PDF Download
3769102.3770623.pdf
07 January 2026
Total Citations: 0
Total Downloads: 155

Latest updates: https://dl.acm.org/doi/10.1145/3769102.3770623

Published: 03 December 2025

Citation in BibTeX format

SEC '25: Tenth ACM/IEEE Symposium on Edge Computing
December 3 - 6, 2025
VA, Arlington, USA

Conference Sponsors:
SIGMOBILE

RESEARCH-ARTICLE

SledgeScale: Load-Aware Dispatch and Deadline-Driven Scheduling for Scalable, Dense Serverless Computing in Edge Data Centers

XIAOSU LYU, The George Washington University, Washington, D.C., United States

EMIL ABBASOV, The George Washington University, Washington, D.C., United States

SEAN MCBRIDE, The George Washington University, Washington, D.C., United States

GABRIEL AMMON PARMER, The George Washington University, Washington, D.C., United States

TIMOTHY WOOD, The George Washington University, Washington, D.C., United States

Open Access Support provided by:

The George Washington University

SEC '25: Proceedings of the Tenth ACM/IEEE Symposium on Edge Computing (December 2025)
https://doi.org/10.1145/3769102.3770623
ISBN: 9798400722387

SledgeScale: Load-Aware Dispatch and Deadline-Driven Scheduling for Scalable, Dense Serverless Computing in Edge Data Centers

Xiaosu Lyu, Emil Abbasov, Sean McBride, Gabriel Parmer, Timothy Wood
Department of Computer Science, The George Washington University
Washington, DC, USA
{lyuxiaosu,eabbasov,seanmcbride,gparmer,timwood}@gwu.edu

Abstract

Serverless and Edge Computing ought to be a perfect match to create flexible, efficient, and responsive applications for emerging areas like augmented reality and autonomous vehicles. Unfortunately, current serverless designs incur high overheads—preventing submillisecond execution—and consume large amounts of resources—preventing dense deployment within constrained edge environments. We present a platform to overcome these challenges, while providing strong isolation and performance in multi-tenant edge data centers.

Our system, SledgeScale achieves this through lifecycle management of lightweight WebAssembly sandboxes that can be rapidly instantiated for each request, a high performance kernel bypass-based communication framework, and load- and deadline-aware request dispatch and scheduling algorithms. Our evaluation shows that we can reach 1M req/second using 15 cores. Under a log normal distribution workload, we can sustain 6x and 48x more load than DARC and Shinjuku scheduling algorithms, respectively, for a target 99.9th percentile slowdown of 200x. We support unprecedented levels of function density—maintaining 99th percentile latency under 60 microseconds and throughput over 250K req/sec for 10,000 distinct functions running on six cores.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SEC '25, Arlington, VA, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2238-7/2025/12
https://doi.org/10.1145/3769102.3770623

CCS Concepts

• Computer systems organization → Real-time systems; **Cloud computing**; **• Networks → Network services**.

ACM Reference Format:
Xiaosu Lyu, Emil Abbasov, Sean McBride, Gabriel Parmer, Timothy Wood. 2025. SledgeScale: Load-Aware Dispatch and Deadline-Driven Scheduling for Scalable, Dense Serverless Computing in Edge Data Centers. In *The Tenth ACM/IEEE Symposium on Edge Computing (SEC '25), December 3–6, 2025, Arlington, VA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3769102.3770623

1 Introduction

Serverless computing provides a "Functions as a Service" (FaaS), allowing software to be deployed, scaled, and managed with minimal regard to the underlying computing infrastructure. Functions are dynamically instantiated in an isolated sandbox such as a virtual machine (VM) or container and are scaled up or down based on incoming workloads. By not having a long-term commitment of resources to each function, Serverless computing seeks to reduce cost for the user and make the cloud platform more efficient.

Serverless computing ought to be a perfect fit for Edge Cloud Computing[5, 6, 59, 80], where tiny data centers are distributed throughout a geographic area. Edge Clouds allow nearby users to access low latency services rather than relying on a distant, centralized cloud. These edge resources typically comprise a small number of servers deployed by a Content Delivery Network or Cellular Network Provider as close to users as possible [73]. They can then be used to support emerging applications with millisecond time scales such as autonomous vehicle coordination, augmented reality, and cyber physical system control. Since each edge data center will be highly resource constrained, the on-demand invocation and autoscaling "from zero" capabilities provided by serverless computing seem an ideal fit. Unfortunately, current serverless platforms cannot provide the necessary performance or efficiency to support edge cloud applications.

The conventional approach of maintaining a "warm" set of functions to reduce initialization latency exacerbates the

resource scarcity issue of edge environments. Keeping functions in a ready state consumes resources that could otherwise be allocated to additional functions or saved to reduce overall energy consumption. This trade-off between initialization latency and resource efficiency is a critical concern that needs addressing to support dense deployments of serverless computing functions at the Edge.

Allowing more functions to be instantiated simultaneously improves efficiency, but also increases the likelihood of interference between functions. Existing serverless platforms reserve CPU resources for each function—e.g., AWS Lambda[3] and Azure Functions[55] allocate CPU time in proportion to memory size—and rely on traditional OS-level schedulers, such as Linux's CFS, to multiplex CPU time across functions. However, prior studies have shown that such general-purpose schedulers are ill-suited for latency-sensitive workloads [24, 32, 36, 43, 47]. More recent work, DARC [17], address this by statically reserving dedicated CPU cores for function groups according to demand. While this reduces interference, it conflicts with our goal of efficiently supporting a large number of concurrent functions on limited edge resources.

Finally, achieving high performance at scale requires efficient, low latency network communication and request dispatch. Current serverless platforms contain multiple layers of indirection (e.g., the ingress gateway, activator, and queue proxy in Knative), which add overhead to each request. They also rely on inefficient protocols like HTTP which further increase costs.

Our system, SledgeScale seeks to resolve these problems through the design of a WebAssembly based serverless platform that makes the following contributions:

- Our work addresses the density challenge by optimizing the lifecycle management of WebAssembly sandboxes, thereby improving memory efficiency and reducing cold start latency. Our approach enables safe reuse of sandboxes across different functions while preserving robust isolation guarantees.
- Our work proposes a user-space, deadline-aware dispatching and scheduling framework that employs Earliest Deadline First (EDF) principles with slack-aware preemption to carefully schedule the WebAssembly sandboxes for different function invocations while meeting task deadlines.
- Our work employs a kernel-bypass RPC framework, combined with scalable, load-aware request dispatching engines to ensure incoming requests can be handled at extremely high scale while efficiently distributing request to workers.

We implement our design by extending Sledge [26, 50], an open source serverless platform. The source code is publicly available at `https://github.com/gwsystems/sledge-serverless-framework/tree/sledgescale`. SledgeScale can scale performance to 1.37 million RPS with 5 dispatchers and a total of 20 cores, a 90X improvement over original Sledge with 1 dispatcher and 19 workers. SledgeScale reduces cold and hot request latency to 400 μs and 13 μs, 2000 and 40 times faster than the popular container-based platform Knative. Compared to state of the art schedulers like Shinjuku [36] and DARC [17], SledgeScale sustains 5.9x and 47.69x more load for a set slowdown target under the log normal request distribution typical in serverless environments [68]. SledgeScale achieves a level of function density far beyond existing systems, maintaining a request rate of 250K req/second when accessing 10,000 different function instances with 99th percentile latency below 60 μs with 6 workers.

2 Background and Motivation

We seek to support low latency serverless for edge applications such as AR/VR, autonomous vehicles, and cyber physical systems. These applications can require end-to-end processing times in the millisecond range, e.g., the 3GPP spec for autonomous vehicle platooning specifies a 10 ms latency requirement [1]. In such a scenario, a vehicle might send outputs from its sensors to an edge data center for processing, invoking a series of serverless functions such as an image processing pipeline and autonomous control algorithms. With the shift towards microservice architectures, one user request with a 10 ms deadline may be broken down into a large number of smaller function invocations which each need to have processing costs of a millisecond or below.

Recent work such as Shinjuku [36] and DARC [17] have demonstrated high performance microsecond scale request processing, but they are not designed for serverless environments where there is a wide range of function types that may change over time – especially at the Edge where different device types will be physically moving in and out of the coverage area, each potentially needing a unique set of functions. To understand how a serverless platform can be adapted to support the performance requirements of these workloads under the resource constraints of the Edge, we must overcome challenges in the following areas.

Cold Start Costs. The cost of initiating a function sandbox for new requests is a significant challenge for low latency serverless platforms, especially in edge Computing where resources are limited and idle functions must be terminated, leading to more cold starts. Traditional methods to mitigate cold start delays, such as keeping sandboxes running or starting them preemptively, are resource-intensive, counter to the goals of an efficient edge platform. This leads to our first research challenge and design choice:

How can we prevent cold start overheads from violating deadlines for latency sensitive applications, without wasting

Table 1: Cold And Hot Start Latency of no-op functions

	Protocol	Cold Start			Hot Start
		Total Time (us)	Network Setup (us)	Func Init + Exec (us)	Total Time(us)
Knative [41]	TCP/HTTP	826,946	160	826,786	532
Nightcore [35]	TCP/HTTP	260	104	156	104
rFaaS [14]	RDMA	128,507	128,431	76	11
Sledge [26]	TCP/HTTP	240	103	137	76
SledgeScale	DPDK	400	339	61	13

edge resources? In SledgeScale, we resolve this issue by building on the Sledge platform that uses WebAssembly based sandboxes that last only as long as an individual request. While starting a new sandbox adds a small overhead to each request, our optimized sandbox lifecycle management significantly reduces the startup overhead and prevents the need for advanced resource reservations.

We evaluate cold start overheads across a range of serverless platforms, with the results summarized in Table 1.[1] *Knative*, a container-based serverless platform developed by Google [41], incurs a function initialization cost of over 826 milliseconds, many orders of magnitude higher than the others, which use lighter weight isolation mechanisms. Both *rFaaS* and *Nightcore* use process based isolation, avoiding the expensive operations needed to create containers and configure their memory and networking namespaces, but creates scheduling issues we discuss in the next section. rFaas also exhibits a very high network setup time due to the high performance networking framework that is uses which incurs a more expensive connection setup in exchange for faster subsequent messaging. *Sledge* uses WebAssembly based sandboxes which operate at user-level within a single process runtime and achieves a similar initialization cost. Our approach, *SledgeScale*, improves upon Sledge by optimizing sandbox lifecycle management and replacing its network subsystem (among other enhancements). These optimizations halve the cold start function initialization to only 61 μs, albeit increasing the total time due to a higher network setup cost (although still orders of magnitude lower than rFaaS which uses a similar design). Subsequent requests experience negligible latency (13 μs, a 5.8x reduction from Sledge), despite each request being assigned a fresh sandbox. The upfront networking cost leads to dramatically higher throughput as we show later.

Communication Protocol Overheads. Serverless functions typically rely on HTTP over TCP, adding latency from connection setup and protocol parsing. This latency is compounded when each edge user request triggers many function invocations as is common with microservice architectures. Upcoming edge applications such as autonomous vehicle coordination and AR/VR workloads, will require end-to-end latencies within a few milliseconds, thus individual serverless function invocations need to be completed at microsecond scale. Further, OS networking stacks can incur significant data copying costs between the kernel and userspace. Based on recent studies [11], as much as 50% of CPU cycles can be consumed by data copying. Thus our second challenge:

How can we optimize communication in a Serverless platform to support sub millisecond request processing while efficiently using edge resources? To address this, we adopt kernel bypass networking techniques, coupled with request dispatch algorithms that ensure workers are evenly balanced and requests with tighter latency requirements can be prioritized. This design is suitable for edge environments where each client request enters through a WAN gateway, which can then trigger a larger set of function invocations within the edge network via our streamlined RPC protocol.

Table 1 also shows the communication overhead of existing serverless platforms. The platforms that use standard TCP/HTTP protocols (Knative, Sledge, and Nightcore) all incur an initial connection latency of about 100 microseconds, with Knative being slightly longer due to the container networking infrastructure. The two userspace networking systems, rFaaS and SledgeScale, incur higher initial network connection costs, rFaaS dramatically so since it has an inefficient startup protocol designed for static HPC environments. SledgeScale is based on eRPC's protocol, which requires an initial UDP session creation [40]. However, this cost is incurred only once to establish a persistent connection between edge nodes or the gateway, and can then be reused.

When sandboxes and network connections are already initialized, all of the systems achieve much lower "hot start" request processing times. SledgeScale is able to complete a no-op request within 13 μs in a cleaned sandbox, 5x to 40x faster than the HTTP-based approaches. rFaaS provides similar latency, but falters when running multiple functions as we show next.

Function Scheduling. Container and process-based serverless platforms rely on the Linux Completely Fair Scheduler (CFS) for OS-based task scheduling. CFS divides CPU time into fine-grained slices by partitioning the scheduling period among all runnable tasks, aiming to allocate a "fair" amount of CPU time to all tasks with the same priority. However, this "fairness" at the operating system level may not achieve application level goals since CFS is unaware of request priority or latency targets. Studies [24, 32, 36, 43, 47] have shown that in the presence of a large number of colocated tasks,

[1] To ensure the best results for these systems, we modified Nightcore to bypass the gateway, modified rFaaS to bypass the resource manager and communicate directly with the executor manager, and remove the connection setup latency for Sledge under hot requests since it lacks HTTP keep-alive support. For Knative we could not bypass the Istio Gateway, as it relies on Istio VirtualService to configure routing rules.

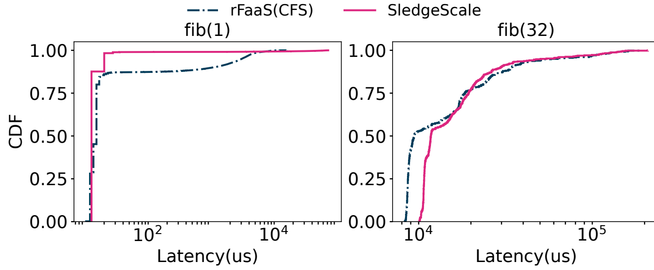


Figure 1: Latency distribution on a single core with 16 short and 16 long colocated concurrent requests at sending rates of 5000 and 48 requests per second, respectively

the frequent context-switch overhead of CFS negatively impacts shorter requests, as they are more sensitive to waiting time. To improve latency handling while preserving fairness, the Linux kernel 6.6 introduced the Earliest Eligible Virtual Deadline First (EEVDF) scheduler [72]. EEVDF selects the task with non-negative lag and the earliest virtual deadline for execution, inherently favoring shorter tasks. However, while it allows users to influence scheduling via nice values, it cannot enforce fine-grained per-request deadlines, as virtual deadlines represent task eligibility rather than actual request deadlines. This poses a new challenge: *How can we ensure requests with different priorities are scheduled to meet their latency targets?* Our approach leverages that WebAssembly sandboxes are a userspace construct, allowing us to perform application-aware scheduling within our serverless runtime instead of relying on the OS. By running at the user level, SledgeScale 's scheduler can incorporate per-request information (e.g., execution time, deadlines), enabling finer-grained scheduling than OS-level schedulers.

In Figure 1 we consider a scenario where a serverless platform must run two types of functions, half represented by short (*fib(1)*) and half by long (*fib(32)*) calculations. The workload causes a continuous stream of 32 concurrent function requests. We compare SledgeScale, which uses our user-space, deadline-driven scheduler for WebAssembly sandboxes, against rFaaS, which has 32 preinitialized processes that are reused by requests and scheduled by Linux CFS. Short requests experience significant interference in rFaaS, with a 90%ile latency of 1055 μs compared to 20 μs in SledgeScale. This is primarily due to CFS assigning time slices in turn, and with more colocated entities, the waiting time rises with many context switches in between, which is harmful to short requests. In contrast, SledgeScale is able to use knowledge of the execution time of each request to prioritize short requests and ensure they meet tighter latency goals.

Shinjuku [36] and DARC [17] also use user-space scheduling and knowledge of execution time to guide scheduling and address head-of-line blocking. Shinjuku uses a fixed microsecond scale scheduling interval to preempt a long running request to schedule a short one, but determining the

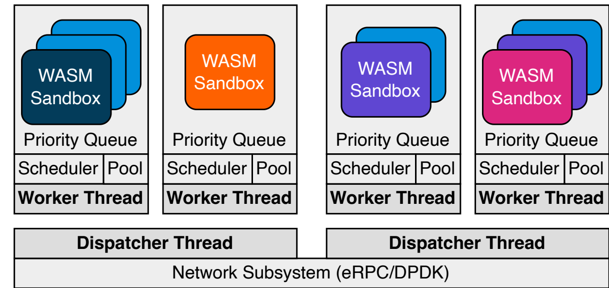


Figure 2: SledgeScale Architecture

optimal quantum is challenging—too short a quantum increases context switches, while a long quantum blocks short requests. DARC groups requests by execution time, allocating cores based on each group's demand. Shorter groups can borrow cycles from longer ones, but not vice versa. While this isolates short and long requests, the lack of preemption can still cause blocking, particularly during bursts. Additionally, this design risks CPU underutilization, as long requests cannot use cores reserved for short requests, even if idle.

3 Design

SledgeScale achieves high performance, resource efficiency and strong per-request isolation through optimized communication, early-binding dispatch, user-space scheduling with precise preemption, and WebAssembly sandboxing. Figure 2 shows the architecture; we provide an overview here and discuss key features in detail in the following sections.

Incoming requests are received and processed by the **network subsystem** (§3.1), which parses the requests and posts them to a **dispatcher thread** . The dispatcher assigns requests to workers (§3.3), employing early-binding to select the best **worker thread**. The best worker is the one that maximizes the likelihood of meeting the request deadline with the shortest wait time and causing minimal disruption to other requests, while simultaneously balancing the load across all workers.

Each worker maintains a **priority queue** based on deadlines to schedule requests (§3.2). Upon popping a request from the queue, a WebAssembly **sandbox** is instantiated to execute it. Unlike most serverless platforms where workers are specific to function types (e.g., a worker for a function is a container or VM spawned specifically for that function), workers in SledgeScale can execute functions of any type, illustrated by the different colors representing distinct function types in Figure 2. The output result of a completed sandbox is returned to the client by the network subsystem, and the sandbox is released and its memory recycled to the worker's pool (§3.4). To reduce context switch overhead, each dispatcher and worker is pinned to its own CPU core.

3.1 Communication Protocol

SledgeScale targets submillisecond applications where communication protocol overhead cannot be ignored. SledgeScale

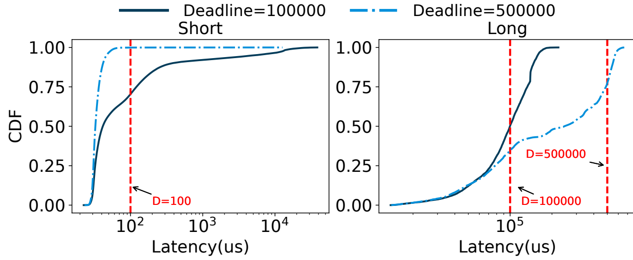


Figure 3: Increasing the deadline for long requests allows SledgeScale to naturally rebalance resources towards short ones

optimizes communication using kernel-bypass I/O technologies such as DPDK and RDMA, which achieve high throughput and low latency by eliminating kernel stack involvement and data copying. Unlike container- or VM-based serverless platforms, SledgeScale offloads all network transfers to the runtime, which forwards data to the WebAssembly sandbox via direct memory copy. This design improves efficiency and eases user-level networking integration, as it simplifies the communication boundaries, allowing developers to focus on application logic rather than managing complex network interactions.

Given the minimal packet loss within modern cloud data centers[2], SledgeScale uses UDP in userspace with a lightweight retransmission mechanism to improve reliability and efficiency. To reduce application-layer overhead, SledgeScale employs RPC instead of HTTP, which uses text-based or metadata-heavy formats; RPC, in contrast, uses compact binary encoding for lower space consumption and faster parsing. We use eRPC [40], an RPC library based on DPDK, as the foundation for our network subsystem.

In SledgeScale, each function module is assigned a unique type ID, included in the RPC request, which the runtime uses to locate and execute the corresponding function code. The request type ID precedes the payload. This optimized communication layer reduces message size and parsing time, improving overall efficiency.

3.2 Scheduling

The use of WebAssembly facilitates user-level scheduling, allowing SledgeScale to make smarter, more flexible scheduling decisions based on request information, such as execution time and deadlines. SledgeScale 's scheduler aims to maximize the likelihood of meeting request deadlines while minimizing tail latency. Sledge uses a simple Round Robin scheduler with a 5 ms polling interval. However, this approach is oblivious to deadlines and fails to prioritize urgent tasks. SledgeScale inherits Sledge's kernel-bypass scheduling framework, but introduces deadline-aware scheduling to efficiently execute tasks. Each worker in SledgeScale independently schedules requests in its local queue using the Earliest Deadline First (EDF) policy, with the dispatcher influencing this process by determining which worker to assign tasks to and whether to trigger preemption to avoid head-of-line blocking. This makes scheduling simple: the dispatcher will preempt a worker upon receiving a more urgent request (i.e., with a sooner deadline), otherwise each worker runs requests to completion in order based on deadline.

Deadlines and Slack. SledgeScale's scheduler is based on the EDF[46] algorithm with a zero remaining slack boundary. This approach assumes that each request has a predefined relative deadline, e.g., requests should complete within 10x of their execution times. In serverless platforms, request execution times can often be predicted from historical profiling data using techniques such as machine learning [9, 21, 29, 45, 75].

SledgeScale will interrupt a request with a longer absolute deadline (i.e., arrival time plus relative deadline) to prioritize the execution of a request with a shorter absolute deadline. However, if the *remaining slack* (i.e., its relative deadline minus execution time and waiting time) of the current request diminishes to zero, the request will not be interrupted to avoid missing its deadline. Remaining slack denotes the time a request can wait for execution without breaching its deadline. Initially, a request's remaining slack is equivalent to its relative deadline minus its execution time. While awaiting execution, the remaining slack decreases; during execution, it remains unchanged. This approach eliminates the need to define a scheduling quantum and instead relies solely on deadlines. Scheduler interrupts occur only when necessary; if subsequent requests have longer absolute deadlines than the current one, no interruptions are made. This ensures efficient handling of requests with minimal disruption and overhead.

Figure 3 illustrates the benefit of having a system capable of explicitly accounting for deadlines in request scheduling. We show the latency distribution of short and long requests under a heavy, extreme bimodal workload, which will be detailed in the evaluation section. In this experiment, short requests have a fixed deadline of 100 usec and we use either a loose or strict deadline for long requests. The results illustrate how SledgeScale automatically adapts its scheduling decisions when the long request deadline is relaxed to 500 msec, allowing it to prioritize more of the shorter requests, and vice versa when the long requests have a stricter 100 msec deadline. Being deadline aware allows SledgeScale to naturally balance the performance needs of different function types, rather than strictly prioritizing short requests over long requests. We believe this is a critical property for a serverless platform where there may be a wide diversity of function types with different performance requirements.

[2] We assume SledgeScale executes in a small edge data center with a gateway that mediates all incoming client requests, thus requests are either initiated by the gateway or other functions within the same edge site.

Sandbox Preemption. A currently executing sandbox can be preempted in order to schedule a more urgent task. This is primarily triggered by the dispatcher, as described in the next section, although it can also be used to interrupt functions that are running excessively long. Preemption is triggered when a worker gets an interrupt (i.e., a signal generated by the dispatcher thread). Upon receiving the interrupt, the worker saves the context (i.e., register values such as the instruction pointer (IP) and stack pointer (SP)) of the current sandbox and switches to the next sandbox in the queue for execution (typically a newly added request with a more urgent deadline from the dispatcher). This user-level context switch overhead is light compared to system-level switches, such as a container/VM context switch, thereby ensuring low overhead and high performance. An interrupted sandbox remains in the worker's queue for later resumption based on its priority.

3.3 Dispatching

The primary function of SledgeScale's dispatcher is to efficiently allocate requests to workers and trigger preemptions. Prior work has described the two primary options available for a dispatcher to assign a request to a worker: early binding and late binding [39]. With late binding, such as Shinjuku [36], Nightcore [35] and DARC [17], a request is retained in the dispatcher's queue until a worker becomes available. While this approach reduces the risk of load imbalance, it may result in worker idleness due to inefficient availability checking and delivery mechanisms, consequently reducing throughput. Conversely, early binding entails the immediate delivery of requests to workers upon their arrival, without queuing in the dispatcher. While this may introduce load imbalance, it ensures that workers are consistently engaged, thereby facilitating optimal throughput. Sledge lacks a dedicated dispatch mechanism: all incoming requests are placed in a global queue, and workers busy-loop to fetch tasks when their local queues are empty. While the lock-free work-stealing deque [12, 42] and busy polling enable high throughput with few workers, this approach is energy-inefficient and processes requests in first-in, first-out (FIFO) order without considering deadlines. Additionally, the so-called lock-free deque is not truly lock-free in practice: it requires locks when accessed by multiple dispatchers and does not scale well with more cores due to synchronization overhead from atomic operations like Compare-And-Swap (CAS).

In SledgeScale, we adopt early binding in the dispatcher while also employing load estimation to effectively balance the workload. Meanwhile, the dispatcher proactively interrupts ongoing request executions based on request urgency to ensure timely processing. Table 2 describes the notation used throughout this section.

Table 2: Notation used to define the scheduling algorithm

Symbol	Description
N	Number of workers in the n-th group.
T_i	Task queue for worker i
W_i	Waiting time for request r in worker i's queue
I_i	Flag to indicate if worker i needs interrupt
C_i	Total remaining execution time in worker i's queue
r	New arrival request
$\mathcal{P}_i^>(r)$	Set of reqs in queue i with a higher priority than request r
Q_i	The set of all queuing requests in worker i
E_j	The remaining execution time of request j
cr_i	Current request being processed in worker i

Load-Aware Dispatch. Since SledgeScale adopts early binding, determining the best worker that can serve a new request with the shortest waiting time based on priority, while also balancing the load across all workers, becomes crucial. A common approach is to use Join the Shortest Queue (JSQ), which assigns a new request to the queue with the least number of unfinished jobs in the system [48][58][67], or Join the Least Loaded Queue (LL(D)) [79], which assigns a new request to the queue with the smallest amount of remaining work. However, neither algorithm can guarantee that the new request will be executed first based on priority in the chosen queue compared to other queues.

To both minimize latency and balance load, SledgeScale employs a greedy algorithm to assign incoming requests to either the worker queue with the least total work or the shortest waiting time for higher priority requests, depending on whether preemption is needed. The scheduling priority, defined by the EDF principle, ensures that requests with the earliest deadlines are processed first.

Algorithm 1 GetWaitTime(T_i, r)

1: **if** is_empty(T_i) **then**
2:     **return** (0, false)
3: **else if** cri.absolute_deadline > r.absolute_deadline and cri.remaining_slack > r.execution_time **then**
4:     **return** (0, true)
5: **else**
6:     **return** (TotalPrecedingExecutionTime(T_i, r), false)
7: **end if**

Algorithm 1 describes the function `GetWaitTime`(T_i, r), which returns the waiting time W_i for the request r if inserted into the queue of worker i, and a boolean flag I_i indicating whether an interruption is necessary. If worker i is empty, then it returns a zero waiting time and flag `false` indicating it is the best worker with no interruption necessary. Otherwise, if the current request has a later deadline, and its remaining slack exceeds the new request's execution time, meaning it still has time to wait, then the function returns a

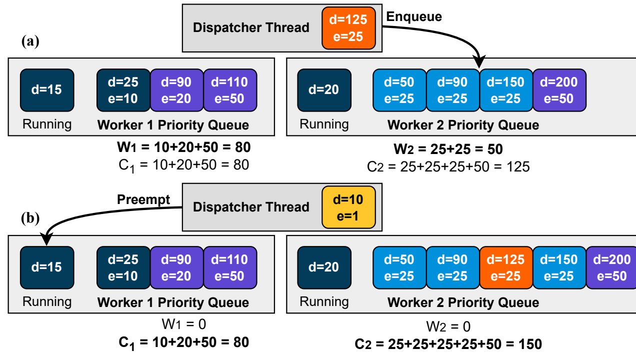


Figure 4: (a) The dispatcher picks the worker with the shortest waiting time (W_2) since the new request is lower priority than the currently executing sandboxes. (b) The dispatcher preempts Worker 1 for this urgent request since it has a lower total queuing time than Worker 2.

waiting time of zero and flag `true` indicating that an interruption is necessary. If neither of the above conditions are met, the function calculates the waiting time for r by calling `TotalPrecedingExecutionTime`(T_i, r), which computes the total remaining execution time of all preceding requests with higher priorities than r in queue T_i, and returns this waiting time and flag `false` to indicate no preemption is needed.

W_i is the total remaining execution time of all requests in the queue i that have a higher priority than the incoming request r and it is formulated as:

$$W_i = \sum_{j \in \mathcal{P}_i^{>}(r)} E_j \tag{1}$$

Each worker queue is structured as a binary search tree (BST), where the absolute deadline serves as the key for sorting requests. To optimize lookups, we also maintain a value in each node that stores the sum of remaining execution times for nodes "left" (i.e., with earlier deadlines) of this node. This allows the worker scheduler to efficiently identify and execute the most urgent request, and enables the dispatcher to quickly compute the waiting time, W_i, for all requests with an earlier deadline than an incoming request.

This load-aware approach facilitates automatic load balancing among workers. If load imbalance occurs, with one worker becoming more heavily loaded than others, the likelihood of subsequent requests being enqueued to this worker diminishes due to its reduced probability of having the shortest waiting time.

Deadline-Driven Interrupts. The dispatcher can make two types of decisions: selecting a worker with a currently running request that has a later absolute deadline than the incoming request (requiring preemption) or selecting a worker with an earlier deadline (resulting in the new request being queued by that worker). Upon making an interrupt decision, the dispatcher sends a signal to the worker to halt its current execution and then delivers the new request to it. Conversely, if an interrupt decision is not made, the dispatcher enqueues the request into the queue of the selected worker, to be completed in priority order.

Algorithm 2 shows the request dispatching algorithm. The dispatcher iterates over all workers to determine the optimal worker for the new arrival request r. If a worker's queue is empty (lines 6-8), request r is promptly inserted into the queue of worker i and the search concludes. Otherwise, workers will fall into one of two categories: 1) their current request is more urgent (earlier deadline) than the new request, in which case we seek to enqueue the new request where it will have the *shortest waiting time, W_i, before execution*, or 2) their current request is less urgent, in which case we select the worker that will *minimize the total queuing cost, C_i, in its queue* and trigger preemption. C_i is formulated as:

$$C_i = \sum_{j \in Q_i} E_j \tag{2}$$

The algorithm evaluates all workers, assessing whether their current request is more or less urgent than the new one, and determines the best worker based on waiting time or total queuing cost, respectively. When the waiting time W_i is zero but interruption is possible, we assess the total queuing cost to identify candidate worker p with the least queuing cost, ensuring workload balance (lines 9-16). If waiting time W_i is not zero, we track the worker c with the minimum waiting time W_c (lines 17-20). If a candidate worker p that can be preempted exists, the dispatcher enqueues the request r to this worker and interrupts its current task to process the new request immediately (lines 22-24). If no such worker exists, then all current requests must have more urgent deadlines or insufficient slack, so the dispatcher enqueues the request r into the worker c that has the minimal total queuing cost (line 26).

Figure 4 illustrates two dispatching possibilities for incoming function requests. For the first request in Figure 4(a), the incoming request has a deadline of 125, which is higher than either of the running requests in the two workers. As a result, the request is dispatched to Worker 2 since it will have a lower waiting time (50 vs 80 time units), calculated based on the sum of remaining execution times (e values) of requests with earlier deadlines. The request is enqueued so that it can be processed in order based on deadline. The second request in Figure 4(b) has an earlier deadline than either executing function, so in this case SledgeScale picks Worker 1 since it has a smaller total sum of execution times for all functions in its queue.

Dispatcher Scalability. For a machine equipped with multiple CPU cores, a single dispatcher may not suffice to manage all workers efficiently. To ensure scalability and performance, SledgeScale partitions workers into groups, each managed

Algorithm 2 Request Dispatch And Preemption

1: $p \leftarrow -1$ // best worker to preempt
2: $c \leftarrow -1$ // best worker to enqueue without interruption
3: $W_c \leftarrow \infty$ // minimum waiting time
4: **for** $i = 1$ to N **do**
5: $W_i, I_i \leftarrow$ GetWaitTime(T_i, r)
6: **if** $W_i = 0$ and $I_i =$ false **then**
7: EnqueueRequest(T_i, r)
8: **return**
9: **else if** $W_i = 0$ and $I_i =$ true **then**
10: **if** $p \neq -1$ **then**
11: **if** $C_i < C_p$ **then**
12: $p \leftarrow i$
13: **end if**
14: **else**
15: $p \leftarrow i$
16: **end if**
17: **else if** $W_c > W_i$ **then**
18: $W_c \leftarrow W_i$
19: $c \leftarrow i$
20: **end if**
21: **end for**
22: **if** $p \neq -1$ **then**
23: EnqueueRequest(T_p, r)
24: PreemptWorker(p)
25: **else**
26: EnqueueRequest(T_c, r)
27: **end if**

by a dedicated dispatcher responsible for handling requests within its group.

We use Receive Side Scaling (RSS) within the network subsystem to randomly assign incoming packets to dispatchers. This design allows performance to scale up well, but potentially introduces imbalance between worker groups if one dispatcher happens to receive more requests with a tighter deadline or a higher cost. Approaches such as defining the RSS distribution rules to account for knowledge of request costs or deadlines could be explored to resolve this limitation.

3.4 Sandbox Lifecycle Management and Isolation

Initializing and Recycling Sandboxes. Frequent sandbox creation and teardown amplify the overhead of traditional memory allocation using `malloc` and `free`, which involve repeated kernel interactions and may cause page faults, cache misses, and memory fragmentation—collectively increasing latency. To reduce cold start overhead and improve memory efficiency, we implement a shared memory pool for sandbox reclamation. Each incoming request is allocated a WebAssembly sandbox with dedicated stack and heap memory, reused from the pool when possible. Completed sandboxes return their stack and heap memory to the pool for reuse. By utilizing available memory from the pool, sandbox initialization is reduced to just a few microseconds. To minimize race conditions, each worker maintains a private memory pool.

For security, the heap memory of reclaimed sandboxes is cleared; the runtime tracks the range of used memory by assisting memory allocation and only zeros the used portion. However, clearing the stack memory would require clearing the full range as exact stack usage cannot be tracked. Therefore, the stack is left uncleared to balance performance and overhead.

Sandbox lifecycle management plays a critical role in achieving high function density, especially under dynamic, short-lived workloads. Slow sandbox startup delays scaling and increases latency, limiting achievable function density. Our approach uses memory pooling to enable fast sandbox teardown and instantiation, ensuring rapid resource reuse and supporting high achievable function density.

Isolation. Sledge employs software fault isolation (SFI) to ensure that each function operates within its own WebAssembly sandbox, preventing a malfunctioning sandbox from affecting the runtime or other functions. Since sandboxes are allocated on a per-request basis, a function that becomes exploited by a malicious request is not able to affect subsequent requests entering the system (unlike other platforms that reuse containers for many requests). SledgeScale further extends SFI isolation by terminating any sandbox that exceeds its allowed execution time or enters an infinite loop, providing both memory and temporal isolation so that buggy or malicious functions cannot monopolize CPU resources.

SledgeScale's runtime precisely tracks each sandbox's execution time by recording when it starts, is preempted, and resumes. When dispatching a new request, the dispatcher checks whether the current sandbox's total execution time exceeds a predefined threshold. If it does, the dispatcher terminates and removes the current sandbox.

4 Evaluation

Testbed Our evaluation utilized two CloudLab [20] d6515 nodes, configured as client and server, respectively. Each node features a 32-core (64-thread) AMD 7452 CPU operating at 2.35 GHz, 160 GB of RAM, and a dual-port Mellanox ConnectX-5 100 GbE NIC (PCIe v4.0). Hyper-threading was disabled. Both nodes ran Ubuntu 20.04 LTS with the Linux 5.4.0 kernel.

Implementation & Optimizations. SledgeScale uses the open source Sledge [26] as a base and is implemented in C/C++. We integrated eRPC [40] into our network subsystem by encapsulating a set of C APIs. We modified 20,684 lines of code in Sledge and 367 lines of code in eRPC.

In contrast to Sledge's busy-loop design, SledgeScale workers sleep when idle to conserve energy and are awakened upon new request arrivals, with synchronization handled via semaphores.

Baselines. We include Sledge as the baseline in most experiments. For comparison purposes, we retain the original Sledge design but replace its TCP stack with eRPC, and selectively enable or disable sandbox lifecycle management (LM). We evaluate scalability using Nightcore [35], rFaaS [14], and Sledge [26], excluding Knative due to its significantly slower performance than the others. For the function density evaluation, rFaaS serves as the baseline. We compare dispatching performance across Round Robin (RR), Join the Shortest Queue (JSQ), join the Least Loaded queue (LL(D)), and eRPC-based Sledge with LM; we omit JSQ results due to its consistently lower performance compared to LL(D). In all cases, each worker schedules using EDF with a fixed 1 ms scheduling quantum triggered by a timer, consistent across RR, LL(D), and eRPC-based Sledge. We evaluate scheduling performance against Shinjuku [36] and DARC [17]. Since Shinjuku and DARC are *not* serverless platforms and assume a single application is running, we integrate their algorithms into SledgeScale for a fair comparison. Both Shinjuku and SledgeScale use a signal-based approach to interrupt requests, ensuring consistent interruption overhead. Preempted requests are moved to the end of the global queue and resumed by the same worker in Shinjuku.

Workload. We employ both synthetic and real-world functions in our evaluation. We use a No-Op function in experiments where we want to stress sandbox lifecycle management. For real-world functions, we utilize Image Resizing, SIFT, Segmentation, and Classification tasks from SD-VBS [77] benchmark. These applications are commonly used in autonomous vehicle benchmarks, such as MEVBench [13], CAVBench [78], and Pylot [27, 28]. Synthetic functions involve hash computations with a loop count parameter, enabling us to model various execution time distributions and analyze scheduling performance across diverse workloads.

Table 3 lists all workloads used for evaluation. We specifically include a log-normal distribution workload because it has been shown to be highly representative of serverless workloads, according to the Azure serverless function trace [68]. We use the same parameters ($\mu = -0.38$ and $\sigma = 2.36$) as the Azure trace, but scale the execution times down by a factor of 1000 since the Azure trace data was collected in 2019, and function costs have been declining. For the exponential distribution, we use $\mu = 1$ but scale up the data by a factor of 100 to reflect a more realistic serverless function execution time.

DARC requires functions to be grouped together based on execution time. We simplify the log-normal and exponential distributions by evenly dividing each into two groups: data points below the median are assigned to group 1, and those above the median to group 2. Each group is allocated an equal number of CPU cores.

Table 3: Benchmark Workloads

Workloads	Request	Runtime(μs)	Ratio
Fixed	N/A	10	100%
High bimodal	Image Resizing	26	50%
	Image Classification	2070	50%
Extreme bimodal	Short	10	99.5%
	Long	10000	0.5%
SD-VBS [77]	SIFT	113	90%
	Segmentation	1100	9%
	Image Classification	2070	1%
Exponential($\mu = 1.0$)	N/A	69 (median)	N/A
Log normal ($\mu = -0.38, \sigma = 2.36$)	N/A	683 (median)	N/A

Execution Time & Deadline. We assume access to a perfect execution time prediction model for all functions except for the real-world benchmark functions; for these we estimate execution time as the median observed execution time during the last 256 requests. This could be replaced with ML models such as [9, 21, 29, 45, 75]. Unless otherwise specified, we set the deadline to be 10x the execution time. In practice, we expect deadlines to be set similar to SLOs, representing a soft target for request latency, and could be customized by users on a per-request basis.

Load generator. We wrote closed and open-loop load generators for rFaaS and SledgeScale in C++ as an additional 7,614 lines of code. The open-loop generator follows a Poisson distribution. SledgeScale ensures that each dispatcher receives an equal number of requests.

We evaluate scalability with closed-loop load generator and others with open-loop. For Knative, Nightcore and Sledge measurement, we use the closed-loop http load generator hey [18]. All experiments run for 20 seconds each.

rFaaS spawns a worker process for each client connection, allowing direct communication between the worker and the client . Due to the inherent limitation of rFaaS, where a single connection does not support concurrent requests, we developed a pseudo open-loop rFaaS client. Unlike a true open-loop generator, which could issue multiple concurrent requests and potentially crash the rFaaS server due to shared memory conflicts, our pseudo open-loop approach tracks when a concurrent request would have been generated. It then uses that timestamp as the request's start time once it can be sent serially. This method effectively simulates the queuing delay of concurrent requests on the client side, rather than on the server side.

Metrics. For dispatching and scheduling evaluation, we measure deadline miss rate on the server side and end-to-end

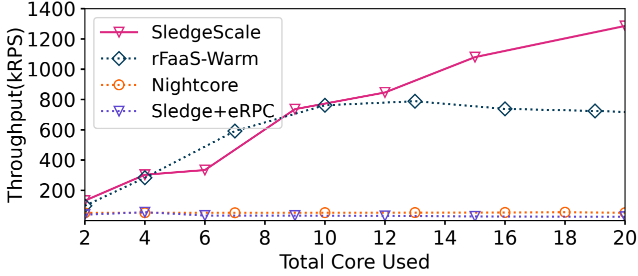


Figure 5: Performance scalability on different platforms

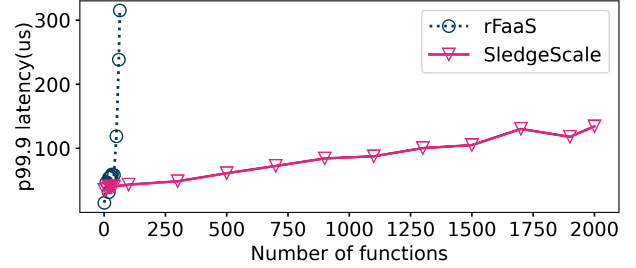


Figure 6: Function density in a single core

latency recorded by the client. We report the 99.9th percentile of the measured latency and assess slowdown [30], defined as the ratio of total client latency to request execution time.

4.1 Scalability

We first quantify the throughput contributions of our network stack (eRPC) and LM optimizations using a no-op function and a closed-loop client.

Versions	1 worker	3 workers
Sledge (TCP, -LM)	13K	14K
Sledge (TCP, +LM)	13K	21K
Sledge (eRPC, -LM)	40K	54K
SledgeScale (eRPC, +LM, +EDF)	120K	300K

The table's first two rows demonstrate that adding our LM optimizations to Sledge improves throughput only when we scale up the number of workers. The third row shows that adding only eRPC provides a more significant gain with a single core, but levels off with more workers since lifecycle operations begin to dominate. Finally, the last row illustrates that SledgeScale, which integrates both eRPC and LM, achieves the highest overall improvement in throughput. This benefit is greater than that of each optimization individually, as it allows us to surpass both bottlenecks.

Figure 5 shows scalability as dispatchers and workers are added. The x-axis indicates total core count, including dispatcher cores. Nightcore and Sledge (with our eRPC optimization) both saturate at 50 kRPS due to a single HTTP-based dispatcher in Nightcore and lack of LM in Sledge (the original Sledge only reaches 15 kRPS). rFaaS scales up to 800 kRPS with 12 cores before hitting a bottleneck. SledgeScale can scale up to 800 kRPS with 3 dispatchers and 12 cores, further reaching 1.37 million RPS with 5 dispatchers and 20 cores. These results demonstrate that SledgeScale can effectively scale performance by adding more cores.

4.2 Density

We assess density with a no-op function, comparing latency between SledgeScale and rFaaS while varying the number of concurrently active functions. We duplicate and rename the no-op function code to simulate different function types.

For rFaaS, each worker process can handle only one type of request, thus to support X function types requires X processes to be started. We use our pseudo open-loop generator that opens a long running connection to the server for each function type.

In contrast, a worker in SledgeScale can handle any type of function, we start only one worker that will execute distinct sandboxes for each function type.

Single core: Initially, we measure function density on a single core (with both systems using a second core for a dispatcher in SledgeScale or an executor manager in rFaaS). We begin with a single function type at a fixed sending rate of 40k RPS, gradually increase the number of function types, and measure the 99.9th percentile latency.

Figure 6 shows that rFaaS achieves a lower 99.9th percentile latency than SledgeScale when the number of function types is small (less than 20), as fewer concurrent requests result in less context switch overhead. However, as the number of function types increases, rFaaS experiences a sharp rise in 99.9th percentile latency due to frequent context switches. In contrast, SledgeScale maintains a 99.9th percentile latency under 150 μs even with 2000 function types, as no thread context switching occurs on the worker core. Additionally, SledgeScale isolates each request in a clean sandbox, while rFaaS shares execution environments among requests of the same function type.

Multi-core: We next evaluate the impact of density on SledgeScale's scalability. We increase the number of request types to 10,000. We aim to assess whether performance scales with worker core count under high function density by gradually increasing the request rate. Figure 7 shows the results, with lines labeled as WXFY, where X is the number of worker cores and Y is the number of function types.

For a target 99th percentile latency of 60 μs, a single worker running one function (W1F1) achieves 100 kRPS, while one worker running 10,000 functions (W1F10000) achieves 25 kRPS. Scaling up to two workers, two running one function reach 170 kRPS, and two running 10,000 functions achieve 72 kRPS, nearly three times the performance of W1F10000. Two workers outperform one worker by effectively handling bursts. Scaling up to three dispatchers, each managing two workers (totaling six workers), results in a five-fold

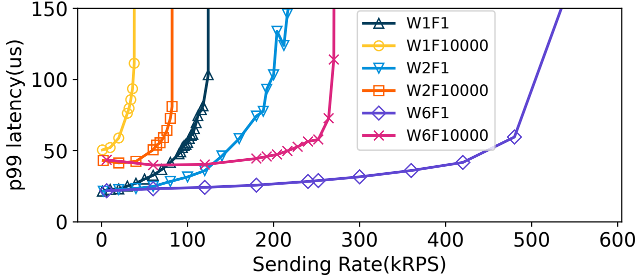


Figure 7: Scaling with W workers and F function types

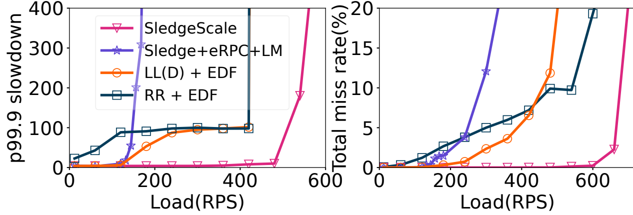


Figure 8: Performance of Dispatching Strategies under a Log-Normal Workload

performance increase compared to W1F1 and a ten-fold increase compared to W1F10000. Six workers running one function achieve 480 kRPS, and six workers running 10,000 functions (W6F10000) achieve 250 kRPS, maintaining the 99th percentile latency at *60 µs*. These results demonstrate that SledgeScale exhibits robust performance scalability even under extremely high function density.

4.3 Dispatching

We evaluate dispatching performance using the Log-normal workload shown in Table 3 since prior work describes it as the most representative workload for serverless, exhibiting a broad, heavy-tailed distribution of execution times [68] – the 99th percentile of requests exceeds 140 milliseconds, while short requests can be as brief as 10 microseconds.

Figure 8 demonstrates the results. Sledge, in which each worker competes for a batch of requests (batch size 5) from the global queue and schedules them locally using Round Robin, exhibits the worst performance due to its lack of deadline awareness. LL(D) initially outperforms RR by making more informed dispatching decisions based on the load of each queue. However, as the workload increases and queues build up, merely selecting a better worker without precise preemption becomes insufficient. Its performance converges with RR because both rely on a fixed 1 ms scheduling quantum, which limits preemption granularity and causes short requests to be blocked.

In contrast, SledgeScale consistently maintains low slowdown and low deadline miss rate under increasing load by always selecting the optimal worker and enabling fine-grained, deadline-aware preemption. This emphasizes the benefit of SledgeScale integrating the dispatcher with the scheduler to allow preemption precisely when it is needed.

4.4 Scheduling under Diverse Workloads

In this section, we evaluate the SledgeScale scheduling algorithm in comparison to Shinjuku and DARC with various workloads. We start 6 workers with 1 dispatcher for all workload tests. We set each task's deadline to 10 times its execution time and observe the 99.9th percentile latency and overall slowdown (client measured latency divided by execution time) as the load increases.

To get the best results for Shinjuku, we employ Single-Queue scheduling for workloads with exponential and log normal distributions, and Multi-queue scheduling for others. Our testing showed that a 15μs quantum yields optimal performance for Shinjuku, so we use this value in all experiments.

Fixed. This workload causes all the schedulers to use an equivalent FIFO order, letting us measure platform overheads under identical scheduling choices. Sledge (with our eRPC and Life Cycle optimizations, but lacking our dispatch and scheduling changes) achieves the highest performance, as shown in Figure 9, due to architectural differences: its workers use a busy-loop to fetch requests from a lock-free global queue, maximizing throughput. SledgeScale, on the other hand, employs a dispatcher that iterates over all workers to select the optimal one, and sleeps between requests to conserve CPU. SledgeScale comes close in busy-loop mode but remains slightly behind due to the cost of worker iteration (not shown). As discussed in §3.3, DARC and Shinjuku use late-binding, causing worker idleness and reduced throughput. When the network subsystem is handling packets, the dispatcher may miss opportunities to check worker availability and assign requests, resulting in idle workers. In contrast, SledgeScale uses early-binding to deliver requests directly to the worker queue, ensuring continuous worker engagement. As a result, for a target 99.9th percentile tail latency of 200μs, SledgeScale can sustain *1.47x* more traffic than DARC and Shinjuku.

High Bimodal. We employ two real-world applications - Image Resizing and Image Classification that have approximately 1:10 execution time ratio. SledgeScale performs best in Figure 12 by prioritizing short requests over long requests based on deadline. Shinjuku suffers from high tail latency for short requests due to frequent context switches and lower overall throughput. DARC requires 0.07 CPU cores for short requests. However, since CPU cores cannot be fractionally allocated, it reserves 1 core for short requests and 5 for long ones, leaving 0.93 cores underutilized, as long requests cannot use the cores reserved for short requests. This limits DARC to 85% of the load before dropping requests. Since short requests are over-provisioned, DARC can maintain a lower tail latency for short requests even under high load, as depicted in figure 12 (b), but this reduces the throughput

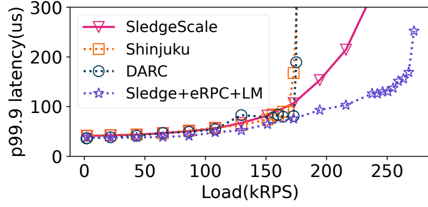


Figure 9: Fixed(10)

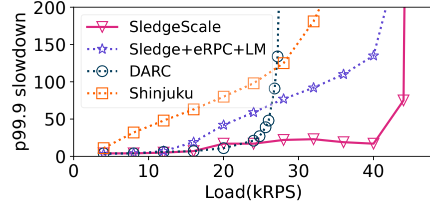


Figure 10: Exponential($\mu = 100$)

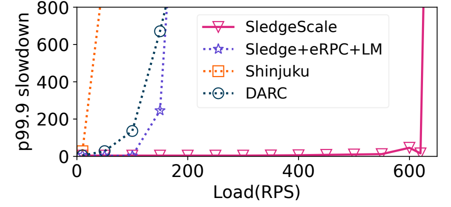
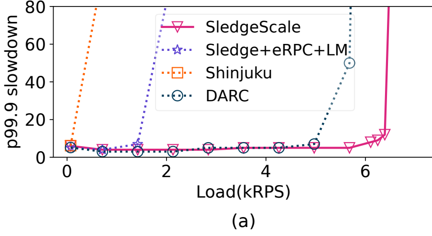
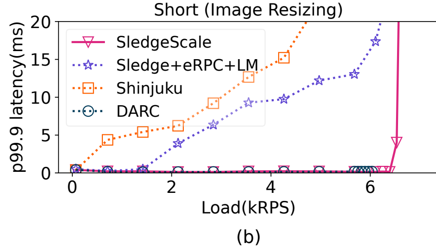


Figure 11: Log normal(-0.38,2.36)





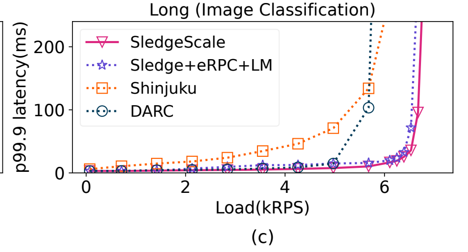


Figure 12: High Bimodal(50:26, 50:2070)

for long requests, as illustrated in figure 12 (c). Shinjuku performs poorly due to frequent preemptions, and Sledge shows similar performance when its scheduling quantum is reduced from 1ms to 15μs. As a result, SledgeScale can sustain *1.16x more load than DARC and 18.35x more than Shinjuku*, with a target 99.9th percentile slowdown of 40x. We omit experiments with a synthetic Extreme Bimodal workload that yielded similar results.

Exponential. This workload exhibits a continuous range of execution times, with most being quick but some being very long, and the result is shown in Figure 10. DARC divides this workload into two groups based on median execution time, each allocated 3 workers. DARC performs poorly with this coarse-grained CPU allocation, sustaining only 26K RPS before dropping requests. Conversely, SledgeScale is able to sustain 44K RPS with a 99.9th percentile latency of less than 10ms, achieving 1.7x the throughput of DARC and 1.2x that of Shinjuku. Moreover, *SledgeScale sustains 3.2x and 1.6x more load than Shinjuku and DARC with a slowdown target of 50.*

Log-normal. We divided the workload into two groups based on median execution time, allocating 3 workers to each group for DARC. Figure 11 displays the results. Similar to the exponential workload, DARC performs poorly with this coarse-grained CPU allocation. Overall, *SledgeScale sustains a load 5.9 times that of DARC and 47 times that of Shinjuku with a target 99.9th percentile slowdown of 200, indicating that the SledgeScale scheduling algorithm is robust even with wide variability in execution time and distributions.* In contrast, DARC and Shinjuku struggle with efficient CPU resource allocation and constant preemption overhead. This result highlights a key benefit of SledgeScale–the ability to seamlessly support a very wide range of request types without requiring a prior knowledge of the request distribution.

4.5 Deadline-Aware Scheduling

Deadlines in edge applications vary significantly depending on the function type. Certain applications demand extremely strict deadlines, while others can tolerate more relaxed timing constraints. For instance, in autonomous vehicles, detecting a person 20 meters away to prevent a collision requires a minimum response time of 200ms [27], whereas applications such as entertainment consoles are typically assigned the lowest priority. This variability requires that runtime schedulers must adapt to diverse deadline requirements. To evaluate the effectiveness of deadline-aware scheduling, we utilized three real-world applications commonly used in autonomous vehicles - SIFT, Segmentation and Classification - comparing our performance to Shinjuku and DARC.[3] Deadlines were set to twice the execution time for Segmentation and Classification, while SIFT's deadline was varied from 1.13 ms to 11.3 ms to illustrate how our deadline aware scheduler adapts. We use a fixed sending rate of 14.4 kRPS.

The results are presented in Figure 13(a-d), with Shinjuku excluded due to its significantly lower performance. These figures show that as SIFT's deadline increases, SledgeScale takes advantage of the slack to increase its latency while lowering the tail latency of the other two functions. Figure 13 (a) highlights that SledgeScale's deadline miss rate decreases as SIFT's deadline extends. In contrast, DARC, a deadline-unaware scheduler favoring shorter requests, exhibits consistent deadline miss rates and tail latencies.

Figures 13 (f-h) present the latency distribution for each function when SIFT's deadline is 11.3 ms (with red dashed lines indicating deadlines). DARC shows a shorter latency distribution for SIFT compared to SledgeScale in figure 13 (f),

[3] Based on the workload distribution (see Table 3) and the DARC algorithm, SIFT requires 2.75 CPUs and is allocated 3 CPUs, the Classification requires 0.56 CPUs and is allocated 1 CPU, and Segmentation requires 2.68 CPUs and is allocated 2 CPUs, given the total of 6 CPUs available.

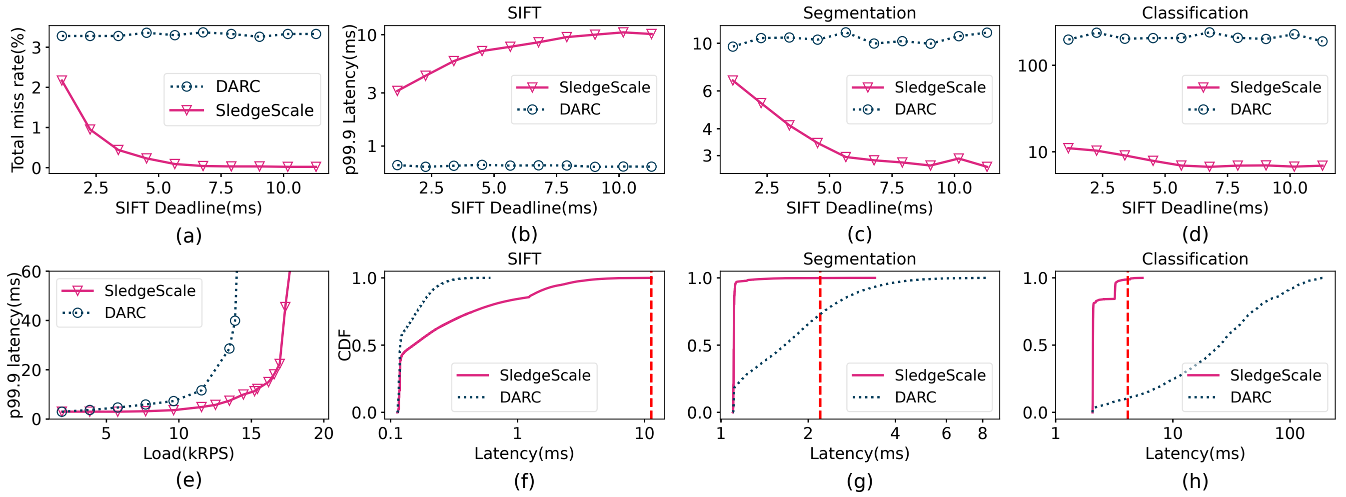


Figure 13: Scheduling performance under varying deadlines.

but this comes at the expense of Segmentation and Classification, which experience longer latencies and miss most of their deadlines as shown in figure 13 (g) and (h). In contrast, SledgeScale prioritizes Segmentation and Classification by postponing SIFT execution, as it is less urgent, resulting in a longer latency for SIFT while still meeting its deadline. Figure 13 (e) demonstrates that SledgeScale supports higher loads than DARC under the same target tail latency. Overall, SledgeScale adapts effectively to varying deadline settings and ensures more requests meet their deadlines.

5 Related Work

Cold Start. Reducing function cold-start latency has been a central focus of prior serverless research. Lightweight VMs such as Firecracker [2] and LightVM [52] reduce startup times from seconds to milliseconds, but this cost is still prohibitive for the sub-millisecond tasks we target. Pure user-space sandboxes using WebAssembly, e.g., Sledge [26] and Faasm [69], offer lower latency; we adopt Sledge due to its focus on low latency requests. Other approaches—including snapshots, checkpointing [4, 10, 19, 70, 76], and prefetching working sets [76]—still incur millisecond-scale startup (~1 ms), over 15× that of SledgeScale 's 60, μs. Pre-warming techniques [56, 57, 64–66] and caching strategies like FaasCache [25] reduce latency but increase resource consumption, conflicting with edge efficiency goals.

Serverless Scheduling. Most prior works on serverless scheduling [39, 44, 51, 71, 74] primarily focus on placement decisions in large datacenter environments—i.e., selecting the optimal node to execute an incoming function request in order to balance load or minimize resource cost. These approaches operate at the infrastructure or cluster level, optimizing where functions should run, while typically leaving intra-node execution to a simple FIFO order. In contrast, our work targets intra-node scheduling, which determines when and in what order functions should execute once they have been placed on a node. Specifically, we focus on meeting per-request deadlines and minimizing tail latency. This complements existing node-selection research by improving per-request deadline satisfaction after placement decisions have been made.

Previous studies [24, 31, 32, 37, 38, 82] have explored intra-node task scheduling. Syrup [37] proposed a generalized user-space framework that allows user-defined scheduling policies. Kaffes et al. [38] designed a centralized request scheduler that allocates tasks based solely on worker availability, without considering task deadlines. ghOSt [31] decouples the scheduling policy from the kernel by executing policy logic in user space, but it still relies on kernel agents for task dispatching and context switching—incurring higher overhead than pure user-space schedulers. Zhao et al. [82] further proposed a hybrid OS-level scheduler built on ghOSt to handle short and long tasks separately, yet this design introduces additional kernel interaction overhead. SFS [24] adopts a two-level scheduling approach using Linux FIFO and CFS, prioritizing short requests via Shortest Remaining Time First (SRTF). However, it extends long request execution times without considering Service Level Objectives (SLOs). Our algorithm uses earliest deadline first scheduling to naturally prioritize requests at user level, with zero remaining slack as a constraint. Isstaif et al. [32] modifies kernel scheduling to prioritize long, low-frequency requests and reduce short request interruptions. This method lacks flexibility for workloads with many long requests and requires kernel modifications, increasing maintenance costs.

rFaaS [14] also targets high-performance functions via kernel-bypass networking but relies on the standard Linux CFS scheduler, making it ineffective when scaling to many processes running low latency requests.

Microsecond Scale Scheduling for Non-Serverless Workloads. Minimizing request latency is critical since tail latency can be the dominant factor in user-perceived performance [16]. Focus on microsecond scale applications has grown in recent years, with many works considering how a single application can handle a mixture of fast and slow requests, such as a key-value store handling both cheap GET and expensive SCAN requests [17, 33, 36, 49, 54]. By optimizing IO paths and scheduling, these systems effectively support applications with single digit microsecond round trip delays. However, these prior works on microsecond-scale round trip latency often consider several different applications in their evaluations, and assume only one application runs per server and generally expect the request type distribution to be known a priori. In contrast, we focus on a multi-tenant serverless environment with a larger and less predictable set of application types.

We evaluate our approach against DARC (the scheduling algorithm from Perséphone) [17]. While DARC's strategy of reserving cores for short requests can improve performance, its effectiveness is highly sensitive to the allocation ratio between short and long cores. Under bursty arrival rates, short requests may experience head-of-line blocking due to the lack of preemption. Additionally, dynamically changing request distributions, common in serverless environments, can alter the core allocation ratio, making accurate prediction of request distributions challenging.

We also compare against Shinjuku [36], which employs preemption at very fine timescales to limit the impact of long requests on short ones. This works well if context switch cost is minimized and there is only few types of requests. However, in serverless environments with numerous request types and varying execution times, frequent, unnecessary preemptions degrade performance. Furthermore, the c-FCFS-based Single-Queue scheduling is less effective, struggling to select the most urgent request to execute and resulting in suboptimal performance.

Two recent projects by Iyer [33] and Luo [49] use compile-time instrumentation to improve latency, particularly for blind workloads with unknown costs. While these approaches minimize interruption overhead, determining the optimal frequency for inserting bookkeeping probes and setting a reasonable quantum remains challenging. SledgeScale promotes using dispatcher triggered preemption to simplify this problem and only trigger interrupts when necessary, but our approach requires workloads to be annotated with deadline information and execution time estimates.

Kernel bypass. Kernel bypass techniques have been widely researched to overcome the limitations of OS-based networking and scheduling [7, 14, 15, 17, 22, 23, 36, 53, 61, 62, 81]. Several works [34, 63] built user space networking stacks to improve throughput and connection scalability, while others like IX [8] and Shenango [60] propose new OS designs to better support latency sensitive applications. Junction [22] is designed for cloud applications and provides strong isolation for unmodified Linux native applications by filtering system calls to restrict access. eRPC [40] designed a general-purpose RPC library for latency sensitive data center networks. SledgeScale builds on top of eRPC based on the observation that the network within an edge site will rarely see packet loss.

6 Future Work

Admission Control. We focus on non-overloaded, soft-deadline scenarios in this paper. Admission control can help prevent overloaded or unschedulable situations—for example, when multiple tasks with tight, overlapping deadlines make it impossible to schedule all on time. Investigating these cases and designing admission control is left for future work.

Resource Adaptation. Adding more dispatchers improves scalability but also consumes CPU cores that could otherwise run workers. Future work will investigate the trade-off between a single dispatcher with N workers and M dispatchers with $N + 1 - M$ workers, and design adaptive scaling policies based on workload dynamics.

Memory Pool Auto-Scaling. Our current sandbox memory management does not reclaim excess memory after bursty or idle periods, potentially causing temporary resource overuse. Future work will explore adaptive memory pool management and auto-scaling policies to enhance resource efficiency.

7 Conclusions

We have presented SledgeScale, a WebAssembly-based serverless platform optimized for latency, throughput scalability, and deployment density. SledgeScale achieves these benefits through the combination of its dispatch and scheduling framework that uses function deadlines to effectively trigger preemptions only when they are needed. This helps eliminate head of line blocking and prevents the need to carefully tune scheduler interrupt quanta. The dispatcher seeks to balance load across workers and minimize waiting time of requests. Leveraging optimized sandbox lifecycle management, we can deploy new WebAssembly sandboxes in tens of microseconds, while ensuring each function invocation receives a sanitized heap for execution. Our implementation can complete a NoOp request within 13 microseconds, scales to over one million requests per second, and can handle running ten thousand distinct functions at the same time, achieving latency, scale, and density far beyond existing serverless platforms.

Acknowledgments: We thank our shepherd, Jayson Boubin, and the reviewers for significantly improving this paper. This work was supported by NSF JUNO 2210380 and ONR N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] 3GPP. 2018. *Study on Enhancement of 3GPP Support for 5G V2X Services.* 3GPP Technical Report 22.886. 3rd Generation Partnership Project (3GPP). Available at: `https://www.3gpp.org/ftp/Specs/archive/22_series/22.886/`.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Amazon Web Services. [n. d.]. Configure Lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html Accessed: 2025-03-24.

[4] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 730–746.

[5] Mohammad S Aslanpour, Adel N Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. 2021. Serverless edge computing: vision and challenges. In *Proceedings of the 2021 Australasian computer science week multiconference*. 1–10.

[6] Luciano Baresi and Danilo Filgueira Mendonça. 2019. Towards a serverless platform for edge computing. In *2019 IEEE international conference on fog computing (ICFC)*. IEEE, 1–10.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65.

[8] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2016), 1–39.

[9] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 23–33.

[10] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[11] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. doi:10.1145/3452296.3472888

[12] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 21–28.

[13] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. 2011. MEVBench: A mobile computer vision benchmarking suite. In *2011 IEEE international symposium on workload characterization (IISWC)*. IEEE, 91–102.

[14] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, St. Petersburg, FL, USA, 897–907. doi:10.1109/IPDPS54959.2023.00094

[15] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. 2018. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 373–387.

[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

[17] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. doi:10.1145/3477132.3483571

[18] Jaana Dogan. [n. d.]. HTTP load generator, ApacheBench (ab) replacement. https://github.com/rakyll/hey.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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*. 467–481.

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[21] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*. 265–276.

[22] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making kernel bypass practical for the cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 55–73.

[23] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[24] Yuqi Fu, Li Liu, Haoliang Wang, Yue Cheng, and Songqing Chen. 2022. SFS: Smart OS scheduling for serverless functions. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[25] Alexander Fuerst and Prateek Sharma. 2021. 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 (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. doi:10.1145/3445814.3446757

[26] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Lightweight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 265–279. doi:10.1145/3423211.3425680

[27] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Joseph E Gonzalez, and Ion Stoica. 2022. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 453–471.

[28] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A Wright, Joseph E Gonzalez, and Ion Stoica. 2021. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 8806–8813.

[29] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C Nachiappan, Mahmut Taylan Kandemir, and Chita R Das. 2020. Fifer: Tackling resource underutilization in the serverless era. In *Proceedings of the 21st International Middleware Conference*. 280–295.

[30] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press.

[31] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[32] Al Amjad Tawfiq Isstaif and Richard Mortier. 2023. Towards latency-aware linux scheduling for serverless workloads. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. 19–26.

[33] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 466–481. doi:10.1145/3600006.3613136

[34] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[35] Zhipeng Jia and Emmett Witchel. 2021. 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 (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. doi:10.1145/3445814.3446701

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {usecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[37] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 605–620.

[38] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*. 158–164.

[39] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 289–305. doi:10.1145/3542929.3563468

[40] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

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

[42] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices* 48, 8 (2013), 69–80.

[43] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.

[44] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-aware, resource-efficient function scheduling for serverless computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 32–47.

[45] Changyuan Lin and Hamzeh Khazaei. 2021. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (March 2021), 615–632. doi:10.1109/TPDS.2020.3028841 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[46] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (jan 1973), 46–61. doi:10.1145/321738.321743

[47] Mingzhe Liu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Ran Zheng, and Liting Hu. 2022. Towards low-latency I/O services for mixed workloads using ultra-low latency SSDs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.

[48] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R Larus, and Albert Greenberg. 2011. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation* 68, 11 (2011), 1056–1071.

[49] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2024. Efficient Microsecond-scale Blind Scheduling with Tiny Quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24, Vol. 2)*. Association for Computing Machinery, New York, NY, USA, 305–319. doi:10.1145/3620665.3640381

[50] Xiaosu Lyu, Emil Abbasov, Gabriel Parmer, and Timothy Wood. 2025. A High-Density, Deadline-Aware, and Scalable Serverless Platform for Sub-Millisecond Functions at the Edge. In *2025 IEEE Cloud Summit*. IEEE, 35–38.

[51] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 303–320.

[52] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233. doi:10.1145/3132747.3132763

[53] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 175–186.

[54] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for {Microsecond-Scale} Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1–18. https://www.usenix.org/conference/nsdi22/presentation/mcclure

[55] Microsoft. [n. d.]. Available instance SKUs. web:https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal#available-instance-skus Accessed: 2025-03-24.

[56] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. 2021. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*. 168–181.

[57] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2021. Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 168–181. doi:10.1145/3472883.3487014

[58] Arpan Mukhopadhyay and Ravi R Mazumdar. 2015. Analysis of randomized join-the-shortest-queue (JSQ) schemes in large heterogeneous processor-sharing systems. *IEEE Transactions on Control of Network Systems* 3, 2 (2015), 116–126.

[59] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. 2017. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing* 21, 4 (2017), 64–71.

[60] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[61] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[62] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2015), 1–30.

[63] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[64] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa $ T: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM symposium on cloud computing*. 122–137.

[65] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

[66] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

[67] K Salchow. 2007. Load balancing 101: Nuts and bolts. *White Paper, F5 Networks, Inc* (2007).

[68] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[69] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[70] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 663–677.

[71] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*. 138–152.

[72] Ion Stoica and Hussein Abdel-Wahab. 1995. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22* (1995).

[73] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. 2017. On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration. *IEEE Communications Surveys Tutorials* 19, 3 (2017), 1657–1681. doi:10.1109/COMST.2017.2705720

[74] Qinqin Tang, Renchao Xie, Fei Richard Yu, Tianjiao Chen, Ran Zhang, Tao Huang, and Yunjie Liu. 2022. Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach. *IEEE Internet of Things Journal* 9, 20 (2022), 19634–19648.

[75] Dimitrios Tomaras, Michail Tsenos, and Vana Kalogeraki. 2023. Prediction-driven resource provisioning for serverless container runtimes. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 1–6.

[76] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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*. 559–572.

[77] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 55–64.

[78] Yifan Wang, Shaoshan Liu, Xiaopei Wu, and Weisong Shi. 2018. CAVBench: A benchmark suite for connected and autonomous vehicles. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 30–42.

[79] Adam Wierman and Bert Zwart. 2012. Is tail-optimal scheduling possible? *Operations research* 60, 5 (2012), 1249–1257.

[80] Renchao Xie, Qinqin Tang, Shi Qiao, Han Zhu, F Richard Yu, and Tao Huang. 2021. When serverless computing meets edge computing: Architecture, challenges, and open issues. *IEEE Wireless Communications* 28, 5 (2021), 126–133.

[81] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 195–211.

[82] Yuxuan Zhao, Weikang Weng, Rob van Nieuwpoort, and Alexandru Uta. 2024. In Serverless, OS Scheduler Choice Costs Money: A Hybrid Scheduling Approach for Cheaper FaaS. In *Proceedings of the 25th International Middleware Conference*. 172–184.