SPRIGHT: High-Performance eBPF-Based Event-Driven, Shared-Memory Processing for Serverless Computing

Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan, *Life Fellow, IEEE, Fellow, ACM*

***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'.**

***Index Terms*— Serverless, eBPF, event-driven, function chain, shared memory.**

I. 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 of serverless computing 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 (QoS) 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. This can be extremely resource-intensive, especially when deployed in a limited capacity environment, *e.g.,* edge cloud [1]. 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 §II) for a **single** request. Moreover, since most serverless frameworks primarily focus on HTTP/REST API [2], [3], additional protocol adaptation is required for specialized use cases, *e.g.,* IoT (Internet-of-Things) with MQTT [4], CoAP [5]. The current design runs protocol adaptation as an individual component, resulting in substantial resource consumption [6]. 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 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

Manuscript received 24 December 2022; revised 24 November 2023; accepted 20 January 2024; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Liu. Date of publication 22 February 2024; date of current version 18 June 2024. This work was supported in part by the U.S. NSF under Grant CRI-1823270, Grant CNS-1818971, and Grant CSR-1763929. *(Corresponding author: Shixiong Qi.)*

Shixiong Qi and K. K. Ramakrishnan are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521 USA (e-mail: sqi009@ucr.edu).

Leslie Monis, Ziteng Zeng, and Ian-Chin Wang were with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521 USA.

Digital Object Identifier 10.1109/TNET.2024.3366561

such as Apache Kafka [7], to coordinate communication between functions, and a load balancer like Istio [8]). 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 level objectives (SLOs).

In this paper, we design SPRIGHT,[1] 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 on Knative [10], 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 design event-driven proxies (*i.e.,* EPROXY and SPROXY) using the eBPF (extended Berkeley Packet Filter [11]), that effectively replace the heavyweight sidecar proxy. We support the functions of metrics collection *etc.*, with much lower CPU consumption. We further utilize the XDP/TC hooks 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.

(3) We implement zero-copy message delivery within a serverless function chain by using event-driven 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-driven shared memory communication helps reduce CPU usage and alleviate penalties when keeping the function chain warm.

(4) SPRIGHT fully exploits the reconfigurability of the eBPF maps to support Direct Function Routing (DFR) within the serverless function chains, which eliminates the dependency on an intermediate routing component (*e.g.,* the message broker in Knative [12]) for function chaining and avoids duplicate processing in the dataplane.

(5) We implement the 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.

(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 resource usage.

[1]This work was first published in the ACM SIGCOMM 2022 [9]. It has been extended here with additional design details and experimental results.

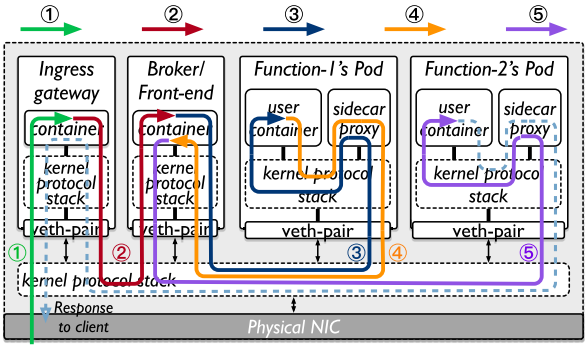


Fig. 1. Networking processing involved in a typical serverless function chain.

II. 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 [12], [13], [14], [15], [16] 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 is transferred from the message broker/front-end proxy 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 are deployed on the same node, with the overhead on the external client-side excluded. We use an NGINX [17]

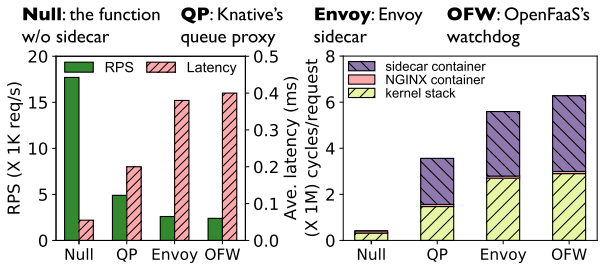


Fig. 2. Performance and overhead breakdown of different sidecar proxies.

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 I. 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: Individual, constantly-running heavy-weight sidecar. 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 [18], Queue proxy from Knative [19], and the OF-watchdog from OpenFaaS [20]. We use these sidecar proxies to work with NGINX [17] 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* [21] 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.

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.

Takeaway#2: Excessive data copies, context switches, and interrupts introduced by kernel-based networking. The existing Knative framework uses kernel-based networking to construct the dataplane for a serverless function chain, which inevitably introduces a number of overheads (data copies, context switches, and interrupts) caused by the kernel-userspace boundary crossing. Looking at the network processing from

TABLE I
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	0	1	1	2	2	2	6	7
# of deserialization	1	1	2	2	2	2	6	8

step ① to step ⑤ in Fig. 1, each request results in 15 data copies, 15 context switches, and 25 interrupts throughout the entire data pipeline. In particular, most of the overhead (80%) comes from networking within the function chain (from ③ to ⑤). The excessive overhead adds up as more messages are exchanged between functions, which have to be handled by the kernel. This can greatly impact the dataplane performance of a serverless function chain.

Takeaway#3: Unnecessary serialization & deserialization. HTTP/REST API requires 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) [22], [23]. 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 I, current designs further exacerbate this overhead with an excessive number of protocol stack traversals, which we describe next.

Takeaway#4: Excessive, duplicate processing. Current approaches for serverless function chains rely on the composition of existing networking components to support asynchronous and reliable message exchange between functions. Traffic within the chain has to go through the message broker/front-end proxy, each time having to cross the kernel-user space boundary. This also leads to duplicate network protocol processing, adding to the overhead. As seen in Table I, networking *within* the function chain in Knative accounts for 75% of the total protocol processing overhead. Protocol processing tasks, including checksum calculation in software and complex iptables processing,[2] contribute to latency and results in poor scaling (especially as the number of iptables rules increases) [25]. Furthermore, many of the other dataplane overheads (*e.g.,* data copies, context switches, interrupts, and serialization & deserialization) are also amplified, as the chain becomes more complex, resulting in very poor scaling.

Summary: The expected benefit of serverless computing was to overcome the inefficiencies of 'serverful' computing through event-driven execution, which helps use resources strictly on-demand and be proportional to the load. 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 discour-

[2]In [24] we showed 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.

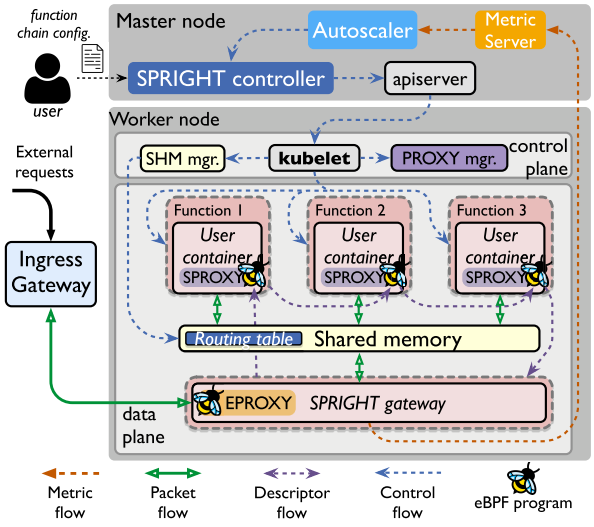


Fig. 3. The overall architecture of SPRIGHT.

aging 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.

III. System Design of SPRIGHT

We now provide an overall view of the SPRIGHT architecture, justifying the design of each component and discussing their benefits for serverless environments. We then discuss each part in detail, starting with lightweight event-driven processing (§III-B), the shared memory processing for communication within serverless function chains (§III-C), and direct function routing (§III-D). We also discuss function startup (§III-E), protocol adaptation (§III-F), security domain (§III-G), and vertical scaling of SPRIGHT gateway (§III-H).

A. Overview of SPRIGHT

For this work, we start with open-source Knative as the base platform [10]. Using an innovative combination of event-driven processing and shared memory, we achieve high performance while being resource-efficient and providing the flexibility to build microservices using serverless function chaining. Fig. 3 shows the overall architecture of SPRIGHT. 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 [11]. 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 [26] and low-overhead metric collection [27]. 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 strictly load-proportional resource usage, a highly desirable toolbox for serverless environments.

SPRIGHT's dataplane: SPRIGHT improves the dataplane of serverless computing by leveraging eBPF-based event-driven processing and shared memory communication, which helps us avoid the use of individual, constantly-running sidecars (Takeaway#1 in §II) and reduces a number of data movement related overheads within the function chain (Takeaway#2 and #3 in §II). SPRIGHT uses Direct Function Routing (DFR) to forward requests directly from one function to the next. This eliminates the need for an intermediate routing component and avoids unnecessary, duplicate processing overheads (Takeaway#4 in §II). These dataplane optimizations make the request handling in SPRIGHT strictly load-proportional and achieve superior performance compared to existing serverless platforms (see evaluation in §IV).

- **Event-driven processing:** We design lightweight, event-driven proxies (EPROXY and SPROXY in Fig. 3) that use eBPF to construct the service mesh instead of a continuously-running sidecar proxy associated with each function instance, as is used by Knative. Thus, we reduce a significant amount of the processing overhead (§III-B1).
  To accelerate the data path to/from the function chain and the ingress gateway which is outside the function chain, we utilize XDP/TC hooks [28] in eBPF (§III-B2). An XDP/TC hook processes packets at the early stage of the kernel receive (RX) path before packets enter the kernel iptables [25], resulting in substantial dataplane performance improvement without the resource consumption of a dedicated sidecar proxy that uses the kernel protocol stack. In addition, protocol adaptation is often required to interface between application layer protocols, such as MQTT and CoAP for IoT, to the HTTP/REST API supported by serverless frameworks (addressed in §III-F).
- **Shared memory communication:** For inter-function communication within the chain, SPRIGHT takes advantage of shared memory that avoids a number of overheads associated with data movement, including protocol processing, serialization/deserialization, memory-memory copies, *etc.* For every incoming request from external clients, a SPRIGHT gateway performs the one-time, consolidated protocol processing for the function chain (§III-C1). SPRIGHT considers event-driven `SKMSG`, which is a socket-related eBPF program type [28], to construct the zero-copy I/O (*i.e.,* descriptor delivery) between functions (see §III-C3).
- **Direct function routing (DFR):** To eliminate the impact of having an intermediate routing component (message broker) within the function chain, we design Direct Function Routing (DFR). DFR leverages the configurability provided by eBPF maps and allows for the dynamic update of routing rules while exploiting shared memory to pass data directly between the functions within the chain (§III-D).

Although these dataplane optimizations are built around the Knative, our concepts and methodology can also be broadly

applied to other serverless platforms. In addition to dataplane optimizations, SPRIGHT incorporates security domains to restrict unauthorized access between different chains, by creating a private shared memory pool for each chain and applying message filtering for inter-function communication (§III-G).

SPRIGHT's control plane: We introduce a SPRIGHT controller (Fig. 3) to coordinate the control plane for SPRIGHT function chains. The SPRIGHT controller runs as a cluster-wide control plane component in the master node, working with serverless orchestration engines, *e.g.,* Knative and Mu [1], and their associated control plane components (*e.g.,* autoscaler, placement engine) to determine the scale and placement of the function chain at the appropriate worker node.

SPRIGHT adopts Kubernetes to manage the lifecycle of function pods (*e.g.,* creation, termination). It cooperates with the *kubelet*, which is a pod management process in the Kubernetes control plane that runs on each worker node to manage the lifecycle of the pods. We also use *kubelet* for function health checks instead of depending on the sidecar (details in Appendix B). Given a function chain creation request from the SPRIGHT controller, the *kubelet* starts up functions in the chain based on the user configuration, working in conjunction with the shared memory manager ('SHM mgr.' in Fig. 3) and PROXY manager to set up the dataplane for the function chain (details in §III-B1). Each worker node has a shared memory manager and a PROXY manager in the control plane, both running as separate Kubernetes pods. To route external requests to the SPRIGHT gateway of each function chain, we use a cluster-wide Ingress Gateway to distribute the traffic.

B. Event-Driven Processing

1) eBPF-Based Event-Driven Proxy (EPROXY/SPROXY): The sidecar proxy in a serverless environment, *e.g.,* the queue proxy in Knative, 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 sidecar 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, as we described earlier. We overcome them with our lightweight, event-driven eBPF-based EPROXY & SPROXY, which replace the sidecar proxy.

The EPROXY is composed of a set of eBPF programs executed at the veth of the SPRIGHT gateway (Fig. 4), using XDP and TC hooks [28]. SPRIGHT uses EPROXY to perform L3 metrics collection and dataplane acceleration (§III-B2). The SPROXY runs as a set of socket-related eBPF programs (SK_MSG [28]) at the socket interface of the SPRIGHT gateway/function pods (Fig. 4). The 'SK_MSG' program supports modification of messages that pass through the attached socket as well as message redirection between sockets (with the help of eBPF's sockmap [28] to provide routing rules), which is an ideal capability to exchange small messages such as packet descriptors in supporting shared memory communication (§III-C). However, the 'SK_MSG' program only works on the TX path of the socket [28]. We use SPROXY for L7

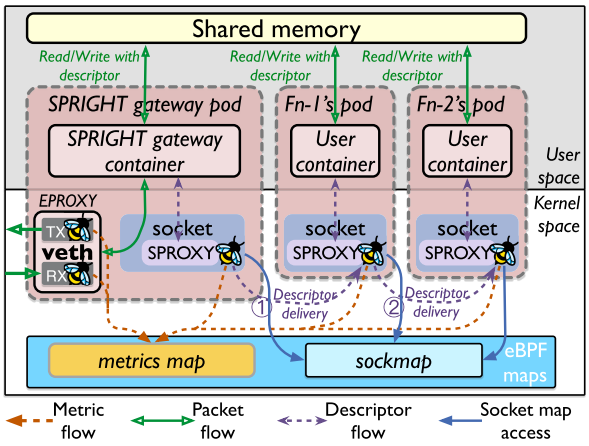


Fig. 4. Event-driven EPROXY & SPROXY, shared memory, and DFR within a chain: (①) the SPRIGHT gateway invokes the head function of chain; (②) the head function calls the next function bypassing the SPRIGHT gateway.

metrics collection, packet descriptor exchange (§III-C3), routing (§III-D), and security (§III-G). More details of the metrics collection can be found in Appendix B.

The goal of the event-driven proxy is to achieve functionality comparable to that of the sidecar proxy, but with lower overhead. Since the event-driven proxy is 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. We do not need the queueing capability in the event-driven proxy 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 sidecar proxy. But, eliminating the additional queuing stage helps reduce request delays.

Initialization of EPROXY & SPROXY: We dedicate a PROXY manager (Fig. 3) on each worker node for attaching EPROXY/SPROXY to the SPRIGHT gateway and/or function pods. The PROXY manager is created during the startup of the worker node's control plane, during which it loads the EPROXY/SPROXY eBPF programs into the kernel (via the *bpf()* syscall) and creates required eBPF maps, including the metrics map and socket map (*sockmap*). The `SKMSG` program is then attached to the sockmap. The sockmap automatically attaches the `SKMSG` program to the function pod's socket interface, once a function's socket interface is registered into it by the sockmap writer in the PROXY manager. The initialization procedure of the EPROXY starts as soon as the SPRIGHT gateway is ready (*i.e.,* when SPRIGHT gateway passes the health check from *kubelet*). *kubelet* instructs the PROXY manager to attach the EPROXY programs to the XDP and TC hooks at the veth of SPRIGHT gateway.

Both the SPRIGHT gateway and functions follow the same procedure to attach to the SPROXY. We use a function as an example to explain the initialization procedure of SPROXY, as depicted in Fig. 5. During startup, the function creates a socket interface for attaching to the SPROXY. The function initiates a connection on its socket to the dummy socket in the PROXY manager. The dummy socket stays active to maintain

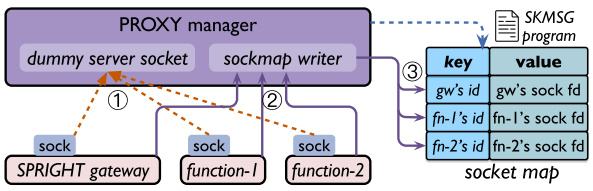


Fig. 5. Initialization of SPROXY: SKMSG program, sockmap.

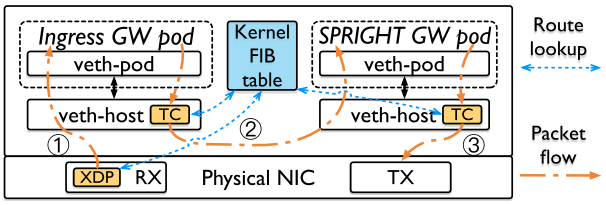


Fig. 6. Dataplane acceleration using eBPF XDP/TC hooks.

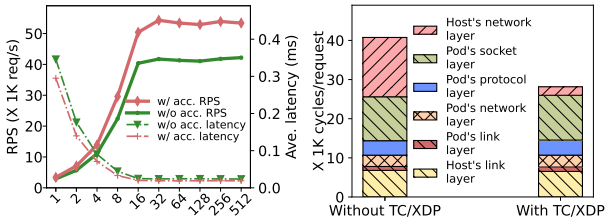


Fig. 7. (Left) Performance impact of TC/XDP redirect: RPS and latency; (Right) CPU overhead breakdown of receiver side kernel stacks: with TC/XDP acceleration (w/ acc.) & without TC/XDP acceleration (w/o acc.).

the connection with function's socket (①). After the connection is established, the function sends the socket's file descriptor and the function ID to the PROXY manager (②). To attach the SKMSG program to function's socket, the PROXY manager uses its sockmap writer to update the function ID (key) and the socket's file descriptor (value) to the sockmap (③), using '*bpf_map_update_elem()*' helper. The SKMSG program is then automatically attached to the socket by the kernel.

2) eBPF-Based Dataplane Acceleration for External Communication: We exploit EPROXY's XDP/TC hooks to accelerate communication by the function chain in SPRIGHT to external components (*e.g.,* cluster-wide Ingress Gateway). We develop an eBPF forwarding program (in EPROXY) 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*[3]) and the physical NIC, as shown in Fig. 6. 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 [29]. 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 of the received packet (using *bpf_fib_lookup()* helper), 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. 6). 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. 6). If the destination function pod is on another node, the TC program redirects the packet to the NIC (③ in Fig. 6).

Improvement with dataplane acceleration based on EPROXY: To estimate the benefit of eBPF's XDP/TC features, we evaluate the networking performance of SPRIGHT when the XDP/TC acceleration is enabled in EPROXY. We use Apache Benchmark [30] to simulate the traffic to/from the cluster-wide ingress gateway running on a different node than the SPRIGHT gateway. We further break down the CPU cycles expended for the kernel stack processing, to accurately quantify the CPU cycles saved by the XDP/TC acceleration.

Fig. 7 (Left) compares the RPS and response latency performance of SPRIGHT when the XDP/TC acceleration is enabled or disabled. With a concurrency of 32, SPRIGHT with XDP/TC acceleration has a 1.3× improvement in RPS compared to SPRIGHT without XDP/TC acceleration. Since XDP/TC acceleration transfers raw packets between network devices (*i.e.,* veth and NIC), the overhead spent in kernel iptables can be avoided, which in turn improves throughput. The response latency of SPRIGHT with XDP/TC acceleration is 19μs with a concurrency of 32, compared to 24μs for SPRIGHT without XDP/TC acceleration. The RPS and response latency improvements remain even as the concurrency increases, allowing SPRIGHT to maintain a peak RPS of 53K when XDP/TC acceleration is enabled. We then break down the CPU cycles spent on processing a request (at concurrency 32), including in the host's kernel stack and the pod's network stack, as shown in Fig. 7 (Right). The client-side overhead is excluded. Bypassing the host's kernel networking stack and associated iptables processing, XDP/TC's acceleration saves 1.45× total CPU cycles spent on each request. This option, however, means the loss of full-featured iptables network policy support, which may not be required for certain cases (*e.g.,* when users require higher dataplane performance, with the infrastructure provider having only a subset of the kernel iptables functionality [31]).

C. Shared Memory Communication Within Function Chains

To support shared memory communication within a serverless function chain, three key building blocks are required: (1) **Protocol processing.** The incoming message to a SPRIGHT function chain requires protocol processing before being moved to shared memory for communication within the function chain. Similar protocol processing to construct outgoing

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

messages is needed. (2) **Shared memory pool.** A private shared memory pool that is initialized for the function chain and is attached to functions during their startup is needed. The message payload is kept in shared memory without being moved between functions. (3) **Zero-copy I/O within the function chain.** 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.

1) Consolidated Protocol Processing: To flexibly manage traffic in and out of the function chain 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 memory between functions.

2) Shared Memory Pool: 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.

SPRIGHT takes advantage of DPDK's multi-process support [32] to create shared memory pools for function chains. At the startup of a SPRIGHT function chain, a DPDK primary process is spun up in the shared memory manager pod. The DPDK primary process has privileged permission to initialize the shared memory pool, using *rte_mempool_create()* API. Each DPDK primary process owns a unique shared data file prefix [32] – a multiprocessing-related option in DPDK. We further extend its use to isolate different memory pools [9]. 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 specific DPDK primary process in the shared memory manager pod.

Note that DPDK's multi-process shared memory is *independent* of other DPDK libs/devices such as DPDK RTE RING and Poll Mode Driver. This gives SPRIGHT the freedom to choose different implementations of zero-copy I/O to support shared memory communication within the function chain.

3) Event-Driven Zero-Copy I/O Within the Function Chain: SPRIGHT extends the use of SPROXY (Fig. 4) to implement the *event-driven* zero-copy I/O for shared memory communication within the function chain. The `SKMSG` program in SPROXY works with eBPF's sockmap to enable message redirection between the socket interfaces of function pods by communicating a packet descriptor from one function to the next in the chain. The packet descriptor used in SPRIGHT is a small 16-byte message that incurs negligible 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 sockmap to retrieve the target socket interface information (*i.e.,* the file descriptor). For the description of the zero-copy based message flow in SPRIGHT, refer to [9].

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

Another implementation option is using polling-based zero-copy I/O, as used by DPDK, which uses polling-based RTE RING [33] to pass packet descriptors. DPDK's RTE RING is implemented as a userspace shared memory queue that offers a low-latency IPC channel between independent processes (*i.e.,* function pods) because it entirely eliminates any kernel-userspace interaction (*e.g.,* context switches, interrupts) and operates at memory speeds, ensuring higher performance. DPDK's RTE RING has been extensively used to build high-performance dataplanes for cloud services [34]. However, using DPDK's RTE RING as the inter-function IPC channel requires expensive busy polling that continuously consumes CPU cycles, independent of traffic intensity.

4) Event-Driven Vs. Polling-Based Shared Memory Processing: To identify the most appropriate zero-copy I/O for shared memory processing in the context of serverless computing, we compare SPRIGHT's event-driven 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 [30] 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. 8, with low concurrency, *e.g.,* at 32, S-SPRIGHT (0.024ms) shows a slightly higher average response delay compared to D-SPRIGHT (0.02ms), but still has a much lower (almost 6×) response latency compared to Knative (0.138ms). In terms of RPS, both D-SPRIGHT (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 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

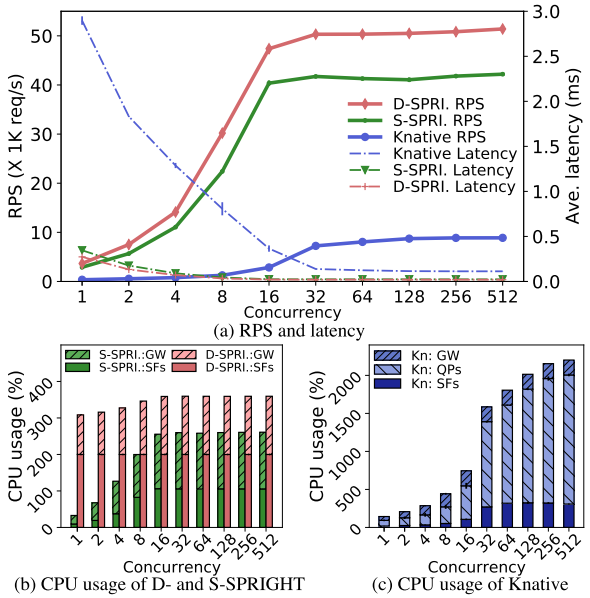


Fig. 8. Comparison between polling-based (D-SPRI.) and event-driven (S-SPRI.) shared memory processing with 1 gateway pod and 2 function pods. Kn: Knative; QPs: Sidecars; SFs: serverless functions; GW: gateway.

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 D-SPRIGHT (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. 8 (c)). The sidecar proxy consumes 70% of Knative's CPU. Even with increasing concurrency ($\geq$ 32), S-SPRIGHT has a consistent and steady saving in CPU compared to the others. Individual, constantly-running components (sidecar 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 event-driven consume *zero* CPU when there is no traffic, making it possible to keep a function pod 'warm' to overcome the 'cold start' delay (§IV-B2). Thus, event-driven shared memory processing is ideal for serverless computing, especially for function chains.

D. Direct Function Routing 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 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. To support DFR, SPRIGHT adopts a two-step routing mechanism. It uses a chain-specific, userspace routing table and an in-kernel sockmap. The userspace routing table helps determine the ID of the next function, while the in-kernel sockmap 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. For details of routing configuration and load balancing with a function chain, refer to Appendix A.

E. Function Startup in SPRIGHT

In Knative, the startup of a function pod consists of several key steps: control plane activity (*e.g.,* pod placement), container runtime initialization (*e.g.,* container image extraction, namespace creation, Cgroups configuration, file system mounting, etc), and dataplane setup (*e.g.,* route setup, veth devices creation, etc). The startup process of a SPRIGHT function pod shares several common steps with a Knative function pod creation in terms of control plane activity and container runtime initialization. However, SPRIGHT differs from Knative in setting up the dataplane because SPRIGHT uses shared memory communication. More importantly, SPRIGHT uses eBPF-based event-driven proxies. Knative, on the other hand, uses the sidecar proxy as an individual container, thus incurring additional startup latency to initialize the sidecar container. The dataplane setup of a Knative function pod is nested within the container runtime initialization and is completed by the Container Network Interface (CNI) plugin [24]. During the dataplane setup of a Knative function pod, the CNI plugin creates a veth-pair (a pod-side veth and a host-side veth) to connect the function pod to the host's network namespace, facilitating inter-pod connectivity. An IP address is assigned to the function pod and the route is configured in the host's iptables to finalize the dataplane setup [24].

SPRIGHT avoids relying on the CNI plugin to set up the dataplane of the function pod for the use of event-driven shared memory processing. The dataplane setup of a SPRIGHT function pod involves the initialization of the SPROXY (§III-B) and attachment to the shared memory pool[4] (§III-C). We compare the startup overhead of SPRIGHT function pods and Knative function pods in §IV-C.

When starting up a function chain in SPRIGHT, a SPRIGHT gateway pod is created, which involves the initialization of SPROXY and attachment to the shared memory pool. Additional dataplane setup (*e.g.,* veth-pair creation, route setup, etc) is performed by the CNI plugin to connect the SPRIGHT gateway pod with the kernel protocol stack, as the SPRIGHT gateway interacts with the kernel protocol stack to perform protocol processing and to attach the EPROXY to the veth device. The initialization of the SPRIGHT gateway pod as well as the startup of functions in the same chain can be performed in parallel to amortize the startup penalty, as we discuss in §IV-C.

[4]*Note:* The shared memory manager in SPRIGHT pre-allocates a number of shared memory objects in the shared memory pool. This avoids the shared memory creation latency during dataplane setup for SPRIGHT functions.

F. Event-Driven Protocol Adaptation

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 [4], CoAP [5]). Current designs use a separate protocol adapter (*e.g.,* Kamelet in Apache Camel-K [35]) for translation between these protocols. However, since SPRIGHT's shared memory processing directly works on payloads independent of the application layer protocols, the protocol adapter can ideally run as an internal event-driven component that is part of the SPRIGHT gateway. This achieves a much more streamlined protocol adapter design, using resources strictly on demand. Please refer to [9] for the details of the event-driven protocol adaptation design.

G. 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 to 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 abstractions to construct a security domain for each function chain: 1) a private shared memory pool for each chain; 2) inter-function (intra-domain) packet descriptor filtering with the SPROXY; 3) inter-domain access control enabled by the SPRIGHT gateway and attached EPROXY. Appendix C provides more details of SPRIGHT's security domain design, including security domain separation, intra-domain and inter-domain access control, and alternatives to in-kernel iptables by directly using SPRIGHT components.

H. Vertical Scaling of SPRIGHT Gateway

The SPRIGHT security domain design requires a dedicated SPRIGHT gateway for each domain. Dedicating CPU cores to the SPRIGHT gateway may lead to CPU wastage at light loads. Therefore, we utilize the concept of "rate proportional scheduling" [36]. It actively determines the CPU quota of the SPRIGHT gateway based on request arrival characteristics.

The SPRIGHT gateway is a networking component, serving as the entry/exit point for function chain. The processing tasks for each request in the SPRIGHT gateway involve a fixed amount of protocol processing, routing, etc., which usually has very little variability for the CPU cycle consumption. This suggests there would be a strong correlation between the CPU core usage of the SPRIGHT gateway and the incoming request rate. Fig. 15 (left) shows this correlation under various load levels (number of external clients at 5K, 12K, 20K, 25K, and 30K). Therefore, we can estimate the CPU core usage of the SPRIGHT gateway based on the request rate,[5] using simple linear regression: $C_{i,t} = \alpha \times r_{i,t}$, where $C_{i,t}$ represents the estimated CPU core usage of the SPRIGHT gateway i at time t, $r_{i,t}$ represents the Exponentially Weighted Moving Average (EWMA) of the request rate of the SPRIGHT gateway i at time t, and α is the regression coefficient. We use the EWMA of the request rate to ensure that the estimated $C_{i,t}$ is not too sensitive to short-term fluctuations. We set the EWMA coefficient to 0.8 as it yields better results in our testing. We compute α offline, as 1.13×10^{-4}. This is consistent for various concurrency levels we tested (see the `LR` line plot in Fig. 15 (left)). Given the estimated CPU core usage of SPRIGHT gateway i, we use the *Cgroups* utility to configure its CPU core quota ("`cfs_quota_us`" and "`cfs_period_us`"), thus avoiding wastage caused by dedicating CPU cores. We execute the scheduling loop over a 2-minute window.

[5]Note that to estimate the CPU usage of general serverless functions, which may involve much more complex application-level computations, a more comprehensive estimator is recommended, such as using a Deep Q Network (DQN) [37] or a Graph Neural Network (GNN) [38].

TABLE II
PER REQUEST DATA PIPELINE OVERHEAD FOR SPRIGHT

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	0	1	1	0	0	0	1	7
# of deserialization	1	1	2	0	0	0	2	8

Note: 1. We audit a '1 broker/front-end + 2 functions' chain and exclude client overhead; 2. as SPRIGHT uses DFR, hence no route ④ and ⑤ in Fig. 1. ④ here means direct route from function-1's pod to function-2's pod.

I. Overhead Auditing of SPRIGHT and Constraints

The overhead auditing of SPRIGHT (Table II) shows that 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, the total number of context switches and interrupts for SPRIGHT is still far less than that of the base Knative design. In addition, the results in Fig. 8 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 However, SPRIGHT's shared memory processing is constrained by the need to have intra-node deployment of function chains, requiring locality-aware placement strategies, as exploited in [39] (more details in Appendix B). Existing applications, which use synchronous HTTP/REST APIs and/or POSIX-like sockets, also require minimal changes to the code to work with SPRIGHT's shared memory processing (details in Appendix D).

IV. EVALUATION & ANALYSIS

A. 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. 9). The details of the setup for each scenario are as follows:

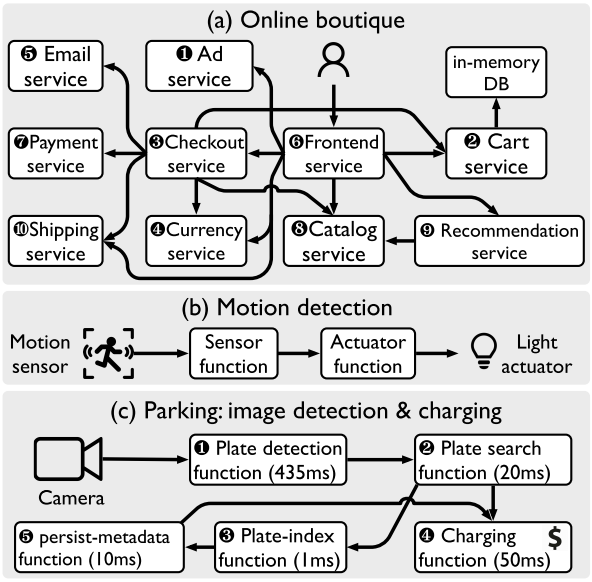


Fig. 9. Serverless function chains setup. The parking workload has two function chain invocation sequences: (Chain-1) ①→②→③→⑤→④; (Chain-2) ①→②→④.

1. Online Boutique is an open-source representative implementation of a microservice-based online store application [40]. It has 10 different functions, communicating with each other using gRPC. We ported these functions to SPRIGHT (in C) and Knative (using Go language) based on the implementation provided in [40]. Functions ported to SPRIGHT use shared memory, while functions in Knative continue to use gRPC, for inter-function communication. We use Locust [41] as the load generator and use the default workload provided in [40] 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 [9]). 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.[6] In Knative mode, we use the Istio ingress gateway to mediate the communication between functions. We disable the activator [42] (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. 9 (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 [43]. 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.

[6]The "frontend service" (Fig. 9 (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).

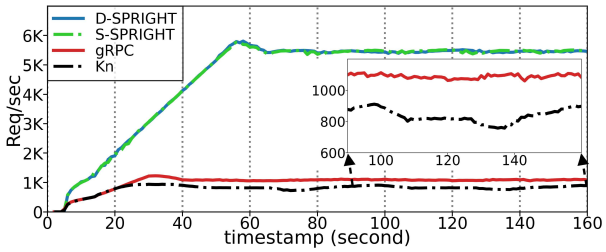


Fig. 10. RPS for online boutique: {Knative, gRPC} at 5K & {D-SPRIGHT, S-SPRIGHT (overlap)} at 25K concurrency.

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 *CNR-Park+EXT* image dataset collected from a parking lot with 164 parking spaces [44]. 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 [45]. The CPU service times of other functions and the sequence of functions being called are shown in Fig. 9.

Testbed setup: The testbed is built on top of a base Knative platform, including 1) Knative serving/eventing components (v0.22.0) [12], [46]; 2) Kubernetes components (v1.19.0), including API server, placement engine, etcd, *etc* [47]. We use the docker engine (v20.10.21) as the container runtime. We consider Calico CNI (Native routing mode) [48] as the underlying networking solution except for the communication within the function chain of Knative. We run the experiments on the NSF Cloudlab with two c220g5 nodes [49]. 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 it can process in parallel.

B. Performance With Realistic Workloads

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 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

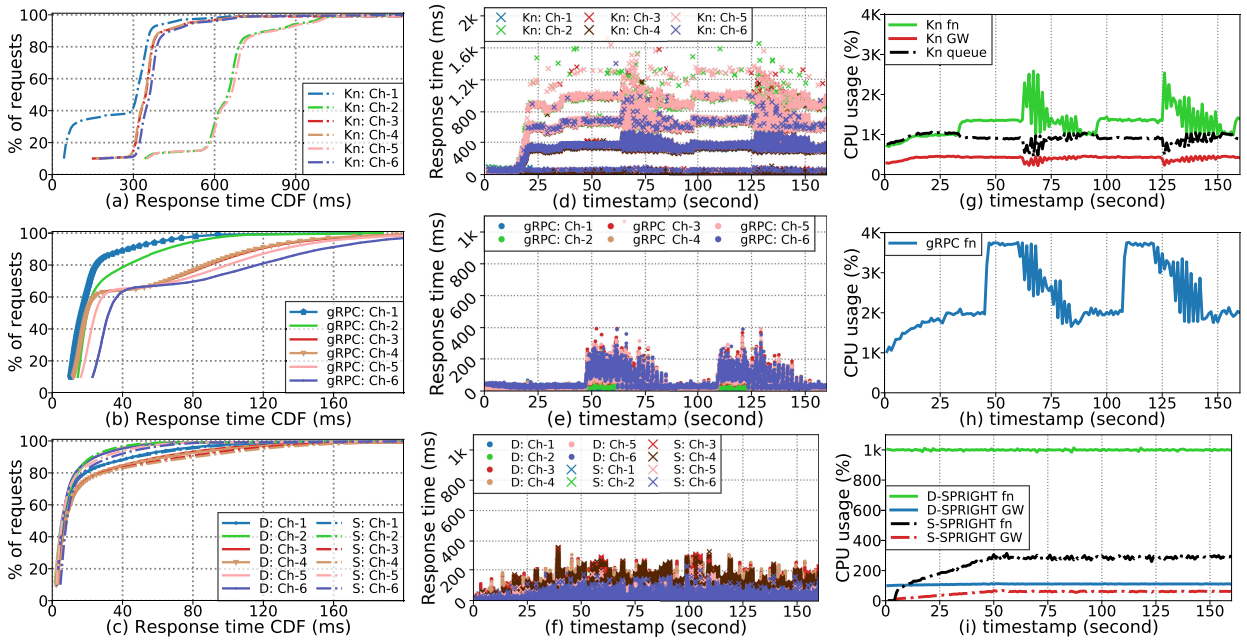


Fig. 11. Online boutique. Top row: Knative, 5K concurrency. Mid. row: gRPC, 5K concurrency. Bottom row: {D-SPRIGHT (*D*), S-SPRIGHT (*S*)}, 25K concurrency. (Left col.) Response time CDF for 6 different function chains; (Mid. col.) Time series of response time, function chains; (Right col.) CPU usage time series, gateway (*GW*), function chains (*fn*), sidecar proxy (Knative).

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. 10), 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. 10) due to the use of sidecars and the 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. 11 (g)). In addition, the use of the gateway and sidecars contributes to additional processing and queuing delays on the request's data path, leading to the reduction in RPS observed (see beyond 30s in Fig. 10). The closed-loop of workload generation and request processing results in the RPS, resource utilization, and response times experiencing overload cycles (occurs 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. 11 (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. 11 (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. 11 (h)). This pattern repeats again, *e.g.,* in the time period 108s - 140s. Overall, this is quite inefficient.

TABLE III
LATENCY COMPARISON AT 5K AND 25K CONCURRENCY

	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**

Note: latency is measured across all the functions of the online boutique.

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 III), significantly less than Knative (690ms) and gRPC (140ms), while utilizing far less CPU. Although D-SPRIGHT constantly consumes CPU cycles when idle, even at 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. 8). 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 §II. S-SPRIGHT *further* reduces CPU usage dramatically by using

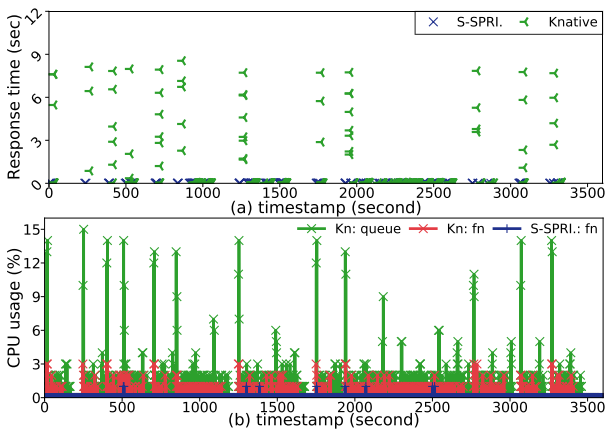


Fig. 12. Time series of response time, and CPU utilization for motion detection workload - 1-hour long experiments.

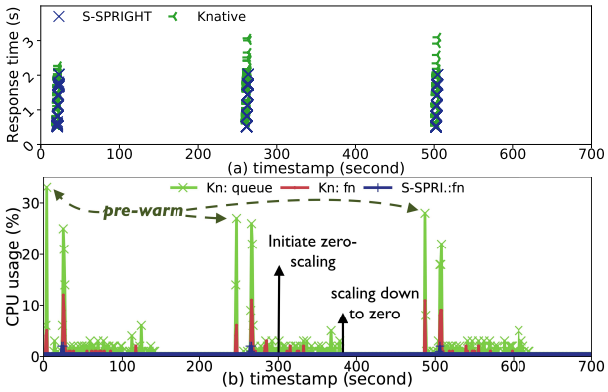


Fig. 13. Parking image detection & charging: (a) Time series of response time of function chains; (b) Time series of aggregate CPU for function chains, sidecar proxy (Knative).

purely event-driven processing compared to D-SPRIGHT. With 5K concurrency, S-SPRIGHT consumes only ∼1 CPU core, 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 maintains low tail response times. Both D-SPRIGHT and S-SPRIGHT maintain a stable RPS of ∼5500 req/sec (Fig. 10), which is 5× higher than the highest stable RPS achieved with Knative and gRPC. Moreover, S-SPRIGHT uses 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. 11(i)), showing the benefit of the eBPF-based event-driven processing.

With SPROXY generating context switches and interrupts for descriptor delivery (Table II), 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. 11(c)). The 95%iles of S-SPRIGHT, measured across all the functions, is 1.2× higher than D-SPRIGHT (more details in Table III). The additional delay for SPROXY's descriptor delivery, adds to the transient queueing and hence slightly longer tail latency. However, as we said in §III-C4, 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 high concurrency levels.

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. 12 (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 [38], 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 sidecar proxy under load more than offsets any benefit of Knative's zero-scaling. E.g., in Fig. 12 (b), the spikes in the CPU usage for the sidecar 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 [50]. However, as observed in Fig. 13 (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

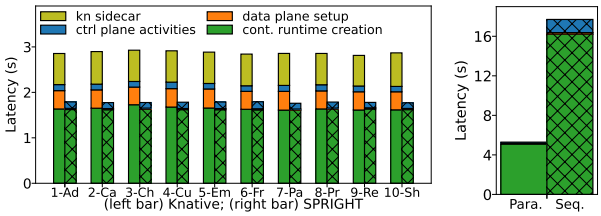


Fig. 14. (Left) Latency comparison of a single function pod startup between SPRIGHT and Knative; and (Right) latency of multiple function pods startup in SPRIGHT (startup in parallel VS. startup in sequence). For function's index, refer to Fig. 9.

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. 13 (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 sidecar 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. 13 (a)). Overall, S-SPRIGHT saves up to 41% CPU cycles in this 700s experiment without resorting to zero-scaling, almost doubling system capacity compared to Knative.

C. Startup Latency Comparison (SPRIGHT Vs. Knative)

We now compare the startup latency for a function chain in SPRIGHT against Knative. For a fair comparison, we use the same control plane and Docker container runtime. The common control plane components include Kubernetes's pod scheduler, controller manager, and API server. We use the online boutique functions [40] for evaluation and reuse the testbed setup in §IV-A.

To measure the latency of control plane activity (t_{cp}), we timestamp when the Kubernetes controller manager (on the master node) receives the pod creation request, and again when the *kubelet* (on the worker node) is signaled by the control plane for the pod initialization tasks, including the dataplane setup and container runtime creation. The difference is the latency for performing control plane activities.

To measure the dataplane setup latency (t_{dp}) of a Knative function pod, we timestamp when the *kubelet* sends the networking setup request to the CNI and another timestamp when the *kubelet* receives an ack. from the CNI indicating successful setup of the function pod's dataplane. We measure t_{dp}, the latency for completing the initialization of the SPROXY (Fig. 5) and its attachment to the shared memory pool.

For both Knative and SPRIGHT, we quantify the difference between the total pod initialization latency (t_{pod}) and the dataplane setup latency (t_{dp}) as the container runtime creation latency ($t_{cr} = t_{pod} - t_{dp}$). t_{pod} is measured as the duration starting from when the *kubelet* is informed of pod initialization by the control plane, until the *kubelet* detects the readiness of the function pod. When profiling the startup latency of SPRIGHT function pods, we disable the CNI to avoid spending unnecessary latency on setting up kernel iptables and configuring veth devices, since SPRIGHT functions do not require these. We keep the CNI plugin enabled for the startup of SPRIGHT gateway pod as discussed in §III-E.

Fig. 14 (Left) shows the startup latency of a single function. SPRIGHT and Knative have same t_{cp} and t_{cr} as both use the same control plane components and container runtime. The primary difference is in the t_{dp}. Knative spends ∼0.4s for setting up the dataplane for a single function pod with an extra ∼0.7s for creating an individual sidecar container for this pod. SPRIGHT takes only ∼0.016s for SPROXY initialization and attaching the shared memory pool. This, again, shows the benefit of SPRIGHT's use of shared memory and event-driven proxies, which eliminates the heavyweight sidecar container creation and slow kernel-based dataplane setup.

The SPRIGHT gateway dedicated to the function chain incurs a significant initialization overhead (∼0.4s) when the SPRIGHT function chain is first created, with an extra ∼0.3 milliseconds to attach the EPROXY. This is mainly caused by the in-kernel configuration (veth-pair and kernel iptables) via CNI, similar to a Knative function pod. But, the startup overhead of the SPRIGHT gateway can be overlapped by starting it in parallel with the SPRIGHT functions in the chain. More importantly, the startup of the SPRIGHT gateway is a one-time task that occurs only during the very first startup of the function chain.

Fig. 14 (Right) compares the startup latency of a complete function chain with the parallel and sequential startup of multiple function pods. Starting up pods in parallel takes 3.3× *less* time than starting up pods sequentially. Thus, it is desirable to have function pods in SPRIGHT start in parallel to amortize the startup penalty. Both dataplane setup in SPRIGHT and control plane activity get the benefit of parallelism. The dominant latency is from container runtime creation, but there is limited benefit from having pods started up in parallel. This is because of the contention for access to the network namespace in the Linux kernel. This forces us to sequentially create and modify the network namespaces [51], [52], [53]. Using a shared network namespace (shared by functions in the same security domain) can eliminate the network namespace contention and further reduce the total container runtime creation time for parallel startup of multiple function pods in a chain [52].

D. Evaluating the Scaling of the SPRIGHT Gateway

To understand the benefits of rate proportional scheduling (denoted `RP`) on the SPRIGHT gateway, we compare it with the "*dedicated cores*" policy (denoted `DC`). The "*dedicated cores*" strategy assigns CPU cores exclusively to the SPRIGHT gateway, which can prevent CPU interference. However, this strategy potentially wastes CPU cores under light load as the assigned CPU cores are not shared with any other SPRIGHT gateway or functions running on the same node. We use three online boutique [40] function chains (`CH-1`, `CH-2`, and `CH-3`), co-located on the same node. Each function chain has a dedicated SPRIGHT gateway. We set the peak concurrency levels of the three function chains as 5K, 12K, and 30K. We set

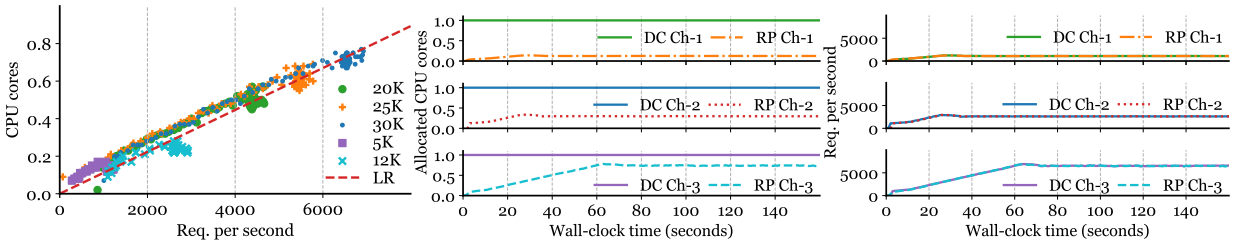


Fig. 15. Comparison of rate proportional scheduling and dedicating cores: (left) correlation between CPU core usage of SPRIGHT gateway and RPS; (middle) CPU allocation time series; (right) RPS time series.

the spawn (increase) rate of the concurrency level at 500/sec. Other configurations are the same as §IV-B1.

Fig. 15 (middle) shows the CPU allocation over time for the different SPRIGHT gateways. It is clear that the SPRIGHT gateway using DC results in serious CPU wastage by consistently occupying the CPU core throughout the entire time. In contrast, RP allocates CPU cores to SPRIGHT gateways of different function chains in proportion to their load. The unused portion of the CPU core can be potentially shared with serverless functions, resulting in better efficiency. In the meantime, RP also does not cause any significant performance (RPS) loss when we look at Fig. 15 (right), which demonstrates the feasibility of using the concept of "rate proportional scheduling" to help address the resource allocation for the SPRIGHT gateway while retains the high performance of SPRIGHT's data plane.

V. Related Work

In recent years, a number of serverless platforms have been launched, *e.g.,* AWS Lambda [54], IBM Cloud Functions [55], Apache OpenWhisk [56], OpenFaaS [57], Knative [10], *etc*, to support cloud-resident applications. Work on understanding the performance impact of commercial or open-source serverless platforms [19], [58] has guided us on the design of SPRIGHT. Li et al. [19] 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). Zhu et al. [59] reported that container-based sidecars cause excessive latency and CPU usage increase, which is also consistent with our observations. Benedetti et al. [58] 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 [60], [61], [62], [63]. The overhead of protocol processing [24] and serialization-deserialization [22], [64] directly impact networking performance, which applies to container-based serverless functions, 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 [26], AF_XDP in OVS [65]) or bypass kernel-based networking (*e.g.,* NetVM for NFV [66]). 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 [51], [67], [68], [69], intelligent resource provisioning, and traffic management [1], [70]. Ditto [39] is a serverless analytics system developed on top of SPRIGHT, maximizing intra-node shared memory processing to alleviate the data plane overheads of a large amount of shuffle-related traffic within function chains used for data analytics. Further, [38], [71], [72] 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. But, 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. Reference [73] 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 [50]) 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 [74]. They can achieve an 85% improvement of the 99%ile latency. Although [74] considerably improves the SLOs, it is built on Knative with heavyweight components (*e.g.,* queue proxy), resulting in excessive resource usage. Fuerst and Sharma [75] 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 CPU usage, [75] can be a good complement to SPRIGHT to reduce memory utilization.

VI. Conclusion

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. In addition, SPRIGHT saves 32% startup latency for a single function pod compared to Knative, which is an ideal capability for serverless computing. SPRIGHT is publicly available at https://github.com/ucr-serverless/spright.git

REFERENCES

[1] V. Mittal et al., "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 168–181.

[2] (2022). *AWS Serverless API*. [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-api.html

[3] (2022). *OpenFaaS API Gateway/Portal*. [Online]. Available: https://docs.openfaas.com/architecture/gateway/

[4] (2022). *MQTT Version 5.0*. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[5] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Comput.*, vol. 16, no. 2, pp. 62–67, Mar./Apr. 2012.

[6] I.-C. Wang, S. Qi, E. Liri, and K. K. Ramakrishnan, "Towards a proactive lightweight serverless edge cloud for Internet-of-Things applications," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Oct. 2021, pp. 1–4.

[7] (2022). *APACHE KAFKA*. [Online]. Available: https://kafka.apache.org/

[8] (2022). *Istio Traffic Management*. [Online]. Available: https://istio.io/latest/docs/concepts/traffic-management/

[9] S. Qi, L. Monis, Z. Zeng, I.-C. Wang, and K. K. Ramakrishnan, "SPRIGHT: Extracting the server from serverless computing! High-performance eBPF-based event-driven, shared-memory processing," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 780–794.

[10] (2022). *Knative*. [Online]. Available: https://knative.dev

[11] The Linux Foundation. (2022). *EBPF*. [Online]. Available: https://ebpf.io/

[12] (2022). *Knative Eventing*. [Online]. Available: https://knative.dev/docs/eventing/

[13] (2022). *OpenWhisk—Creating Action Sequences*. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences

[14] (2022). *OpenWhisk Composer*. [Online]. Available: https://github.com/apache/openwhisk-composer

[15] (2022). *Chaining OpenFaaS Functions*. [Online]. Available: https://ericstoekl.github.io/faas/developer/chaining_functions/

[16] Microsoft. (2022). *Azure—Function Chaining in Durable Functions*. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-sequence?tabs=csharp

[17] (2022). *NGINX*. [Online]. Available: https://www.nginx.com/

[18] (2022). *Istio Architecture*. [Online]. Available: https://istio.io/latest/docs/ops/deployment/architecture/

[19] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proc. 5th Int. Workshop Serverless Comput.*, 2019, pp. 37–42.

[20] (2022). *Of-Watchdog*. [Online]. Available: https://github.com/openfaas/of-watchdog

[21] (2021). *Wrk*. [Online]. Available: https://github.com/wg/wrk

[22] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soule, "Zerializer: Towards zero-copy serialization," in *Proc. Workshop Hot Topics Operating Syst.*, 2021, pp. 206–212.

[23] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather," in *Proc. Workshop Hot Topics Operating Syst.*, Jun. 2021, pp. 199–205.

[24] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 656–671, Mar. 2021.

[25] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, "Securing Linux with a faster and scalable iptables," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, pp. 2–17, Nov. 2019.

[26] T. Høiland-Jørgensen et al., "The express data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. Experiments Technol.*, 2018, pp. 54–66.

[27] J. Levin and T. A. Benson, "ViperProbe: Rethinking microservice observability with eBPF," in *Proc. IEEE 9th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2020, pp. 1–8.

[28] Red Hat. (2022). *Understanding the EBPF Networking Features in RHEL*. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel-8_configuring-and-managing-networking

[29] (2022). *EBPF XDP: The Basics and a Quick Tutorial*. [Online]. Available: https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/

[30] (2021). *Apache Benchmark*. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[31] M. Abranches, O. Michel, and E. Keller, "Getting back what was lost in the era of high-speed software packet processing," in *Proc. 21st ACM Workshop Hot Topics Netw.*, 2022, pp. 228–234.

[32] (2023). *Multi-process Support of DPDK*. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html

[33] (2023). *Ring Library*. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/ring_lib.html

[34] W. Zhang et al., "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, Aug. 2016, pp. 26–31.

[35] (2023). *Apache Camel*. [Online]. Available: https://camel.apache.org/camel-k/2.1.x/kamelets/kamelets.html

[36] S. G. Kulkarni et al., "NFVnice: Dynamic backpressure and scheduling for NFV service chains," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 639–652, Apr. 2020.

[37] Z. Wang et al., "DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 16–30.

[38] J. Park, B. Choi, C. Lee, and D. Han, "GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices," in *Proc. 17th Int. Conf. Emerg. Netw. Experiments Technol.*, 2021, pp. 154–167.

[39] C. Jin et al., "Ditto: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 406–419.

[40] (2023). *Online Boutique by Google*. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[41] (2022). *Locust*. [Online]. Available: https://locust.io/

[42] (2022). *Knative Serving—Activator*. [Online]. Available: https://github.com/knative/serving/blob/main/docs/scaling/SYSTEM.md#activator

[43] C. R. Wren, Y. A. Ivanov, D. Leigh, and J. Westhues, "The MERL motion detector dataset," in *Proc. Workshop Massive Datasets*, 2007, pp. 10–14.

[44] G. Amato, F. Carrara, F. Falchi, C. Gennaro, and C. Vairo, "Car parking occupancy detection using smart camera networks and deep learning," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2016, pp. 1212–1217.

[45] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "ECML: Improving efficiency of machine learning in edge clouds," in *Proc. IEEE 9th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2020, pp. 1–6.

[46] (2022). *Knative Serving*. [Online]. Available: https://knative.dev/docs/serving/

[47] (2021). *Kubernetes Components*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[48] (2022). *Project Calico*. [Online]. Available: https://www.tigera.io/project-calico/

[49] D. Duplyakin et al., "The design and operation of CloudLab," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1–14.

[50] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.

[51] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 57–70.

[52] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: Ephemeral endpoints for serverless networking," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 16–29.

[53] V. Jain, S. Qi, and K. K. Ramakrishnan, "Fast function instantiation with alternate virtualization approaches," in *Proc. IEEE Int. Symp. Local Metrop. Area Netw. (LANMAN)*, Jul. 2021, pp. 1–6.

[54] (2022). *AWS Lambda*. [Online]. Available: https://aws.amazon.com/lambda/

[55] (2022). *IBM Cloud Functions*. [Online]. Available: https://cloud.ibm.com/functions/

[56] (2022). *Apache OpenWhisk*. [Online]. Available: https://openwhisk.apache.org/

[57] (2022). *OpenFaaS*. [Online]. Available: https://www.openfaas.com/

[58] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Experimental analysis of the application of serverless computing to IoT platforms," *Sensors*, vol. 21, no. 3, p. 928, Jan. 2021.

[59] X. Zhu et al., "Dissecting overheads of service mesh sidecars," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2023, pp. 142–157.

[60] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 65–77.

[61] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, 1997.

[62] J. Lei, M. Munikar, K. Suo, H. Lu, and J. Rao, "Parallelizing packet processing in container overlay networks," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 1–16.

[63] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop Experim. Comput. Sci.*, Jun. 2007, pp. 1–4.

[64] S. Kanev et al., "Profiling a warehouse-scale computer," in *Proc. 42nd Annu. Int. Symp. Comput. Architecture*, 2015, pp. 158–169.

[65] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff, "Revisiting the open vSwitch dataplane ten years later," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 245–257.

[66] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. 11th USENIX Symp. Networked Syst. Design Implement.*, Seattle, WA, USA, Apr. 2014, pp. 445–458.

[67] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, Jul. 2018, pp. 923–935.

[68] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 265–279.

[69] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th USENIX Symp. Networked Syst. Design Implement.*, 2020, pp. 419–434.

[70] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 138–152.

[71] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 153–167.

[72] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 311–327.

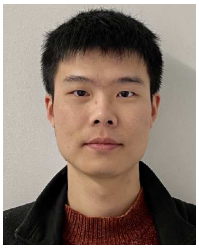
[73] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proc. 3rd USENIX Workshop Hot Topics Edge Comput.*, 2020, pp. 1–7.

[74] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," 2019, *arXiv:1903.12221*.

[75] A. Fuerst and P. Sharma, "FaasCache: Keeping serverless computing alive with greedy-dual caching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 386–400.

Shixiong Qi received the B.Sc. degree in electronic and information engineering from Nanjing University of Posts and Telecommunications, China, in 2015, and the M.Sc. degree in communication and information systems from Xidian University, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of California at Riverside. His current research interests include cloud computing, 5G, and network function virtualization.

Leslie Monis received the B.Tech. degree in computer science and engineering from NITK, India, in 2019, and the M.S. degree in computer science from the University of California at Riverside in 2022. He is currently a Software Engineer with NVIDIA. His research interests include cloud computing and network function virtualization.



Ziteng Zeng received the B.Sc. degree in computer science from Zhejiang University, China, in 2020, and the M.S. degree in computer science from the University of California at Riverside in 2022. He is currently a Software Engineer with Google. His research interests include cloud computing and network function virtualization.

Ian-Chin Wang received the bachelor's degree in computer science from National Chiao Tung University, Taiwan, in 2018, and the M.S. degree in computer science from the University of California at Riverside in 2021. He is currently a Software Engineer with Oracle. His research interests include serverless computing and the IoT.

K. K. Ramakrishnan (Life Fellow, IEEE) received the M.Tech. degree from the Indian Institute of Science in 1978 and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, USA, in 1981 and 1983, respectively. He is currently a Professor of computer science and engineering with the University of California at Riverside. Previously, he was a Distinguished Member of the Technical Staff with AT&T Labs-Research. Before 1994, he was the Technical Director and a Consulting Engineer of networking with Digital Equipment Corporation. From 2000 to 2002, he was the Founder and the Vice President with TeraOptic Networks Inc. He has published nearly 300 articles and has 183 patents issued in his name. He is an ACM Fellow and an AT&T Fellow, recognized for his fundamental contributions to communication networks, including his work on congestion control, traffic management, and VPN services.