SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing

Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan
University of California, Riverside, CA

ABSTRACT

Serverless computing promises an efficient, low-cost compute capability in cloud environments. However, existing solutions, epitomized by open-source platforms such as Knative, include heavyweight components that undermine this goal of serverless computing. Additionally, such serverless platforms lack dataplane optimizations to achieve efficient, high-performance function chains that facilitate the popular microservices development paradigm. Their use of unnecessarily complex and duplicate capabilities for building function chains severely degrades performance. 'Cold-start' latency is another deterrent.

We describe SPRIGHT, a lightweight, high-performance, responsive serverless framework. SPRIGHT exploits shared memory processing and dramatically improves the scalability of the dataplane by avoiding unnecessary protocol processing and serialization-deserialization overheads. SPRIGHT extensively leverages event-driven processing with the extended Berkeley Packet Filter (eBPF). We creatively use eBPF's socket message mechanism to support shared memory processing, with overheads being strictly load-proportional. Compared to constantly-running, polling-based DPDK, SPRIGHT achieves the same dataplane performance with 10× less CPU usage under realistic workloads. Additionally, eBPF benefits SPRIGHT, by replacing heavyweight serverless components, allowing us to keep functions 'warm' with negligible penalty.

Our preliminary experimental results show that SPRIGHT achieves *an order of magnitude* improvement in throughput and latency compared to Knative, while substantially reducing CPU usage, and obviates the need for 'cold-start'.

CCS CONCEPTS

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

KEYWORDS

Serverless, eBPF, event-driven, function chain

ACM Reference Format:
Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3544216.3544259

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544259

1 INTRODUCTION

Serverless computing has grown in popularity because users have to only develop their applications while depending on a cloud service provider to be responsible for managing the underlying operating system and hardware infrastructure. The typical costs borne by the user are only for processing incoming requests. This event-driven consumption of resources is attractive for cloud users, especially when their demand is intermittent. It does, however, place the burden on the cloud service provider to provide adequate resources on-demand and ensure the quality of service requirements are met.

In many cases, serverless frameworks are profligate in their resource consumption. They provide the needed functionality by loosely coupling serverless functions and middleware components that run as a separate container and/or pod.[1] This can be extremely resource-intensive, especially when deployed in a limited capacity environment, *e.g.,* edge cloud [57]. There are still a number of shortcomings to be overcome for building a high-performance, resource-efficient, and responsive serverless cloud. Some contributors to this overhead are the following.

Use of heavyweight serverless components. In a serverless environment, each function pod has a dedicated sidecar proxy, distinct from its application container. Sidecar proxies help build an inter-function service mesh layer with extensive functionality support, *e.g.,* metrics collection and buffering, facilitating serverless networking and orchestration. However, the existing sidecar proxy is heavyweight since it is continuously running and incurs excessive overheads, including 2 data copies, 2 context switches, and 2 interrupts (see §2) for a **single** request. Moreover, since most serverless frameworks primarily focus on HTTP/REST API [6, 25, 47], additional protocol adaptation is required for specialized use cases, *e.g.,* IoT (Internet-of-Things) with MQTT [19, 73], CoAP [38]. The current design runs protocol adaptation as an individual component, resulting in substantial resource consumption. Having such a heavyweight design may overload serverless environments, especially in resource-limited edge clouds or when handling infrequent workloads (*e.g.,* IoT). Instead, going a step further and invoking code for execution on a completely event-driven basis without using an individual component can result in substantial resource savings.

Poor dataplane performance for function chaining. Modern cloud-native architectures decompose the monolithic application

[1] "one-container-per-Pod" is the most common model used by Kubernetes for running a function instance.

into multiple loosely-coupled, chained functions with the help of platform-independent communication techniques, *e.g.,* HTTP/REST API, for the sake of flexibility. But, this involves context switching, serialization and deserialization, and data copying overheads. The current design also relies heavily on the kernel protocol stack to handle the routing and forwarding of network packets to and between function pods, all of which impact performance. Although function chaining brings flexibility and resiliency for building complex serverless applications, the decoupled nature of these chains also requires additional components (*e.g.,* a message broker such as Apache Kafka [34], to coordinate communication between functions, and a load balancer like Istio [13]). The resulting complex data pipelines add more network communications for the function chain. All of this contributes to poor dataplane performance (lower throughput, higher latency), potentially compromising service service level objectives (SLOs).

In this paper, we design SPRIGHT, a high-performance, event-driven, and responsive serverless cloud framework that utilizes shared-memory processing to achieve high-performance communication within a serverless function chain. We base the design of SPRIGHT[2] on Knative [14], a popular open-source serverless framework. Evaluation results are presented for SPRIGHT and compared with Knative under various realistic serverless workloads in a cloud environment. Our event-driven shared memory processing, includes event-driven proxies (we call them the EPROXY and SPROXY) that significantly reduce the high resource utilization in the Knative design. This results in much lower latency. SPRIGHT overcomes the challenges of existing serverless computing with the following innovations:

(1) We design the SPRIGHT gateway, a chain-wide component, to facilitate shared memory processing within a serverless function chain. The SPRIGHT gateway consolidates protocol stack processing in the Linux kernel and distributes the payload to the chain.

(2) We implement zero-copy message delivery within a serverless function chain by using event-based shared memory communication. This avoids the unnecessarily duplicated in-kernel packet processing between functions, achieving high-speed, highly scalable packet forwarding within a serverless function chain. Event-based shared memory communication helps reduce CPU usage and alleviate penalties when keeping the function chain warm.

(3) We design event-driven proxies (*i.e.,* EPROXY and SPROXY) using the eBPF (extended Berkeley Packet Filter [66]), that effectively replace the heavyweight sidecar proxy. We support the functions of metrics collection *etc.*, with much lower CPU consumption.

(4) We implement separation at the function-chain level in SPRIGHT's shared memory processing by restricting access to a private shared memory to trusted functions of only that chain. The SPROXY further restricts unauthorized access by applying message filtering for inter-function communication.

(5) We utilize the packet redirection function provided by eBPF to improve packet forwarding performance outside the serverless function chain. Compared to the kernel networking stack, the eBPF-based dataplane dramatically lowers latency and CPU consumption.

[2]SPRIGHT is publicly available at https://github.com/ucr-serverless/spright.git

(6) We optimize protocol adaptation by running it as an event-driven component attached to the SPRIGHT gateway, to avoid unnecessary networking protocol stack processing overhead. This optimization can significantly reduce latency.

2 BACKGROUND AND CHALLENGES

There are a variety of implementations for function chaining since there is no standard for a general solution architecture for serverless applications. The data pipeline patterns for function chaining of different open-source serverless platforms are slightly different, depending on the messaging model applied, *e.g.,* a publish/subscribe model typically uses a message broker as the intermediate component for coordinating invocations within the function chain, while the request/response model typically employs a front-end proxy to perform invocations within the function chain. We examined the design of several proprietary and open-source serverless platforms [8, 15, 27, 28, 56] and developed a common abstract model of the typical data pipeline pattern they use, as shown in Fig. 1.

The data pipeline for function chains uses a message routing as follows: ① Clients send messages (requests) to a message broker/front-end proxy through the ingress gateway of the cluster. ② The messages are queued in the message broker/front-end proxy and registered as an event. ③ The message broker/front-end proxy sends the message to an active pod of the head (first) function in the chain, as defined by the user. ④ The function pod is invoked to process the incoming request. After the first function processes the request, a response is returned and queued in the message broker/front-end proxy, registered as a new event for the next function in the chain. ⑤ The message broker/front-end proxy sends this new event to an active pod for the next function in the chain.

Unfortunately, this data pipeline poses several challenges that are common across the different serverless platforms. The core dataplane components, including the ingress gateway, message broker/front-end proxy, sidecar proxy, *etc.*, are usually implemented as individual, constantly-running, loosely coupled components. In addition, for internal calls within the chain, each involves context switching, serialization/deserialization, and protocol processing.

We quantify the overheads in the representative open-source platform, Knative, through systematic auditing performed with a '1 broker/front-end + 2 functions' chain setup based on the current design depicted in Fig. 1. We assume all evaluated components

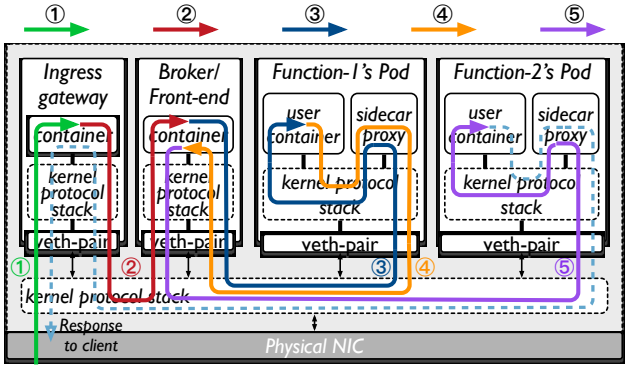


Figure 1: Networking processing involved in a typical serverless function chain setup.

Table 1: Per request Knative overhead auditing of data pipelines for a '1 broker/front-end + 2 functions' chain.

Data Pipeline No.	*External*			*Within chain*				**Total**
	①	②	*total*	③	④	⑤	*total*	
# of copies	1	2	3	4	4	4	12	15
# of context switches	1	2	3	4	4	4	12	15
# of interrupts	3	4	7	6	6	6	18	25
# of protocol processing tasks	1	2	3	3	3	3	9	12
# of serialization	1	1	2	2	2	2	6	8
# of deserialization	0	1	1	2	2	2	6	7

are deployed on the same node, with the overhead on the external client-side excluded. We use a NGINX [21] server function for this audit. However, our results are generally applicable, as these basic overheads are independent of the function used. We examine the different overheads incurred in the data pipeline processing of one request (from ① to ⑤), including # of copies, # of context switches, *etc.* as listed in Table 1. Due to implementation-specific differences, *e.g.,* running multiple threads on the same CPU core, there may inevitably be additional context switches. Our audit aims to quantify the minimum value of each type of overhead. Based on these observations, we list the following key takeaways:

Takeaway#1: Excessive data copies, context switches, and interrupts. With the existing Knative framework, each request results in 15 data copies, 15 context switches, and 25 interrupts throughout the entire data pipeline. Surprisingly, most of the overhead (80%) comes from networking within the function chain (from ③ to ⑤). Current approaches for serverless function chaining rely on the composition of existing networking components to support asynchronous and reliable message exchange between functions, and traffic within the chain has to go through the message broker/front-end proxy each time over the kernel. This inevitably introduces additional data copies, context switches, and interrupts, thus increasing overhead. Furthermore, as the chain becomes more complex, the number of data copies, context switches, and interrupts increase linearly, resulting in very poor scaling.

Takeaway#2: Excessive, duplicate protocol processing. Protocol processing is another major source of overhead. As seen in Table 1, networking *within* the function chain accounts for 75% of the total protocol processing overhead, reflecting the problematic design of current serverless function chains. Protocol processing tasks, including checksum calculation in software and complex iptables processing,[3] contribute to latency and results in poor scaling (especially as the number of iptables rules increases) [55].

Takeaway#3: Unnecessary serialization/deserialization. REST API and HTTP require additional serialization and deserialization operations to convert application data to byte streams before being transmitted over the network. These operations incur significant overhead (lowering throughput and adding latency) [71]. Each step in the data pipeline for the function chain (from ③ to ⑤) introduces 2 serialization and 2 deserialization operations. As shown in Table 1, current designs further amplify this degradation with an excessive number of protocol stack traversals.

Takeaway#4: Individual, constantly-running heavyweight components. Serverless platforms equip each function pod with an individual, constantly-running sidecar proxy to handle inbound and outbound traffic. The presence of this sidecar proxy introduces a significant amount of overhead. Just going through step ④, the sidecar proxy introduces 2 data copies (50%), 2 context switches (50%), and 2 interrupts (33%). To understand the impact of this overhead on dataplane performance, we evaluate several sidecar proxies, including the Envoy sidecar from Istio [12], Queue proxy from Knative [53], and the OF-watchdog from OpenFaaS [22]. We use these sidecar proxies to work with NGINX [21] as a representative HTTP server function. We also use this NGINX HTTP server function without sidecar proxies as the baseline to quantify the additional overhead introduced by the sidecar proxy. We disable autoscaling and limit ourselves to a single function instance. We use *wrk* [3] as the workload generator and send variable-size HTTP traffic (2% 10KB requests, 98% 100B requests) directly to the function pod (including sidecar). Both *wrk* and the function pod are running on the same node.

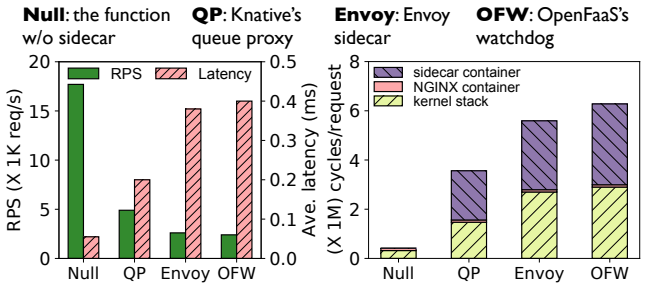


Figure 2: Performance and overhead breakdown of different sidecar proxy implementations.

Our experimental results are shown in Fig. 2. Equipping a sidecar proxy results in a 3×–7× reduction in throughput, 3×–7× higher latency, and a significant increase (3×–7×) in CPU cycles per request. Even though the overhead varies, it is common across all the evaluated sidecar proxies. Looking deeper at the CPU overhead breakdown, the kernel stack for the sidecar proxy consumes 50% of CPU cycles. This substantial overhead of sidecar proxies undercuts the benefit of serverless computing and calls for a more lightweight serverless capability to provide the same functionality.

Summary: The expected benefit of serverless computing was to overcome the inefficiencies of 'serverful' computing. However, the excessive overhead in current serverless frameworks shows that the 'server' is still entrenched in serverless computing. Our auditing shows that the loosely coupled construction of existing components for serverless computing results in substantial unnecessary processing overhead, possibly discouraging the implementation of microservices as function chains. This poor dataplane design and having individual, constantly-running components in the function chain prompt us to create a more streamlined, responsive serverless framework by considering high-performance shared memory processing and lightweight event-driven optimizations to help extract the 'server' out of serverless computing.

3 SYSTEM DESIGN OF SPRIGHT

In this section, we start with the overall architecture of SPRIGHT by justifying the design of each component and discussing the benefits it achieves in improving serverless environments. We then discuss each part separately, including the shared memory processing for

[3] [61] reports that the overheads for iptables processing in a typical Kubernetes environment (also applicable to Knative) using the Container Network Interface accounts for 60% of the total networking overhead.

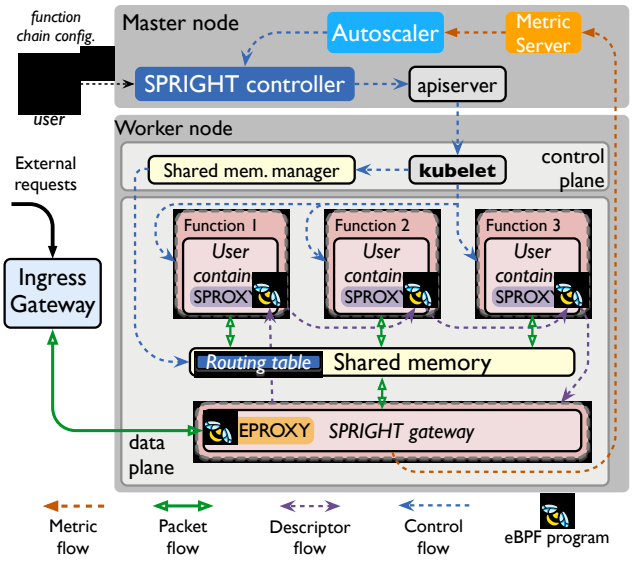


Figure 3: The overall architecture of SPRIGHT

communication within serverless function chains, the lightweight event-driven proxy, the security domain design, the dataplane acceleration for communication outside the function chain, lightweight protocol adaptation, and intelligent autoscaling.

3.1 Overview of SPRIGHT

In this work, we start with open-source Knative as the base platform [14]. Using an innovative combination of event-driven processing and shared memory, we support high performance while being resource-efficient and providing the flexibility to build microservices using serverless function chaining. Importantly, we extensively use eBPF in SPRIGHT for networking and monitoring. eBPF is an in-kernel lightweight virtual machine that can be plugged in/out of the kernel with considerable flexibility, efficiency, and configurability [66]. The execution of eBPF programs is triggered only whenever a new event arrives, thus working naturally with the event-driven serverless environment. Using eBPF, various event-driven programs can be attached to kernel hook points (*e.g.,* the network or socket interface). This enables high-speed packet processing [45, 70] and low-overhead metric collection [51, 75]. eBPF achieves its configurability through eBPF maps – a configurable data structure shared between the kernel and userspace. With eBPF maps, a more flexible dataplane can be implemented with customized routing. The good features of eBPF help us provide functionality with resource use that is strictly load-proportional, a highly desirable toolbox for serverless environments.

Fig. 3 shows the overall architecture of SPRIGHT. We introduce a SPRIGHT controller to coordinate the control plane for functions working in conjunction with the orchestration engine (*i.e.,* Kubernetes and Knative). The SPRIGHT controller runs as a cluster-wide control plane component in the Kubernetes master node. It cooperates with the *kubelet*, which is an indispensable pod management process in the Kubernetes control plane that runs on each worker node, to manage the lifecycle of the pods. In addition, the SPRIGHT controller works with the autoscaler and placement engine (*i.e.,* Kubernetes scheduler) to determine the scale of the function chain and placement of the function chain at the appropriate worker node.

Given a function chain creation request from the user, the SPRIGHT controller instructs a *kubelet* on a selected worker node to create and assign necessary control and data plane components for the function chain, including the shared memory manager and the SPRIGHT gateway, and start up the functions in the chain based on the user configuration. To route external requests to the SPRIGHT gateways of different function chains, we use a cluster-wide Ingress Gateway to distribute the traffic.

To flexibly manage traffic in and out of the function chain in SPRIGHT and avoid duplicate protocol processing within the chain, we create a SPRIGHT gateway. It acts as a reverse proxy for the function chain to consolidate the protocol processing. The SPRIGHT gateway relies on the kernel protocol stack for protocol processing and extracts the application data (*i.e.,* Layer 7 payload). It intercepts incoming requests to the function chain and copies the payload into a shared memory region. This enables zero-copy processing within the chain, avoids unnecessary serialization/deserialization and protocol stack processing. The SPRIGHT gateway invokes the function chain for requests, processes the results, and constructs the HTTP response to external clients. SPRIGHT assumes that functions in the same chain run within the same node, to derive the benefits of sharing the memory between functions. We dedicate a SPRIGHT gateway for each function chain (§3.4). To mitigate the concern of overhead when there are many chains in the cluster, we emphasize that the SPRIGHT gateway is a lightweight component with a relatively small memory footprint (27KB compared to each (even simple) Golang-based function that is more than 2MB). The CPU consumption of the SPRIGHT gateway is also not a significant concern. We share the requisite CPU cores across multiple SPRIGHT gateways. In addition, since the SPRIGHT gateway processes requests based on kernel interrupts, its event-driven nature results in the CPU usage being largely load-dependent.

To eliminate impact of additional networking components for function chaining, we design Direct Function Routing (DFR). DFR exploits shared memory and leverages the configurability provided by eBPF maps. DFR allows dynamic update of routing rules and uses shared memory to pass data directly between functions (§3.2.3).

We design a lightweight, event-driven proxy (EPROXY and SPROXY) that uses eBPF to construct the service mesh instead of a continuously-running queue proxy associated with each function instance, as is used by Knative. Thus, we reduce a significant amount of the processing overhead (§3.3).

SPRIGHT also incorporates security domains to restrict unauthorized access between different chains by coordination with *kubelet*. SPRIGHT's security domain design isolates different chains by creating a private shared memory pool for each chain and applying message filtering for inter-function communication. We discuss the security considerations in more detail in §3.4.

To accelerate the data path outside the function chain, we utilize XDP/TC hooks [67] in eBPF to forward packets between other serverless dataplane components, *e.g.,* ingress gateway and to/from the chain. An XDP/TC hook processes packets at the early stage of the kernel receive (RX) path before packets enter into the kernel iptables [36, 45], resulting in substantial dataplane performance improvement without dedicated resource consumption, compared to a constantly running queue proxy that depends on the kernel protocol stack (§3.5).

Event-driven processing can help tremendously in interfacing serverless frameworks, which have an HTTP/REST API, with a variety of application-specific protocols (*e.g.,* for IoT with MQTT [19], CoAP [38]). Current designs use a separate protocol adapter for translation between these protocols. However, since SPRIGHT's shared memory processing directly works on payloads independent of the application layer (L7) protocols, the protocol adapter can ideally run as an internal event-driven component that is part of the SPRIGHT gateway. This way, we achieve a much more streamlined protocol adapter design, using resources strictly on demand (§3.6).

Although these optimizations are built around the Knative-based environment, our concepts and methodology can also be broadly applied to other serverless platforms.

3.2 Optimizing communication within serverless function chains

3.2.1 Shared memory within a function chain. SPRIGHT allocates a private shared memory pool with Linux HugePages for each serverless function chain. Using HugePages can reduce the access overhead of in-memory pages, thus improving the performance of serverless functions when accessing data in the shared memory pool. In addition, the shared memory pool within the function chain supports queueing to help sustain traffic bursts.

To enable zero-copy data movement between functions, shared memory processing relies on packet descriptors to pass the location of data in the shared memory pool, which is then accessed by the function. One implementation option is DPDK, which uses polling-based RTE rings [29] to deliver the packet descriptor through its multi-process support [20]. DPDK has been extensively used to build up high-performance dataplane for cloud services [74]. While DPDK allows for fast packet processing and low latency, it continuously consumes significant CPUs independent of traffic intensity.

Instead of using heavyweight polling-based shared memory processing, SPRIGHT dynamically extends the use of the socket interface at the function pod by attaching an eBPF Socket Message program (SPROXY in Fig. 4) [62]. SPROXY works with eBPF's socket

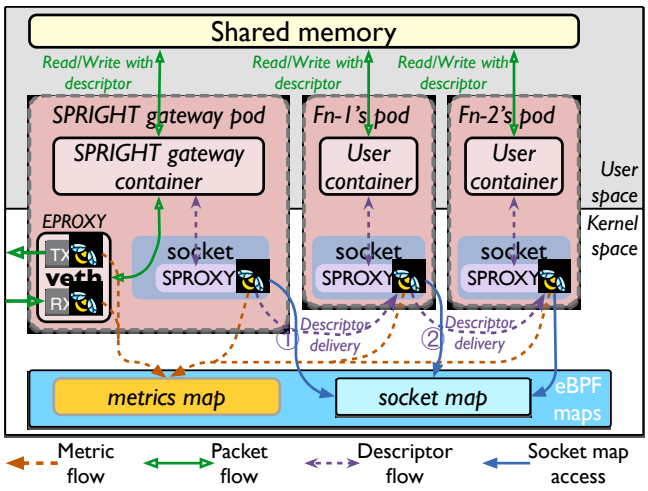


Figure 4: Event-driven EPROXY & SPROXY, shared memory, and DFR within a chain: (①) the SPRIGHT gateway invokes the head function of chain; (②) the head function calls the next function bypassing the SPRIGHT gateway.

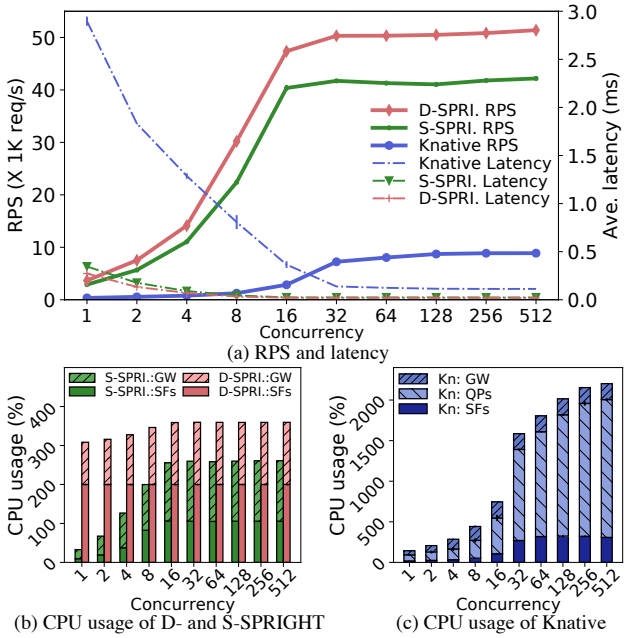
map to enable message redirection between socket interfaces of function pods. The packet descriptor used by SPROXY is a small 16-byte message to minimize overhead. A packet descriptor contains two fields: the instance ID of the next function and a pointer to the data in shared memory. Once the SPROXY receives a packet descriptor, it extracts the instance ID of the next function, which is then used to query the eBPF's socket map to retrieve the target socket interface information (*i.e.,* the file descriptor). Appendix-A describes the zero-copy based packet flow of SPRIGHT. SPRIGHT's gateway maintains the in-kernel eBPF's socket map (Fig. 4). When a new function pod instance starts, the SPRIGHT gateway updates its instance ID and socket interface information in the socket map to support redirection between socket interfaces.

The packet descriptor redirection performed by SPROXY bypasses the kernel protocol stack, incurring minimal latency overhead. SPROXY operates in a purely event-driven manner, avoiding the need to busy-poll packet descriptors and saving CPU resources. Thus communication overhead is entirely load-dependent.

3.2.2 Event-based vs. polling-based shared memory processing. To identify the most appropriate shared memory processing mechanism in the context of serverless computing, we compare SPRIGHT's event-based shared memory processing based on SPROXY (hereafter referred to as S-SPRIGHT) with polling-based shared memory processing based on DPDK (hereafter referred to as D-SPRIGHT), with a function chain containing 2 function pods. We use Apache Benchmark [1] on a second node as the workload generator. We additionally set up a function chain with the base Knative environment and use NGINX as the front-end proxy to coordinate the communication within the chain. Both the SPRIGHT gateway and NGINX proxy are configured with two dedicated cores for a fair comparison. Note: We collect the results from 10 repetitions. All results also show the 99% confidence interval.

As shown in Fig. 5, with low concurrency, *e.g.,* at 32, S-SPRIGHT (0.024ms) shows a slightly higher average response delay compared to DPDK (0.02ms), but still a much lower (almost 6×) response delay compared to Knative (0.138ms). In terms of RPS, both DPDK (50.3K) and S-SPRIGHT (41.7K) are substantially higher than Knative (7.2K), with a significant 5.7× improvement.

As S-SPRIGHT relies on the in-kernel eBPF program (*i.e.,* SPROXY) to deliver packet descriptors, it incurs the overheads for context switching, contributing to the extra latency. However, the SPROXY processing latency is masked when the concurrency increases (≥ 32), because the context switching latency overlaps with the other processing. Throughput increases rapidly, up to 5× that of Knative. Although S-SPRIGHT has a 1.2× lower peak throughput than D-SPRIGHT, S-SPRIGHT has a substantially lower CPU usage because it is purely event-driven. Both of those approaches have a much lower overhead compared to Knative. With a concurrency of 1, S-SPRIGHT consumes 32% CPU, which is 9.6× and 4.5× less than D-SPRIGHT (308%, or more than 3 CPU cores fully used) and Knative (143%), respectively. When the concurrency increases to 32, S-SPRIGHT consumes 259% CPU, which is still less than DPDK (359%). Comparatively, the CPU usage of base Knative increases to a shocking 1585% (more than 15 CPU cores used) at a concurrency of 32 (see Fig. 5 (c)). The queue proxy consumes 70% of Knative's CPU. Even with increasing concurrency (≥ 32), S-SPRIGHT has a



(a) RPS and latency

(b) CPU usage of D- and S-SPRIGHT

(c) CPU usage of Knative

Figure 5: Comparison between polling-based (D-SPRI.) and event-driven (S-SPRI.) shared memory processing with 1 gateway pod and 2 serverless function pods. Kn: Knative; QPs: Queue proxies; SFs: serverless functions; GW: gateway

consistent and steady saving in CPU compared to the others. Individual, constantly-running components (queue proxy with Knative or DPDK's poll mode using up CPUs) have excessive overhead. More importantly, S-SPRIGHT consumes negligible CPU resources when there is no traffic. We observed that S-SPRIGHT's gateway and function pods that are SPROXY-based consume zero CPU when there is no traffic, making it possible to keep a function pod 'warm' to overcome the 'cold start' delay (§4.2.2). Thus, SPROXY-based shared memory processing is ideal for serverless computing, especially for function chains.

3.2.3 Direct Function Routing & Load balancing within a function chain. To optimize the invocations within a function chain, we use Direct Function Routing (DFR), which enables the upstream function in the chain to directly invoke/communicate with the downstream function. As shown in Fig. 4, the SPRIGHT gateway only invokes the head function in the chain once (① in Fig. 4). When the first function completes the request message processing (② in Fig. 4), it directly calls the next function without going through the SPRIGHT gateway. The rest of the function invocations in the chain also bypass the SPRIGHT gateway, thus significantly reducing the invocation latency (and overhead) for the function chain. To support DFR, SPRIGHT adopts a two-step routing mechanism. It uses a chain-specific, userspace routing table, and an in-kernel socket map. The userspace routing table helps determine the ID of next function while the in-kernel socket map uses that function ID to find its corresponding socket file descriptor, which is then used by the SPROXY to perform the actual packet descriptor delivery between the sockets of the source and destination function.

We use the SPRIGHT controller (Fig. 3) to manage DFR within the function chain. The SPRIGHT controller configures the routing table based on the user-defined sequence for the function chain. We keep the routing table in shared memory to reduce access latency. To support multiple downstream functions, we use a 'topic' (extracted from the message payload) based publish/subscribe messaging model, and dynamically route requests using the routing table. The message topic and the ID of the current function serve as the key to looking up the ID of the next-hop function in the routing table. For load balancing, we select the active pod instance with the maximum residual service capacity[4] and pack its instance ID into the packet descriptor. The invocation is then performed through the SPROXY based on configured instance ID, without going through the SPRIGHT gateway.

3.3 Event-driven proxy (EPROXY & SPROXY)

In Knative, the queue proxy runs as an additional container in a function pod distinct from the user container. It buffers incoming requests before forwarding them to the user container, to help handle traffic bursts and maintain throughput. The queue proxy is also responsible for collecting metrics for the pod (*e.g.,* request rate, concurrency level, response time) and exposing them to a metrics server to facilitate control plane decision-making, *e.g.,* autoscaling. However, this design has several drawbacks we described earlier. We overcome these with our lightweight, event-driven eBPF-based EPROXY & SPROXY, replacing the queue proxy.

The goal of EPROXY & SPROXY is to achieve functionality comparable to that of the queue proxy but with lower overhead. We do not need the queueing capability in the EPROXY as the shared memory within the function chain already provides that queueing. Thus, SPRIGHT still provides the same functionality to improve concurrency and handle traffic bursts as a queue proxy. But, eliminating the additional queuing stage helps reduce request delays.

To collect the required metrics for the Knative control plane, we attach eBPF-based monitor programs to the EPROXY (at the SPRIGHT gateway pod) and SPROXY (at function pods), as shown in Fig. 4. The EPROXY at the SPRIGHT gateway pod collects L3 metrics, *e.g.,* packet rate, bytes received, while the SPROXY at function pods collect L7 metrics, *e.g.,* request rate. In addition, we assign a 'metrics map' in the eBPF maps that serves as a local metrics storage on each node. When a new request or response occurs, the monitor programs are triggered to collect and update the metrics to the metrics map. The SPRIGHT gateway has a built-in metrics agent responsible for reading the metrics map periodically and providing the latest metrics to the metrics server. We further extend the SPRIGHT gateway with internal event-driven metrics collection capabilities as an enhancement of EPROXY to provide function-chain-level metrics such as the request rate and execution time on a chain basis. Since the EPROXY and SPROXY are only triggered when there are incoming requests, there is no CPU overhead when idle. Although EPROXY and SPROXY work in the kernel, they are created by the cloud service provider rather than the user, which does not affect the isolation of the user function. This is similar to how serverless platforms attach a sidecar to a user function.

[4]The residual service capacity ($RC_{i,t}$) of a function pod i at time t is calculated by $RC_{i,t} = MC_i - r_{i,t}$, where MC_i is the maximum service capacity (*i.e.,* maximum request rate that can be served) of the function pod i and $r_{i,t}$ is the request rate to the function pod i at time t. Both MC_i (estimated at maximum load) and $r_{i,t}$ can be monitored by SPRIGHT's event-driven proxy.

Kubernetes natively supports function pod health checks via the *kubelet*. Working in conjunction with the *kubelet*, SPRIGHT can enable TCP or HTTP probes in functions for health checks. Enabling the TCP or HTTP probes requires a minimal change of opening an additional socket or HTTP server in the function to listen to health check requests from the control plane. Thus, we can dispense with Knative's queue proxy doing a health check to check on function pods, using TCP or HTTP probing.

3.4 Security domains in SPRIGHT

SPRIGHT recognizes the need for isolation between serverless functions in a shared cloud environment, especially with the use of shared memory processing. It is necessary to restrict access of a shared memory pool to only trusted functions. The trust model in SPRIGHT assumes that the functions within a chain trust each other, but the functions in different chains may not. To limit unauthorized access across function chains, SPRIGHT provides two abstractions to construct a security domain for each function chain: 1) a private shared memory pool for each chain; 2) inter-function packet descriptor filtering with the SPROXY.

SPRIGHT takes advantage of DPDK's multi-process support [20] to isolate shared memory pools between chains. The *kubelet* works with the SPRIGHT controller to assign each chain with a dedicated shared memory manager and a SPRIGHT gateway at startup. The shared memory manager runs as a DPDK primary process to have the privileged permission to initialize the shared memory pool (using `rte_mempool_create()` API). Each shared memory manager owns a unique shared data file prefix – a multiprocessing-related option in DPDK used to isolate different memory pools [20]. By specifying the correct prefix, the gateway and functions in SPRIGHT, which run as DPDK secondary processes, can attach to the memory pool (use `rte_memzone_lookup()` API) created by the chain's shared memory manager. The SPRIGHT gateway of each chain is used to consolidate protocol processing and move the payload to this private shared memory pool. Functions in the same chain are assigned the same shared data file prefix upon creation and are identified as trusted functions in their dedicated security domain.

SPRIGHT's DFR allows function-to-function communication. Malicious functions may crash other functions by intentionally sending a packet descriptor pointing to an unauthorized memory address. SPRIGHT leverages the extensibility of the SPROXY to enforce inter-function message filtering, where we check carefully which function has write and read access to each descriptor. Upon the reception of a descriptor, the SPROXY performs a rule look-up in the filtering map (built on eBPF maps) to check if the destination of this packet descriptor is allowed. If the destination is not authorized, SPROXY discards the packet descriptor. During the function startup,

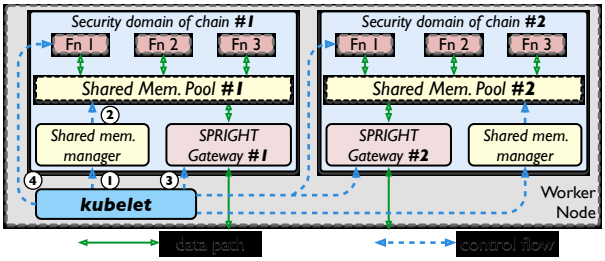


Figure 6: Security domains isolating function chains

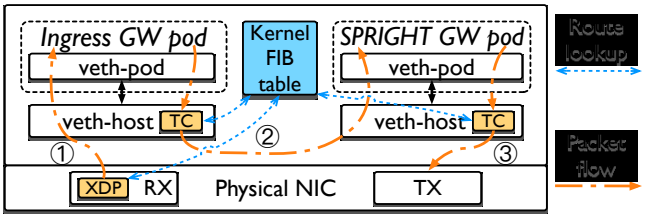


Figure 7: Dataplane acceleration using eBPF XDP/TC hooks

the *kubelet* configures the rules in the filtering map. SPRIGHT also supports dynamic configuration of rules during runtime.

Fig. 6 shows the startup flow of a function chain in SPRIGHT: ①) On receiving a request to create a function chain, the SPRIGHT controller starts a shared memory manager dedicated to the chain. ②) The shared memory manager initializes a private memory pool for the chain. ③) The SPRIGHT controller creates a dedicated SPRIGHT gateway for the chain. ④) The SPRIGHT controller starts the functions in the chain and attaches a SPROXY to each function while also configuring the filtering rules in the eBPF map.

3.5 eBPF-based dataplane acceleration for external communication

We exploit eBPF's XDP/TC hooks to accelerate the communication by the function chain in SPRIGHT to external components. We develop an eBPF forwarding program and attach it to the XDP/TC hook that is positioned on the RX path of the network interface, including the host-side veth of the pod (*i.e., veth-host*[5]) and the physical NIC, as shown in Fig. 7. eBPF offers packet redirect features (*i.e.,* 'XDP_REDIRECT' and 'TC_ACT_REDIRECT') that support passing raw frames between the virtual network interfaces, or to and from the physical NIC without going through the kernel protocol stack [11]. This helps save CPU cycles consumed by iptables. The eBPF forwarding program has two functions: 1) Look up the kernel FIB (Forwarding Information Base) table to find the destination network interface based on the FIB parameters [7] of the received packet, including the IP 5-tuple, index of source interface, *etc.* 2) Forward the raw packet frame to the target (*veth-host* or NIC) interface via 'XDP_REDIRECT' or 'TC_ACT_REDIRECT'. The communication could be either in the same node or across different nodes, supported by an eBPF-based dataplane via the eBPF forwarding program. An XDP program at the physical NIC processes all inbound packets received by the NIC. It redirects the packet to the *veth-host* of the destination function pod after a routing table lookup (① in Fig. 7). The TC program at the *veth-host* handles the outbound packet from the function pod. Depending on the packet's destination, the TC program may take different routes. If the destination of the packet is to another function pod (*e.g.,* traffic between ingress gateway pod and SPRIGHT gateway pod) on the same node, the TC program directly passes the packet to the *veth-host* of the destination function pod via 'TC_ACT_REDIRECT' (② in Fig. 7). If the destination function pod is on another node, the TC program redirects the packet to the NIC (③ in Fig. 7). Our evaluation shows that XDP/TC redirection helps achieve a 1.3× improvement in throughput and a 20% reduction in latency under peak load.

[5]A function pod is connected to the host through a pair of veths, *i.e.,* the host-side veth and pod-side veth.

3.6 Event-driven protocol adaptation

To run a protocol adapter as an internal, lightweight event-driven component, we predefine 'protocol adaptation hook points' on the packet datapath inside the SPRIGHT gateway, just before the gateway sends messages to the function pod. The protocol adaptation hook is a function call entry point that can be invoked to execute customized protocol adaptation programs that are attached. Once an application-specific message arrives at the hook point, the protocol adaptation program is triggered and executed. With internal event-driven execution, invocations are integrated into the same component without extra context switching and networking overhead. Our design supports attaching the protocol adapter program at runtime by exploiting dynamic code injection [10]. Different programs are pre-compiled into a dynamic library and can be loaded to, or unloaded from, the hook point at runtime according to the protocol adapter's requirements. This facilitates compatibility when handling traffic specific to each distinct protocol. In addition, dynamic loading of the program helps reduce startup time compared to initializing a separate protocol adapter function pod.

Our adapter works seamlessly with stateless protocol adaptation, *e.g.,* HTTP, since managing the transport layer (L4) connections is offloaded to the SPRIGHT gateway. However, some adaptation scenarios with stateful protocols, *e.g.,* MQTT, require an additional L7 connection establishment before exchanging messages [19]. To retain stateless protocol adaptation, we use the SPRIGHT gateway to handle the L7 connection establishment rather than the internal protocol adapter. Then, the SPRIGHT gateway passes the received application messages to the event-driven protocol adapter, which extracts the payload from the application message and delivers it to shared memory. To improve interoperability and compatibility with current serverless platforms, our adapter is designed to be compatible with the CloudEvent specification [9], an event data format widely adopted by serverless platforms [15, 26].

3.7 Intelligent autoscaling

In the control plane, the autoscaler scrapes metrics from the metrics server to determine the load intensity, based on which serverless function instances are automatically scaled up or down to serve requests on demand. Knative's autoscaler depends primarily on the users to specify the requested resources for their functions based on a single metric and is unaware of the complexity of function chains. To improve on Knative, several approaches, *e.g.,* Mu [57], GRAF [60], *etc.* have been proposed to more effectively scale cloud resources for serverless applications. SPRIGHT can be used in conjunction with these advanced autoscalers. In addition, SPRIGHT supports each function being scaled independently using vertical pod scaling, especially with adding more CPU cores for the function as needed.

3.8 Discussion

Overhead auditing (contd.) - SPRIGHT: We now perform an audit of overhead for SPRIGHT, following the same methodology used before in §2, and compare it against the base design depicted in Fig. 1. As can be seen in Table 2, SPRIGHT significantly reduces overheads for processing within the function chain. With shared memory processing, SPRIGHT achieves **0** data copies, **0** additional protocol processing, and **no** serialization/deserialization overheads within the chain. Although the use of SPROXY generates context switches and interrupts, which do add latency for processing, the total number of context switches and interrupts for SPRIGHT is still much less than that of the base Knative design (repeated in the last column of Table 2). In addition, the results in Fig. 5 show that the context switches and interrupts introduced by SPROXY have a limited impact on the performance with concurrent processing of just a few sessions. The event-based shared memory processing substantially reduces resource usage, more than compensating for any of the added context switches and interrupts.

Table 2: Per request data pipeline overhead for SPRIGHT for '1 broker/front-end + 2 functions' chain (excluding client overhead). Note-2: DFR: hence no route ④ and ⑤ (Fig. 1). ④: direct route from function-1's pod to function-2's pod.

Data Pipeline No.	*External*			*Within chain*			Total	Total of Kn
	①	②	*total*	③	④	*total*		
# of copies	1	2	3	0	0	0	3	15
# of context switches	1	2	3	2	2	4	7	15
# of interrupts	3	4	7	2	2	4	11	25
# of proto. processing tasks	1	2	3	0	0	0	3	12
# of serialization	1	1	2	0	0	0	2	8
# of deserialization	0	1	1	0	0	0	1	7

Deployment Constraints: In SPRIGHT, functions in the same chain need to be placed on the same node. This requires the placement engine to deploy functions on the basis of a chain. This allows shared memory communication between functions in a chain. In addition, scaling SPRIGHT across multiple nodes requires all the function of a chain to be deployed on each node. This may lead to more resource fragmentation compared to deployment of each function. Additionally, we need to load balance between different function chain units in a multi-node deployment.

Application Porting Requirements: Porting an HTTP/REST-based application to SPRIGHT requires replacing HTTP/REST-based I/O with SPRIGHT's SPROXY-based event-driven shared memory I/O. However, SPRIGHT does not support synchronous calls between functions, *e.g.,* where the client sends a request to the server and waits for a response before moving on to the next step. SPRIGHT's existing programming model assumes that a function's code runs to completion after invocation, is purely event-driven, and inherently supports asynchronous calls between functions. In SPRIGHT, a synchronous call needs to be broken down into multi-step asynchronous calls. For instance, a synchronous "request-response" transaction needs to be treated as two separate asynchronous calls: (1) The calling function sends a request (associated with a caller ID) to the serving function. The calling function can continue processing other requests without pausing and waiting for the response. (2) The serving function generates a response and sends it back to the calling function (based on the caller ID in the request). The calling function receives the response and then continues to process the transaction.

We ported the online boutique application[6] [23] to SPRIGHT and evaluate it in §4.2.1. The source code is available in [30]. Currently, SPRIGHT only supports C-based implementations of user functions. We expect to extend SPRIGHT's language support as part of our ongoing development of SPRIGHT.

[6] [23] is a popular application used to demonstrate the utility of serverless computing.

4 EVALUATION & ANALYSIS

4.1 Experiment Setup

To examine the improvement of SPRIGHT and its components, we consider several typical serverless scenarios, including (1) a popular online shopping boutique, (2) An IoT environment of motion detectors, and (3) a more complex processing of image detection & charging for an automated parking garage. For each scenario, we set up a function chain to execute the serverless application (Fig. 8). The details of the setup for each scenario are as follows:

1. Online Boutique is an open-source representative implementation of a microservice-based online store application [23]. It has 10 different functions communicating with each other using gRPC. We ported these functions to SPRIGHT (in C) and Knative (using the Go language) based on the implementation provided in [23]. Functions ported to SPRIGHT use shared memory, while functions in Knative continue to use gRPC, for inter-function communication. We use Locust [18] as the load generator and use the default workload provided in [23] to generate a realistic web-based shopping application's request pattern. The default workload utilizes a total of 6 different sequences of function chains (see Table 3 in Appendix-B). We compare four alternatives to run the online boutique application, including gRPC, Knative, S-SPRIGHT, and D-SPRIGHT. In the "gRPC" mode ('server-full' approach), the function runs as a Kubernetes pod without a sidecar and uses the built-in gRPC server for functions to talk to each other directly without involving a broker/front-end.[7] In Knative mode, we use the Istio ingress gateway to mediate the communication between functions. We disable the activator [17] (a cluster-wide queuing component in Knative) to avoid additional queuing delays.

2. IoT - Indoor motion detection for automated lighting requires tracking a sequence of events utilizing multiple sensors. The simple function chain contains 2 functions (Fig. 8 (b)). Motion sensors going 'on' triggers an actuator function to turn on the light. The light may be automatically turned off after a period of no activity. We consider the MERL motion detector dataset [72]. We use a traffic generator developed in Python to send motion events based on the timestamps in the dataset. The CPU service time of the sensor function and actuator function are both set at 1ms. For the base Knative setup, we use NGINX to coordinate the communication within the function chain.

3. Parking - image detection & charging takes snapshots of each parking spot as input for visual occupancy (of parking spots) detection in parking lots. It detects the vehicle's license plate and determines whether the plate metadata is stored in the database through a plate search function. If it is not stored, a 'persist-metadata' function is invoked to store the plate metadata in the database. Finally, it charges parking fees based on the license plate's metadata. We consider the *CNRPark+EXT* image dataset collected from a parking lot with 164 parking spaces [33]. We use the same load generator used for IoT workload to send snapshot images (150×150 pixels, ~ 3KB each) through HTTP/REST API call. Every 240-second interval, 164 snapshots are sent to the function chain. We use NGINX to coordinate the message exchanges within the chain. We use VGG-16 as the image detection algorithm, and the CPU service time of the image detection function is set to 435ms [40]. The CPU service times of other functions are listed in Table 4 (Appendix-B).

[7] The "frontend service" (Fig. 8 (a)) in the online boutique runs as a user function, which is distinct from the general broker/front-end. The latter is a system component that used to mediate the communication between functions (*e.g.*, the Istio ingress gateway in Knative mode).

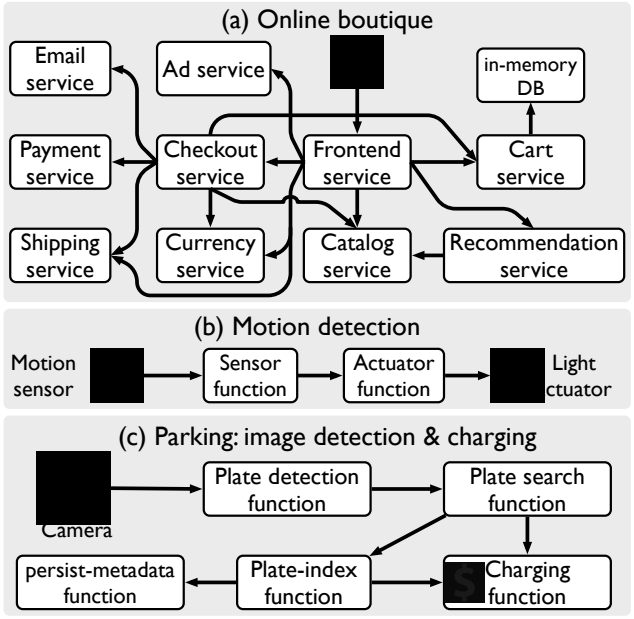


Figure 8: Serverless function chains setup

We use these serverless applications to quantify the performance gain brought by each of SPRIGHT's optimization. We evaluate it based on several metrics, including CPU usage, RPS, and response time. To understand in detail, we show the time series and CDF when appropriate.

Testbed setup: The testbed is built on top of a base Knative platform, including 1) Knative serving/eventing components (v0.22.0) [15, 16]; 2) Kubernetes components (v1.19.0), including API server, placement engine, etcd, *etc* [2]. We consider Calico CNI (Native routing mode) [68] as the underlying networking solution except for the communication within the function chain of SPRIGHT. We run the experiments on the NSF Cloudlab with two c220g5 nodes [41]. Each node has a 40-core Intel CPU@2.2 GHz, 192GB memory, and a 10Gb NIC. We use Ubuntu 20.04 with kernel version 5.16. We configure the concurrency of both Knative and SPRIGHT function as 32. The concurrency level of a function pod determines the # of requests that can be processed in parallel at each time.

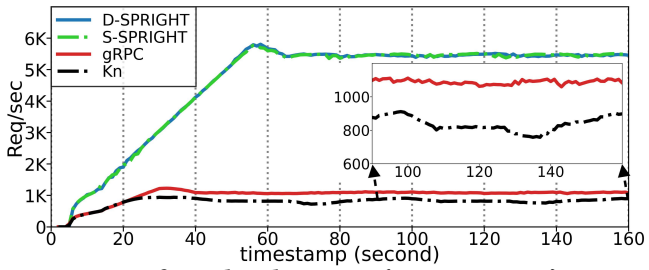


Figure 9: RPS for online boutique: {Knative, gRPC} at 5K & {D-SPRIGHT, S-SPRIGHT (overlap)} at 25K concurrency.

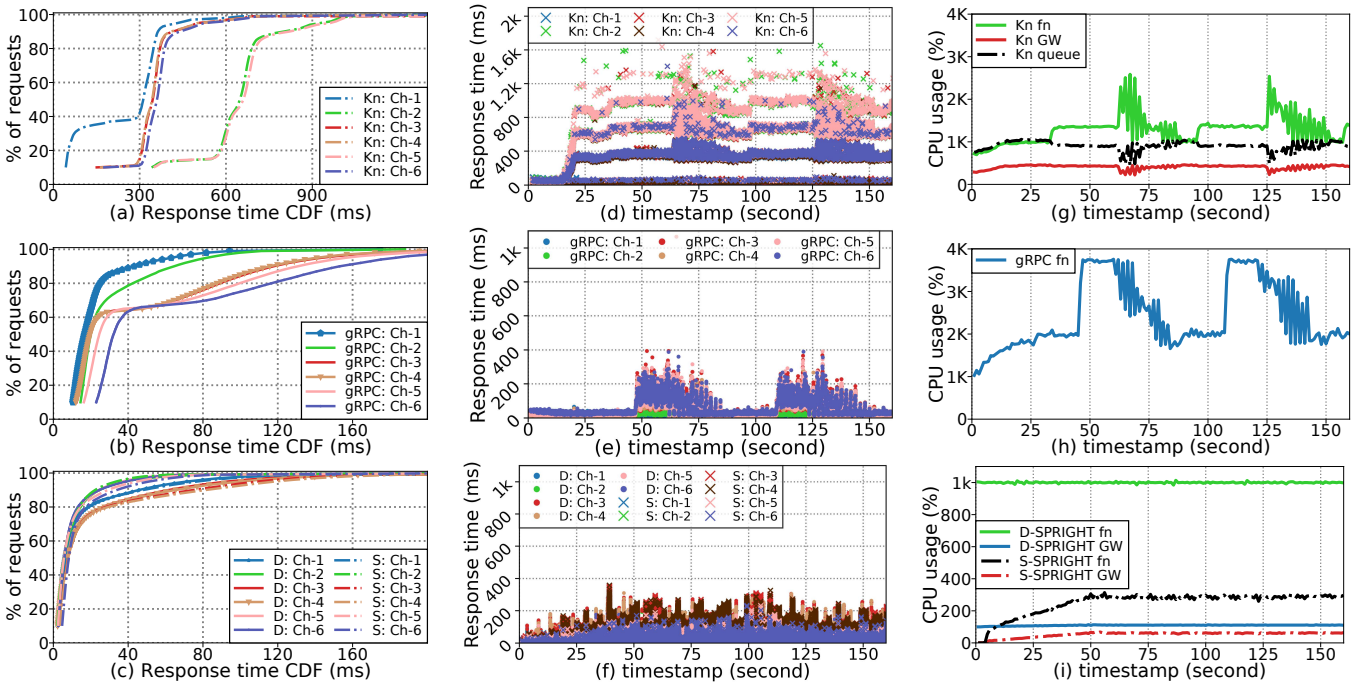


Figure 10: Online boutique. Top row: Knative, 5K concurrency. Mid. row: gRPC, 5K concurrency. Bottom row: {D-SPRIGHT (*D*), S-SPRIGHT (*S*)}, 25K concurrency. (Left col.) Response time CDF for 6 different function chains; (Mid. col.) Time series of response time, function chains; (Right col.) CPU usage time series, gateway (*GW*), function chains (*fn*), queue proxy (Knative)

4.2 Performance with Realistic Workloads

4.2.1 Comparing SPRIGHT, Knative, and gRPC mode. We now compare D-SPRIGHT (using DPDK's RTE rings) and S-SPRIGHT (using SPROXY) against Knative and the gRPC mode for several different function chains (*i.e.,* Ch−1 to Ch−6) of the online boutique application. We configure different concurrency levels (*i.e.,* # of concurrent users) of requests from the Locust load generator. We select two concurrency levels, 5K and 25K, to show here. To achieve the 5K concurrency, we set the spawn rate of 200/sec. concurrent requests. The spawn rate controls the # of concurrency steps increased every second. Above 5K, Knative's performance becomes highly variable with time, indicating overload (also results in very high tail response times). Both S-SPRIGHT and D-SPRIGHT have stable performance at a 25K concurrency level, after which they begin to show behavior indicating a slight overload. To achieve the 25K concurrency, we set the spawn rate of concurrency at 500/sec.

Even at 5K concurrency, Knative already begins to be overloaded. From 0s to 35s (Fig. 9), as the concurrency level of the load generator is ramping up to 5K, and the requests/sec (RPS) increases to ~900 req/sec. Knative begins to overload (see at 35s in Fig. 9) due to the use of sidecars and use of the Istio ingress gateway (hereafter referred to simply as 'gateway') to mediate the communication between functions. At this 5K concurrency, the gateway and sidecars consume ~13 CPU cores (from 35s onwards), which is 50% of the entire Knative setup. It finally leads to CPU contention with the functions, whose CPU utilization soon reaches saturation at 62s (using up ~13 CPU cores, Fig. 10 (g)). In addition, the use of the gateway and sidecars adds additional processing and queuing delays on the request's data path, leading to the reduction in RPS observed (see beyond 30s in Fig. 9). The closed-loop nature of the workload generation and request processing results in the RPS, resource utilization, and response times going through cycles of overload (occurring again between 100s - 140s).

Compared to Knative, gRPC has a more stable RPS and better overload behavior at 5K as gRPC has no sidecars and bypasses the gateway. By removing these heavyweight components, functions in the gRPC mode make full use of CPU resources. The shortened request data path further reduces latency and alleviates overload and queuing problems. As shown in Fig. 10 (a) and (b), the resulting tail latency of gPRC, *i.e.,* 95%ile, of 141ms, measured across all the functions of the online boutique service, which is 4.9× lower than Knative (whose 95%ile is 693ms). Fig. 10 (d) and (e) further demonstrate the benefits of removing sidecars and the gateway. For requests sent between 35s and 75s, the response time of Knative increases significantly while the gRPC shows a delayed overload (only 45s onwards) and its response time during the overload (45s to 75s) is much lower than Knative. However, as gRPC depends on the kernel protocol stack for networking and requires serialization/deserialization. These overheads are not negligible. The entire gRPC setup consumes 91% of the total CPU cores available on the physical node in order to drain the queued requests (*e.g.,* 45s to 75s in Fig. 10 (h)). This pattern repeats again, *e.g.,* in the time period 108s - 140s. Overall, this is quite inefficient.

Compared to Knative and gRPC, D-SPRIGHT and S-SPRIGHT both have stable RPS throughout the experiment, for concurrency levels ranging from 5K all the way to 25K. At 5K concurrency, The 95%ile latency of D-SPRIGHT and S-SPRIGHT are 11ms and 13ms (see Table 5 in Appendix-B), significantly less than Knative (690ms) and gRPC (140ms), while utilizing far less CPU. Although D-SPRIGHT constantly consumes CPU cycles when idle, even at

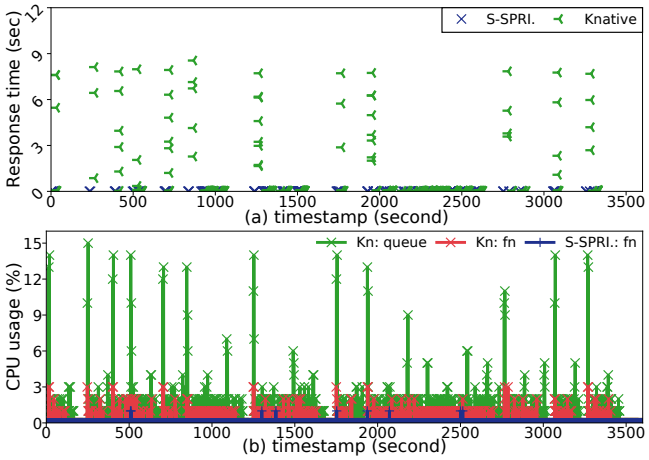


Figure 11: Time series of response time, and CPU utilization for motion detection workload - 1-hour long experiments.

maximum load, it consumes only 11 total CPU cores at a concurrency level of 5K, which is ~2.5× less than Knative (similar to Fig. 5). This again validates the benefits of SPRIGHT's shared memory processing, saving CPU resources by avoiding the needless processing overheads with Knative discussed previously in §2. S-SPRIGHT *further* reduces CPU usage dramatically by using a purely event-driven processing compared to D-SPRIGHT. With 5K concurrency, S-SPRIGHT consumes only ~1 CPU cores, including the gateway and all the functions, getting comparable performance (throughput, response time) to D-SPRIGHT. We further increase the concurrency level of the load generator to 25K for D-SPRIGHT and S-SPRIGHT. This increases the utilization, but still maintaining low tail response times. Both D-SPRIGHT and S-SPRIGHT maintain a stable RPS of ~5500 req/sec (Fig. 9), which is 5× higher than the highest stable RPS achieved with Knative and gRPC. Moreover, S-SPRIGHT far less CPU resources than D-SPRIGHT, even as the load increases. At 25K concurrency, S-SPRIGHT consumes only ~3.5 CPU cores, which is 3× less than D-SPRIGHT (Fig. 10(i)), showing the benefit of the eBPF-based event-driven processing.

With SPROXY generating context switches and interrupts for descriptor delivery (Table 2), there is some additional latency in S-SPRIGHT's shared memory processing, and is slightly worse than D-SPRIGHT in terms of tail latency (Fig. 10(c)). The 95%iles of S-SPRIGHT, measured across all the functions, is 1.2× higher than D-SPRIGHT (more details in Table 5, Appendix-B). The additional delay for SPROXY's descriptor delivery, adds to the transient queueing and hence slightly longer tail latency. However, as we said in §3.2.2, the impact of this additional latency introduced by SPROXY is quite limited. Further, the processing time within the functions, which usually are non-trivial, will likely dwarf the extra latency introduced by SPROXY, in relative terms. Importantly, the throughput (RPS) of S-SPRIGHT is very close to D-SPRIGHT at the high concurrency levels.

4.2.2 Bypassing the impact of cold start and zero scaling. We set up an experiment with zero scaling enabled in Knative to study the impact of cold start. Without incoming requests, Knative scales functions down to zero to save resources and reduce costs. We set the 'grace period' for scaling down to zero as 30 seconds. In contrast, we keep functions in SPRIGHT 'warm' by having a minimum number of active function pods, knowing that our purely event-driven processing will not consume CPU resources when idle. We use the motion detection workload to study the impact of cold start because of the intermittent nature of such IoT traffic.

Fig. 11 (a) clearly shows the impact of cold start in Knative, with large response times that possibly render the motion detection application ineffective and severely violate SLOs. E.g., starting from 1950s, a number of motion events occur one after another (inter-arrival time of a few seconds) that are sent to the currently zero-scaled function chain. The first motion event that arrives at the gateway is queued and triggers the instantiation of the functions. Since a serverless function pod takes some time to start, subsequent requests have to be queued. The cascading effect during the cold start of the entire function chain further degrades the response time [60], resulting in a long tail latency going up to 9s. Once the function is active, Knative has a reasonably small response time when there are consecutive incoming events (*e.g.,* before the grace period terminates between 2000s and 2500s), which keeps the functions 'warm'.

In contrast, SPRIGHT shows consistently low response times over the entire workload duration since there is always an active pod to serve the request without leaving requests waiting in the queue (we can sidestep going down to zero-scale). More importantly, although SPRIGHT keeps one (or more) function warm, the event-driven nature of SPRIGHT leads to negligible CPU consumption when there is no traffic. In fact, with Knative, the higher resource usage of the queue proxy under load more than offsets any benefit of Knative's zero-scaling. E.g., in Fig. 11 (b), the spikes in the CPU usage for the queue proxy (*e.g.,* at the 1500s mark), even when handling small traffic, is quite wasteful and is eminently avoidable with SPRIGHT's event-driven design.

Since the 'Parking: image detection & charging' workload has a distinct periodic arrival pattern (*e.g.,* monitoring and billing every 4 minutes), we configure a 'pre-warm' phase for Knative functions 20 seconds before the next burst is scheduled to arrive. 'Pre-warming' helps avoid the penalty of the cold start delay of serverless functions while trading off a small amount of the resource savings of shutting down the pods in serverless computing with zero-scaling [63]. However, as observed in Fig. 12 (b), the CPU usage for each function instantiation at the pre-warming stage in fact exceeds the CPU usage consumed by request processing (*i.e.,* observe the CPU usage spike for the pre-warming and the function execution 20 seconds later). Thus, while zero-scaling reduces CPU usage if the idle period is long, a CPU cost for frequent creation/destruction of functions must be considered. Knative also is quite inefficient for scaling functions down to zero. When there is no traffic for a grace period of 30s (*e.g.,* 270s to 300s in Fig. 12 (b)), Knative begins scaling down the functions to zero. But, functions remain in a 'terminating' state until 380s without being really terminated or releasing CPU resources. Thus, the scaling down process lasts as long as 80s, during which all the Knative queue proxies and functions are consuming CPU resources, which is unnecessary and wasteful.

For comparison, S-SPRIGHT consumes only a small amount of CPU throughout the entire period, in fact with slightly lower (about 16%) response time (both average and 95%, Fig. 12 (a)). Overall, S-SPRIGHT saves up to 41% CPU cycles in this 700s experiment

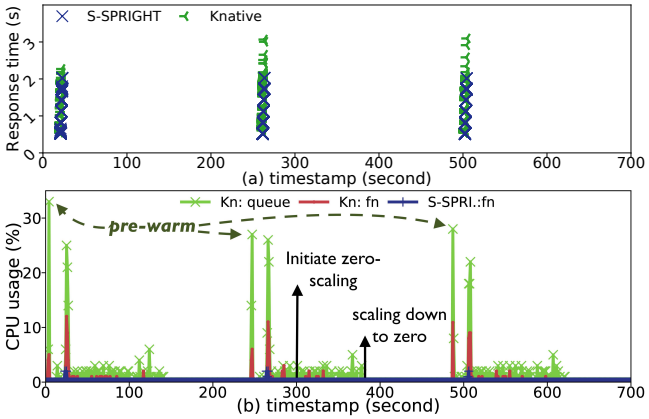


Figure 12: Parking image detection & charging: (a) Time series of response time of function chains; (b) Time series of aggregate CPU for function chains, queue proxy (Knative).

without resorting to zero-scaling, almost doubling system capacity compared to Knative.

5 RELATED WORK

In recent years, a number of serverless platforms have been launched, *e.g.,* AWS Lambda [5], IBM Cloud Functions [48], Apache OpenWhisk [4], OpenFaaS [24], Knative [14], *etc*, to support cloud-resident applications. Work on understanding the performance impact of commercial or open-source serverless platforms [35, 53] has guided us on the design of SPRIGHT. Li *et al.* [53] showed that the overhead of the ingress gateway reduced the throughput by 13%, compared to the performance of function invocation using the 'direct call' mode (*i.e.,* the client directly invokes the function instance, bypassing the ingress gateway). Priscilla *et al.* [35] studied the suitability of different serverless function startup modes (*i.e.,* cold and warm) for supporting IoT applications, indicating that cold start can have significant resource-saving benefits but can impact response time. This prompts us to examine the resource consumption and overheads of each component carefully.

Several past works have examined the inefficiency and overheads that exist in Linux networking, including data copies and context switching [39, 50, 52, 58]. The overhead of protocol processing [61] and serialization-deserialization [49, 71] directly impact networking performance, which applies to the container-based serverless function, including function chains. A variety of optimizations have been proposed to improve the network performance for different application scenarios, which can be complementary to current Linux networking (*e.g.,* XDP [45], AF_XDP in OVS [69]) or bypass kernel-based networking (*e.g.,* NetVM for NFV [46]). Our work combines the advantages of kernel-bypass zero-copy networking where essential for serverless function chains, and leveraging eBPF-based event-driven processing.

Multiple proposals optimize different aspects of serverless frameworks, *e.g.,* runtime overhead reduction [31, 32, 44, 59], intelligent resource provisioning, and traffic management [57, 64]. Further, [60], [37], [65] aim to optimize resource allocation and deployment of serverless functions on the basis of a chain, which improves the efficiency and flexibility of building microservices using serverless function chaining. However, they do not focus on optimizing the dataplane, which as we show has a significant impact.

'Cold start' in serverless: The cold start latency of serverless functions detracts from their being an ideal framework for building microservices. [42] proposes a startup latency optimization specifically for Kubernetes-based environments by placing pods on nodes that have container image dependencies locally to avoid the latency of pulling images. However, their 95%ile startup latency after optimization is still around 23s, severely impacting the QoS. In addition, startup (either cold start or pre-warm [63]) adds additional costs, as we have observed, making optimizations built around cold start less desirable. A policy of 'keep-warm' of pods has been an alternative to mitigate the cold start latency in serverless [54]. They can achieve an 85% improvement of the 99%ile latency. Although [54] considerably improves the SLOs, it is built on Knative with heavyweight components (*e.g.,* queue proxy), resulting in excessive resource usage. Fuerst *et al.* [43] consider greedy-dual caching to determine which functions should be kept as warm. By factoring in several key indicators of a function, *e.g.,* memory footprint, invocation frequency *etc.*, they can prioritize functions to be kept warm, thus limiting memory consumption to keep a minimum number of warm functions and achieve SLOs. Since SPRIGHT primarily contributes to controlling the CPU usage, [43] can be a good complement to SPRIGHT to reduce memory utilization.

6 CONCLUSIONS

SPRIGHT demonstrated the effectiveness of event-driven capability for reducing resource usage in serverless cloud environments. With extensive use of eBPF-based event-driven capability in conjunction with high-performance shared memory processing, SPRIGHT achieves up to 5× throughput improvement, 53× latency reduction, and 27× CPU usage savings compared to Knative when serving a complex web workload. Compared to an environment using DPDK for providing shared memory and zero-copy delivery, SPRIGHT achieves competitive throughput and latency while consuming 11× fewer CPU resources. Additionally, for intermittent request arrivals typical of IoT applications, SPRIGHT still improves the average latency by 16% while reducing CPU cycles by 41%, when compared to Knative using 'pre-warmed' functions. This makes it feasible for SPRIGHT to support several 'warm' functions with minimum overhead (since CPU usage is load-proportional), sidestepping the 'cold-start' latency problem. Across several typical serverless workloads, SPRIGHT shows higher dataplane performance while avoiding the inefficiencies of current open-source serverless environments, thus getting us closer to meeting the promise of serverless computing.

To provide isolation between serverless functions with the use of SPRIGHT's shared memory processing, SPRIGHT supports function-chain-level separation by restricting access of a private shared memory pool to only trusted functions of that chain. SPRIGHT further provides traffic isolation by enabling message filtering in the SPROXY of each function. SPRIGHT is publicly available at `https://github.com/ucr-serverless/spright.git`

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank the US NSF for their generous support through grants CRI-1823270 and CSR-1763929. We thank our shepherd, Prof. Marco Canini for his extremely thorough, helpful guidance, and the anonymous reviewers for their valuable suggestions and comments.

REFERENCES

[1] 2021. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. [online].

[2] 2021. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/. [online].

[3] 2021. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. [online].

[4] 2022. Apache OpenWhisk. https://openwhisk.apache.org/. [online].

[5] 2022. AWS Lambda. https://aws.amazon.com/lambda/. [online].

[6] 2022. AWS Serverless API. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-api.html. [online].

[7] 2022. BPF-HELPERS - list of eBPF helper functions. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. [online].

[8] 2022. Chaining OpenFaaS functions. https://ericstoekl.github.io/faas/developer/chaining_functions/. [online].

[9] 2022. CloudEvents Spec. https://github.com/cloudevents/spec. [online].

[10] 2022. Dynamically Loaded (DL) Libraries. https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html. [online].

[11] 2022. eBPF XDP: The Basics and a Quick Tutorial. https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/. [online].

[12] 2022. Istio Architecture. https://istio.io/latest/docs/ops/deployment/architecture/. [online].

[13] 2022. Istio Traffic Management. https://istio.io/latest/docs/concepts/traffic-management/. [online].

[14] 2022. Knative. https://knative.dev. [online].

[15] 2022. Knative Eventing. https://knative.dev/docs/eventing/. [online].

[16] 2022. Knative Serving. https://knative.dev/docs/serving/. [online].

[17] 2022. Knative Serving Autoscaling System - Activator. https://github.com/knative/serving/blob/main/docs/scaling/SYSTEM.md#activator. [online].

[18] 2022. Locust: An open source load testing tool. https://locust.io/. [online].

[19] 2022. MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. [online].

[20] 2022. Multi-process Support of DPDK. https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html. [online].

[21] 2022. NGINX. https://www.nginx.com/. [online].

[22] 2022. of-watchdog. https://github.com/openfaas/of-watchdog. [online].

[23] 2022. Online Boutique by Google. https://github.com/GoogleCloudPlatform/microservices-demo. [online].

[24] 2022. OpenFaaS. https://www.openfaas.com/. [online].

[25] 2022. OpenFaaS API Gateway / Portal. https://docs.openfaas.com/architecture/gateway/. [online].

[26] 2022. OpenFaaS Triggers. https://docs.openfaas.com/reference/triggers/. [online].

[27] 2022. OpenWhisk - Creating action sequences. https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences. [online].

[28] 2022. OpenWhisk Composer. https://github.com/apache/openwhisk-composer. [online].

[29] 2022. Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html. [online].

[30] 2022. SPRIGHT. https://github.com/ucr-serverless/spright.git. [online].

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

[32] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[33] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, and Claudio Vairo. 2016. Car parking occupancy detection using smart camera networks and Deep Learning. In *2016 IEEE Symposium on Computers and Communication (ISCC)*. 1212–1217. https://doi.org/10.1109/ISCC.2016.7543901

[34] Apache Software Foundation. 2022. APACHE KAFKA. https://kafka.apache.org/. [online].

[35] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. 2021. Experimental Analysis of the Application of Serverless Computing to IoT Platforms. *Sensors* 21, 3 (2021). https://doi.org/10.3390/s21030928

[36] Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, and Massimo Tumolo. 2018. Toward an eBPF-based clone of iptables. *Netdev'18* (2018).

[37] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. *Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3472883.3486992

[38] Carsten Bormann, Angelo P Castellani, and Zach Shelby. 2012. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing* 16, 2 (2012), 62–67.

[39] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[40] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. ECML: Improving Efficiency of Machine Learning in Edge Clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–6. https://doi.org/10.1109/CloudNet51028.2020.9335804

[41] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[42] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and Efficient Container Startup at the Edge via Dependency Scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association. https://www.usenix.org/conference/hotedge20/presentation/fu

[43] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

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

[45] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[46] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 445–458. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/hwang

[47] IBM. 2022. Creating serverless REST APIs. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-apigateway. [online].

[48] IBM. 2022. IBM Cloud Functions. https://cloud.ibm.com/functions/. [online].

[49] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[50] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing Packet Processing in Container Overlay Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 261–276. https://doi.org/10.1145/3447786.3456241

[51] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–8. https://doi.org/10.1109/CloudNet51028.2020.9335808

[52] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the Cost of Context Switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (San Diego, California) *(ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, 2–es. https://doi.org/10.1145/1281700.1281702

[53] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3366623.3368139

[54] Ping-Min Lin and Alex Glikson. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. arXiv:1903.12221 [cs.DC]

[55] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a Faster and Scalable Iptables. *SIGCOMM Comput. Commun. Rev.* 49, 3 (nov 2019), 2–17. https://doi.org/10.1145/3371927.3371929

[56] Microsoft. 2022. Azure - Function chaining in Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-sequence?tabs=csharp. [online].

[57] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2021. *Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds.* Association for Computing Machinery, New York, NY, USA, 168–181. https://doi.org/10.1145/3472883.3487014
[58] Jeffrey C Mogul and KK Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252.
[59] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes
[60] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network Based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21).* Association for Computing Machinery, New York, NY, USA, 154–167. https://doi.org/10.1145/3485983.3494866
[61] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2021. Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 656–671. https://doi.org/10.1109/TNSM.2020.3047545
[62] Red Hat, Inc. 2022. Understanding the eBPF networking features in RHEL. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel_configuring-and-managing-networking. [ONLINE].
[63] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad
[64] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. *Atoll: A Scalable Low-Latency Serverless Platform.* Association for Computing Machinery, New York, NY, USA, 138–152. https://doi.org/10.1145/3472883.3486981
[65] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20).* Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306
[66] The Linux Foundation. 2022. eBPF. https://ebpf.io/. [ONLINE].
[67] Tigera, Inc. 2022. About eBPF. https://projectcalico.docs.tigera.io/about/about-ebpf. [ONLINE].
[68] Tigera, Inc. 2022. Project Calico. https://www.tigera.io/project-calico/. [ONLINE].
[69] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open VSwitch Dataplane Ten Years Later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21).* Association for Computing Machinery, New York, NY, USA, 245–257. https://doi.org/10.1145/3452296.3472914
[70] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. https://doi.org/10.1145/3371038
[71] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21).* Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3458336.3465283
[72] Christopher R. Wren, Yuri A. Ivanov, Darren Leigh, and Jonathan Westhues. 2007. The MERL Motion Detector Dataset. In *Proceedings of the 2007 Workshop on Massive Datasets* (Nagoya, Japan) *(MD '07).* Association for Computing Machinery, New York, NY, USA, 10–14. https://doi.org/10.1145/1352922.1352926
[73] Muneer Bani Yassein, Mohammed Q Shatnawi, Shadi Aljwarneh, and Razan Al-Hatmi. 2017. Internet of Things: Survey and open issues of MQTT protocol. In *2017 international conference on engineering & MIS (ICEMIS).* IEEE, 1–6.
[74] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization.* ACM.
[75] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. 2021. Antelope: A Framework for Dynamic Selection of Congestion Control Algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP).* 1–11.

Appendices are supporting material that has not been peer-reviewed.

A PACKET DATA FLOW IN SPRIGHT

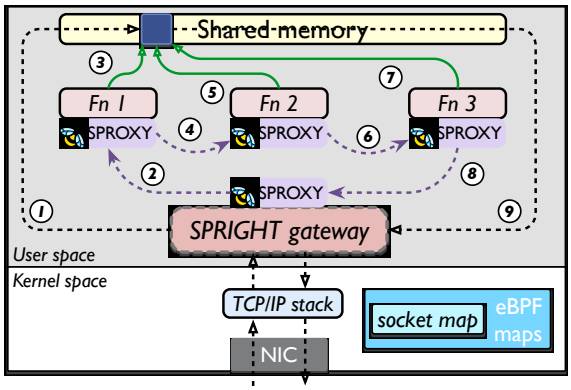


Figure 13: Packet data flow in S-SPRIGHT (SPROXY). NOTE: The SPROXY is an in-kernel component. We put it in the userspace for simplicity.

We describe the zero-copy packet delivery in S-SPRIGHT and D-SPRIGHT. Fig. 13 shows the packet flow in a sequential chain with three functions using S-SPRIGHT. Each function attaches a SPROXY containing the eBPF's Socket Message program at startup. After protocol processing, ①) and receipt of the payload from the kernel protocol stack, the SPRIGHT gateway writes the payload to shared memory. ②) The SPRIGHT gateway then generates a packet descriptor, which contains the location of the payload in shared memory, and executes the `send()` system call to send the packet descriptor on its socket interface. The Socket Message program at the socket interface intercepts the packet descriptor and parses it to retrieve the ID of the next function (*i.e.,* of Fn-1 in Fig. 13). The Socket Message program then looks up the eBPF socket map, using Fn-1's ID as the key to find the file descriptor of Fn-1's socket interface. The Socket Message program invokes the `bpf_msg_redirect_map()` eBPF call to transfer the packet descriptor to Fn-1 without traversing the kernel protocol stack. ③) Once Fn-1 receives the packet descriptor, it can use the location information contained in the packet descriptor to access the payload in shared memory. The same procedure is followed for the subsequent packet data flow (④ to ⑦) within the function chain. ⑧) After processing within the function chain is complete, the packet descriptor is sent back to the SPRIGHT gateway. ⑨) The SPRIGHT gateway fetches the payload from the shared memory and sends it out through the kernel protocol stack for processing and transmission of the response.

Fig. 14 shows the packet flow when passing packet descriptors between functions using DPDK's RTE rings. It follows the same procedure as the S-SPRIGHT, except for the way it is used to exchange packet descriptors. With the D-SPRIGHT setup, functions and the SPRIGHT gateway are assigned an RTE ring at startup, using `rte_ring_create()`. We configure the ring to run in a multi-producer/multi-consumer mode by specify the "flags" parameter [29]

of `rte_ring_create()` as 0. The D-SPRIGHT supports direct routing from one function to another by looking up the routing table, which is configured by the shared memory manager during the initialization of the function chain. The source (SPRIGHT gateway or function) looks up the routing table in the shared memory to find its destination. It then calls `rte_ring_enqueue()` to move the packet descriptor to the destination's RTE ring. The destination polls the RTE ring (using `rte_ring_dequeue()`) to retrieve the packet descriptor and then accesses the payload in shared memory.

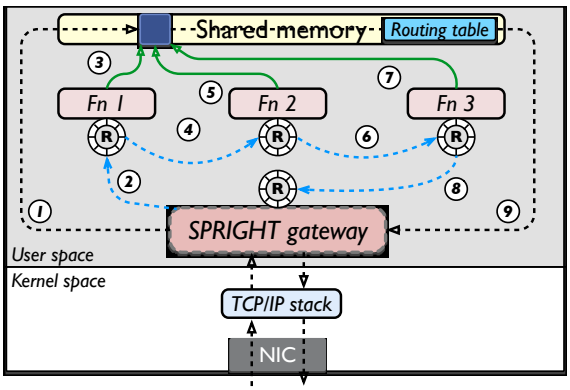


Figure 14: Packet flow in D-SPRIGHT (DPDK's RTE rings).

B ADDITIONAL EXPERIMENT DETAILS

Table 3: Sequence of different function chains in online boutique. ①: Frontend service; ②: Currency Service; ③: Product Catalog Service; ④: Cart Service; ⑤: Recommendation Service; ⑥: Shipping Service; ⑦: Checkout Service; ⑧: Payment Service; ⑨: Email Service; ⑩: Ad Service.

Index	API call	Call sequence in the chain
Ch-1	GET "/"	1,2,1,3,1,4,1,2,1,10,1
Ch-2	POST "/setCurrency"	1
Ch-3	GET "/product/$ID"	1,3,1,2,1,4,1,2,1,5,1,4,1,10,1
Ch-4	GET "/cart"	1,2,1,4,1,5,1,6,1,2,1,3,1,2,1
Ch-5	POST "/cart"	1,3,1,4,1
Ch-6	POST "/cart/checkout"	1,7,4,7,3,7,2,7,6,7,2,7,8, 7,6,7,4,7,9,7,1,5,1,2,1

Table 4: CPU service time and sequence of different function chains of Parking: Image Detection & Charging workload. ①: Plate detection, ②: Plate search, ③: Plate index, ④: Charging, ⑤: Persist metadata.

	Sequence of the function chain
Ch-1	① (435ms), ② (20ms), ③ (1ms), ⑤ (10ms), ④ (50ms)
Ch-2	① (435ms), ② (20ms), ④ (50ms)

Table 3, 4, and 5 provide additional details of experiments in §4. Table 3 describes the sequence of different function chains in online boutique workload. Based on the REST API call made by the load generator, difference functions are invoked following a certain sequence (*i.e.,* Ch-1 to Ch-6). Table 4 defines the CPU service time of each function and the sequence of functions being called in the "Parking: Image Detection & Charging" workload. The "Parking: Image Detection & Charging" workload has two function chains, *i.e.,* Ch-1 and Ch-2 in Table 4. Table 5 compares the latency between different alternatives in online boutique experiment at different concurrency levels (5K and 25K).

Table 5: Latency comparison between S-SPRIGHT, D-SPRIGHT, Knative and gRPC mode at 5K and 25K concurrency. Note: latency is measured across all the functions of the online boutique service.

	Latency @ 5K (ms)			Latency @ 25K (ms)		
	95%	99%	Mean	95%	99%	Mean
Knative	**693**	**965**	**382**	-	-	-
gRPC	**141**	**199**	**45.6**	-	-	-
D-SPRIGHT	**11.1**	**45.1**	**5.8**	**80.8**	**144**	**17.7**
S-SPRIGHT	**13.4**	**49.2**	**7.2**	**96.1**	**159**	**20.0**

C ARTIFACT APPENDIX

Abstract

We have made the artifact of SPRIGHT publicly available. It contains the environment setup and the provides the instructions to conduct the set of experiments to reproduce the results in §4.

Scope

The goal of our submitting the artifact is to make the software of SPRIGHT available as open source and allows the reproduction of all the results and claims made in this paper. One can run SPRIGHT and potentially extend the functionality of SPRIGHT based on the source code and documentation provided.

Contents

The artifact consists of the full source code, and all necessary scripts to run SPRIGHT as well as the experiments described in §4.

Hosting

SPRIGHT is publicly available at https://github.com/ucr-serverless/spright.git. The artifact documentation can be found in the next branch (commit 98434fd) in "`ARTIFACTS.md`".

Requirements

Hardware Dependencies: This artifact was tested on NSF Cloudlab [41] using c220g5 nodes. However, it can be generally hosted on any DPDK-compatible machines (required for D-SPRIGHT). For those who only want to run S-SPRIGHT, any commercial off-the-shelf machines would be adequate.

Software Dependencies: This artifact requires Ubuntu 20.04 with Linux kernel version 5.16, Knative v0.22.0, Kubernetes v1.19.0, DPDK v21.11, and libbpf v0.6.0.

More details can be found in the artifact documentation.