INVITED PAPER

Will Serverless Computing Revolutionize NFV?

This article explores a network scenario that is emerging as a different type of a novel cloud computing paradigm in which applications are decomposed into smaller and modular functions.

By Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, *Senior Member IEEE*, Ivica Rimac, Klaus Satzke, *Member IEEE*, and Manuel Stein

ABSTRACT | Communication networks need to be both adaptive and scalable. The last few years have seen an explosive growth of software-defined networking (SDN) and network function virtualization (NFV) to address this need. Both technologies help enable networking software to be decoupled from the hardware so that software functionality is no longer constrained by the underlying hardware and can evolve independently. Both SDN and NFV aim to advance a software-based approach to networking, where networking functionality is implemented in software modules and executed on a suitable cloud computing platform. Achieving this goal requires the virtualization paradigm used in these services that play an important role in the transition to software-based networks. Consequently, the corresponding computing platforms accompanying the virtualization technologies need to provide the required agility, robustness, and scalability for the services executed. Serverless computing has recently emerged as a new paradigm in virtualization and has already significantly changed the economics of offloading computations to the cloud. It is considered as a low-latency, resource-efficient, and rapidly deployable alternative to traditional virtualization approaches, such as those based on virtual machines and containers. Serverless computing provides scalability and cost reduction, without requiring any additional configuration overhead on the part of the developer. In this paper, we explore and survey how serverless computing technology can help building adaptive and scalable networks and show the potential pitfalls of doing so.

Manuscript received July 6, 2018; revised November 26, 2018 and January 21, 2019; accepted January 29, 2019. Date of publication February 26, 2019; date of current version March 25, 2019. *(Corresponding author: Paarijaat Aditya.)*

The authors are with Nokia Bell Labs, 70435 Stuttgart, Germany (e-mail: paarijaat.aditya@nokia-bell-labs.com).

Digital Object Identifier 10.1109/JPROC.2019.2898101

KEYWORDS | Application virtualization; cloud computing; edge computing; network function virtualization (NFV); serverless computing; software-defined networking (SDN).

I. INTRODUCTION

Today's cloud environments offer the possibility of rapidly introducing new services and adding additional resources for existing services within seconds. In contrast, the traditional way of using dedicated middleboxes to deploy network services, such as firewalls, intrusion detection systems, caches, and load balancers, does not offer the same agility as cloud computing environments. As a result, today's networking resources do not typically offer the same rapid flexibility of cloud services.

To close the gap between cloud computing services and network services and to fulfill the promise of agile networks, the networking industry needs to undergo a seismic technology shift. With the goal of evolving the network services at a similar pace as today's cloud computing services, the last few years have seen an explosive increase in interest for adopting the software-defined networking (SDN) [1] and network function virtualization (NFV) [2] technologies. These technologies enable networking software to be decoupled from the hardware so that software is no longer constrained by the hardware that delivers it and can evolve independently. For example, SDN separates the network's control and forwarding planes and provides a centralized view of the distributed network for more efficient orchestration and automation of network services. NFV focuses on virtualizing network services and realizing network functions in software to accelerate innovation and deployment. Such virtualized network functions (VNFs) include deep packet inspection, domain name system, and caching.

However, despite the technical promise, the progress of NFV and SDN adoption has been slow [3], [4]. We make the following two observations for this slow pace. First, VNFs are typically deployed inside the network, where computing resources may not be as broadly available as in a typical data center. To satisfy low-latency requirements of various applications, VNFs are usually hosted on the network edge, for example, on edge data centers with only a few server racks or even with equipment put on customer premises, such as edge routers or home gateways. These environments have limited computing resources that need to be managed carefully to keep the service quality high. In fact, initial attempts to realize VNFs via virtual machines (VMs) are being abandoned in favor of other virtualization techniques such as containers that require fewer resources [5].

Our second observation is that network services needed by many new applications are set up to serve a particular session and, thus, are short-lived and event-driven. A few examples are transcoding tasks [6], [7], anomaly detection [8], and pluggable modules of SDN controllers. Unlike traditional network services, whose setup time is amortized over the longevity of the service, these new network services cannot afford high startup latencies and require the virtualization platform to facilitate fast scheduling of tasks.

We think that the virtualization paradigm and its corresponding computing platform will play an important role in the transition to software-based networks that can match the speed of evolution in the cloud computing environments. Based on our observations, we derive a set of general requirements that a virtualization platform must satisfy to effectively support the SDN- and NFV-based services, which are listed in the following.

1) The platform should be able to match the demand of a service by scaling up fast enough to provision additional compute resources for the service so that the service is able to process all incoming traffic, even if that traffic is increasing rapidly.
2) The platform should make efficient use of the available resources. Ideally, all resources allocated to a service should be actively used to process incoming traffic, implying that services are never overprovisioned and idle service capacity is released immediately.
3) The platform should require minimal configuration and management from the developers while provisioning resources for the services, allowing developers to focus on application logic.
4) The platform should isolate services and their provisioned resources from each other, such that faults or load spikes in one service should not affect others.

Several virtualization technologies have been developed in the cloud computing domain that tries to meet these requirements, such as VMs, containers, and unikernels. In this paper, we survey the applicability of one such technology, serverless computing, as the underlying platform for networking services.

Serverless computing has recently emerged as a promising paradigm in virtualization, with the primary objective of providing seamless scalability and enabling developers to focus completely on their business logic. In serverless computing, developers do not need to provision server capacity in advance, because all aspects related to the service management, including resource allocation, placement, and scaling, are handled by the computing platform. This easy management makes serverless computing a well-suited candidate for services with varying demands.

In the last few years, serverless computing has seen an explosion of interest, with every major cloud provider now offering a serverless computing platform [9]–[12]. For the cloud computing industry, serverless computing has already significantly changed the economics of offloading computations to the cloud. It is considered a rapidly deployable alternative to VMs and containers, providing scalability and cost reduction, without requiring any additional configuration overhead on the part of the developer.

One popular realization of serverless computing is Function-as-a-Service (FaaS). In contrast to traditional virtualization techniques which require developers to bundle their applications as servers inside VMs or containers, FaaS allows developers to write their applications as sets of stand-alone functions. In today's FaaS platforms, these functions may be invoked via HTTP requests or other events that happen internally or externally with respect to the platform. The platform is responsible for allocating the resources necessary for individual function executions according to the demand the function receives. In this paper, we will be referring to FaaS when we use the term serverless computing.

Conceptually, it may appear that the event-driven nature and the finer granularity of resource provisioning (compared with other virtualization techniques, such as VMs and containers) in FaaS make it a suitable platform for deploying adaptive and scalable network services. In this paper, we survey how serverless computing, in general, can help building such networks and also explore the potential pitfalls of doing so.

In the following sections, we first provide a background on different virtualization technologies (in Section II) such as VMs, containers, and unikernels. Afterward, we present (in Section III) an in-depth overview of the serverless computing paradigm. We then explore (in Section IV) areas where serverless computing is applicable and how it can be employed to build adaptive network services. Finally, we discuss (in Section V) the research challenges for the application of serverless computing in the networking domain.

II. BACKGROUND

In this section, we briefly describe some existing virtualization technologies and their properties that are relevant to applications in the networking domain.

A. Virtual Machines

A VM [13] simulates hardware to allow an unmodified "guest" operating system (OS) to be run in isolation. A hypervisor running on the actual physical machine (referred to as the host) manages the sharing of the physical hardware resources between VMs, allowing multiple VMs to be run on the same host. An application running inside a VM interacts with the guest OS, which in turn interacts with the hardware via the hypervisor on the host.

Employing VMs is one of the most popular approaches for deploying services on the cloud. Most cloud providers allow the customers to provision VMs on demand and typically charge them for how long a VM runs. Since VMs need to boot the guest OS when they start, they typically have long startup latencies, making them well suited for long running services.

Following the popularity of VMs for applications running in the cloud, VNFs have been traditionally realized as VMs. Due to the long startup times of VMs, in order not to reduce the service quality, VNFs are often overprovisioned to make sure that they can meet quickly varying workloads and demand spikes without the need for a scale-out operation. Furthermore, even when overprovisioned, applications running in VMs require continuous monitoring for their orchestration.

B. Containers

Another popular approach in virtualization is to use containers [14], [15]. Containers allow developers to package applications with their entire runtime environment. Unlike VMs, containers do not simulate the hardware interface, such that applications running in the containers directly interact with the system call interface exposed by the underlying OS of the host. Essentially, applications inside containers run natively on the host as processes, with an additional layer of protection for resource isolation [16], [17]. Although all containers on the host share the same system kernel, they can have isolated filesystems, networking, CPU, and memory resources.

Not having a "guest" OS, the memory footprint is much smaller than a VM. This property also enables the containers to start an order of magnitude faster than VMs [18]. These advantages make containers a competitive and resource-efficient alternative to VMs for realizing VNFs [5].

Similar to VMs, services deployed using containers still need to be monitored continuously for their orchestration. As such, they are also overprovisioned for higher levels of load to handle sudden increases in incoming traffic.

C. Unikernels

More recently, unikernels [18]–[20] have attracted attention from the research community. Unikernels are tiny runtime environments where a target application is statically linked to a minimalistic library OS, such as MirageOS [19]. This static linking allows the application to run directly on top of a virtual hardware abstraction, where a hypervisor handles the resource sharing between different unikernels. As a result, unikernels can provide similar isolation properties such as VMs, which is stronger than containers.

On the one hand, unikernels offer a lighter weight alternative to VMs because of the reduction in the OS image size and can boot up faster than the traditional VMs and containers [18]. On the other hand, Unikernels can be considered less flexible, because dynamically adding/removing functionality requires a recompilation of the entire unikernel image.

III. WHAT IS SERVERLESS COMPUTING?

In this section, we review the general state of serverless computing technology as well as its advantages and disadvantages. Since the launch of Amazon Lambda in 2014 [9], serverless computing has attracted a significant interest from both industry and academia as a new cloud computing paradigm. In this paradigm, application developers no longer have to manage servers (hence, the name), provision resources, and decide where the application runs—-all these tasks are the responsibility of the platform provider.

One popular realization of serverless computing paradigm is the FaaS model, where the applications are written as individual functions that can be separately managed. These functions can be invoked via various types of events (e.g., external Web requests and internal database triggers), such that the platform allocates an ephemeral runtime environment for the associated function(s) and executes them. In contrast to VMs or containers, the unit of execution in FaaS is a function. This finer granularity enables the platform operators to schedule the execution faster, making applications running in such a platform more responsive to load variations. This responsiveness, combined with the ephemerality of the execution environment, makes the applications more resource-efficient.

A. Serverless Platform Architecture

As of this writing, there are several commercial serverless offerings, including Amazon Lambda [9], IBM Cloud Functions [10], Microsoft Azure Functions [11], and Google Cloud Functions [12], among others. Although these platforms make somewhat different design choices and tradeoffs regarding the execution environment, they have adopted a similar overall architecture.[1] This architecture consists of the following main modules (Fig. 1).

1) *Front End:* The front end is the interface for developers to deploy their applications onto the serverless computing platform. It also contains an interface, such as an application programming interface (API) gateway [22], for users to send REST or RPC requests to these applications and receive responses. For scalability, multiple front-end servers can run behind a standard load balancer.

[1]Conceptually, Erlang [21] has a similar message-passing and event-driven architecture.

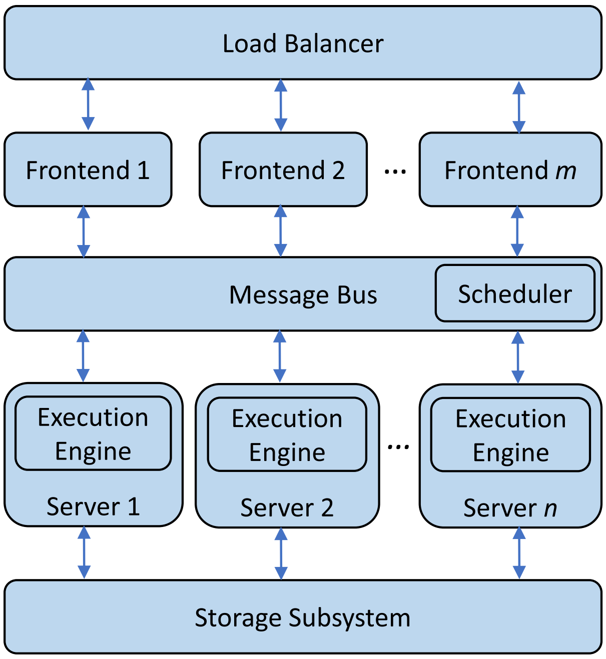

Fig. 1. ***Serverless platform architecture.***

2) *Execution Engine:* In many serverless computing platforms, there is an execution engine on each server that takes an incoming request as input and launches a runtime environment (normally a container[2]) with the associated function code and supporting libraries to handle the request. When the function finishes executing (i.e., the request has been processed), the container will be terminated. The term referring to this practice is launching a "cold container," because a new container is started for each incoming, concurrent request. This approach inevitably incurs long startup latencies for the function execution.
   To improve the function startup latency, one common practice is to reuse the launched containers by keeping them "warm" for a configurable period of time at the expense of occupying system resources while idling [23], [24]. With this approach, the first request to a function will be processed by a "cold" container, but the subsequent requests to this function can be processed by the "warm" container. Note that most serverless platforms allow each container to execute only one function at a time for fault isolation. Therefore, multiple concurrent requests to a function are either processed in their individual "cold" containers or processed in one or more "warm" containers in a sequential manner.
3) *Message Bus and Scheduler:* Generally, there is a message bus mediating between the front end and the execution engine. The front end publishes the requests into the message bus. According to the system status, a request scheduler at the message bus dispatches the requests into different message queues that are individually subscribed by execution engines on each server. Each execution engine will retrieve requests from its associated message queue and launch runtime environments to process them.
   Note, also, that application logic often consists of sequences of multiple functions. A request to a function can be from the external users or internally from the previous function in a sequence. Serverless computing platforms normally treat these two types of requests the same (from a load balancing perspective), and both of them are dispatched through the message bus.
4) *Storage Subsystem:* A storage subsystem is needed when there are states or data that need to persist or different functions need to share the data (especially, a big chunk of data).

[2]Alternative runtime environments (e.g., unikernels [19], [20] and NetBricks [3]) can also be used.

B. Advantages of Serverless Computing

In the following, we list a number of major benefits of the serverless paradigm for application developers.

1) *No Server Management:* Managing and provisioning server resources (also including VMs and containers) for applications present significant overhead and responsibility for traditional application developers and operators. Serverless computing alleviates this problem by shifting this responsibility completely to the platform and allows developers to focus on their application logic. In doing so, the serverless computing platform can also better leverage its entire resource pool and optimize the application performance.
2) *Resource Efficiency and Low Cost:* With serverless computing, system resources are allocated whenever there is a request that needs to be processed and deallocated when the request processing has finished (as opposed to, for instance, a VM-based application, which still occupies system resources when idling). This practice minimizes the usage of system resources and, thus, is more resource-efficient compared with the alternatives. Developers are charged based on how much resource is actually used by their applications. In other words, the charged cost is associated with only the amount of meaningful work that has been done, and there is no cost resulting from idling resources (in contrast to a VM-based application, which still has to pay for unused, idling VMs).
3) *Built-In Scalability:* Developers only need to upload their applications onto the serverless computing platform that deploys these applications according to certain requirements. Whenever there is an incoming request, the platform instantly and precisely scales up the application to handle the request. When the request has been handled, the platform automatically scales down (immediately or after a configurable period of time).

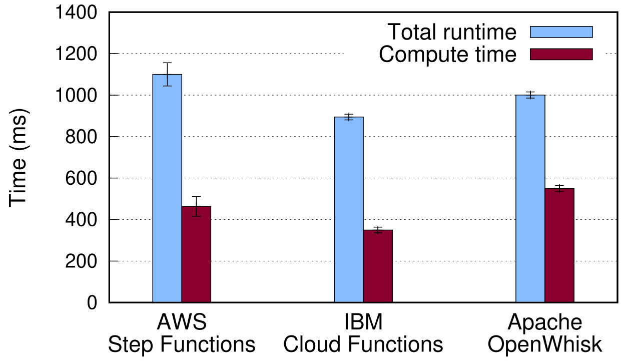

Fig. 2. ***Total runtime and compute time of executing an image processing pipeline with four functions on the existing commercial serverless platforms. The results show the mean values with a 95% confidence interval over ten runs, after discarding the initial (cold) execution.***

C. Disadvantages of Serverless Computing

Despite the numerous benefits of the serverless computing paradigm, there is one primary disadvantage for applications that employ serverless computing—the startup latency per-invocation to handle a request. As described in Section III-A, serverless computing platforms allocate an ephemeral runtime environment (normally a container) to execute a function when a request (i.e., an event) to invoke that function is received. The time spent by the platform in preparing this environment is incurred as the startup latency for each invocation of the function.

These startup latencies could be on the order of tens to hundreds of milliseconds or even longer depending on the underlying virtualization technology being used and whether the function invocation happens to incur a "warm" or a "cold" start. For short-lived functions, with their actual compute times less than a few hundred milliseconds, these startup latencies could mean a significant overhead. Furthermore, these startup latencies are accumulated as application response delay when a sequence of multiple functions are executed to handle a request.

The effect of startup latencies in the existing commercial platforms can be seen in Fig. 2, which shows the overheads of running an image processing pipeline [25] on these platforms. The pipeline consists of four consecutive function executions that extract image metadata, verify and transform it to a specific format, tag objects via image recognition, and produce a thumbnail.

We ran this pipeline using AWS Step Functions [26], IBM Cloud Functions with Action Sequences [10], and Apache OpenWhisk [27], all of which provide a method to connect multiple functions into a single service.[3] We found that the total runtime (which includes the per-function invocation overhead) is significantly more than the actual computation time required for the function executions. This large difference indicates that the per-function invocation latencies can collectively amount into large overheads for applications, whose execution spans several functions. As a result, the range of applications well-suited for the existing serverless computing platforms depends on their latency requirements. We discuss these requirements further in Section IV-A1.

[3]As of this writing, other major serverless providers, e.g., Microsoft and Google, do not support Python, which was used for this pipeline.

IV. WHERE DOES SERVERLESS COMPUTING APPLY?

In this section, we explore the scenarios where serverless computing can be considered a good fit. We first describe this aspect from a more general perspective. We then focus on specific applications from the networking domain.

A. Generality of Serverless Applications

While serverless computing offers several advantages for application developers, it may not be generally applicable. Here, we categorize the biggest factors that we believe may have an influence on the decision whether to use serverless computing. In Section V, we also describe some other factors that could be considered as well.

1) Latency: One important factor developers would consider when deciding whether to use serverless computing is the latency that their applications are going to experience. Application latency requirements are manifold and show a broad variation over several orders of magnitude, ranging from latencies in the range of a few milliseconds for applications such as packet forwarding and high-frequency trading to latencies beyond the 100-ms range for applications such as batch or asynchronous event processing (see Fig. 3).

On the one hand, applications that do not have very strict latency requirements, on the order of a few tens of milliseconds or more (shown in the green region in Fig. 3), can potentially be implemented in a serverless manner. These include stream processing, SDN control plane functions, background tasks, and Internet of Things-related tasks. On the other hand, applications with very strict latency budget (shown in the gray and red regions in Fig. 3), such as online gaming and augmented reality, may not work well on the existing serverless platforms.

For latency-sensitive applications, as described in Section III-C, the latency overhead imposed by the serverless platform may prove problematic for creating serverless applications that are complex with multiple interacting functions. As a result, developers of such applications would be inclined to merge functions to suffer fewer latency penalties making their applications less modular or not to use serverless computing at all. Either way, they would not benefit from the full potential of serverless computing. Nevertheless, recent approaches [28], [29] for high-performance serverless computing can broaden its use and help more developers benefit from it.

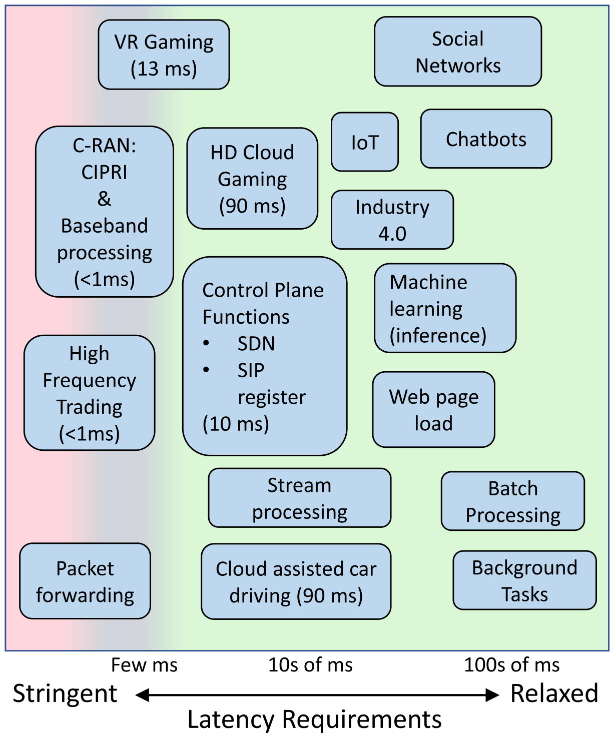

Fig. 3. ***Latency requirement ranges for various applications.***

2) Workload Characteristics: Another important aspect to be considered by the developers is the workload characteristics of their applications. At the high level, because of the event-driven execution nature, serverless computing is especially well-suited for applications that have variable workloads.

As described in Section III-B, in the serverless computing paradigm, the developer only supplies the function code to the serverless platform provider, and the provider is responsible for managing resources to execute the function. Furthermore, the platform allocates the resources in fine increments to match the exact demand for the function. In case the demand for a function experiences a sudden increase (i.e., spike), the platform can quickly create additional function execution instances to handle the increased demand. In doing so, the serverless paradigm is very well-suited to dynamically scale up to handle load spikes. This approach differs from a more traditional approach where developers must overprovision, for example, VMs for the expected peak demand. Furthermore, with serverless computing, the resources can be deallocated once the demand decreases and need not to be kept idle. As a result, serverless computing can be very resource-efficient and economical.

From the developers' perspective, resource efficiency also leads to cost savings, since one need not pay for idle resources. As a concrete example, in Fig. 4, we compare the monthly expenditure for running a Web service on Amazon Lambda [9] (a commercial serverless offering) and Amazon EC2 (which allows users to rent VMs). We assume that the service carries out a short task, such as resizing an image, which takes roughly 100 ms to execute. We also assume a constant workload of 1 request/second for a month, with an increasing number of load spikes per day, each of size 1000 requests/second lasting for 1 min. From Fig. 4, we find that as the number of spikes per day increases, the monthly cost for running the service on Amazon Lambda becomes increasingly more economical than renting VMs provisioned for peak load on Amazon EC2.[4]

Despite these advantages, however, if an application has a very high constant workload (and hence is also latency-sensitive), provisioning VMs or containers in advance to match this load is a better option, since the latency overheads associated with every function invocation in a serverless platform (see Section III-C) could adversely affect the throughput of the application.

3) Application State: One requirement of serverless computing is that each function is stateless. This requirement entails two options for an application that wants to preserve state across function executions. First, the state can be fully encapsulated within the event message passing between the function executions. As such, if there are multiple functions executing in a workflow, each function will find the full application state it needs in the event message it receives, operate on it, and pass it on to the next function. One such example is AWS Lambda@Edge which only operates on the HTTP requests it receives [30]. Unfortunately, this approach may not be applicable for all applications, especially when the application state is large.

The second option for an application to persist state across function executions is to utilize a storage system, such that each function will retrieve the application state from the storage system at the beginning of its execution, operate on it, and then save it in the storage system

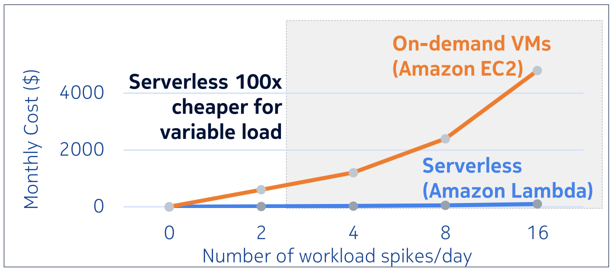

Fig. 4. ***Cost comparison between Amazon Lambda (serverless) and Amazon EC2 (VMs) for spiky workload. In the gray region, serverless is 100x cheaper.***

[4]For Amazon Lambda, we use the price for a function running for 100 ms, with 2-GB RAM. For Amazon EC2, we use two t2.nano instances to handle the constant-rate requests; to handle load spikes, we use 13 m4.4xlarge instances allocated for an hour when the spike occurs.

for other function executions to retrieve. Many serverless functions operate in conjunction with a storage service using this principle. To enable easier development, serverless computing platforms normally provide such a storage service [9], [27], [28], either offered by the ecosystem of the platform (e.g., AWS Lambda users usually utilize storage services offered by Amazon such as S3, DynamoDB, or Kinesis) or by explicit external requests to other storage services (e.g., via REST calls).

However, being able to architect the application with the above-mentioned principles may not be enough for an application to employ serverless computing. The state retrieval and storage operations can incur overhead, which might affect the application performance. Applications with multiple functions may suffer from such overhead several times while handling a single request. Similarly, if the state is relatively large, applications with fewer functions but with stringent latency requirements may not be able to benefit from serverless computing. A real-time video encoding application, in which functions might need to pass a high-resolution frame to the next function, is such an example. Therefore, the ability to handle such cases becomes important for the serverless computing platform to broaden its applicability.

B. Applications From the Networking Domain

In this section, we describe a few specific applications from the networking domain and explore how they could benefit from the serverless computing paradigm.

1) Software-Defined Networking: SDN promises to make communication networks more adaptive and easier to manage. The primary idea is to separate the network's control logic (the control plane) from the underlying routers and switches that forward the traffic (the data plane). With such a separation of the control and data planes, network switches become simple forwarding devices and the control logic is written in software, as network applications, running on a logically centralized controller (or network OS), simplifying policy enforcement and network (re)configuration and evolution [31].

Although forwarding in the network switches happens in an event-driven fashion (i.e., packet arrival triggers route lookup and then routing), the latency requirements of this action cannot be met by today's serverless computing technologies. On the other hand, the SDN controller is a prime candidate. A typical function of the SDN controller is the flow management on network switches. The controller exercises the direct control over the state in the switches via a well-defined API. The most notable example of such an API is OpenFlow [32], [33].

An OpenFlow switch has one or more tables of packet-handling rules (i.e., flow table). Each rule matches a subset of the traffic and performs certain actions (e.g., dropping, forwarding, and modifying) on the traffic. These rules can be installed by the controller, and depending on which rules are installed, an OpenFlow switch can behave like a router, switch, and firewall or perform other roles (e.g., load balancer, traffic shaper, and, in general, those of a middlebox).

There are three requirements of SDN controllers that make serverless computing a suitable alternative for their implementation. In the following, we present these requirements as well as how they are realized today in detail.

a) Modularity: SDN controllers are typically designed as a set of pluggable modules that provide basic networking functionality. For example, one module can provide management of flows, while another is responsible for managing the topology. These modules also provide a mechanism to add custom business logic to develop new networking services [34]. Events coming from the southbound API, such as OpenFlow, trigger the execution of these modules. For example, when a link or port change event is detected, a message is sent by the forwarding device to the controller. Similarly, a new packet with no matching rules is sent to the controller.

These modules also respond to the events received via the northbound API, which interfaces with network applications running on top of the controller. Examples of these network applications include load balancers, firewalls, other security services, and orchestration/automation applications across cloud resources, such as OpenStack [35], Puppet [36], Chef [37], and Ansible [38]. When these applications would like to update the controller and, in turn, the rules in the forwarding devices, they communicate with the controller modules that can provide the desired functionality.

b) Parallelism: The SDN controller serves events from both the southbound and northbound APIs. As a result, the rate at which these events arrive can vary significantly. A single controller may not be enough to manage a network with a large number of data plane elements. Consequently, a considerable effort has been dedicated to engineering controllers as highly concurrent, multithreaded systems to scale up to enterprise-class networks (e.g., NOX-MT [39], Maestro [40], Beacon [41], and Floodlight [42]). These systems aim to increase parallelism by minimizing synchronization needed between the modules via various mechanisms.

Furthermore, SDN controllers are also designed to be deployed in a distributed manner, either in a centralized cluster of nodes or a physically distributed set of nodes, to make them scalable and failure resilient. Onix [43], HyperFlow [44], and ONOS [45] are a few examples of such distributed event-based controllers.

c) Isolation: Another important requirement of SDN controllers is that they isolate faults within the pluggable modules without affecting the controllers. On a single node, modules operate within the same address space as that of the controller. Therefore, the controller must ensure that the faults both within these modules and the resources consumed by the modules are isolated from each other.

For many popular open-source controllers [34], [41], [42], [46], providing such an isolated environment that also performs well is still an open challenge [47]. An alternative controller, Rosemary [47], proposes to isolate network applications by starting them as processes in different containers to prevent propagation of failures through the SDN controller.

Serverless Computing in the SDN Domain: Being highly event-driven, modular, and parallel, the SDN controllers are a great fit for the serverless computing paradigm. The modules in an SDN controller can be implemented as separate, stand-alone functions and deployed on a serverless platform. Such an architecture would offer scalability and fault isolation of modules out-of-the-box. These properties mean that modules can be scaled up and down on-demand and in a resource-efficient manner, without the operators having to provision the resources for peak load in advance. As such, the developers of the SDN controllers can focus on the application logic. One caveat is that the latency requirements of a controller's southbound API could be fairly restrictive. Therefore, even though the modules interacting with the southbound API can be written as event-driven functions, developers may choose to implement them with traditional approaches.

2) Network Function Virtualization: VMs have traditionally been used to instantiate VNFs, but VMs tend to be resource heavy and have long startup latencies, particularly in scenarios where numerous short-running services are needed on demand [6]. Containers [14] are increasingly being deployed as a resource-efficient alternative [5] to VMs, but they also suffer from relatively long cold startup latencies. Consequently, containers, similar to VMs, also need to be provisioned to higher load, to handle short-term traffic spikes.

a) Serverless computing in the NFV domain: Serverless computing promises to provide a resource-efficient, low overhead alternative to VMs and containers, and can be a good fit for the NFV architecture. Latency-sensitive, long-running VNFs, such as firewalls, might not be a good fit for serverless computing. However, we envision serverless computing to be a great candidate for orchestrating VNFs as well as for applications that require running short, on-demand tasks operating on data collected from the data plane (e.g., anomaly detection). In the following, we describe the characteristics of one such anomaly detection approach [8] and motivate where serverless computing could be applied.

b) Serverless anomaly detection VNF: Kostas *et al.* [8] propose a modular anomaly detection and a distributed denial-of-service mitigation architecture that exploits the network programmability of SDN within the context of NFV. Fig. 5 shows the high-level architecture of this system. Edge routers within an enterprise network convey monitoring data to an anomaly detection and identification component that consists of multiple stages implemented as a set of tasks (shown in orange). When an attack is detected, an on-demand network function instructs the edge router to forward all traffic destined to the victim to another switch, which then filters malicious traffic while preserving benign flows.

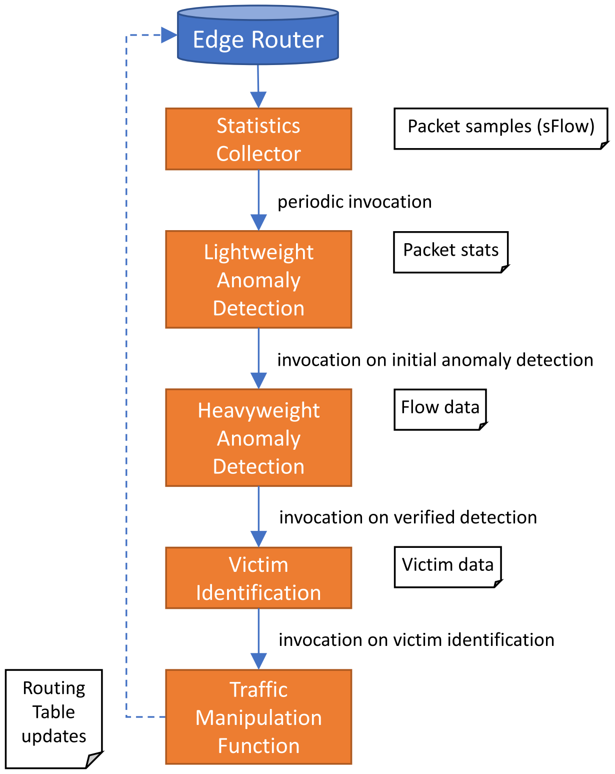
Edge Router
Statistics Collector
Packet samples (sFlow)
periodic invocation
Lightweight Anomaly Detection
Packet stats
invocation on initial anomaly detection
Heavyweight Anomaly Detection
Flow data
invocation on verified detection
Victim Identification
Victim data
invocation on victim identification
Routing Table updates
Traffic Manipulation Function

Fig. 5. ***High-level design of an event-driven architecture for network anomaly detection. It employs the invocations of different functions for anomaly detection, victim identification, and traffic manipulation. Boxes in orange can be deployed on a serverless platform.***

The computation steps required for anomaly detection are decoupled from the data plane on the edge router via the sFlow API [48], which randomly samples the packets from the data plane and reports them. The detection step is divided into a few computation tasks that are executed in a chain, as shown in Fig. 5, each with increasing computational requirements.

More specifically, the first step (i.e., statistics collector) harvests statistics from the packet samples and periodically exports them to the second component for further investigation. The second component (i.e., lightweight anomaly detection) performs coarse-grained, entropy-based anomaly detection on these statistics and informs the third component if an anomaly is detected. The third component (i.e., heavyweight anomaly detection) uses a computationally heavy bidirectional count sketch algorithm to verify the attack and then identifies the victim (i.e., victim identification). Finally, the traffic manipulation function is triggered to update the routing table of the edge router to redirect traffic away from the victim.

These computation tasks required for anomaly detection are short-lived and event-driven. We envision that such short-lived VNFs could be launched in a serverless manner, which would make these instantiations significantly resource-efficient and flexible.

c) Serverless media processing VNFs: The 5G Media project [6], [7] plans to use serverless VNFs for carrying out short tasks running in parallel with the container-based VNFs. One such short task is "repeat clip creation" for online spectators that are watching a live game being played between two players in a virtual world. A spectator may wish to replay a certain portion of the game from a specific camera angle. This task requires collecting the necessary buffered game play data and rendering it from one or more camera angles desired by a specific spectator. This replay creation task can be instantiated anytime during the game play. Although it may not be as time-sensitive as the game play itself, multiple instances of this task may be needed to be created quickly to handle a sudden increase of spectators wishing to replay a particular portion of the game (e.g., a goal event in a soccer game). All these properties make this task a good fit for the serverless computing paradigm.

d) Serverless orchestration of VNFs: Serverless functions could also be used to orchestrate multiple VNFs for short-lived sessions. For example, consider a short-lived media-intensive game between two players, where the entire game session lasts only a few minutes [6], [7]. During that session, many VNFs need to be instantiated, such as transcoding functions to process the media stream, buffering functions to buffer the last few seconds of the game for on-demand clip creation, rendering functions for the two players and for the online spectators, and the aforementioned repeat clip creation function that can be instantiated any time throughout the session. The application logic for orchestrating these VNFs can be launched in a serverless manner, without having to provision another VNF to execute this task.

V. DISCUSSION

Despite its advantages, serverless computing still needs to address certain challenges to improve its applicability across the general application landscape, including the communication services. Many of these challenges are due to the recency of serverless computing; as application developers and engineers continue using serverless computing, they notice the lack of certain features or capabilities. These issues are then addressed via additional tools. On the other hand, some challenges are more fundamental and need to be addressed by higher level design decisions and architectural considerations in the serverless computing platform. Here, we discuss these challenges and point to work that is relevant to address them.

A. New Programming Model

To take full advantage of serverless computing benefits, applications need to be designed with a new programming model that is based on events and asynchronous calls. Designing an application as a group of stateless functions requires a new mindset, in which all shared state must be externalized. This requirement may introduce additional overhead and constraints that might not exist in traditional, server-based application development. On the one hand, if the functions are too fine-grained, then the overhead of interacting with many other functions will affect performance. On the other hand, if the functions are too big, the flexibility about management and scaling will be limited.

In the context of SDN controllers, serverless computing is actually a good fit, since a significant effort has already been put in to engineer SDN controllers to be highly event-driven, modular, and concurrent (with minimal sharing of state between the modules). For NFVs, though they are also event-driven (reacting to packet arrival event), serverless computing still presents significant startup latency overheads (Section III-C) to be generally applicable. As discussed in Section IV-B2, serverless computing can be a great candidate for orchestrating VNFs and also for applications that require running on-demand short tasks that operate on data collected from the data plane, such as for anomaly detection [8].

B. Distributed Testing and Debugging

Another important aspect to consider when building serverless applications is testing and debugging of the applications. Individual functions might become easier to test because their logic is self-contained. On the other hand, testing and debugging applications consisting of several functions might become more difficult for the developers, because they do not usually have access to the underlying physical servers running the serverless computing platform. As a result, it becomes crucial to the operator of the platform providing capabilities to log, test, and debug serverless applications. Graphical user interfaces can provide developers with an overview of the functions and their interactions [26]. Locally running tools can help developers test their functions before deploying them [49], [50].

In communication services such as SDNs or NFVs, it is possible the platform provider and developer of the functions may very well be the same entities. As a result, the developers can have access to the physical servers for testing, debugging, and optimization purposes in addition to the benefits of serverless applications. Even in these cases, visibility into the platform without requiring access to the physical servers would significantly speed up the development process.

C. Load Balancing Versus Locality

Load balancing is beneficial for the operator of a serverless platform to increase resource utilization by distributing the function executions to available resources.

Unfortunately, such load balancing schemes may not necessarily improve the performance for applications consisting of several functions due to the loss of locality. For example, frequently interacting functions may share data faster if they were to be on the same physical server. Since load balancing decisions are typically hidden from the developer, it is possible that load balancing strategies that do not consider such interactions may introduce undesired latency and network overhead by assigning the function executions belonging to the same session to multiple servers. For communication services, since most of the network functions run inside the network, it is likely that they are placed and executed, separated from each other in a distributed network.

Such overhead might not be acceptable for latency-sensitive communication services and will require careful placement of VNFs by allowing developers to specify placement hints. For example, in SAND [28], developers can indicate locality requirements by grouping functions as a single application. The platform then ensures that these functions run within the same container, and the platform also provides the local message-passing mechanism to speed up communication between the colocated functions.

D. Startup Latency

As described in Section III-A, cold starts can introduce long delays for applications and have been identified as an important concern [51]–[53]. Warm starts, where idle containers from the previous executions are reused, help mitigate long tail latencies but come at a cost of reduced resource efficiency for the platform operator. Furthermore, even with warm starts, startup delays can accumulate to a large value, when building applications that consist of several functions that interact with each other (Section III-C).

For communication services, latency overhead is a significant concern, and long startup latencies will narrow down the range of applications that can benefit immediately from serverless computing. In the last few years, a significant research effort has been directed toward reducing startup latencies, which will make serverless computing a viable option for a larger range of applications. We describe some of these research directions in the following.

One line of work optimizes the underlying, most popularly used virtualization technique, namely, containers. Reducing container startup times can have a direct impact on invocation latencies, especially when a new container needs to be launched during cold starts, as well as resource efficiency—making cold starts fast enough would mean that containers need not be kept idle. Slacker [54] identifies the most critical library packages during the initial load of a container so that they can be prioritized to speed up the start of a function execution. PipSqueak [55] and SOCK [29] optimize the containers for serverless functions by keeping a cache of Python interpreters that preload libraries required by the functions. CNTR [56] reduces the container image size by separating the tools needed for testing and debugging from the application code so that the container load times are improved.

Another line of work takes a different approach by rethinking the architecture of the Serverless platform and adapting a different type of virtualization than function-in-a-container. With unikernels [19], [20], [57], each function is compiled into a custom system software, which is then invoked with every request. Because the unikernel is customized to the function requirements, the overhead associated with unnecessary functionality is minimized to the point where unikernel-based VMs can launch faster than containers [18]. SAND [28] employs a fine-grained sandboxing mechanism, whereby functions belonging to the same application are run in the same sandbox as separate processes and new function executions are created via OS forking, which is significantly faster than launching a new container. Boucher *et al.* [58] exploit language runtimes to provide the isolation between function executions rather than using today's virtualization techniques.

E. Legacy Applications

A general question for the serverless computing approach is how can legacy applications be supported. Decomposing application logic into smaller functions may require rearchitecting the application, which may not be a trivial undertaking. The first thing that would be considered is whether such a redesign would make sense economically. Although serverless computing may decrease operational costs, companies may have already invested in the traditional application (e.g., physical servers and developer effort). Furthermore, as with designing new applications, performance is critical. An open question is how legacy applications can be decomposed and run without sacrificing performance [59].

To be feasible and incrementally applicable, such a redesign would lead to a hybrid architecture, whereby the application would be decomposed into parts that can run in serverless manner and parts that will run as it is today. Achieving this transformation would require identifying the application components that need high elasticity the most, as well as ensuring that they can satisfy the serverless computing requirements (i.e., externalized state without losing performance). As described in Section IV-E2, we foresee such a hybrid architecture to be a good starting point for NFVs to explore serverless computing.

VI. CONCLUSION

Fueled by the need to evolve network services at a similar speed as today's cloud computing services, both SDN and NFV push for a fully software-based approach to networking. Networking functionality is implemented in software modules and executed on a suitable cloud computing platform. The virtualization paradigm used in the design of virtualized services and the corresponding computing platforms will play an important role in the transition to software-based networks. They need to provide the required agility, robustness, and scalability for the services executed.

In this paper, we explored the possibility of using serverless computing as a candidate platform for networking services which meets the aforementioned requirements. For the cloud computing industry, serverless computing has already significantly altered the economics of offloading computations to the cloud and is considered a rapidly deployable (redeployable) alternative to VMs and containers, without requiring any additional configuration overhead on developers.

For deploying networking services, we show that the serverless computing paradigm is conceptually a great fit for implementing SDN controllers and, practically, serverless computing also offers the scalability and resource isolation needed by these controllers. NFV can also greatly benefit from serverless computing, and we discuss, with examples, how VNFs that are not on the latency-critical path can be implemented in a serverless manner. Latency-critical VNFs are currently not suited for serverless adoption, but numerous research efforts show a promising improvement on the latency overheads of serverless platforms. We are confident that in the near future, serverless computing will become a viable option for a vast range of communication services, and service providers should seriously consider using the serverless computing paradigm for implementing new networking services. ■

REFERENCES

[1] D. Kreutz, F. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[2] B. Yi, X. Wang, S. K. Das, K. Li, and M. Huang, "A comprehensive survey of network function virtualization," *Comput. Netw.*, vol. 133, pp. 212–262, Mar. 2018.

[3] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 203–216.

[4] (Sep. 2018). *SDN: Time to Move on, Gartner Says*. [Online]. Available: https://www.networkcomputing.com/networking/sdn-time-move-gartner-says/927734697

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

[6] (Jun. 2018). *5G Media Project*. [Online]. Available: http://www.5gmedia.eu/

[7] (Jun. 2018). *Programmable Edge-to-Cloud Virtualization Fabric for the 5G Media Industry*. [Online]. Available: http://www.5gmedia.eu/cms/wp-content/uploads/2018/03/5G-MEDIA-D3.1-Initial-Design-of-the-5G-MEDIA-Operations-and-Configuration-Platform_v1.0.pdf

[8] G. Kostas, A. George, and M. Vasilis, "A scalable anomaly detection and mitigation architecture for legacy networks via an OpenFlow middlebox," *Secur. Commun. Netw.*, vol. 9, no. 13, pp. 1958–1970, 2016.

[9] (Jun. 2015). *AWS Lambda—Serverless Compute*. [Online]. Available: https://aws.amazon.com/lambda/

[10] (Dec. 2017). *IBM Cloud Functions*. [Online]. Available: https://www.ibm.com/cloud/functions

[11] (Jan. 2018). *Microsoft Azure Functions*. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[12] (Jan. 2018). *Google Cloud Functions*. [Online]. Available: https://cloud.google.com/functions/

[13] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[14] (Jun. 2015). *Docker*. [Online]. Available: https://www.docker.com/

[15] (Jan. 2018). *LXC (Linux Containers)*. [Online]. Available: https://en.wikipedia.org/wiki/LXC

[16] (Jan. 2018). *Cgroups (Control Groups)*. [Online]. Available: https://en.wikipedia.org/wiki/Cgroups

[17] (Jan. 2018). *Linux Namespaces*. [Online]. Available: https://en.wikipedia.org/wiki/Linux_namespaces

[18] F. Manco *et al.*, "My VM is lighter (and safer) than your container," in *Proc. 26th Symp. Oper. Syst. Princ. (SOSP)*, 2017, pp. 218–233.

[19] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Distrib. Comput.*, vol. 11, no. 11, pp. 30:30–30:44, 2014.

[20] A. Madhavapeddy *et al.*, "Unikernels: Library operating systems for the cloud," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2013, pp. 461–472.

[21] (Dec. 2017). *Erlang Programming Language*. [Online]. Available: https://www.erlang.org/

[22] (Jun. 2017). *Amazon API Gateway*. [Online]. Available: https://aws.amazon.com/api-gateway/

[23] T. Wagner. (Dec. 2014). *Understanding Container Reuse in AWS Lambda*. [Online]. Available: https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[24] (Apr. 2017). *Squeezing the Milliseconds: How to Make Serverless Platforms Blazing Fast!* [Online]. Available: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

[25] (Dec. 2017). *Serverless Reference Architecture: Image Recognition and Processing Backend*. [Online]. Available: https://github.com/awslabs/lambda-refarch-imagerecognition/

[26] (Jun. 2017). *AWS Step Functions*. [Online]. Available: https://aws.amazon.com/step-functions/

[27] (Jun. 2017). *Apache OpenWhisk*. [Online]. Available: http://openwhisk.apache.org/

[28] I. E. Akkus *et al.*, "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 923–935.

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

[30] (Jan. 2018). *Lambda@Edge*. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

[31] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[32] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[33] (Jan. 2018). *Open Networking Foundation*. [Online]. Available: https://www.opennetworking.org

[34] (Jun. 2018). *OpenDaylight Project*. [Online]. Available: https://www.opendaylight.org

[35] (Jun. 2018). *OpenStack*. [Online]. Available: https://www.openstack.org/

[36] (Jun. 2018). *Puppet*. [Online]. Available: https://puppet.com

[37] (Jun. 2018). *Chef*. [Online]. Available: https://www.chef.io

[38] (Jun. 2018). *Ansible*. [Online]. Available: https://www.ansible.com/

[39] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. 2nd USENIX Conf. Hot Topics Manage. Internet, Cloud, Enterprise Netw. Services (Hot-ICE)*, 2012, p. 10.

[40] (Jun. 2018). *Maestro: A System for Scalable OpenFlow Control*. [Online]. Available: https://www.cs.rice.edu/ eugeneng/papers/TR10-11.pdf

[41] D. Erickson, "The beacon openflow controller," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 13–18.

[42] (Jun. 2018). *Floodlight OpenFlow Controller*. [Online]. Available: http://www.projectfloodlight.org/floodlight

[43] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2010, pp. 351–364.

[44] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proc. Internet Netw. Manage. Conf. Res. Enterprise Netw. (INM/WREN)*, 2010, p. 3.

[45] P. Berde *et al.*, "ONOS: Towards an open, distributed SDN OS," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 1–6.

[46] (Jun. 2018). *POX Network Software Platform*. [Online]. Available: https://github.com/noxrepo/

[47] S. Shin *et al.*, "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (ACM CCS)*, 2014, pp. 78–89.

[48] (Jun. 2018). *sFlow*. [Online]. Available: https://sflow.org/

[49] (Jun. 2018). *AWS SAM CLI*. [Online]. Available: https://github.com/awslabs/aws-sam-cli

[50] (Oct. 2017). *Serverless Framework*. [Online]. Available: https://serverless.com/

[51] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2016, pp. 33–39.

[52] (Jan. 2018). *CNCF WG-Serverless Whitepaper V1.0*. [Online]. Available: https://github.com/cncf/wg-serverless

[53] G. McGrath, J. Short, S. Ennis, B. Judson, and P. Brenner, "Cloud event programming paradigms: Applications and analysis," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 400–406.

[54] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 181–195.

[55] E. Oakes, L. Yang, K. Houck, T. Harter,

A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Pipsqueak: Lean lambdas with large libraries,” in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 395–400.

[56] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, “CNTR: Lightweight OS containers,” in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 199–212.

[57] R. Koller and D. Williams, “Will serverless end the dominance of linux in the cloud?” in *Proc. 16th Workshop Hot Topics Oper. Syst. (HotOS)*, 2017, pp. 169–173.

[58] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Putting the ‘micro’ back in microservice,” in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 645–650.

[59] G. McGrath and P. R. Brenner, “Serverless computing: Design, implementation, and performance,” in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 405–410.

ABOUT THE AUTHORS

Paarijaat Aditya received the Ph.D. degree from Max Planck Institute for Software Systems, Saarbrücken, Germany.

Before starting his Ph.D. degree, he was a Member of Technical Staff with Adobe Systems India, Noida, India. He is currently a Researcher with the Autonomous Software Platforms Team, Nokia Bell Labs, Stuttgart, Germany. He has published and presented his works at several conferences, including USENIX NSDI, USENIX OSDI, ACM MobiSys, and ACM IMC. His current research interests include intersection of cloud computing, mobile computing, deep learning, and applications of cryptography.

Istemi Ekin Akkus received the Ph.D. degree from Max Planck Institute for Software Systems, Saarbrücken, Germany, in cooperation with the Technical University of Kaiserslautern, Kaiserslautern, Germany.

In the past, he was involved in Web privacy, data recovery for Web applications, and peer-to-peer systems. He is currently a Researcher with Nokia Bell Labs, Stuttgart, Germany. He is also involved in designing, analyzing, and implementing systems. His research was published in several conferences, including Middleware, CoNEXT, SIGCOMM, the ACM Conference on Computer and Communications Security, and the IEEE/IFIP International Conference on Dependable Systems and Networks. His current research interests include computer networks and distributed systems with a current focus on cloud computing.

Andre Beck received the M.S. degree in computer science from the University of Mannheim, Mannheim, Germany.

He joined Bell Labs, Holmdel, NJ, USA, and then moved to Naperville, IL, USA, in 2006. He is currently a Researcher in the Application Platforms and Software Systems Research Lab, Nokia Bell Labs, Stuttgart, Germany. He has several publications in peer-reviewed conferences and journals, including two IETF RFCs. He holds six patents. His current research interests include the area of networked software systems, in particular cloud computing and content distribution networks.

Ruichuan Chen received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2009.

He was a Postdoctoral Researcher at Max Planck Institute for Software Systems, Saarbrücken, Germany. He is currently a Researcher with Nokia Bell Labs, Stuttgart, Germany. His works have been published at various prestigious venues, including SIGCOMM, OSDI, NSDI, CoNEXT, USENIX ATC, the ACM Conference on Computer and Communications Security, and the IEEE International Conference on Computer Communications. His current research interests include the areas of networking and distributed systems, in particular cloud computing and data analytics.

Volker Hilt (Senior Member, IEEE) received the M.S. degree in commercial information technologies and the Ph.D. degree in computer science from the University of Mannheim, Mannheim, Germany, in 1996 and 2001, respectively.

In 2002, he joined Bell Labs, Holmdel, NJ, USA, and then moved to Stuttgart, Germany, in 2012. He is currently a Bell Labs Fellow and the Senior Director of Nokia Bell Labs, Stuttgart, where he leads research teams on autonomous software systems in Germany, Israel, and Ireland. He has published over 100 peer-reviewed papers, Internet drafts, and RFCs. He holds over 30 patents. He has designed the overload control mechanism for session initiation protocol (SIP), which has become a key part of today’s telecommunication systems. His current research interests include networked software systems, in which he has made contributions in the areas of cloud computing, distributed multimedia systems, content distribution networks, peer-to-peer applications, and SIP.

Ivica Rimac received the Ph.D. degree in electrical engineering and information technology from the Darmstadt University of Technology, Darmstadt, Germany.

He is currently a Distinguished Member of Technical Staff and the Head of the Autonomous Software Platforms Research Department, Nokia Bell Labs, Stuttgart, Germany. In 2005, he joined Bell Labs, Holmdel, NJ, USA, and then moved to Stuttgart in 2011. He has published numerous papers and patents in the areas of cloud computing, network function virtualization, and content caching and distribution. His current research interests include the fields of computer networking and distributed systems.

Klaus Satzke (Member, IEEE) received the Diploma and Ph.D. degrees in physics from the Philipps-University of Marburg, Marburg, Germany, in 1990.

He spent postdoctoral fellowships in Paris, France, and the U.K. In 1992, he joined the Alcatel SEL Research Centre, Stuttgart, Germany, where he was responsible for the research projects in the context of prototype realization, measurement, and performance evaluations for broadband transmission system. He is currently a Researcher with Nokia Bell Labs, Stuttgart. His current research interests include cloud computing, content distribution, infrastructure virtualization, and computer system performance analysis.

Dr. Satzke is a member of the Deutsche Physikalische Gesellschaft.

Manuel Stein received the B.S. degree in computer science from the University of Cooperative Education, Stuttgart, Germany, in 2005.

In 2002, he has joined Alcatel SEL, Stuttgart. He is currently a Researcher with Nokia Bell Labs, Stuttgart. His current research interests include heterogeneous access management, distributed service delivery platforms for IP Multimedia Subsystem (3GPP IMS) evolution, network virtualization, semantic web, autoscaling, network management, service chaining in distributed cloud infrastructures, infrastructure virtualization, network science, distributed computing system design, scheduling, machine learning, and reliability and performance analysis.