USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Hierarchical Integration of WebAssembly in Serverless for Efficiency and Interoperability

Mohammadamin Baqershahi, Changyuan Lin, and Visal Saosuo, *University of British Columbia;* Paul Chen, *Huawei Technologies Canada;* Mohammad Shahrad, *University of British Columbia*

https://www.usenix.org/conference/nsdi26/presentation/baqershahi

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Hierarchical Integration of WebAssembly in Serverless for Efficiency and Interoperability

Mohammadamin Baqershahi
University of British Columbia

Changyuan Lin
University of British Columbia

Visal Saosuo
University of British Columbia

Paul Chen
Huawei Technologies Canada

Mohammad Shahrad
University of British Columbia

Abstract

Modern serverless systems suffer from low resource efficiency, which maps to high per-unit costs. This comes from the high isolation overhead (e.g., low resource sharing, slow startup, etc.), as well as resource wastage incurred by conservative resource scaling (e.g., keep-alive). Language runtimes such as WebAssembly (Wasm) can reduce isolation overhead without compromising security. Existing Wasm-based serverless approaches fall into one of these categories: supporting only Wasm workloads, failing to leverage existing capabilities of modern serverless and cloud platforms, or falling short of leveraging Wasm's true potential. This work introduces a dense hierarchical architecture to securely co-locate Wasm-based applications from different customers within the same container sandbox. We show how this design preserves the container-based serving model, which allows leveraging existing platform capabilities and supporting non-Wasm-based workloads. Our system, Wasabi, leverages this architecture alongside resource-aware scaling, queuing, and overbooking to offer much higher density than state-of-the-art serverless systems with similar or better performance.

1 Introduction

Serverless computing has become increasingly popular and widely adopted over the past few years. With serverless, developers are able to quickly create scalable, event-driven applications, use massively parallel processing in cloud systems, and pay only for what they use. Datadog's August 2023 survey of cloud user telemetry [70] indicates serverless offerings have been adopted by over 70% of AWS users and over 50% of Azure and Google Cloud users. A defining feature of serverless workloads is their short execution time and low resource consumption per invocation [84, 99, 110, 125]. This leads to a high degree of co-tenancy, increasing the relative overhead of isolation mechanisms [109].

WebAssembly (Wasm) has gained traction in recent years as a solution to this barrier. Originally developed for high-performance and portable code execution in web browsers, Wasm offers unique server-side benefits. It boasts shorter initiation times compared to containers or lightweight VMs [81, 112], and a sandboxed environment with robust security features [88], protecting against buggy or malicious code.

In the context of serverless, there have been few proposals to use Wasm to increase density and efficiency. An approach is building Wasm-customized serverless platforms [112]. While Wasm provides fast, secure, and efficient execution, many existing applications are already developed and tested for containerized runtimes, and migrating them to Wasm can require significant developer effort, including code modifications and library adaptation, necessitating simultaneous support for the existing container-based architecture. Supporting both Wasm and containers allows users to run legacy applications without costly rewrites, while enabling gradual migration of workloads to Wasm to take advantage of its lightweight execution. Moreover, modern serverless platforms provide a wide range of advanced capabilities such as highly available (HA) control planes, or rich security, networking, and monitoring policies managed by sidecar proxies [106, 108]. This makes it difficult to justify rebuilding and maintaining entire Wasm-customized platforms from scratch to replicate existing capabilities.

Another approach to use Wasm in serverless without redesign is directly integrating Wasm runtimes into container-based serverless platforms [32, 45, 57]. These systems rely on container runtimes and use Open Container Initiative (OCI) image specification to seamlessly pack Wasm modules into container-based platforms. As we show in this paper, this common approach is too crude and leaves substantial resource efficiency untapped.

We pursue a third approach and explore the right balance between system specialization and ease of integration into existing platforms. We present Wasabi[1], a system for high-density integration of Wasm in existing serverless systems. **The core enabler of this high density in Wasabi is its novel two-level isolation architecture, where many Wasm modules from different applications and users are securely**

[1] https://github.com/ubc-cirrus-lab/wasabi

co-located within coarse-grained units of traditional isolation abstractions (containers, pods, etc.). Inherently, there are many benefits to this approach, including: (1) seamless integration with existing container-based platforms, (2) amortizing the containerization overhead across many Wasm modules, (3) efficiency gains via statistical cancellation of demand fluctuations across applications, and (4) eliminating high cold start times of containers for the majority of invocations.

Wasabi's new architecture has implications for resource allocation (e.g. resource fragmentation and scaling). We show how minimal modifications to the platform effectively address these challenges and employ a combination of techniques such as resource-aware autoscaling and queuing, as well as contention-aware overcommitment to unlock the potential of this architecture. Beyond the system itself, we provide empirical insights into how workload aggregation impacts burstiness, how platforms can better absorb demand spikes, and the implications for capacity planning.

We prototype Wasabi on Knative [36], a production-grade, open-source serverless platform used by serverless offerings such as Google Cloud Run and IBM Code Engine. Using production invocation traces [110] to compare Wasabi with default Knative and state-of-the-art Wasm integration approaches, we observe 8.15-11.55x lower memory allocation and 9.67-13.73x lower CPU allocation, translating into higher packing density. Increasing density without designing new platforms enables serverless providers to offer more affordable services, improve resource utilization, and enhance overall system efficiency.

2 Background and Motivation

2.1 Serverless is not necessarily cheap!

Serverless architecture's scale-down-to-zero and pay-per-use features can significantly reduce costs for developers, particularly for applications with highly variable traffic. Serverless offers unparalleled scalability [82]. Alternative architectures are slow to scale and incur costs even when idle. However, this cost-saving comes with a higher per-unit-time resource cost in serverless. A recent study [96] shows that AWS Lambda costs 2.43x and 2.09x as much as AWS Fargate and EC2, respectively. A notable example of high serverless costs is AWS Prime Video moving from Lambda to a monolithic architecture, reducing costs by 90% [77].

Serverless being expensive is partially due to its pay-per-use cost model, which limits providers to generate revenue only when functions are running. This is unlike VMs that usually have a constant hourly fee, profitable just by being alive. Yet, serverless comes with added system overhead on the provider side for autoscaling, container keep-alive/pre-warming, event handling, and more [66]. Additionally, code executions from different users must be isolated for security and performance reasons. This isolation introduces significant overhead, both spatially and temporally. Spatially, the small

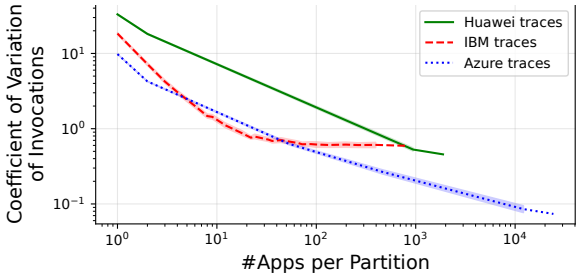

Figure 1: Load variability decreases as a larger fraction of applications are consolidated. Wasabi uses this statistical principle to increase the resource efficiency.

resource consumption of serverless workloads [84, 99, 110] increases the relative resource consumption ratio by the isolation mechanism. Temporally, the short execution time of serverless workloads [84, 99, 101, 110, 125] amplifies the impact of isolation initiation (i.e., cold starts).

The performance penalty of cold starts is substantial enough that most providers keep containers alive for a period after execution [101, 110]. Even on platforms where keep-alive time is not billed directly, its cost is passed to developers via higher per-unit-time charges during billing window. Researchers have proposed reducing keep-alive costs through predictive [101, 105, 110] or caching [73] techniques. However, keep-alive costs will persist as long as cold start durations remain on the same order of magnitude as function execution times [85, 110]. Notably, incorporating cold start latency into the billed duration is now common among public serverless platforms, including AWS Lambda [79] and Google Cloud Run [69].

Resources reserved to keep an application instance warm (e.g., memory) cannot always be used by other applications, thereby amplifying the overhead of keep-alive. However, if these resources could be securely shared across applications, the benefits would be twofold. First, the likelihood of reusing warm containers would increase due to higher traffic volume. Second, combining traffic from different applications would reduce relative traffic variations, demanding less container scaling. This statistical property is the cornerstone of the cloud's economic model. To demonstrate this effect, we calculated the Coefficient of Variation (CV) of invocations over time for various fractions of Azure Functions traces [110], IBM Cloud Code Engine Traces [101], and Huawei FaaS Traces [85]. Fig. 1 shows reduced load variability as a larger number of applications get grouped. Scaling up results in cold starts, while scaling down leads to wastage from keep-alive, regardless of how minimal. **An architecture that securely and efficiently shares resource allocations across different applications is fundamentally less wasteful**.

2.2 WebAssembly

WebAssembly [80] (Wasm) is a binary instruction format, originally designed to enable near-native execution perfor-

mance in browsers. Wasm offers portability across devices and architectures, providing safe and sandboxed execution through memory safety and control flow integrity. It ensures secure execution using a linear memory model, with strict boundaries enforced by Wasm runtimes during memory access [56]. Code from various languages, such as C, C++, Go, C#, and Rust, can be compiled to Wasm and executed safely.

Using Wasm modules for isolation offers faster startup times and a smaller memory footprint compared to VMs and containers. Consequently, Wasm has been recently adopted for isolation on the server-side [15,26,75,76,81,103,107,112,117,122]. A group of systems [22,45,57] shifts from container and VM-based architectures to Wasm and directly integrates Wasm runtimes into existing serverless platforms. However, this introduces a few major challenges. Sidecar containers (e.g., Knative's queue proxy) are often used in orchestration systems and serverless platforms for different purposes, such as traffic management, observability, secure communication, and more [33, 36, 108]. Neither implementing these rich functionalities directly in Wasm is practical, nor is it economically viable to re-implement them externally from scratch in a Wasm-specific manner. As a result, integrating Wasm directly at the final hop (e.g., within containers) is recommended as an alternative by another group of systems [45,57]; the container hosting user code is replaced with a Wasm sandbox, and sidecar container(s) are allocated per Wasm sandbox. However, this approach undermines most of the resource efficiency and startup performance gains offered by Wasm. Wasabi seeks a middle ground in this design spectrum.

3 Architecture

Wasm should not be viewed as an all-or-nothing proposition that is either ready for all applications or none at all. Instead, we see it as a powerful enabler of high-density computing today, with its potential set to expand as the technology matures. We aim for an architecture that maximizes the use of Wasm while co-existing with existing container-based applications.

3.1 Hierarchical Two-Level Isolation

Wasabi has a hierarchical two-level architecture for resource and runtime isolation (Fig. 2). Coarse-grained isolation is provided by the container runtime (e.g., *containerd*) and multiple Wasm modules from different applications and users share each container. This design ensures no platform redesign and supports runtime heterogeneity, since the core control plane still deals with containers as the basic unit of orchestration and scaling. It also offers an additional layer of security. As we show later in the paper, the container hosting Wasm module is special, however. It comes with extra features and is right-sized to balance scaling and resource fragmentation. We refer to the unit of coarse-grained resource scaling as a *WasmBox* in this paper, not to confuse the reader with regular sandboxes concurrently served in the platform. Each WasmBox has a

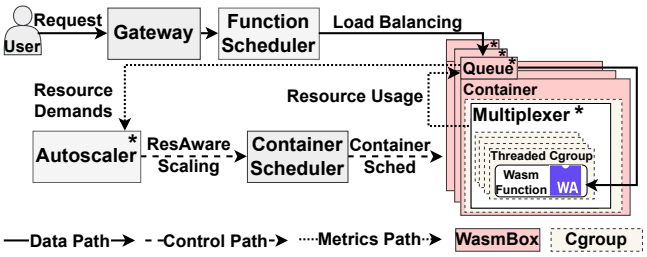

Figure 2: The serverless system architecture for high resource efficiency without platform overhaul. Asterisks mark components we implement or modify.

multiplexer container to run different Wasm modules, as well as a sidecar container.

Fine-grained isolation comes with encapsulating each Wasm-compatible serverless function as a separate Wasm module. In each WasmBox, many Wasm modules run in the secure, isolated sandbox provided by software-based fault isolation (SFI) [118]. Wasm ensures that each function runs in its own dedicated environment with predefined memory boundaries and limits, preventing any function from accessing or interfering with the memory space of the co-located functions. Additionally, the multiplexer places each Wasm module in a threaded Cgroup for CPU limit enforcement. This is needed, as resource isolation is crucial to mitigate noisy neighbor effects and to provide cheap, sub-core allocation, which is popular among serverless workloads [84, 101]. Wasabi does not require any modifications to the source code beyond the standard Wasm compilation process.

Besides ensuring resource isolation and security, the two-level architecture allows for seamless integration with widely used container-based serverless platforms like Knative and OpenFaaS by leveraging mature container orchestration platforms such as Kubernetes. This integration enables our system to leverage the extensive capabilities of Kubernetes and cloud-native serverless platforms—including resource planning, container scheduling, and monitoring—without requiring us to build this functionality from scratch. It also enables heterogeneous workloads (e.g., Wasm-based functions and non-Wasm-based workloads) to operate on the same cluster. Additionally, this architecture eliminates the need for separate sidecar containers (e.g., networking, monitoring, and queuing) for each individual function, reducing the overall resource footprint and increasing the deployment density, thus increasing the resource efficiency of the system.

WasmBox size is a tunable parameter that impacts performance and resource allocation efficiency. Smaller WasmBoxes increase scaling and increase the relative resource overhead of sidecar containers. Conversely, larger WasmBoxes can lead to fragmented, underutilized resources, reducing overall efficiency. §6.4 provides a detailed characterization of this and other exposed parameters and their associated trade-offs. Providers can maximize gains by periodically tuning them via offline simulation on representative workload traces.

The following sub-sections detail the techniques we use to unlock the potential of Wasabi's two-level isolation architecture. The goal is to increase the packing density as much as possible without compromising performance.

3.2 Resource-Aware Autoscaling

As different functions running in each WasmBox can have different resource consumption patterns, using existing concurrency-based scaling methods [16, 17, 29] for scaling WasmBoxes would be meaningless. A bulky function with high resource demands and long execution can put a heavy load on its host WasmBox while concurrency level remains low. On the other hand, in existing container orchestration platforms like Kubernetes, using resource usage metrics to scale WasmBoxes is challenging due to multiple factors. For example, Kubernetes' default usage-based autoscaling solution, `HorizontalPodAutoscaler` (HPA) [29], does not natively support scaling down to zero [12], undermining the cost-efficiency of serverless. Also, the minimal metrics collection interval of Kubernetes Metrics Server for autoscaling is 10 seconds due to the excessive performance overhead of `kubelet` metrics collection [31, 49], which limits the control loop speed of the autoscaler (e.g., HPA [29] and KEDA [39]) and can lead to decreased throughput and responsiveness.

For the above reasons, instead of using concurrency levels or actual usage, we propose a demand-driven, resource-aware autoscaling mechanism in Wasabi. This approach is conceptually similar to Kubernetes node autoscaling, widely used on public clouds, which provisions nodes based on pod resource demands [41]. However, we propose applying demand-driven, resource-aware autoscaling at a lower abstraction level (i.e., container/pod). Specifically, the autoscaling becomes based on the sum of the resource demands of all the Wasm-based functions that are currently running ($R_{Running}$) or waiting in the queue (R_{Queued}) in all WasmBoxes. The queue (further discussed in §3.3) in each WasmBox performs fast bookkeeping of the total resource demands of in-flight requests in real time. The autoscaler polls the resource demand counter from these queues periodically (e.g., every second), calculates the average across all counters, and computes the moving average for the predefined window size to smooth out transient fluctuations in resource demands. By calculating the ratio between this moving average and a predefined per WasmBox resource allocation (R_{WB}), the autoscaler determines the number of WasmBoxes needed to serve all Wasm modules:

$$n = \frac{\sum_{i \in WBs}(R^i_{Running} + R^i_{Queued})}{R_{WB} \times (1 + r_{OC})} \tag{1}$$

r_{OC} represents the degree of resource overcommitment. We use similar memory and CPU overcommitment levels to simplify parameter tuning, but they can be adjusted independently. We discuss how Wasabi handles overcommitment later in this section and evaluate its effect in §6.4. If the number of desired WasmBoxes (n) does not match the currently running number, the autoscaler informs the scheduler to scale up or down the WasmBoxes, as illustrated by the control path in Fig. 2. When a WasmBox remains idle without in-flight requests, the resource demand counter reaches zero, and it is scaled down.

3.3 Request Queues & Safe Overcommitment

Each WasmBox has a queue responsible for (1) tracking the resource demands of in-flight requests and the actual resource usage of the multiplexer container, and (2) queuing the requests based on both metrics in a resource-aware manner. Essentially, it acts as a reverse proxy within the WasmBox that receives the request from the function scheduler, holds or throttles the request in case of insufficient capacity, and forwards the request to the multiplexer container when available. The queue maintains a counter that tracks the resource demands for all requests in-flight in WasmBox. When a request enters the queue, its user-defined resource demand is added to the counter, and when completed, the demand is subtracted upon returning the response. The queue also receives the actual resource usage metrics (e.g., CPU utilization rate and memory currently in use) from the multiplexer container.

As discussed in §3.2, Wasabi allows for reduced scaling based on degree of overcommitment (r_{OC}) to run more Wasm functions concurrently, even when the total resource demands of these function requests exceed the actual capacity of the container. Overcommitment is a classic technique to enhance the efficiency of cloud systems. However, it also introduces the risk of resource shortage. CPU resource shortage results in performance degradation, while memory shortage can lead to out-of-memory (OOM) kills. Two design aspects allow WasmBoxes to sustain capacity shortage:

A. Resource-aware admission control at request queues: When deciding to admit a request (r) from the queue into the multiplexer (mux), the request queue checks that both of the following conditions are met:

$$\begin{cases} 1: mem_r < \left((1 + r_{OC}) \times mem^{capacity}_{mux} - mem^{alloc}_{mux}\right) \\ 2: cpu^{used}_{mux} < \left((1 + r_{OC}) \times cpu^{capacity}_{mux}\right) \end{cases} \tag{2}$$

B. Swap memory: For memory, Eq. (2) considers the difference between capacity and *allocated* memory (not *used* memory) for admission control. Functions already running in the multiplexer may use more memory (within their allocation) after a new request is admitted. This creates the risk of the WasmBox going out of memory if a large fraction of running Wasm modules have simultaneous memory usage bursts. To handle this possibility gracefully, Wasabi allows each multiplexer container to use the swap space for handling transient memory usage spikes. Swap memory is more of a safety valve, and we did not observe any swapping being triggered in our experiments (§6).

3.4 Memory Pre-allocation

Per-request resource isolation reduces noisy neighbor effects. However, since memory allocation in Wasm is more CPU intensive than native execution [13,23,27], sub-core per-request CPU limits can cause disproportional performance degradation. Wasabi overcomes this by memory pre-allocation. Each WasmBox maintains a memory pool to accelerate memory pre-allocation during Wasm module initiation. Pre-allocating the full memory limit wastes resources and introduces unnecessary delays, as functions often do not use their entire requested amount [84]. We empirically found that pre-allocating around 20% of memory limit strikes a balance between performance and resource utilization gains (full study in §6.4).

4 Security Model

Software-based fault isolation (SFI) with Wasm is now used in multi-tenant production environments, including at Cloudflare (Workers [8] and Pages [7] services) and Fastly Compute [62]. Wasm offers isolation features such as strong memory safety guarantees and control-flow integrity [80, 120]. It follows a capability-based security model through WebAssembly System Interface (WASI) [51,89] – the host explicitly controls what resources each module can access, such as specific files or network sockets/endpoints, based on what is actually needed [88, 89]. This supports the principle of least privilege, limiting the attack surface. Unlike containers, Wasm modules run in sandboxed execution environment with a restricted system interface, reducing the potential for privilege escalation or cross-function interference [81, 102]. Security analyses show that the Wasm attack surface is generally small, especially when the host/runtime is properly configured and unnecessary capabilities are not exposed [89, 102, 119].

We consider a multi-tenant setting in which attackers can submit arbitrary untrusted functions for execution. The attacker aims to (1) break tenant isolation (e.g., reading/modifying another tenant's memory or request context) and/or (2) access host resources beyond those explicitly granted, such as filesystem paths or network endpoints. Our security goal is to provide a baseline level of isolation comparable to existing production and research Wasm-based systems: memory isolation via Wasm SFI and a deny-by-default, capability-based host interface where only explicitly granted resources are available to a module (e.g., specific files and endpoints). We assume the host OS/kernel and the Wasm runtime are part of the trusted computing base; Wasabi does not protect against a compromised host or kernel.

Similar to the production systems mentioned earlier, Wasabi uses **Wasm SFI** and exposes only the minimal set of interfaces and resources to each function. Access is granted solely to the files and endpoints explicitly defined. Wasabi employs additional isolation layers for added security. It enforces both **container-level and thread-level cgroup isolation** compared to the V8 engine used by Cloudflare Workers [4, 28], and an additional container-level cgroup layer compared to Faasm. Wasabi further strengthens isolation by creating **separate runtime instances for requests**. This contrasts with typical serverless systems, which can reuse a sandbox for multiple executions of the same function, whether concurrently or over time. Wasabi **destroys the ephemeral execution context** (e.g., Wasmtime Engine [2], WasmEdge VM [6], and threaded cgroup) after each request to avoid potential state leakage. Wasabi routes requests through queuing/load balancing middleware (e.g., queue proxy and activator in Knative [5]) to **avoid direct WasmBox HTTP endpoint exposure**, and the optional Kubernetes gateway-level security extensions (e.g., Guard Gate [3] and Knative Security-Guard [1]) can be configured to block/terminate misbehaving requests and WasmBoxes. Hardening isolation beyond this baseline is orthogonal to our main contribution.

Depending on use case and threat model, a provider can deploy Wasabi in four varying isolation models. *Model-1*: functions from different users share WasmBoxes with Wasm-level isolation; *Model-2*: sharing is restricted to functions of the same organization/subscription; *Model-3*: only functions of the same application can share a WasmBox; and lastly *Model-4*: for highly sensitive use cases only instances of the same function are allowed to share a WasmBox.

5 Implementation

We build Wasabi on top of the popular cloud-native serverless platform Knative [36]. Knative adds a series of components (e.g., Autoscaler, Queue-Proxy, and Activator) to Kubernetes [38] to simplify the deployment, scaling, and management of serverless applications in Kubernetes clusters. Kubernetes is a mature container orchestration platform that automates the deployment, scaling, and management of containerized applications at scale. This allows Wasabi to leverage extensive capabilities of Knative and Kubernetes to manage the entire life cycle of serverless workloads effectively, such as resource definition, request routing, container scheduling, resource allocation, autoscaling, event handling, and monitoring. Fig. 3 illustrates the system implementation of Wasabi. Wasabi relies on Kubernetes for its traffic routing, resource management, and container orchestration capabilities. We adapt several Knative components (i.e., Queue-Proxy, Autoscaler, and Activator) for resource-aware autoscaling, request queuing, and resource overcommitment, and implement the multiplexer for running Wasm-based functions within the Knative service user container.

5.1 WasmBox Implementation

The WasmBox is deployed as a pod of Knative service (ksvc). We added a new request serving layer with resource monitoring and Cgroup management logics in each WasmBox; implemented in about ~1.55k SLOC of Golang. It consists of an HTTP *request multiplexer* for handling the request-response

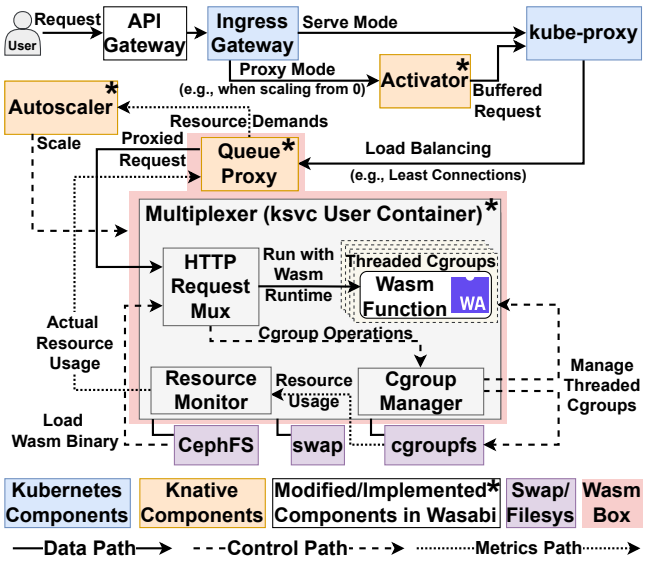

Figure 3: The system implementation of Wasabi. Asterisks denote new or adapted Knative components.

cycle for function invocation requests, a *Cgroup manager* for controlling threaded Cgroups dedicated to Wasm-based functions, and a *resource monitor* to collect actual resource usage metrics. The multiplexer runs in the Knative service user container and exposes the HTTP endpoint for receiving function invocation traffic and the readiness probing. A persistent volume provided by the Ceph file system [14, 121] is attached to and shared by each container where the multiplexer runs, storing the Wasm binary files of Wasm-based functions.

The host-level Cgroup file system is attached to each multiplexer container and accessible to the Cgroup manager, which controls threaded Cgroups in the container's Cgroup hierarchy. The resource monitor runs in a separate goroutine that periodically collects the actual resource usage by reading the Cgroup interface files (e.g., `cpu.stat` and `memory.current`) located in `/sys/fs/cgroup` inside the container. The resource monitor periodically sends the resource usage to each WasmBox's queue proxy for resource-aware queuing.

The HTTP request multiplexer has a request handler that upon receiving each request, spawns a new goroutine and pins it to the OS thread assigned by the Go scheduler (i.e., `runtime.LockOSThread`). Without pinning, the Go scheduler may move it between OS threads, compromising cgroups enforcement, which requires a fixed thread ID (TID). Then, the handler extracts the resource demand (i.e., memory and vCPU cores) from the request header, pre-allocates the memory proportional to the limit, and calls the Cgroup manager to create a threaded cgroup. The Cgroup manager accesses the host cgroup file system (i.e., the root cgroup of the container), creates a threaded subtree, and places the thread in the threaded cgroup by writing its TID to `cgroup.threads`. The created threaded cgroup is configured with resource limits based on the extracted resource demands to enforce the resource limit for the request. At this point, the system-level thread behind the handler goroutine runs in the threaded cgroup. Then, the handler loads the Wasm binary file from the Ceph file system, limits the Wasm sandbox linear memory and execute it using the Wasm runtime SDK [55].

We use Wasmedge [53, 55], a lightweight and high-performance state-of-the-art Wasm runtime, to run Wasm binaries in isolated sandboxes. However, the multiplexer design is not tied to a specific Wasm runtime. After execution, the handler captures the Wasm function output from *stdout*, removes the associated threaded cgroup, and returns the response to the client.

5.2 Adapted Knative Components

We adapt the following Knative Serving components to accommodate the hierarchical two-level isolation architecture and enable effective resource management and resource-aware traffic control for Wasabi: Queue-Proxy (QP), Autoscaler, and Activator. Our modifications consist of ~1.7k SLOC, while Knative Serving consists of ~2M SLOC. This minimal change (less than 0.1%) ensures that production-grade serverless platforms remain unaffected by the integration. We modify the Kubernetes Custom Resource Definitions (CRDs) of Knative to add Wasabi-specific configuration options, such as the in-flight resource demand, scaling target, and hard limit on containers' in-flight resource demands. To ensure compatibility, we implement a feature gate so that only Knative services with the Wasabi feature enabled use the adapted Knative components. This means that standard Knative services remain unchanged and agnostic to Wasabi, operating simultaneously for non-Wasm functions.

The original queuing logic of the QP relies on concurrency or requests per second (RPS) metrics to control the incoming traffic, which may not accurately reflect each WasmBox's actual load, as discussed in §3.3. Therefore, we adapt the QP by adding a middleware layer before the reverse proxy that forwards requests to the WasmBox. This middleware extracts the resource demand from the request HTTP header and maintains a counter to track the current resource demands of all in-flight requests within the QP and the multiplexer. For resource-aware traffic control, the middleware implements a breaker mechanism using a semaphore that represents resource demands. Each unit in the semaphore corresponds to a unit of resource, such as 1 MB of memory or 0.1 vCPU cores. Depending on the current resource demand levels and the predefined threshold (i.e., the hard limit on in-flight resource demands), the middleware temporarily holds or throttles requests, as discussed in §3.3. Also, the in-flight resource demand metrics tracked by the middleware, along with the concurrency and RPS metrics, are made available to the Autoscaler. The Activator is adapted similarly to track and report resource demands under the proxy mode [37], in which the requests are buffered and routed through the Activator when

scaling from zero or the container capacity in terms of concurrency or resource is not enough for the incoming requests.

Knative's Autoscaler scales based on requests per second (RPS) or concurrency (in-flight requests). These metrics can be ineffective under Wasabi's hierarchical two-level isolation architecture, as discussed in §3.1. Therefore, we adapt the Autoscaler to add the in-flight resource demand as an additional scaling metric following the system design presented in §3.2. The Autoscaler periodically (every second) retrieves the in-flight resource demand of each WasmBox through the metrics endpoint exposed by the QP, calculates the average resource demand across all WasmBoxes belonging to the same Knative service, and smooths the data using a windowed average. The Autoscaler then calculates the desired pod count based on the ratio between the windowed average of the resource demands and the scaling target predefined in the configuration, making resource-aware autoscaling decisions.

5.3 Kubernetes and External Components

Wasabi uses several Kubernetes networking components and configurations for traffic management. *Istio* [33] is used as the Kubernetes ingress gateway to manage incoming traffic to relevant Knative services and Wasabi containers. It brings advanced traffic management features for cloud-native applications, such as service mesh and encryption in transit [34]. The *kube-proxy* is configured to operate in the IPVS mode for its high performance and advanced scheduling algorithms [50]. We configure the IPVS scheduler to use the least connections (LC) load balancing algorithm, which can select the Wasabi pod with the fewest in-flight requests to handle new incoming requests for coarse-grained load balancing. In production, an external API gateway/load balancer is set up before the Kubernetes ingress gateway (Fig. 3) for further traffic management functionalities, such as authentication, logging, caching, rate limiting, and load balancing across multiple clusters [59, 60]. The API gateway/load balancer also defines rewrite and request redirect rules to add/modify required HTTP headers (e.g., resource demands) and change the endpoint (e.g., */event* endpoint for functions in workflows discussed in §5.4), while keeping these details hidden from clients.

5.4 Supporting Serverless Workflows

As Wasabi is built on the popular Knative platform, it can leverage Knative Eventing's orchestration features such as workflow registry, event brokering, state management, trigger handling, and message channels. This enables Wasabi users to easily define and run serverless workflows in a way similar to that of container-based functions with Knative Eventing. Users can define serverless workflows using the Knative Eventing Flows schema [11] and specify the HTTP endpoint (addressable [24]) for Wasm-based function components in the workflow with a special endpoint */event*. Wasm-based function requests to the */event* endpoint are processed normally within Wasabi, but the multiplexer only accepts POST requests and adds the media type *application/cloudevents+json* to responses, as required by Knative Eventing and the specification of the JSON Event Format for CloudEvents, a standardized and protocol-agnostic structure and metadata definitions of events [35]. Specifically, Wasm-based function components in the workflow unmarshal the event message passed by the multiplexer, execute the business logic, and serialize the response in JSON following the CloudEvents schema. The multiplexer captures the serialized JSON and returns an HTTP response with the required media type. Knative Eventing channels then forward the event message returned by the multiplexer to the appropriate destinations (e.g., Wasm/container-based functions or workflows) as defined by the workflow. This schema enables developers to build DAGs with only Wasm functions, or hybrid DAGs combining Wasm and container-based functions.

6 Evaluation

This section evaluates Wasabi's effectiveness in meeting its goals by answering: (1) Does Wasabi improve resource efficiency and throughput? (2) Can it achieve efficiency gains without affecting performance? (3) How much does each component contribute to improving resource efficiency and throughput? (4) How much overhead does Wasabi introduce?

6.1 Methodology

Testbed. We deploy Wasabi and baseline systems on a Kubernetes cluster with one controller and seven worker nodes. The controller node has 8 vCPUs (Intel Xeon Gold 6248) and 30 GB of memory. Each worker has 16 vCPUs (Intel Xeon E5-2680 v4) and 32 GB of memory. Some resources (~5 GB of memory) are reserved for per-node processes (e.g., Kube-proxy). We use a separate node configured similar to the controller to generate and send traffic.

Workload. When testing Wasabi or baselines for scenarios involving co-location of more than one application, we use the Azure Functions traces [110] to stress the system with real invocation patterns and resource consumptions. We use randomly sampled sub-traces to match our cluster's scale, and use FaaSProfiler [25, 109] to replay traces.

Since Azure Functions traces and other public serverless traces lack the actual function code, we employ workload-emulating functions that accurately reproduce the CPU usage, memory consumption, and execution durations specified in the trace data. This ensures the execution time and resource usage of our evaluation accurately reflect the original workload. This methodology, which is consistent with prior work [86], allows us to evaluate interference and density under realistic pressure. When conducting controlled experiments without Azure Traces, we use real-world serverless workloads, including five individual serverless functions [78] and a multi-function image processing workflow [87]: (1) RSA-keygen,

which generates an RSA key pair, commonly used for cryptographic purposes, (2) JSON-compression, which compresses JSON data into a compact format, commonly used in web and API-based applications, (3) Scrypt, a key-derivation function for secure password hashing, (4) Image-blur, which adds a blur effect to an image, used for image processing tasks (5) Genpdf, which generates a PDF bill by including the item names, prices, and images, and (6) Image Processing, an image processing pipeline with five transformation operations; we ported its Python implementation to Rust and compiled it into five Wasm functions orchestrated in a workflow.

6.2 End-to-End Comparison

We run 300 randomly sampled applications from the Azure Functions traces with more than 125 k invocations on all baseline systems deployed on the same Kubernetes cluster. This traffic volume was bound by the most resource-intensive baseline. The function list (trace hashapp) and time window are available in Wasabi's public repository for reproducibility. Each experiment lasted two hours.

Baselines. We compare Wasabi against three baselines:

1. Knative: Knative [36] is a state-of-the-art serverless platform enabling production offerings such as Google Cloud Run and IBM Code Engine. It uses containerd [18] as its default runtime, which relies on runc to launch containers [44]. Given that Wasabi's design is not restricted to using Wasm and includes several control plane contributions, this baseline serves as a reference for the default Knative control plane. We configure Knative with per-pod concurrency levels of 1 and 2, the best-performing settings for our trace. In the higher concurrency setting (2), each pod is allocated double the resources.

2. Knative+Wasmedge: Wasmedge [53] is a cloud-native high-performance runtime used for serverless applications [40, 53, 57, 58]. We set up `crun` [21] as the underlying runtime together with `containerd` [18] to allow us to integrate Wasmedge in Knative. This is a standard approach [20,46,47,54,100]. As Wasabi also uses Wasmedge, this baseline enables us to compare its effectiveness with an existing approach to use Wasm in serverless.

3. Spin: Spin [22] is a framework designed for developing and executing event-driven Wasm-based applications. We use SpinKube [32], an open-source, Kubernetes-native project, combining the Spin operator [48] and containerd shim Spin [19], to deploy and run Wasm-based applications in Kubernetes. We use KEDA [39] and the KEDA HTTP add-on [9, 10] to enable event-driven autoscaling (request concurrency based) and scale down to zero.

Faasm uses Wasm for lightweight isolation, originally for serverless [112], and now for longer-running applications [30]. We were unable to fully compare with it, despite months of effort. After following with the project maintainers, we narrowed it down to Faasm not supporting fine-grained resource allocation for functions (also observed by others [94]). Instead, it allocates resources to functions by comparing in-flight requests with "slots", a parameter set to the number of CPU cores per worker node. Using a static parameter is oblivious to varying functions resource needs and infrastructure memory availability. Failure occurs when concurrent functions' memory usage exceeds node capacity, triggering the OOM Killer. To do the best effort deployment of Azure functions on Faasm without redesigning it, we deployed Faasm with two settings: (1) use default setting of slots, (2) set slot count based on the average requested CPU amount in the Azure subtrace. The former led to significant underutilization and request failures from insufficient slots (~98% failure rate), and the latter led to OOM kills (~94% failure rate).

Resource Allocation Reduction. Fig. 4 shows the memory and CPU usage and allocation of applications. Allocated resources refer to the reserved capacity unavailable to other applications. We collect the data using Prometheus [43] with a 5 s scrape interval. Fig. 5 shows aggregated results from Fig. 4, showing that Wasabi allocates 10.47x, 11.55x, 9.75x, and 8.15x lower total memory-seconds, and 13.54x, 13.73x, 12.62x, and 9.67x lower total CPU-seconds compared to Knative [conc=1], Knative [conc=2], Knative+Wasmedge, and Spin, respectively. The improvements stem from eliminating per-app container scaling and self-cancellation of resource demand fluctuations across applications. Combined, they lead to far fewer pods being created for Wasabi (Fig. 5-Top-Right). Increasing pod concurrency in Knative reduces pod creation but harms resource allocation, as pods are not shared across applications, and larger pods needed for higher concurrency lead to greater keep-alive wastage.

Resource Usage Reduction. The resource consumption of all applications over time, shown in Fig. 4 bottom subplots, is also collected with Prometheus. Aggregate usage is shown in hatched bars in Fig. 5. Wasabi reduces memory usage by factors of 9.91, 7.88, 19.89, and 12.7 compared to Knative [conc=1], Knative [conc=2], Knative+Wasmedge, and Spin, respectively, indicating a significantly lower memory footprint. The higher memory usage of Knative+WasmEdge compared to Knative can be attributed to (1) longer service times, shown in Fig. 5, primarily caused by pod initialization overhead, and (2) distributed pod keep-alive, keeping resources in use for a longer period. Wasabi's CPU usage is comparable to the baselines, except for Spin, which is higher.

Service Time Reduction. The service time is defined as the total time taken to process a request, including all platform delays as well as the execution time. Using Wasm for isolation across applications allows Wasabi to reuse costly pods that are already created. As shown in the *Pod Creation Count* subplot in Fig. 5, pod creations are reduced by over 98% in Wasabi compared to baselines. As creating and starting

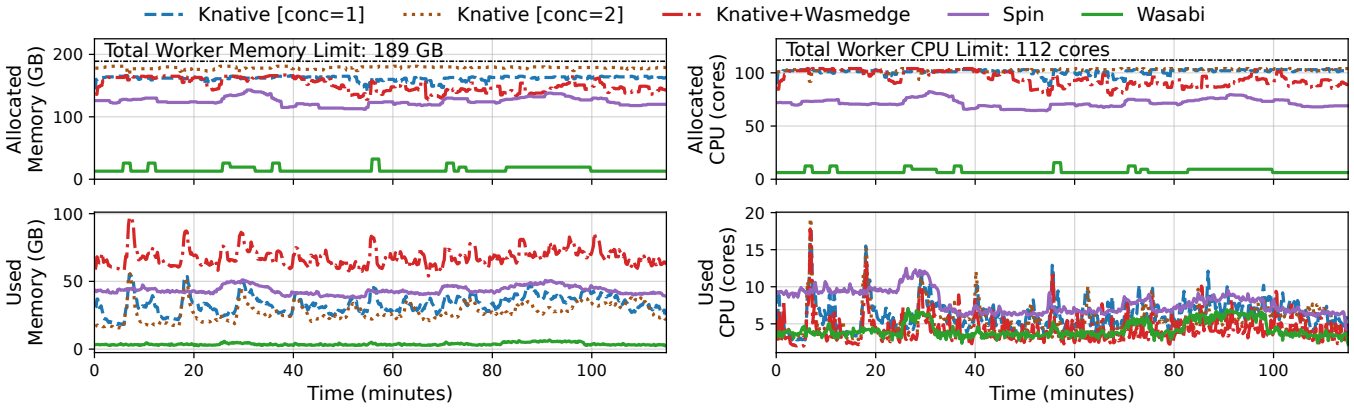

Figure 4: Comparison of resource usage and resource allocation over time between Knative+Wasmedge and Wasabi.

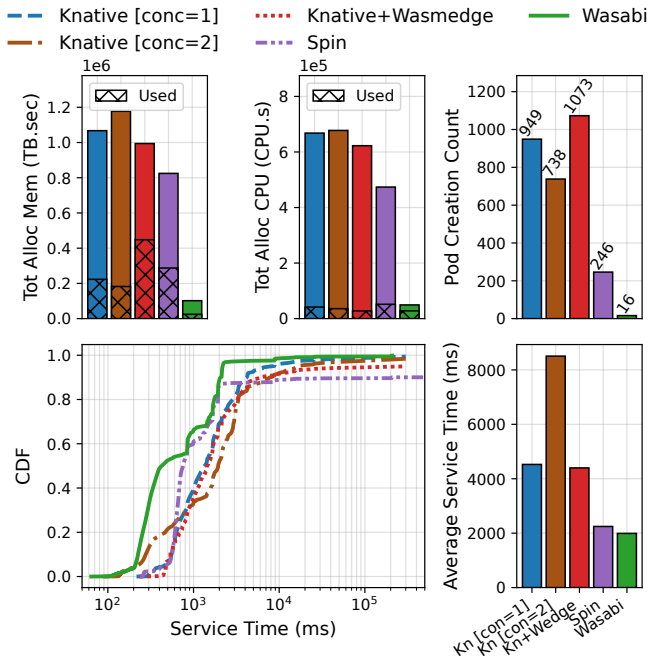

Figure 5: Comparison of total resource usage, resource allocation, and service time distribution between Knative, Knative+Wasmedge, Spin, and Wasabi.

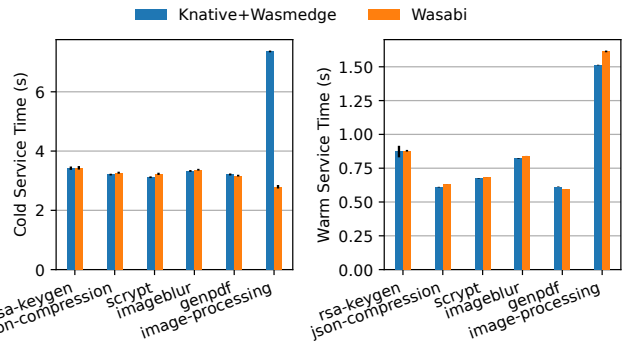

Figure 6: At low traffic rate (similar scaling), Knative+Wasmedge and Wasabi have similar performance. Wasabi significantly reduces workflow cold execution time.

pods is one of the slowest aspects of full-fledged serverless platforms such as Knative [101], this reduction in pod creation significantly reduces the platform delay. The distribution of service times shown in Fig. 5 reflects this reduction for almost all invocations. When running the traces on Spin, ~10% of requests failed. These occurred when requests reach backend pods before they're fully initialized, often during scale-from-zero events, where traffic outpaces pod readiness.

Heterogeneity. To demonstrate simultaneous execution of Wasm-based and container-based functions, we evenly (random) distribute the workload across Knative containers [conc=2] or as Wasm modules on Wasabi. Compared to only running on Knative [conc=2], total memory- and CPU-seconds dropped by 13.7% and 15.2%, respectively, and average service time was improved by 1.63x. The improvements are lower than when the entire workload runs with Wasabi (e.g., 87.3% memory-seconds reduction), as the non-Wasm portion of the workload does not gain density. Specifically, the containerized portion retains its resource overhead and isolation costs, which dilutes the overall efficiency.

6.3 A Closer Look into Wasabi's High Density

§6.2 showed that Wasabi's multiplexed execution outperforms directly integrating the Wasm runtime into Knative pods in terms of resource allocation and service time. A major reason is Wasabi's ability to use already created pods to quickly run Wasm modules, which in turn reduces pod creations and associated overhead of cold starts and +keep-alives (Fig. 5). However, it is unclear how much of the gains stem from reducing containerization overhead by packing multiple Wasm modules versus from lowering pod-scaling overhead. In this section, we run additional controlled experiments to answer this question. We use those serverless applications introduced in §6.1, as the Azure Dataset 2019 (or any other trace) does not include actual function implementations.

Median service time of Knative+Wasmedge and Wasabi are shown in Fig. 6. Cold execution includes pod initialization and

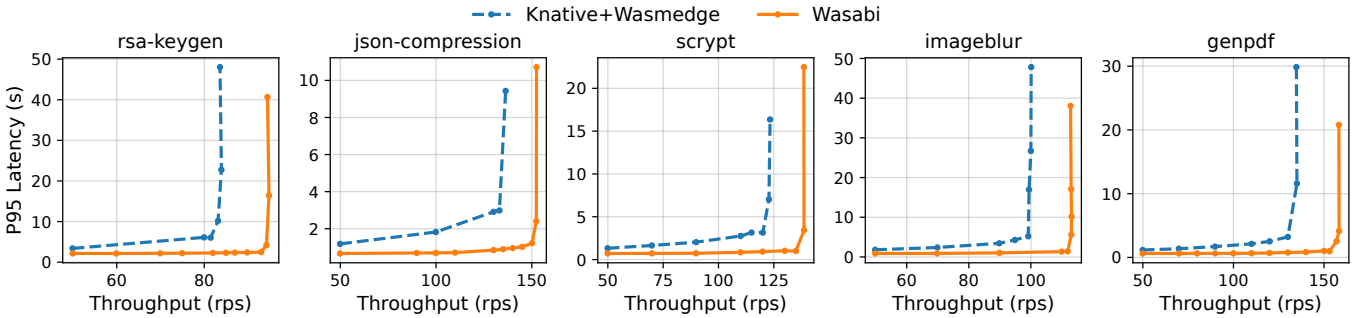

Figure 7: Comparison of Throughput and 95th percentile (p95) service time between Knative+Wasmedge and Wasabi.

function execution, while warm execution excludes pod initialization. Each experiment was conducted separately with low traffic to avoid overload or contention affecting service times. We see that the service times of Knative+Wasmedge and Wasabi are almost identical at this low rate. Comparable warm pod service times indicate identical execution times, which is expected given both systems use Wasmedge. When executing the image processing workflow, Wasabi avoids chained cold starts by reusing an existing WasmBox for subsequent functions after the first, whereas the Knative+Wasmedge experiences cascading cold starts for each function, resulting in significantly higher overall service time.

We now compare the 95th percentile (p95) service time of the two systems near saturation throughput. Unlike in §6.2, where 300 applications shared the cluster, each benchmark here is run individually on the entire cluster. While not reflecting real-world scenarios, this allows us to keep all pods alive and prevent scaling. Fig. 7 shows that Wasabi achieves slightly better service times across all observed rates and outperforms Knative+Wasmedge by achieving higher density (a geometric mean of 19.31%) without degrading service times. Since each data point is collected at a sustained throughput with a fixed traffic rate, no new pods are created during the study. Thus, we can attribute this 19.31% gain to only amortization of containerization overhead across Wasm modules. This throughput gain is equal to the density gain, since both experiments use the same allocation of resources (i.e., the entire cluster). Comparing this 1.19x allocation density gain to the 8.98x gain achieved when running real traces with varying load from many applications (in §6.2) shows that most gains stem from the Wasabi's ability to reduce keep-alive wastage through improved resource sharing across tenants.

Lastly, we observe that at maximum throughput, the cluster is bottlenecked by CPU for these benchmarks in both setups. Fig. 8 shows the maximum memory usage and the saturation throughput for both systems across different benchmarks. Wasabi has 1.95x (geometric mean) lower GB-per-rps compared with Knative+Wasmedge. As Wasabi has fewer large pods and higher throughput, we attribute this reduced memory usage to the lower relative overhead of sidecar containers.

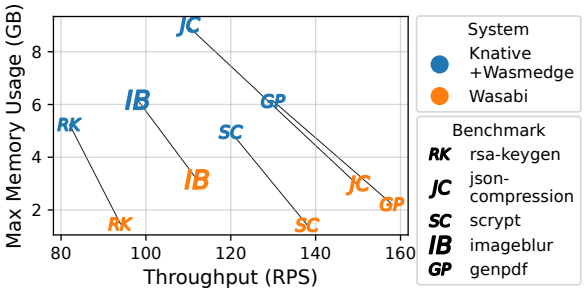

Figure 8: Compared with Knative+Wasmedge, Wasabi delivers higher throughput while using less memory.

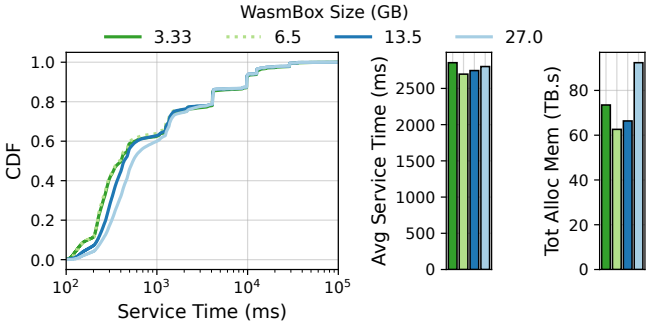

Figure 9: Using very small or large WasmBoxes harms performance and efficiency.

6.4 Ablation Study and Sensitivity Analyses

This section evaluates the impact of each system component on overall resource efficiency and performance of Wasabi. To assess the impact of each component, we perform multiple experiments, disable one component in each, and measure their effectiveness. Here, we use a larger traffic volume than in §6.2, as we are not constrained by the limitations of baseline systems. We replay traces of 500 randomly sampled applications, with over 164,000 invocations over 2 hours.

WasmBox Right-Sizing. As noted in §3.1, WasmBox right-sizing affects efficiency and performance. Fig. 9 shows this trade-off. It lists memory sizes for different configurations, and CPU is configured proportional to the size of each worker node (half a core per GB). We selected 27 GB as the largest memory configuration for WasmBox, slightly below each

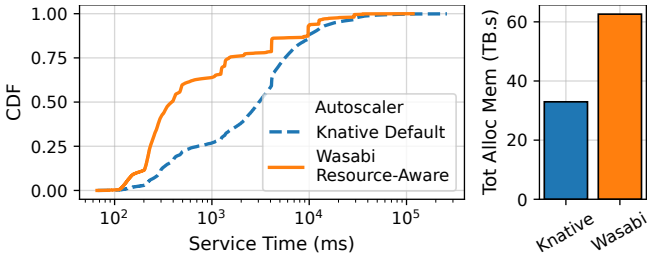

Figure 10: Comparison of service time distribution and total allocated memory between Wasabi and Knative autoscalers.

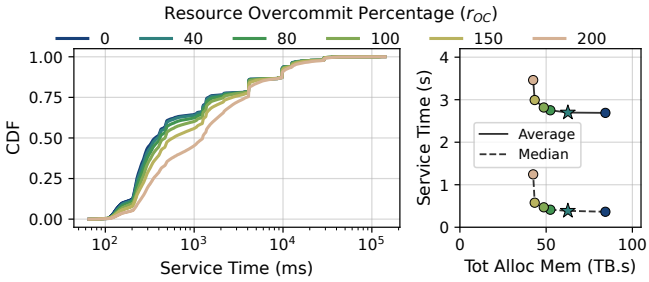

Figure 11: Wasabi is capable of resource overcommitment. Moderate overcommitment (~40%) can substantially lower resource consumption with minimal effect on service time.

worker node's capacity to reserve resources for per-node processes (e.g., Kube-proxy, Prometheus Node Exporter). Memory configurations of 13.5 GB, 6.75 GB, and 3.33 GB follow a geometric progression. We observe that using a small WasmBox leads to increased service time, as there are more pod initiations. This increase in pod counts also increases resource allocation due to accumulated keep-alive wastage. Increasing the WasmBox size solves both issues, but at some point introduces another. Beyond 6.5 GB, service time distributions get shifted to the right for a large fraction of requests. The cause is excessive memory allocation and deallocation syscalls by Wasm modules, degrading performance. Slower function executions in turn reduce the throughput of each WasmBox, requiring more WasmBoxes. In all other experiments in the paper, the default size of 6.5 GB is used.

Autoscaling. Fig. 10-Left shows that Wasabi's resource-aware autoscaler significantly improves service time compared to the original Knative autoscaler. The main reason is that Wasabi's novel autoscaler accounts for the actual resource demands of in-flight requests, which is a more accurate representation of container load compared to concurrency metrics used in Knative, enabling more precise scaling decisions. However, in this experiment, Knative's concurrency-based autoscaler does not sufficiently scale up the containers, causing resource contention and higher service time, as reflected by the total allocated memory (GB·seconds) in Fig. 10-Right.

Resource Overcommitment. This experiment evaluates how varying overcommitment ratios affects service time distribution and resource usage. Fig. 11 shows the service time distribution and total allocated memory for different ratios,

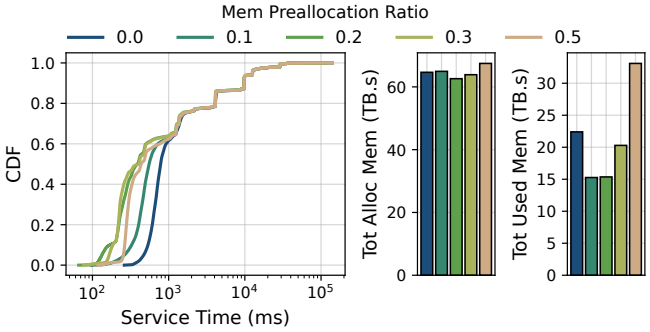

Figure 12: Wasabi is capable of memory pre-allocation which can improve service times and resource efficiency.

ranging from 0% (no overcommitment) to 200% (where concurrent requested resources are three times the allocated resources), applied uniformly to CPU and memory. Higher ratios increase service times but reduce total allocated memory, creating a trade-off between resource allocation and performance. However, we observed that a moderate degree of overcommitment can significantly enhance resource efficiency without significantly impacting the average service time. Accordingly, our system uses an overcommitment ratio of 40%, striking a balance between improved resource efficiency and maintaining good performance. This aligns with the fact that requested resources are often higher than the actual needs, as users tend to overconfigure resources [104, 113, 116].

Memory Pre-allocation. Fig. 12-Left shows that memory pre-allocation can substantially improve service time by addressing the performance challenges mentioned in §3.4. At the same time, by improving performance and enabling faster execution, memory pre-allocation reduces the total allocated memory over time. However, excessive pre-allocation increases overall memory usage and may result in unused reserved memory and therefore memory inefficiency (Fig. 12-Right) and leads to higher service times. Empirically, we found that a pre-allocation ratio of 0.2 strikes a balance between performance improvement and resource overhead.

Request Queue. The resource-aware request queue in Wasabi tracks requests' resource demands and container capacity, holding requests when resources are insufficient to prevent system failures. We observed that 193 of 450 requests (42%) failed during the initial 5-second request burst (i.e., scaling from zero) in the traces with the default Knative queue proxy due to resource shortage. No failures occurred with Wasabi's resource-aware request queue. Default queue proxy's failures stem from using concurrency metrics, which are unsuitable when containers host different functions (§3.2).

6.5 Multiplexer Delay Characterization

Table 1 shows the latency breakdown of the multiplexer overhead for single-function benchmarks. Cgroup operations include creating the cgroup, changing its type to threaded (necessary for assigning threads), applying CPU limits, and delet-

Benchmark	Multiplexer Latency Breakdown (median statistics in microseconds)			
	Cgroup	Thread Pinning	Wasm	Total
rsa-keygen	528.65	0.57	418.4	947.62
json-compression	516.16	0.56	383.62	900.34
scrypt	532.18	0.55	385.97	918.7
imageblur	506.7	0.56	381.8	889.06
genpdf	538.99	0.58	536.6	1076.17

Table 1: Latency breakdown of the multiplexer.

Function Sandbox	Scaling Unit	System(s)	Per-req Isolation	Multi Language	Resource Isolation Level	Memory Pre-alloc
Container	Container	OpenWhisk [42]	N	Y	Request	N
Container	Pod (container(s))	Knative [36]	N	Y	Container	N
Wasm	Pod (Wasm modules and Containers)	Wasmedge on K8s [57]	N	Y	Container	N
		Spin on K8s [32]				
		WasmCloud on K8s [52]			Wasm Module	
V8 Isolate & Wasm	V8 Isolate	Cloudflare Workers [4]	N	Y	V8 Isolate	N
Unikernel	Unikernel	SEUSS [68]	Y	Y	N/A	N
Lang. Runtime	Process	Boucher et al. [65]	N	N	Process	N
Lang. Runtime	N/A	Flock [126]	Y	N	Request	N
Wasm	Wasm module	Sledge [75]	Y	Y	Request	N
		Faasm [112]				
		GRANNY [107]				
Lang. Runtime	Container	Photons [71]	N	Y	Container	N
Wasm	Wasm Module and Container	Wasabi	Y	Y	Request	Y

Table 2: Comparison of existing isolation models.

ing the cgroup post execution. Thread pinning, needed for resource limiting (§5.1), comes with small overhead. Wasm operations include loading the module, limiting its linear memory, and gathering the results upon execution completion. Module loading times differ across benchmarks due to variations in size and dependencies. The multiplexer adds a minimal median latency of ~1 ms, negligible compared to existing platform delays and client-server RTTs.

7 Related Work

Table 2 compares existing platforms by isolation granularity, scaling units, and system boundaries. Container-based systems such as OpenWhisk and Knative place the boundary at the container or pod level, using a single scaling unit and sharing resources across requests. Kubernetes-based Wasm platforms (e.g., WasmEdge and Spin) largely inherit this model, while WasmCloud isolates at the Wasm-module level but still relies on container orchestration. In contrast, unikernel-based, language-runtime, and prior Wasm systems (e.g., SEUSS, Flock, Sledge, Faasm, GRANNY) shift the boundary to individual requests, enabling fine-grained isolation with a single scaling unit. Wasabi differs by supporting two scaling units, containers and Wasm modules, while enforcing per-request isolation, thereby combining coarse-grained container management with fine-grained Wasm-level execution to enable secure execution with efficient resource management.

Enhanced isolation. Various techniques aim to reduce cold start latency in serverless. Some focus on providing light-weight isolation to reduce cold start times [60, 92, 95, 115]. Unikernels offer fast startup times and high throughput for serverless functions [68, 72, 90, 91], while Wasabi relies on the light-weight language runtime-based isolation provided by Wasm. Some systems leverage language runtimes (e.g., JVM, Wasm, Rust) [65, 71, 74, 75, 81, 112, 122, 126]. Some rely on homogeneous resource allocation mechanisms [71, 112], which is not ideal for serving functions with diverse resource needs and execution times in production settings. In this work, however, we use a combination of techniques to make the platform resource-aware at the granularity of each function execution to push resource efficiency to its limits. Some others [65, 71, 126] rely on language runtimes (e.g., JVM, Rust), which introduces challenges in supporting a wide range of programming languages and applications. Some systems [15, 26, 124] leverage Wasm and V8 Isolates [28], and do not address resource isolation. Some systems [75, 124] focus solely on single-host deployment.

Sandbox/runtime sharing. Sharing sandboxes (e.g., container), VMs or language runtimes (e.g., JVM) has been explored in the context of serverless to amortize the costs of cold starts and improve resource efficiency [61, 63, 65, 67, 67, 71, 83, 94, 98, 114, 123, 126], as well as in other contexts to balance isolation, enable flexible resource management, and improve performance [64, 111]. Some systems focus on sharing for the same function, application, or DAG [61, 63, 71, 94, 98], whereas Wasabi safely co-executes functions from different users in shared containers. Some systems rely on weak isolation mechanisms, such as process isolation [61, 114]. Another set of approaches include using specific language runtimes such as JVM or Rust [65, 71, 126], whereas Wasabi leverages Wasm, enabling polyglot programming. Wasabi leverage container reuse differently, incorporating hierarchical isolation for co-execution of Wasm-based functions in shared, fine-grained containers along with a combination of techniques to enable resource awareness in the platform.

Resource overcommitment using prediction models and heuristics has been explored in serverless [93, 97, 116] to improve resource efficiency. Wasabi leverages simple ratio-based resource overcommitment in a new context, but can potentially benefit from those approaches.

8 Conclusion

We introduce a new two-level isolation architecture to densely pack Wasm modules in container-based serverless platforms. This architecture enables fast and efficient resource sharing. We prototype our system on a production-grade serverless platform. It shows high potential to increase packing density and reduce cold starts. Parts of this work have been deployed in production within Huawei Cloud's CDN serverless edge computing service and used by customers since late 2025.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Eric Eide, for helping us improve the paper. We thank William J. Bowman for constructive early-stage discussions, and Ghazal Sadeghian for preliminary explorations. This project was funded by Huawei Technologies Canada.

References

[1] About security-guard - knative. https://knative.dev/docs/serving/app-security/security-guard-about/, Accessed on 2025-09-16.

[2] Engine in wasmtime. https://docs.wasmtime.dev/api/wasmtime/struct.Engine.html, Accessed on 2025-09-16.

[3] Guard gate. https://pkg.go.dev/knative.dev/security-guard/pkg/guard-gate, Accessed on 2025-09-16.

[4] How Workers works | Cloudflare docs. https://developers.cloudflare.com/workers/reference/how-workers-works/, Accessed on 2025-09-16.

[5] Knative serving architecture. https://knative.dev/docs/serving/architecture/, Accessed on 2025-09-16.

[6] WasmEdge VM. https://wasmedge.org/docs/embed/go/reference/latest/#wasmedge-vm, Accessed on 2025-09-16.

[7] Cloudflare Pages docs | module support. https://developers.cloudflare.com/pages/functions/module-support/, Accessed on 2025-09-17.

[8] Cloudflare Workers docs | WebAssembly (Wasm). https://developers.cloudflare.com/workers/runtime-apis/webassembly/, Accessed on 2025-09-17.

[9] https://kedacore.github.io/http-add-on/, Accessed on 2025-09-18.

[10] https://github.com/kedacore/http-add-on, Accessed on 2025-09-18.

[11] About flows - Knative. https://knative.dev/docs/eventing/flows/, Accessed on 2025-09-18.

[12] Allow HPA to scale to 0. https://github.com/kubernetes/kubernetes/issues/69687, Accessed on 2025-09-18.

[13] Bulk memory operations and conditional segment initialization. https://github.com/WebAssembly/bulk-memory-operations/blob/master/proposals/bulk-memory-operations/Overview.md, Accessed on 2025-09-18.

[14] ceph/ceph: Ceph is a distributed object, block, and file storage platform. https://github.com/ceph/ceph, Accessed on 2025-09-18.

[15] Cloudflare. cloudflare workers. https://workers.cloudflare.com/, Accessed on 2025-09-18.

[16] Concurrency in Azure Functions | Microsoft learn. https://learn.microsoft.com/en-us/azure/azure-functions/functions-concurrency, Accessed on 2025-09-18.

[17] Configuring concurrency - Knative. https://knative.dev/docs/serving/autoscaling/concurrency/, Accessed on 2025-09-18.

[18] Containerd. https://containerd.io/, Accessed on 2025-09-18.

[19] Containerd shim spin. https://github.com/spinframework/containerd-shim-spin/, Accessed on 2025-09-18.

[20] Containerization on the edge. https://www.cncf.io/blog/2021/11/11/containerization-on-the-edge/, Accessed on 2025-09-18.

[21] crun. https://github.com/containers/crun, Accessed on 2025-09-18.

[22] Develop serverless webassembly apps with spin. https://www.fermyon.com/spin, Accessed on 2025-09-18.

[23] Do we need a memcpy opcode? https://github.com/WebAssembly/design/issues/977#issue-204960079, Accessed on 2025-09-18.

[24] Duck types - Knative. https://knative.dev/docs/concepts/duck-typing/#knative-duck-types, Accessed on 2025-09-18.

[25] FaaSProfiler. https://github.com/PrincetonUniversity/faas-profiler, Accessed on 2025-09-18.

[26] Fastly. https://www.fastly.com, Accessed on 2025-09-18.

[27] Feature suggestion (memcpy/memset). https://github.com/WebAssembly/design/issues/236#issuecomment-283279499, Accessed on 2025-09-18.

[28] Google. v8 engine. https://github.com/v8/v8, Accessed on 2025-09-18.

[29] Horizontal pod autoscaling | kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, Accessed on 2025-09-18.

[30] How to measure performance? https://github.com/faasm/faasm/issues/797, Accessed on 2025-09-18.

[31] How to shorten the collecting interval(resolution) ? · issue 1483 · kubernetes-sigs/metrics-server. https://github.com/kubernetes-sigs/metrics-server/issues/1483, Accessed on 2025-09-18.

[32] Hyper-efficient serverless on kubernetes, powered by webassembly. https://www.spinkube.dev/, Accessed on 2025-09-18.

[33] Istio. https://istio.io/, Accessed on 2025-09-18.

[34] Istio / the istio service mesh. https://istio.io/latest/about/service-mesh/, Accessed on 2025-09-18.

[35] JSON event format for CloudEvents - version 1.0.2. https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/formats/json-format.md, Accessed on 2025-09-18.

[36] Knative. https://knative.dev/docs/, Accessed on 2025-09-18.

[37] Knative serving autoscaling system. https://github.com/knative/serving/blob/main/docs/scaling/SYSTEM.md, Accessed on 2025-09-18.

[38] Kubernetes. https://kubernetes.io/, Accessed on 2025-09-18.

[39] Kubernetes event-driven autoscaling. https://keda.sh/, Accessed on 2025-09-18.

[40] A lightweight, high-performance, and extensible webassembly runtime. https://github.com/WasmEdge/WasmEdge?tab=readme-ov-file, Accessed on 2025-09-18.

[41] Node autoscaling. https://kubernetes.io/docs/concepts/cluster-administration/node-autoscaling/, Accessed on 2025-09-18.

[42] Open source serverless cloud platform. https://openwhisk.apache.org/, Accessed on 2025-09-18.

[43] Prometheus. https://prometheus.io/, Accessed on 2025-09-18.

[44] runc. https://github.com/opencontainers/runc, Accessed on 2025-09-18.

[45] runwasi. https://github.com/containerd/runwasi, Accessed on 2025-09-18.

[46] Second state. AI inference on the edge. https://www.secondstate.io/, Accessed on 2025-09-18.

[47] Shaping the future towards a fully connected, intelligent world. https://www.futurewei.com/, Accessed on 2025-09-18.

[48] Spin operator. https://github.com/spinframework/spin-operator/, Accessed on 2025-09-18.

[49] Steady state kubelet CPU usage is high due to excessive allocation in prometheus scraping of cadvisor · issue 104459 · kubernetes/kubernetes. https://github.com/kubernetes/kubernetes/issues/104459, Accessed on 2025-09-18.

[50] Virtual IPs and service proxies | kubernetes. https://kubernetes.io/docs/reference/networking/virtual-ips/#proxy-mode-ipvs, Accessed on 2025-09-18.

[51] WASI: WebAssembly System Interface. https://wasi.dev/, Accessed on 2025-09-18.

[52] wasmcloud on kubernetes. https://wasmcloud.com/docs/kubernetes/, Accessed on 2025-09-18.

[53] Wasmedge. https://wasmedge.org/, Accessed on 2025-09-18.

[54] WasmEdge + crun is more powerful and easy to use than Krustlet. https://www.secondstate.io/articles/kind-wasmedge/#wasmedge--crun--is-more-powerful-and-easy-to-use-than-krustlet, Accessed on 2025-09-18.

[55] WasmEdge for Go package. https://github.com/second-state/WasmEdge-go, Accessed on 2025-09-18.

[56] WebAssembly Core Specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, Accessed on 2025-09-18.

[57] WebAssembly on Kubernetes: from containers to Wasm (part 01). https://www.cncf.io/blog/2024/03/12/webassembly-on-kubernetes-from-containers-to-wasm-part-01/, Accessed on 2025-09-18.

[58] WebAssembly serverless functions in AWS Lambda. https://wasmedge.org/docs/start/usage/serverless/aws, Accessed on 2025-09-18.

[59] What is an API gateway? | IBM. https://www.ibm.com/think/topics/api-gateway, Accessed on 2025-09-18.

[60] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[61] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[62] Hannah Aubry. Fastly can teach you about the Wasm future in just 6 talks. https://www.fastly.com/blog/fastly-can-teach-you-about-the-wasm-future-in-just-6-talks, Accessed on 2025-09-17.

[63] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. Propack: Executing concurrent serverless functions faster and cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 211–224, 2023.

[64] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[65] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the" micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, 2018.

[66] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[67] Rodrigo Bruno, Serhii Ivanenko, Sutao Wang, Jovan Stevanovic, and Vojin Jovanovic. Graalvisor: Virtualized polyglot runtime for serverless applications. *arXiv e-prints*, pages arXiv–2212, 2022.

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

[69] Cloud Run Documentation | Google Cloud. Billing settings for services. https://cloud.google.com/run/docs/configuring/billing-settings, Accessed on 2025-09-18.

[70] Datadog. The state of serverless 2023. https://www.datadoghq.com/state-of-serverless/, Accessed on 2025-09-18.

[71] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[72] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. USETL: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 23–30, 2019.

[73] Alexander Fuerst and Prateek Sharma. 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*, pages 386–400, 2021.

[74] Philipp Gackstatter, Pantelis A Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE, 2022.

[75] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st international middleware conference*, pages 265–279, 2020.

[76] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615. IEEE, 2019.

[77] Rafal Gancarz. Prime Video switched from serverless to EC2 and ECS to save costs. https://www.infoq.com/news/2023/05/prime-ec2-ecs-saves-costs/, May 2023.

[78] Samuel Ginzburg, Mohammad Shahrad, Michael J Freedman, Zhaoduo Wen, Sidharth Kumar, Binyu Zang, Ken Gordon, Xiaochuan Tang, Balaji Vembu, Zbigniew T Kalbarczyk, et al. VectorVisor: A binary translation scheme for throughput-oriented GPU acceleration. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1017–1037, 2023.

[79] Shubham Gupta and Jeff Gebhart. AWS Lambda standardizes billing for init phase. https://aws.amazon.com/blogs/compute/aws-lambda-standardizes-billing-for-init-phase/, Accessed on 2025-09-18.

[80] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[81] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, page 225–236. ACM, 2019.

[82] Vlad Ionescu. Scaling containers on AWS in 2022. https://www.vladionescu.me/posts/scaling-containers-on-aws-in-2022/, Accessed on 2025-09-18.

[83] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 symposium on cloud computing*, pages 445–451, 2017.

[84] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. How does it function? characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 443–458. ACM, 2023.

[85] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 938–953. ACM, 2025.

[86] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 289–305, 2022.

[87] Jeongchul Kim and Kyungyong Lee. FunctionBench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[88] Minseo Kim, Hyerean Jang, and Youngjoo Shin. Avengers, assemble! survey of WebAssembly security solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 543–553, 2022.

[89] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *Journal of Grid Computing*, 21(3):34, 2023.

[90] Ricardo Koller and Dan Williams. Will serverless end the dominance of Linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 169–173, 2017.

[91] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A Linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[92] Nicolas Lacasse. Open-sourcing gVisor, a sandboxed container runtime, Accessed on 2025-09-10.

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

[94] Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. Rethinking deployment for serverless functions: A performance-first perspective. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[95] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: a lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, 2022.

[96] Changyuan Lin, Yuanzhi Ma, and Mohammad Shahrad. Demystifying serverless costs on public platforms: Bridging billing, architecture, and OS scheduling. In *Proceedings of the European Conference on Computer Systems (EuroSys' 26). ACM*, 2026.

[97] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R Larus, and Haibo Chen. Harmonizing efficiency and practicability: optimizing resource utilization in serverless computing with Jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1–17, 2024.

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

[99] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. WISEFUSE: Workload characterization and DAG transformation for serverless workflows. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), jun 2022.

[100] Cynthia Marcelino and Stefan Nastic. CWASI: A WebAssembly runtime shim for inter-function communication in the serverless edge-cloud continuum. In *2023 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 158–170. IEEE, 2023.

[101] Nima Nasiri, Nalin Munshi, Simon Moser, Marius Pirvu, Vijay Sundaresan, Daryl Maier, Thatta Premnath, Norman Böwing, Sathish Gopalakrishnan, and Mohammad Shahrad. In-production characterization of an open source serverless platform and new scaling strategies. In *Proceedings of the European Conference on Computer Systems (EuroSys' 26). ACM*, 2026.

[102] Steven Pham, Kaue Oliveira, and Chung-Horng Lung. WebAssembly modules as alternative to docker containers in iot application development. In *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, pages 519–524, 2023.

[103] Robby Qiu. WebAssembly serverless functions in AWS lambda. https://www.cncf.io/blog/2021/08/25/webassembly-serverless-functions-in-aws-lambda/, Accessed on 2025-09-18.

[104] Ran Ribenzaft. What AWS Lambda's performance stats reveal. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/, Accessed on 2025-09-18.

[105] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 753–767. ACM, 2022.

[106] Prateek Sahu, Shijia Wei, Neeraja J. Yadwadkar, and Mohit Tiwari. Understanding sidecars in cloud orchestration. In *Proceedings of the 3rd Workshop on SErverless Systems, Applications and MEthodologies*, SESAME' 25, page 1–11. ACM, 2025.

[107] Carlos Segarra, Simon Shillaker, Guo Li, Eleftheria Mappoura, Rodrigo Bruno, Lluís Vilanova, and Peter Pietzuch. GRANNY: Granular management of Compute-Intensive applications in the cloud. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 205–218, 2025.

[108] Vishwanath Seshagiri, Abhinav Gupta, Vahab Jabrayilov, Avani Wildani, and Kostis Kaffes. Rethinking the networking stack for serverless environments: A sidecar approach. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, SoCC '24, page 213–222. ACM, 2024.

[109] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1063–1075. ACM, 2019.

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

[111] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[112] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

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

[114] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[115] Ariel Szekely, Adam Belay, Robert Morris, and M Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 385–402, 2024.

[116] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 78–93, 2022.

[117] Kenton Varda. Webassembly on cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Accessed on 2025-09-18.

[118] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[119] Zhen Wang, Jianda Wang, Zhendong Wang, and Yang Hu. Characterization and implication of edge webassembly runtimes. In *2021 IEEE 23rd Int Conf on High Performance Computing Communications; 7th Int Conf on Data Science Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*, pages 71–80, 2021.

[120] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65. ACM, 2018.

[121] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320. USENIX Association, 2006.

[122] Yosh Wuyts and Ralph Squillace. Introducing hyperlight: Virtual machine-based security for functions at scale. https://opensource.microsoft.com/blog/2024/11/07/introducing-hyperlight-virtual-machine-based-security-for-functions-at-scale/, Accessed on 2025-09-18.

[123] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 335–350, 2024.

[124] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[125] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739. ACM, 2021.

[126] Ziming Zhao, Mingyu Wu, Xujie Cao, Haibo Chen, and Binyu Zang. Flock: Towards multitasking virtual machines for function-as-a-service. *IEEE Transactions on Computers*, 72(11):3153–3166, 2023.