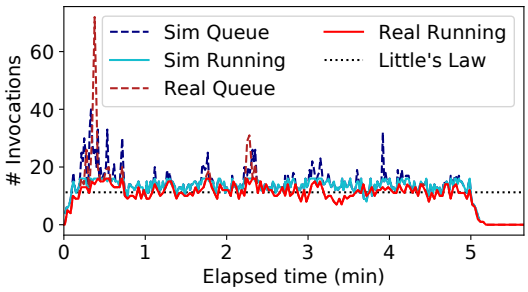


Figure 10: Ilúvatar running in-silico closely models the in-situ performance. Making it a viable exploration opportunity supplementing real experiments.

They tackle the competing demands of modularity and features, along with supporting function executions in generic environments. OpenWhisk's cold and warm performance has been analyzed in many prior works such as [48] and also as part of other systems [15, 29, 30, 52]. OpenWhisk scheduling design and improvements can be found in [30, 38]. Tighter latency requirements exist when deploying functions at the edge, and OpenWhisk's use on lower powered devices presents even more latency troubles [32, 44, 45, 66]. Interestingly, public cloud latencies are also significant, of the order of 50 ms [64], hinting that the problems also extend their control planes.

Function Scheduling. Concurrent to our efforts, queuing of function invocations has been proposed in [72], which implements various size-aware policies like SJF. Surprisingly, and perhaps due to OpenWhisk overheads, their function slowdowns are extremely high: of more than $10,000\times$. An earlier theoretical queueing analysis of flow and stretch metrics is also presented in [73]. In contrast to Ilúvatar's worker-centric design, a centralized core-level allocation design is presented in [36]. In FaaS clusters, the tradeoffs in load balancing and early/late binding are evaluated in [37]. Locality [30] and ML-based [69] techniques for FaaS load-balancing take advantage of the high temporal locality and predictability of the FaaS workloads. Our effort is more focused on reactive systems, and adding predictive allocation will only improve it.

OS scheduler improvements can also improve FaaS workloads [28]. Regulating Linux CPU cgroups shares is also effective in overcommitment [62]. Evaluating the effectiveness of these scheduling improvements when juxtaposed with queueing will be interesting. Scheduling function workflows and DAGs are a growing area [41, 57, 71], and we focus on single-invocation optimizations.

8 CONCLUSION

Ilúvatar a fast, modular, and extensible FaaS control plane. It is implemented in Rust in about 13,000 lines of code, and introduces only 3ms of latency overhead under a wide range of loads. Its worker-centric architecture, resource caching based design, queue-based overcommitment and scheduling, and careful asynchronous implementation, all contribute to low latency and jitter.

Ilúvatar is open source, and intended to serve as a platform for future high-performance FaaS research and deployments. In the near future, we intend to incorporate support for Firecracker [12] VMs and GPUs; investigate load balancing optimizations; and deploy Ilúvatar on HPC and cloud clusters.

Acknowledgements. This research was supported by the NSF grant 2112606.

REFERENCES

[1] Open Container Initiative. https://opencontainers.org/, 2015.
[2] containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/, 2019.
[3] Apache Kafka: Open Source Distributed Event Streaming Platform. https://kafka.apache.org/, 2020.
[4] Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/, 2020.
[5] AWS Lambda. https://aws.amazon.com/lambda/, 2020.
[6] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2020.
[7] crun: A fast and low-memory footprint OCI Container Runtime fully written in C. https://github.com/containers/crun, 2020.
[8] Google Cloud Functions. https://cloud.google.com/functions, 2020.
[9] OpenFaaS : Server Functions, Made Simple. https://www.openfaas.com, 2020.
[10] Automate the Data Science Pipeline with Serverless Functions. https://nuclio.io/, 2023.
[11] Adzic, G., and Chatley, R. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 884–889.
[12] 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.
[13] Akhtar, N., Raza, A., Ishakian, V., and Matta, I. COSE: Configuring Serverless Functions using Statistical Learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (July 2020), pp. 129–138. ISSN: 2641-9874.

[14] 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.

[15] Alzayat, M., Mace, J., Druschel, P., and Garg, D. Groundhog: Efficient Request Isolation in FaaS, May 2022. arXiv:2205.11458 [cs].

[16] Ao, L., Porter, G., and Voelker, G. M. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 730–746.

[17] Bender, M. A., Chakrabarti, S., and Muthukrishnan, S. Flow and stretch metrics for scheduling continuous job streams. In *SODA* (1998), vol. 98, pp. 270–279.

[18] Bruno, R., Ivanenko, S., Wang, S., Stevanovic, J., and Jovanovic, V. Graalvisor: Virtualized polyglot runtime for serverless applications, 2022.

[19] Carraway, D. Lookbusy – a synthetic load generator. http://www.devin.com/lookbusy/.

[20] Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19* (Santa Cruz, CA, USA, 2019), ACM Press, pp. 13–24.

[21] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. The rise of serverless computing. *Communications of the ACM 62*, 12 (2019), 44–54.

[22] Chard, R., Babuji, Y., Li, Z., Skluzacek, T., Woodard, A., Blaiszik, B., Foster, I., and Chard, K. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2020), HPDC 20, Association for Computing Machinery, pp. 65–76.

[23] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 153–167.

[24] 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.

[25] Eismann, S., Bui, L., Grohmann, J., Abad, C., Herbst, N., and Kounev, S. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference* (2021), pp. 248–259.

[26] Fouladi, S., Romero, F., Iter, D., Li, Q., and Chatterjee, S. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. *USENIX Annual Technical Conference* (2019), 15.

[27] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), pp. 363–376.

[28] Fu, Y., Liu, L., Wang, H., Cheng, Y., and Chen, S. Sfs: Smart os scheduling for serverless functions. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2022), IEEE Computer Society, pp. 584–599.

[29] 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.

[30] Fuerst, A., and Sharma, P. Locality-aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2022), HPDC 2022, Association for Computing Machinery.

[31] Guo, Z., Blanco, Z., Shahrad, M., Wei, Z., Dong, B., Li, J., Pota, I., Xu, H., and Zhang, Y. Decomposing and Executing Serverless Applications as Resource Graphs, Dec. 2022. arXiv:2206.13444 [cs].

[32] Hall, A., and Ramachandran, U. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19* (Montreal, Quebec, Canada, 2019), ACM Press, pp. 225–236.

[33] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[34] Jia, Z., and Witchel, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 152–166.

[35] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 345–360.

[36] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Centralized Core-granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz CA USA, Nov. 2019), ACM, pp. 158–164.

[37] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco California, Nov. 2022), ACM, pp. 289–305.

[38] Kim, D. K., and Roh, H.-G. Scheduling Containers Rather Than Functions for Function-as-a-Service. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (May 2021), pp. 465–474.

[39] 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)* (2019), IEEE, pp. 502–504.

[40] Kotni, S., Nayak, A., Ganapathy, V., and Basu, A. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference (USENIXATC 21)* (2021), pp. 805–820.

[41] Mahgoub, A., Yi, E. B., Shankar, K., Minocha, E., Elnikety, S., Bagchi, S., and Chaterji, S. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 6*, 2 (May 2022), 1–28.

[42] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, Association for Computing Machinery, p. 265âĂŞ276.

[43] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. 14.

[44] Palade, A., Kazmi, A., and Clarke, S. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)* (2019), vol. 2642-939X, pp. 206–211.

[45] Pfandzelter, T., and Bermbach, D. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *2020 IEEE International Conference on Fog Computing (ICFC)* (Sydney, Australia, Apr. 2020), IEEE, pp. 17–24.

[46] Popa, L., Ghodsi, A., and Stoica, I. HTTP as the narrow waist of the future internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Monterey California, Oct. 2010), ACM, pp. 1–6.

[47] Prekas, G., Kogias, M., and Bugnion, E. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 325–341.

[48] Quevedo, S., MerchÃąn, F., Rivadeneira, R., and Dominguez, F. X. Evaluating Apache OpenWhisk - FaaS. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)* (Nov. 2019), pp. 1–5.

[49] Ristov, S., Hollaus, C., and Hautz, M. Colder than the warm start and warmer than the cold start! experience the spawn start in faas providers. In *Proceedings of the 2022 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems* (New York, NY, USA, 2022), ApPLIED '22, Association for Computing Machinery, p. 35âĂŞ39.

[50] Roy, R. B., Patel, T., Gadepally, V., and Tiwari, D. Mashup: making serverless computing useful for hpc workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2022), pp. 46–60.

[51] Roy, R. B., Patel, T., and Tiwari, D. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 753–767.

[52] Scheuner, J., Eismann, S., Talluri, S., van Eyk, E., Abad, C., Leitner, P., and Iosup, A. Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications, May 2022. arXiv:2205.07696 [cs].

[53] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM 64*, 5 (Apr. 2021), 76âĂŞ84.

[54] Shahrad, M., Balkind, J., and Wentzlaff, D. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus OH USA, Oct. 2019), ACM, pp. 1063–1075.

[55] Shahrad, M., Fonseca, R., Goiri, Ã., 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.

[56] Sharma, P. Challenges and opportunities in sustainable serverless computing. *HotCarbon 2022: 1st Workshop on Sustainable Computer Systems Design and Implementation*.

[57] 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)* (July 2021), pp. 194–204. ISSN: 2575-8411.

[58] Shillaker, S., and Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX{ATC} 20)* (2020), pp. 419–433.

[59] Silva, P., Fireman, D., and Pereira, T. E. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft Netherlands, Dec. 2020), ACM, pp. 1–13.

[60] Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., and Akella, A. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 138–152.

[61] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Faleiro, J. M., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

[62] 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.

[63] Tian, H., Li, S., Wang, A., Wang, W., Wu, T., and Yang, H. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco California, Nov. 2022), ACM, pp. 78–93.

[64] Ustiugov, D., Amariucai, T., and Grot, B. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *2021 IEEE International Symposium on Workload Characterization (IISWC)* (Storrs, CT, USA, Nov. 2021), IEEE, pp. 51–62.

[65] 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.

[66] Wang, B., Ali-Eldin, A., and Shenoy, P. Lass: running latency sensitive serverless computations at the edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (2021), pp. 239–251.

[67] Xu, F., Qin, Y., Chen, L., Zhou, Z., and Liu, F. λ-dnn : Achieving predictable distributed dnn training with serverless architectures. *IEEE Transactions on Computers* (2021).

[68] Yang, Y. R., and Lam, S. S. General aimd congestion control. In *Proceedings 2000 International Conference on Network Protocols* (2000), IEEE, pp. 187–198.

[69] 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.

[70] Zhou, Z., Zhang, Y., and Delimitrou, C. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (2022), pp. 1–14.

[71] Zhou, Z., Zhang, Y., and Delimitrou, C. QoS-Aware Resource Management for Multi-phase Serverless Workflows with Aquatope, Dec. 2022. arXiv:2212.13882 [cs].

[72] Zuk, P., Przybylski, B., and Rzadca, K. Call Scheduling to Reduce Response Time of a FaaS System. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept. 2022), pp. 172–182. ISSN: 2168-9253.

[73] Zuk, P., and Rzadca, K. Scheduling Methods to Reduce Response Latency of Function as a Service. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (Sept. 2020), pp. 132–140. ISSN: 2643-3001.