

# Junctiond: Extending FaaS Runtimes with Kernel-Bypass

Enrique Saurez<sup>§</sup>  
 Íñigo Goiri<sup>§</sup>  
*MIT CSAIL*

Joshua Fried  
 Sameh Elnikety<sup>\*</sup>  
*Azure Research – Systems*

Gohar Irfan Chaudhry  
 Adam Belay  
*Rodrigo Fonseca<sup>§</sup>*  
*\* Microsoft Research*

## Abstract

This report explores the use of kernel-bypass networking in FaaS runtimes and demonstrates how using Junction [9], a novel kernel-bypass system, as the backend for executing components in faasd can enhance performance and isolation. Junction achieves this by reducing network and compute overheads and minimizing interactions with the host operating system. Junctiond, the integration of Junction with faasd, reduces median and P99 latency by 37.33% and 63.42%, respectively, and can handle 10 times more throughput while decreasing latency by 2× at the median and 3.5 times at the tail.

## 1 Introduction

Serverless is one of the main paradigms for cloud-native programming. It simplifies cloud usage by minimizing operational complexity, allowing fine-grained pricing, and scaling the capacity automatically. The main serverless offering is Function as a Service (FaaS). With FaaS, the customer writes the code for functions that are triggered by certain events (*e.g.*, HTTP invocations or timers). The platform provider handles resource allocation, request routing, and function execution. The customer does not need to manage the infrastructure and platforms that host the services. The invocations are stateless, and the data related to the requests are stored in external data services within the same cloud provider. Since the cloud controls the execution of FaaS, it can optimize it better than most other types of service.

One of the main infrastructure components that affect FaaS performance is networking [6]. All FaaS invocations involve at least one remote procedure call, and usually more, as multiple software components are involved in routing requests to the process running the functions, including gateways and sidecars. Each network round-trip is in the critical path of an invocation and consumes CPU time that could be better utilized to serve more functions or to improve the throughput of a specific function instance. As such, FaaS can greatly benefit from having an efficient network stack for all its infrastructure and platform components.

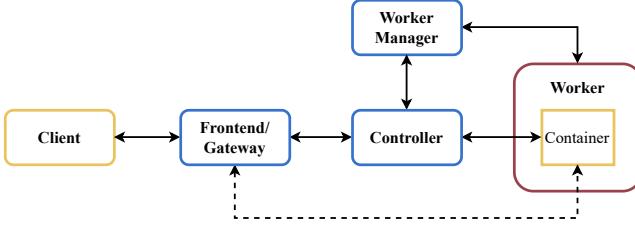
Kernel-bypass networking is utilized in data centers to enhance the performance of software services [3, 10, 12, 16, 20, 25]. Bypassing the operating system kernel, this technology reduces the number of layers involved and minimizes expensive context switches that occur when using regular network stacks, thereby improving performance. However, due to the complexity of implementation and additional computational costs involved [9, 25], most FaaS platforms do not leverage kernel-bypass networking for general applications [21].

Kernel-bypass networking requires the dedicated use of resources to poll network queues, as notifications of packet reception are typically unavailable in user space (where kernel-bypass code resides). In FaaS, the naive use of kernel-bypass incurs a significant penalty in the number of resources allocated to polling. This is because one polling core is required per hosted function instance, and commonly used libraries (*e.g.*, DPDK [18]) cannot be securely shared for polling across different tenant functions. This issue is further exacerbated by the fact that most functions are not frequently invoked [22], resulting in more resources being spent on polling than on performing computations for the functions.

Newer kernel-bypass systems, like Junction [9], bring the performance benefits without the complexities and resources overhead involved. In this work, we demonstrate how we can utilize these systems to increase throughput and reduce warm end-to-end latency in FaaS runtimes. Specifically, we evaluate how Junction [9] can be seamlessly integrated into faasd as its primary execution runtime, reducing tail latencies by up to 81% and increasing throughput by up to 10 times, without significantly increasing the number of allocated cores per server.

## 2 Background

In this section, we describe the core building components of our prototype. We first describe the basic architecture used by many FaaS frameworks, and more concretely the architecture of faasd. Finally, we discuss kernel-bypass networking, and the properties of Junction that make it a good fit for FaaS.



**Figure 1:** Common FaaS architecture.

## 2.1 FaaS architecture

Figure 1 shows the high-level architecture for the typical FaaS platform [1, 2, 4, 17, 19, 23]. A client typically reaches a stateless load-balancer or gateway, which authenticates the request and routes it to the appropriate component. If the function is not currently active, the gateway will request the controller to deploy the corresponding function instance. This operation may also involve adding more workers to the pool via the worker manager if there is insufficient capacity.

Once the function instance is ready to handle requests in a worker, the request is forwarded either directly from the gateway or through the controller to the corresponding function instance. Most FaaS runtimes execute the function code inside either containers or virtual machines. Additionally, outside of the critical path, the controller will perform auto-scaling operations for both the pool and the function instances to properly handle the load being handled by a given function.

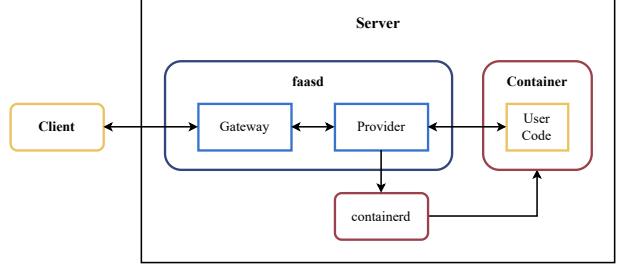
Each of the components in Figure 1, including the gateway, controller, and worker manager, are replicated services deployed on different servers for fault-tolerance. Workers are also typically deployed on separate servers, with their number determined by the overall load of the FaaS platform.

A key aspect of this architecture is that a request must pass through one or more components, such as the gateway, before reaching the container hosting the application. Each additional component in the invocation path adds an additional RPC call and its associated overhead. In some cases, a sidecar may even be present next to the application container to route the request from outside the worker to the process running the function, as it is the case in Kubernetes-based FaaS runtimes [17].

### 2.1.1 faasd

As a concrete implementation of the architecture presented in Section 2.1, we use faasd as the base building block in our prototype. faasd [8] is an open-source single-node serverless orchestration framework based on OpenFaaS [19].

As shown in Figure 2, faasd employs Linux containers, deployed by `containerd` [5], to sandbox untrusted user applications. It includes two orchestration services, both written in Go: a front-end gateway and a provider that communicates with `containerd`. Each of these orchestration services runs as an independent process within the same server.



**Figure 2:** Architecture of faasd.

An invocation in faasd always traverses the gateway and the provider before reaching the container running the user function code. The communication between each of the components is done via gRPC [11], which means each invocation involves at least three gRPC invocations, plus any additional request to external storage that is common in the context of FaaS applications.

## 2.2 Kernel-bypass networking

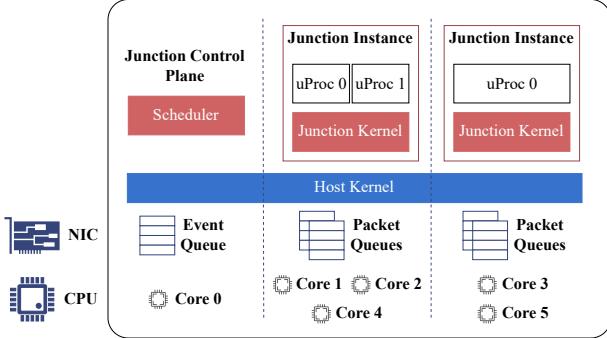
Kernel-bypass networking is a technique that allows user-space applications to communicate with the network hardware directly, without going through the operating system’s network stack. This can improve the throughput and latency of network-intensive applications. Kernel-bypass networking requires specialized hardware and software support, such as network interface cards (NICs) that can access user memory, and user-space libraries that can handle packet processing (e.g., DPDK [18]).

Kernel-bypass networking employs polling mode drivers (PMDs) to directly access the network interface cards (NICs), thereby avoiding the overhead of context switching and interrupt processing. PMDs keep a core busy by continuously polling the NIC for incoming or outgoing packets and transferring them between the NIC and user memory. Newer systems are building abstractions atop kernel-bypass libraries to simplify the programming and enable secure multi-tenant usage of the polling resources. One such system is Junction [9], which is explained in the next section.

### 2.2.1 Junction

Junction is a libOS-based, kernel-bypass system that can run process-isolated, unmodified Linux applications at a high density, and with practically no performance trade-offs. Figure 3 illustrates the Junction architecture. The three main components in Junction that are essential for integration with a FaaS runtime are: (1) a Junction instance, (2) the scheduler, and (3) the I/O management. For a complete explanation of its architecture, refer to Junction [9].

**Junction instance** In Figure 3, a Junction instance represents a container for executing one or more user applications.



**Figure 3:** Architecture of Junction.

Each Junction instance is a process in the host kernel. Each executable within a Junction instance runs in a user-level, process-like abstraction called a uProc. All uProcs within an instance share the Junction kernel. The Junction kernel provides a Linux syscall abstraction to the uProc, akin to a library OS [7], enabling it to run existing applications without modifications.

The use of a libOS-style user-space kernel, in combination with kernel-bypass devices, allows most system calls to be handled entirely within the Junction instance, except for those necessary to multiplex resources by the host kernel (*i.e.*, cores and memory). Shifting OS functionality into user space improves performance by reducing the frequency of context switches and limits the attack surface by allowing untrusted programs to exercise only very small parts of trusted host kernel code.

**IO management** The Junction kernel uses kernel bypass hardware, including both networking queues and CPU features, to provide its OS abstractions. Specifically, for the NIC queues, each Junction instance is assigned one or more NIC send and receive queue pairs, proportional to its maximum core allocation. To process these queue’s packets, the Junction kernel provides a high-performance network stack. By directly handling the NIC queues in each Junction instance, it can provide full concurrency across independent instances.

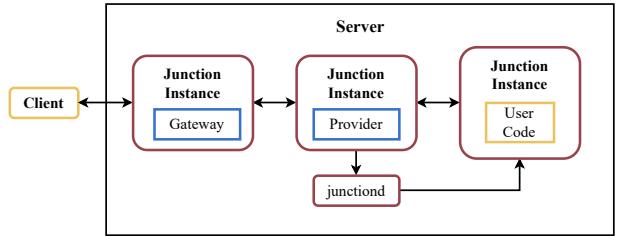
As shown in Figure 3, there are two types of queues that are directly assigned to userspace processes: the packet queues and the event queue. The packet queue, as described in the previous paragraph, is used by instances to communicate with any external entity. The event queue is used to signal when new packets are available in the NIC and is one of the main drivers of the scheduler in Junction.

**Scheduler** The scheduler in Figure 3 is used to manage core allocation for all the Junction instances in a server. The scheduler runs in a reserved core and busy polls on different signals to determine the core allocation for each instance. The allocation of cores varies across time based on demand, up to a configured limit for each instance.

The Junction kernel provides user-space threads to Junction instances. This allows the scheduler to have visibility into the state of each instance’s threads (whether they are active or idle) and the signals from the NIC event queues. As a result, the scheduler can centralize polling across all instances. The polling is scalable, as it is mostly proportional to the number of cores active with the instances, rather than being proportional to the total number of instances. This is because the scheduler only needs to make decisions about which task to assign to each core. Additional optimizations are performed to keep the overhead of this decision-making process bounded. The scheduler is also responsible for ensuring fair allocation of cores to each instance and preempting them to assign them to other instances.

### 3 Extending FaaS with Kernel-Bypass

In this section, we explain how Junction can be utilized as the backend for executing components in faasd, thereby improving its performance and isolation. Given the similarity of faasd’s architecture to the general architecture presented in Section 2.1, our results can be generalized to other systems.



**Figure 4:** Extending faasd architecture with junctiond.

Figure 4 shows the mapping of the architecture in Figure 2 to components of Junction. There are two key changes when using Junction: a new function manager junctiond, and the use of Junction instances.

junctiond serves as a direct replacement for the local container manager, *containerd*. Instead of deploying processes within a container sandbox, they are deployed within a Junction instance. junctiond also manages requests to increase the concurrency of a given function. The scale factor of a function can be modified in two ways, depending on the programming language of its implementation. For language runtimes that do not support native parallelism, such as Python, multiple processes can be deployed within the same Junction instance. For other languages, the maximum core assignment to a given *uProc* can be modified. If isolation is required across instances of the same function, they can be deployed as independent Junction instances. The scale policy decisions are still performed externally by other components in the FaaS runtime.

Junction instances are utilized not only to host the function code, but also to run the various services in the FaaS runtime (*e.g.*, gateway and provider). This design choice improves the

end-to-end latency and throughput of function invocations, as shown in Section 5. The use of Junction instances is well-suited to the overall architecture of a FaaS runtime for two key reasons: it provides isolation and increases resource utilization efficiency when compared to using containers and other kernel-bypass systems.

Junction instances provide greater isolation for function execution compared to containers. This is due to the Junction kernel’s ability to interpose syscalls and minimize interaction with external components (*i.e.*, host kernel), thus reducing the attack surface. Furthermore, Junction delivers packets directly to services and functions through hardware, bypassing the need for software switching. This reduces the likelihood of malicious software accessing it. Given the need for multi-tenancy in FaaS runtimes, reducing the amount of trusted code that needs to be reviewed and is vulnerable to attack is highly significant.

Junction’s scheduler offers improved resource efficiency compared to standard kernel bypass systems when hosting functions. Typically, kernel-bypass systems require a core to poll for each independent, isolated application instance. However, Junction’s scheduler’s computational cost is proportional to the number of cores being managed, rather than the number of functions hosted on a server. For instance, Junction can use a single dedicated core to manage thousands of functions on a 36-core server.

## 4 Implementation

We implemented junctiond in C++. It is a simple component that manages the configuration of junction instances (including network settings), the deployment of instances via the custom ‘junction\\_run’ command, and the monitoring the running state of all functions. Junctiond is the only component that runs outside of a Junction instance, allowing it to properly spawn isolated junction instances for each function (otherwise the process spawn is handled by the Junction kernel). Additionally, we implemented a new provider extension that connects the provider process in faasd to junctiond.

We also extended faasd’s provider with a caching mechanism. In mainline faasd, the provider forwards any state request to *containerd*. However, for our evaluations, we cache these decisions in the provider, assuming that all requests modifying a running function instance will go through faasd’s gateway. This improves the overall faasd performance, as requests to *containerd* *can be slower than the function invocation itself and can be on the critical path*. Currently, we only cache the number of active replicas of a function, as well as the associated local IP and port for contacting a function. We use the same caching mechanism with junctiond to have a proper comparison. While it is possible that this caching can improve the performance of faasd in general, it is beyond the scope of this work to evaluate it in other contexts.

The code is available in GitHub for the benchmark function [15], the modified provider [14], and junctiond [13].

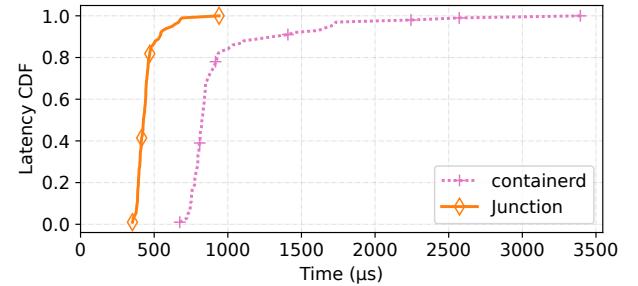
## 5 Evaluation

In this section, we show the benefits in latency and throughput of using Junction as the execution runtime for faasd components and for function execution.

**Methodology.** This experiment runs on 2 machines with 10 core Xeon 4114 CPUs running at 2.2GHz, 48GB of RAM, and 100GbE NICs. We evaluate the setup using invocations of a serverless function from vSwarm [23, 24] that encrypts a 600-byte input with AES. The evaluation compares faasd with both junctiond and containerd, both with the function metadata caching explained in Section 3.

**Functions benchmark.** To conduct the evaluations, we adapted functions from vSwarm [23, 24] to work with the templates provided by faasd. For Go functions, we utilized a custom target for the Go compiler, allowing syscalls to be efficiently handled by the Junction kernel and avoiding the overhead of a trap. For C++ functions, we employed ‘LD\\_PRELOAD’ to override glibc with a custom version when loaded into memory, which similarly forwards requests to the Junction kernel.

**Average latency.** Figure 5 shows the latency distribution for 100 sequential invocation to the AES function with 600-byte random inputs. Junction reduces both median and P99 by 37.33% and 63.42%, respectively. The execution time for the function is also improved, as Junction performs many of the operating system operations in user-space. The median of the function execution latency is reduced by 35.3%, while the P99 is reduced by 81%. Junction. The improvements shown in both figures 5 and 6 are due to both the improvements in networking and compute latencies. The compute optimization are related to better thread multiplexing and reduction in context switches due to the Junction kernel.



**Figure 5:** faasd [8] latency distribution as observed from the gateway for 100 sequential invocations to an AES function [23, 24]. Junction significantly improves the median and tail latency.

**Tail latency vs load.** Figure 6 shows the tail latency across varying request rates offered via the front-end load balancer. Junction can sustain 10× more throughput while lowering the



**Figure 6:** faasd response-time at varying offered loads. Junction offers higher throughput and lower tail latency.

latency by  $\sim 2\times$  at the median and  $\sim 3.5\times$  at the tail. This reflects the compounding end-to-end benefit of using Junction across multiple components running in separate instances.

**Cold starts.** We do not evaluate cold-starts for junctiond, but we separately profiled the startup costs for a single-threaded Junction instance and found that Junction takes 3.4 ms to initialize them.

## 6 Conclusion

We discussed the use of kernel-bypass networking in Function as a Service (FaaS) runtimes, specifically using Junction as the backend for executing components in faasd. Our results show that using Junction can improve performance by reducing latency and increasing throughput, while also reducing the attack surface when compared to containers. Junction reduces both median and P99 latency by 37.33% and 63.42%, respectively, and can sustain  $10\times$  more throughput while lowering the median latency by  $2\times$  and the tail by  $3.5\times$ .

## References

- [1] Apache. Openwhisk. <https://openwhisk.apache.org/>, 2024.
- [2] Azure. Functions. <https://azure.microsoft.com/en-us/products/functions/>, 2024.
- [3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.
- [4] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand Container Loading in AWS Lambda. In *USENIX ATC*, 2023.
- [5] containerd. containerd overview. <https://containerd.io/docs>, 2024.
- [6] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefer. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *IPDPS*, 2023.
- [7] D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*, 1995.
- [8] faasd. A lightweight & portable faas engine. <https://github.com/openfaas/faasd>, 2024.
- [9] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *NSDI*, 2024.
- [10] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*, 2020.
- [11] gRPC. gRPC Remote Procedure Call. <https://grpc.io/>, 2024.
- [12] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sungewan Ihm, Dongsu Han, and KyounghSo Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *NSDI*, 2014.
- [13] Junction. Junctiond. [https://github.com/esaurez/junction/tree/junction\\_manager\\_prometheus/manager](https://github.com/esaurez/junction/tree/junction_manager_prometheus/manager), 2024.
- [14] Junction. Junctiond faas provider. <https://github.com/esaurez/faasd>, 2024.
- [15] Junction. vHive faas benchmark. [https://dev.azure.com/msresearch/Serverless-Efficiency/\\_git/faasd\\_functions](https://dev.azure.com/msresearch/Serverless-Efficiency/_git/faasd_functions), 2024.
- [16] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for  $\mu$ Second-Scale Tail Latency. In *NSDI*, 2019.
- [17] Knative. Knative documentation. <https://knative.dev/docs/>, 2024.
- [18] Linux Foundation. Data Plane Development Kit (DPDK). <https://www.dpdk.org>, 2024.
- [19] OpenFaaS. Serverless functions made simple. <https://docs.openfaas.com>, 2024.
- [20] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. In *OSDI*, 2015.

- [21] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *SIGCOMM*, 2022.
- [22] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Lau-reano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*, 2020.
- [23] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*, 2021.
- [24] vSwarm. Serverless benchmarking suite. <https://github.com/vhive-serverless/vSwarm>, 2024.
- [25] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Data-path OS Architecture for Microsecond-Scale Datacenter Systems. In *SOSP*, 2021.