

NFV Platforms: Taxonomy, Design Choices, and Future Challenges

Tianzhu Zhang, Han Qiu, *Member, IEEE*, Leonardo Linguaglossa, *Member, IEEE*,
Walter Cerroni, *Senior Member, IEEE*, Paolo Giaccone, *Senior Member, IEEE*,

Abstract—Due to the intrinsically inefficient service provisioning in traditional networks, Network Function Virtualization (NFV) keeps gaining attention from both industry and academia. By replacing the purpose-built, expensive, proprietary network equipment with software network functions consolidated on commodity hardware, NFV envisions a shift towards a more agile and open service provisioning paradigm. During the last few years, a large number of NFV platforms have been implemented in production environments which typically face critical challenges, including the development, deployment, and management of Virtual Network Functions (VNFs). Nonetheless, just like any complex system, such platforms commonly consist of abounding software and hardware components and usually incorporate disparate design choices based on distinct motivations or use cases. This broad collection of convoluted alternatives makes it extremely arduous for network operators to make proper choices. Although numerous efforts have been devoted to investigating different aspects of NFV, none of them specifically focused on NFV platforms or attempted to explore their design space. In this paper, we present a comprehensive survey on the NFV platform design. Our study solely targets existing NFV platform implementations. We begin with a top-down architectural view of the standard reference NFV platform and present our taxonomy of existing NFV platforms based on what features they provide in terms of a typical network function life cycle. Then we thoroughly explore the design space and elaborate on the implementation choices each platform opts for. We also envision future challenges for NFV platform design in the incoming 5G era. We believe that our study gives a detailed guideline for network operators or service providers to choose the most appropriate NFV platform based on their respective requirements. Our work also provides guidelines for implementing new NFV platforms.

Keywords—*Network Function Virtualization, Service Function Chaining, Service Management and Orchestration, NFV Infrastructure, VNF Life Cycle*

I. INTRODUCTION

Traditionally, network services are provisioned using purpose-built, proprietary hardware appliances (or middle-

boxes). Middleboxes embody a large variety of specialized functions to forward, classify, or transform traffic based on packet content. Examples of middleboxes include L2 switching, L3 Routing, Network Address Translation (NAT), Firewall (FW), Deep Packet Inspection (DPI), Intrusion Detection System (IDS), Load Balancers (LB), WAN optimizers, and stateful proxies. Nowadays, middleboxes are ubiquitous in enterprise networks [1]. With the increasingly diversified user requirements, as well as the rapid growth of Internet traffic in terms of both volume and heterogeneity [2], hardware middleboxes begin to exhibit several fundamental disadvantages. First off, middleboxes are generally expensive to acquire and usually require domain-specific knowledge to manage them, resulting in large capital expenditure (CapEx) and operational expenditure (OpEx). Also, adding customized functionalities is extremely time-consuming if not impossible, and it sometimes takes an entire purchase cycle (e.g., four years) to bring in equipment with new features [3]. Such tight coupling with the hardware production cycle considerably hampers network innovation and prolongs time-to-market. Deploying new Network Services (NSs) is also a tedious process, as technicians are required to visit specific sites and place the middleboxes in a pre-defined order to form the correct Service Function Chains (SFCs). Service instantiation might even take days. Worse still, service maintenance usually involves constant repetition of the same process. Furthermore, because of the inherent inflexibility, it is not trivial for hardware middleboxes to elastically scale in and out based on the shifting demand or other system dynamics. Consequently, network operators usually resort to peak-load (over-)provisioning, which in turn leads to inefficient resource utilization and high energy consumption.

To improve resource utilization and overcome the network ossification, telecommunication operators began to pursue new solutions that can guarantee both cost-effectiveness and flexibility. The advent of Software Defined Networking (SDN) [4]

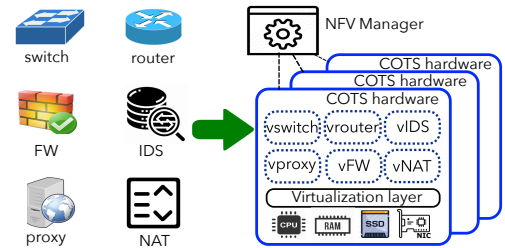


Fig. 1: Traditional vs. NFV paradigm

T. Zhang was with the Department of Network and Computer Science, Telecom Paris. He is now with Nokia Bell Labs, 91620 Nozay, France. (email: tianzhu.zhang@nokia-bell-labs.com)

H. Qiu and L. Linguaglossa are with the Department of Network and Computer Science, Telecom Paris, 91120 Paris, France. (email: han.qiu@telecom-paris.fr, linguaglossa@telecom-paris.fr)

W. Cerroni is with Department of Electrical, Electronic, and Information Engineering "Guglielmo Marconi" of University of Bologna, 40126 Bologna BO, Italy. (email: walter.cerroni@unibo.it)

P. Giaccone is with the Department of Electronics and Telecommunications, Politecnico di Torino, 10138 Torino TO, Italy. (email: paolo.giaccone@polito.it)

and Network Function Virtualization (NFV) [5] opened new alternative approaches for network management and service provisioning. SDN decouples the control plane from the data plane and leverages a logically centralized controller to configure the programmable switches based on a global view, while NFV aims at replacing specialized middleboxes with software-based Virtual Network Functions (VNFs) deployed on Commodity Off-The-Shelf (COTS) hardware. The key to their success lies in separating the evolution timeline of software network functions from that of specialized hardware, completely unleashing the potential of the former. An illustrative example contrasting the NFV paradigm with traditional network infrastructures is shown in Fig. 1. Compared to the traditional service provisioning paradigm based on hardware middleboxes, NFV manages to achieve cost-effectiveness by leveraging multiple instances of VNFs on high-volume yet less expensive COTS servers, routers, or storage. Service provisioning in NFV is thus highly simplified, as the previously mentioned troublesome tasks, such as middlebox deployment, monitoring, migration, and scaling, can be optimally automated and flexibly managed through software control mechanisms. It is thus convenient for NFV solutions to exploit available resources and management tools typical of cloud or edge computing infrastructures. In addition, NFV remarkably promotes network innovation and accelerates the time-to-market process as network function development is cut down to writing software programs using standard application programming interfaces (APIs).

Thanks to these benefits, NFV keeps gaining momentum from both industry and academia. The first concerted effort towards NFV standardization began in 2012, with the appointment of the Industry Specification Group on NFV as part of the European Telecommunications Standards Institute (ETSI) [6]. Currently, ETSI consists of more than 500 members across the world, including major telecommunication operators, service providers, manufacturers, as well as universities. Meanwhile, the continuous advancement of COTS hardware capabilities and the emergence of high-speed packet processing techniques have managed to reduce the previously huge performance gap between software network functions and specialized middleboxes. Resources of other hardware components, such as Graphics Processing Unit (GPU) and in-path programmable network devices, can also be exploited to share the workload and alleviate the CPU burden. These technical opportunities considerably stimulate the growth of NFV and foster its adoption by telecom operators. During the last eight years, a large variety of NFV platforms have been developed and implemented to spur the innovation and evolution of NFV.

However, just like any complex system, existing NFV platforms usually encompass many closely interacting software and hardware components and embrace divergent design choices driven by their respective motivations, use cases, and application fields. To deal with the non-trivial network function life cycle, the design space of these platforms can be very wide, with choices ranging from high-level VNF development to low-level infrastructure details. The former category includes VNF execution models, state management schemes, or genres of APIs, while the latter category includes system design choices like packet I/O frameworks, VNF interconnecting

methods, or virtualization technologies, as well as various datapath acceleration techniques such as batch processing, zero-copy packet transfer, data prefetching, and computation offloading. Such a broad range of platform implementations coupled with even more extensive design space makes it extremely difficult (if not impossible) for network operators to choose the most suitable solution to their needs. The tradeoffs and caveats between different design choices are also unclear, making new platforms laborious and error-prone to implement.

This paper presents a comprehensive survey of existing NFV platforms and their design. The main contributions can be summarized as follows:

- We conduct a literature review and classify the existing NFV platforms according to what features they provide in terms of a typical VNF life cycle. We also briefly review the internals of each platform.
- We explore the NFV design space and discuss the various design choices adopted by existing platforms.
- We discuss potential challenges of bringing Artificial Intelligence (AI), network slicing, and Internet of Things (IoT) into NFV.

Several existing literature surveys investigated some particular aspects of NFV, including VNF placement [7], resource allocation [8], fault management [9], service function chaining [10], and security [11], but none of them specifically focused on the design aspects of NFV platforms from a VNF life cycle perspective, nor did they attempt to explore the design space or review different implementation choices. In [12], the authors investigated several industrial NFV Management and Orchestration (MANO) projects, whereas our work additionally considers the NFV management frameworks from academia. In [13] and [14], a subset of the state-of-the-art NFV platforms were generally reviewed, while our work considers, to the best of our knowledge, all the existing platforms and focuses on their tailored implementation.

This paper is organized as follows: in Sec. II, we give an architectural overview of the components of NFV platforms. Then we present our taxonomy on existing platforms in Sec. III based on a typical VNF life cycle. In Sec. IV, we propose a collection of critical design choices and survey the solutions adopted by different platforms. We envision future directions and draw our conclusion in Secs. V and VI, respectively.

II. NFV PLATFORM: AN ARCHITECTURAL OVERVIEW

We devote this section to present a general architectural overview of a typical NFV platform and to review the key components in depth. Although a reference architecture has been defined by the ETSI specification [15], most of the existing NFV platforms did not strictly follow it. For example, some key ETSI components were not implemented by some industrial NFV platforms [12], whereas others implementations focused only on partial management aspects. As a result, we seek to combine the ETSI reference architecture with those of the existing platforms and present a generic view, as illustrated in Fig. 2. An NFV platform generally consists of three primary components, namely the NFV MANO plane, the service plane, and the NFV Infrastructure (NFVI). The MANO

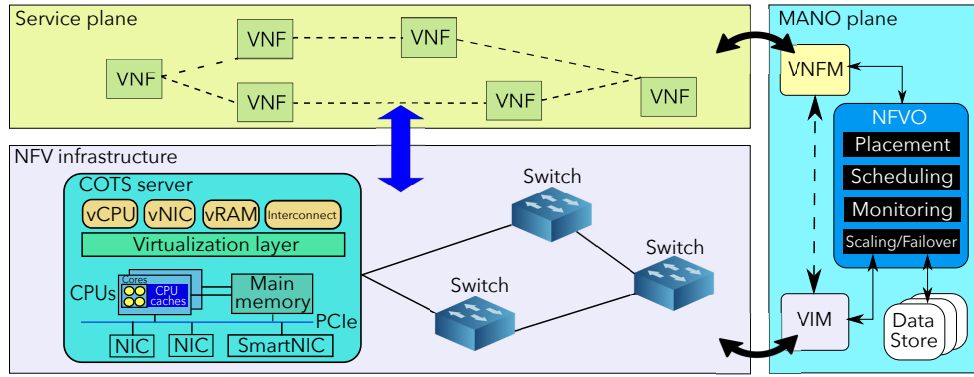


Fig. 2: The architecture of a general NFV platform

plane provides centralized control over service provisioning and management. The NFVI contains a collection of computation, storage, and network resources that are distributed across different infrastructural nodes. MANO plane components systematically monitor and schedule the resources to build the virtualized environment and accommodate different network services. The service plane contains a diversified collection of VNFs that are ordered in the form of service chains to fulfill the desired network services. These service chains are also carefully monitored and adjusted by the MANO plane components to efficiently multiplex the NFVI resources. In general, the service plane is enabled through the concerted operations from both MANO plane and NFVI.

A. MANO plane

The NFV MANO plane is the central point for service provisioning in NFV. A MANO system typically consists of three sub-systems: NFV Orchestrator (NFVO), Virtualized Infrastructure Manager (VIM), and VNF Manager (VNFM). As shown in Fig. 2, the NFVO is responsible for the instantiation, management, and termination of network services. At present, an NFVO commonly encompasses different modules to apply different MANO operations. On the right part of Fig 2, we illustrate four example modules. The placement module is in charge of rendering the best deployment, possibly in an incremental fashion. When new services need to be deployed, the placement module analyzes the service descriptions or requirements specified by network operators, constructs an aggregated service representation (e.g., service processing graph), performs necessary optimizations (e.g., function merging, redundant elimination), and calculates the best possible placement strategy by determining the nodes on which to deploy the related VNFs and their chaining order. The monitoring module is responsible for collecting statistics and events from both the service plane and the infrastructure, and provides runtime feedback to other NFVO modules. Based on the traffic condition and resource utilization collected on-the-fly, the placement module can recalculate a new placement to achieve better performance. The scheduling module can dynamically make fine-grained resource allocation to attain

resource efficiency. The scaling/failover module can also collaborate with the placement module to scale in/out particular VNFs or service chains to accommodate traffic fluctuations, or instantiate new VNF replicas upon failure. Based on the decisions made by the aforementioned modules, the NFVO interacts with other MANO plane components to realize the intended service configurations and resource allocations.

The VIM is designed to configure infrastructure components to accommodate the heterogeneous VNFs or service chains instantiated in the service plane. In specific, it directs the provision/release/upgrade of NFVI resources and manages the mapping between virtual and physical resources. It also manages the data path for network services by creating/deleting/updating virtual interfaces and logical links, and collects the NFVI software and hardware resource status on behalf of the NFVO monitoring module. Note that a VIM instance might control either all the resources of the whole NFVI or those of multiple NFVI-Nodes. In some cases, a VIM might also just control a specific type of resource.

On the other hand, VNFM interacts with the service plane and takes care of the lifecycle (i.e., instantiating, scaling, upgrading, and terminating) of individual VNFs and service chains. It also needs to synchronize with VIMs to allocate or release the related infrastructural resources. According to ETSI specification, the MANO system might also maintain several data stores to hold configuration information such as network service descriptors, VNF templates, NFVI resource repositories, etc.

B. NFV Infrastructure

The NFV Infrastructure (NFVI) contains all the essential hardware and software components to compose virtual network services. The infrastructure might belong to Internet service providers, cloud/edge operators, or simply infrastructure providers. It usually embodies a large variety of computing nodes and network equipment. Each computing node or network equipment is commonly referred to as NFVI-Node. Network equipment in NFVI can be traditional purpose-built switches/routers or the emerging programmable switches that can be remotely orchestrated with SDN or P4 [16] semantics. The most typical form of computing nodes in NFVI is

represented by COTS servers. These servers normally contain several critical hardware components, including multicore CPUs, the main memory, and the physical Network Interface Controllers (NICs), which are interconnected through PCI buses. The physical NICs are capable of operating at Gigabit rates with multiple queues promoting parallelization. High-speed packet I/O techniques are also integrated by the NICs to transport packets to the service plane. Inside the server, multicore CPUs are distributed across non-uniform memory access (NUMA) nodes to speed up traffic processing. Aside from CPU, other computing units such as smartNICs and GPUs are also widely utilized by existing NFV platforms to further boost performance. The virtualization layer in the COTS server provides the environment to accommodate network functions. The virtualization can be at the hardware level relying on bare-metal hypervisors or at OS level using container engines. Some platforms even execute network functions as ordinary processes, which are addressed as Physical Network Functions (PNFs) in some works. In this paper, we universally refer to them as VNFs for the sake of simplicity. To ensure efficient communication between the VNFs and the external network, virtual interconnects need to be precisely established. This is typically accomplished using state-of-the-art software-based virtual switches or customized forwarding tables. Note that we consider physical links between COTS servers and network equipment as part of the NFVI as well.

C. Service plane

The service plane is populated with a variety of VNFs implementing different processing to constitute various network services. The distribution of VNFs inside virtual environments is quite flexible. For instance, a VNF or a whole service chain can be mapped to a single VM for execution, but a VNF can also be split into finer-grained processing elements and deployed across multiple NFVI-Nodes. In addition, VNFs are usually constructed using different programming abstractions and operate in different runtime execution models. Some platforms provide complete primitives to build and manage stateful VNFs or SFCs.

III. TAXONOMY OF NFV PLATFORMS

To provide a consistent taxonomy of NFV platforms, we consider the typical life cycle of network functions, as shown in Fig. 3. The initial step is *prototyping*, which produces a first instance of the desired network function typically running in a controlled environment. Then it follows the extensive *testing* and *benchmarking* phase to validate the correctness and performance of the implemented function. Afterward, during the *deployment* phase, the network function is instantiated to execute in the production environment. Finally, *management* and *execution* of the newly created network function can occur at runtime and along its entire life span. Based on our literature review and the reference architecture, we observed that most of the principles and objectives of existing platforms are shared among different implementations, despite that such growing

ecosystem has not followed any predefined direction. Therefore, we opted to classify existing NFV platforms according to their main focus for the VNF life cycle.

The remainder of this section is organized as follows. Sec. III-A discusses the aspects related to the design and implementation of network functions. As a representative of this category, we have integrated NFV platforms providing a complete suite of solutions for end-to-end service provisioning, which involves the adoption of concepts such as modularity and reusability. In Sec. III-B we discuss the testing and benchmarking features of NFV platforms. For instance, MANO systems that tackle various issues such as the correctness and the performance bottleneck assessments fall in this category. Sec. III-C contains NFV platforms that specifically target the problem of resource allocation, service composition, and scaling and migration of network functions, as represented by MANO systems for service and resource management. Finally, in Sec. III-D we analyze (i) how NFV platforms can specialize to provide unique management capabilities (e.g., service composition systems to supply high-level APIs, failover recovery, scheduling, and monitoring of the running service instances) and (ii) the execution of network functions (i.e., optimized runtimes for VNF/SFC execution) and performance improvements (i.e., NFVI acceleration platforms aimed at optimizing the critical data path involving diversified hardware and software components).

A. Prototyping

The softwarization of network functions has massively reduced the time-to-market of VNFs. Consequently, rapid prototyping of functions and services plays an important role in the growing popularity of NFV platforms. Furthermore, the software nature of network functions allows the programmer to borrow concepts from regular software products such as reusability and modularity. Thanks to the *reusability* of network functions, VNF engineers can avoid “reinventing the wheel” by accessing a collection of pre-developed components (either from the open-source communities or after purchasing a license) that suits their need. Most of NFV platforms adopt also a *modular* approach, i.e., they allow VNF developers to arrange and compose existing modules to perform complex procedures. Another aspect linked to rapid VNF prototyping is the *complexity* of the underlying infrastructure. To achieve efficient packet processing, developers usually resort to the callback-based asynchronous programming model. However, this model presents several drawbacks. Firstly, it makes VNF state maintenance more complex [17]. Secondly, callback functions are unintuitive for developers to trace VNF execution. Third, callback functions introduce redundant error handling code [18]. It is common for NFV platforms to employ advanced techniques to hide such complexities and relief programmers from this burden. We classify in this section all efforts that specifically target rapid prototyping of NFV applications, by facilitating the access to shared resources, simplifying the network function composition, and providing complexity abstractions to VNF engineers.

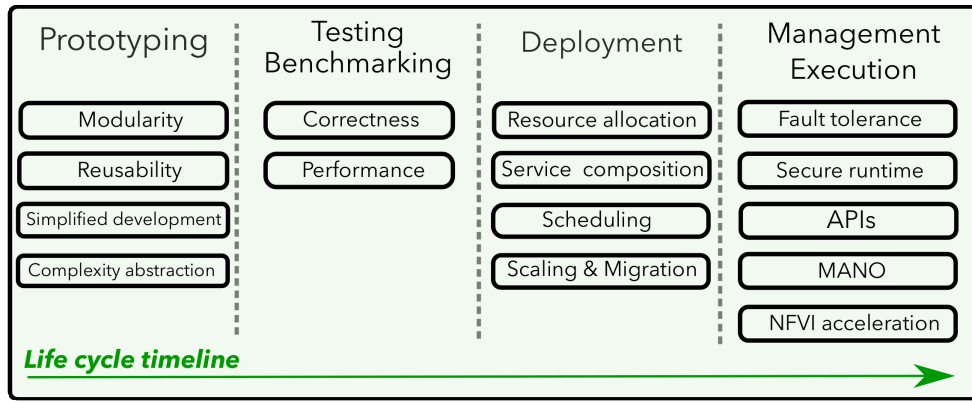


Fig. 3: Our taxonomy is inspired by the typical life cycle of network functions, namely: prototyping, testing and benchmarking, deployment, management and execution. We highlight in boxes some of the principles and objectives, and their main positioning within the network function life cycle.

Modularity and reusability: A typical class of work that adopts the modular and reusable approach for VNF prototyping is represented by the *integrated NFV platforms*, which are commonly adopted to provide high-level abstractions to compose network services, and frameworks to implement VNFs and SFCs. There exist several prior industrial projects specialized for integrated NFV platforms. For instance, **CloudBand** [19] is a carrier-grade NFV platform from Nokia. It consists of two components: the CloudBand Management System and the CloudBand Node. The former functions as the NFVO that interfaces with the latter through standard OpenStack APIs. CloudBand node integrates the OpenStack Platform as VIM and other Red Hat virtualization solutions to construct NFVI. CloudBand also implements different abstractions that support network services optimized for distributed cloud infrastructure and VNF lifecycle management. **CloudNFV** [20] is an open-source NFV platform based on cloud computing and SDN. It employs a data model named “active virtualization” to represent network services and infrastructural resources. Based on the current resource usage and specific service profiles, CloudNFV’s MANO plane deploys VNFs to the most suitable NFVI PoPs and configures the corresponding routes. The MANO plane also inspects resources and traffic conditions in real-time to make orchestration decisions according to pre-defined management policies. **SONATA** [21] implements a service development toolchain for service composition and integrates a service platform and a modular orchestration system to deploy and manage network services. **OPNFV** [22] is a Linux Foundation project which integrates several open-source sub-components for the development of NFV systems. It can provide a large variety of tasks including continuous components integration, function verification, performance benchmarking, and service automation, as well as cycle management, dynamic service provisioning, fault recovery, and vendor-agnostic deployment.

There exist also many integrated NFV platforms implementations from academia. For instance, **OpenBox** [23] allows developers to implement VNF logic through the northbound

API of the OpenBox controller, which in turn deploys the logic to the data plane and implements the intended processing sequence through the OpenBox protocol. The OpenBox controller merges the core control logic of multiple VNFs to avoid duplicated processing and spare NFVI resources for other tasks. The OpenBox data plane is extensible with specialized hardware or pure software. By default, OpenBox contains over 40 processing blocks that can be chained to realize various VNFs, and it can further seamlessly integrate custom blocks at runtime. **fNF** [24] proposes the concept of flyweight network functions that construct slices on the shared IoT infrastructure for applications with diversified QoS requirements. **VLH** [25] combines NFV and edge computing to handle the complex IoT application call-graphs with resource efficiency. It also adopts container techniques and microservice architecture for remote function sharing. **MicroNF** [26] builds modularized SFCs based on element dependency analysis and places them with minimal inter-VM data transfer. It also employs two algorithms to achieve load-balanced scaling and introduces an infrastructure to realize high-speed forwarding and fair scheduling. **μ NF** [27] builds VNFs and SFCs using disaggregated, reusable components, and employs a centralized orchestrator to convert service policies to equivalent forwarding graphs. It also instructs per-server agents to install and manage the VNFs.

Simplified development: Some NFV platforms are specifically purposed to facilitate the development of VNFs and SFCs. **xOMB** [28] is the earliest endeavor to build VNFs on COTS servers by pipelining a set of programmable modules. The execution pipeline is monitored by the xOMB control plane for failover. **Eden** [29] brings application semantics into VNF development by providing an abstraction to classify traffic and using “enclaves” to apply target VNFs. **FlowOS** [30] is specialized for per-flow VNF development with an API to conceal kernel-level complexities. **NetBricks** [31] facilitates the VNF development process by implementing and reusing a set of optimized and customizable elements. It also employs

safe language, runtime libraries, and unique types to ensure isolation. **Scylla** [32] is a declarative language for per-flow custom processing in wireless networks. It also provides a set of programming abstractions to express management intents, which are realized in NFVI by Scylla runtime. **Flick** [33] proposes a domain-specific language to assist VNF development on multi-core COTS servers. It translates the Flick programs into task graphs with bounded resource usage and executes them through cooperative scheduling. **NFMorph** [34] decouples VNF logics from packet processing optimizations and proposes a domain-specific language with coherent processing pipelines. It also optimizes VNF execution based on workload and system constraints.

Complexity abstractions: The development of VNFs and SFCs becomes more lightweight when the programmers can “ignore” some implementation details, as they are already handled by the NFV platform. For example, the data communication of internal modules can be delegated to the core of the platform as done by **Microboxes** [35], which implements a modular, customizable, asynchronous TCP stack for each flow to avoid redundant SFC processing. It also provides a publish/subscribe channel to chain network functions and realize complex network services. A similar approach may delegate to SDN-like structures for data exchange. **Polycube** [36] builds reconfigurable SFCs in kernel space. It separates the processing pipeline into a fast kernel path and a slow path and exposes an API to handle fast/slow-path processing and system events. **SpeedyBox** [37] utilizes a match/action table to consolidate VNF actions at runtime and eliminate redundant processing for SFCs. **libVNF** [17] proposes a generic API to develop L2-L7 VNFs by seamlessly integrating kernel and third-party network stacks. It devises a request object abstraction to keep persistent VNF states across multiple callbacks. It also leverages multi-level data stores for state management across VNF replicas. **ClickNF** [38] augments the Click router [39] with a modular TCP stack to build L2-L7 VNFs and devises a blocking socket to ease VNF development. **S6** [40] extends distributed state objects (DSO) with a programming model to build scalable VNFs. S6 runtime manages shared VNF states distributed in DSO space. Additionally, the NFV platform can provide the tools to automatically detect redundant code and parallelize the jobs that could be performed in parallel. Within this class of work, **NFCompass** [41] adopts a two-level SFC reorganization technique to parallelize VNFs and eliminate redundant processing. It also adopts a task allocation scheme to balance the load and minimize latency. **ParaBox** [42] utilizes a dependency analysis module to identify parallelizable VNFs and implements mirror/merge functions to distribute and aggregate packet copies across the parallelization stages. **NFP** [43] incorporates an orchestrator to analyze VNF dependencies and build optimized service graphs. The NFP infrastructure handles graph execution while dealing with traffic steering, load balancing, and parallel execution. **E2** [44] exposes a “pipeline” abstraction to express network policies. E2 manager merges multiple pipelinelets into a graph and instructs local agents to place VNFs across servers and interconnect them through a high-speed data plane. E2 also provides hooks to VNFs and

data plane to make dynamic adjustments.

B. Testing and benchmarking

After the prototyping phase, VNF developers are required to perform an extensive campaign of sanity checks and performance assessments, which will help validate the correct execution of the network functions, as well as the compliance to the desired service requirements. Network function *correctness* includes all the aspects related to the correct workflow execution (i.e., the application behaves as expected without errors) and the fulfillment of isolation requirements (i.e., services in a multi-tenant environment should not interact or interfere with each other). Correctness checks can be obtained by analyzing the source code or by experimental validation. On the other hand, the developed VNFs should pass a *performance benchmarking* step, to ensure the correct behavior when given some performance constraints.

Correctness: A subset of existing NFV platforms adopts code analysis as a way to ensure the correct execution of VNFs and SFCs. For example, **CHC** [45] ensures service correctness without degrading performance by offloading VNF states to distributed data stores with local caching and employing update algorithms. **SFC-Checker** [46] verifies the behavioral correctness for SFC and employs a model to manage VNF internals and two algorithms for policy validation and state aggregation. Other tools such as **Symperf** [47] bring code analysis even further, by providing also a quantitative prediction of the VNF performance. Alternatively, a more empirical approach is adopted by **Dysco** [48], which explores session protocol to perform reconfigurable TCP service chaining with no packet loss and minimal service disruption. **MiddleClick** [49] targets fast VNF development, based on the Click language, to build and validate SFCs and experimentally assess performance. It also analyzes traffic classes for an SFC to build a synthesized flow classifier and associates states to VNFs on the chain. **NNF** [50] extends UNIFY’s data plane to directly execute simple VNFs at Customer Premise Equipment (CPE). It uses a native controller to instantiate service graphs according to the corresponding VNF templates and employs network namespaces to guarantee isolation and multi-tenancy. In [51], the authors proposed a proof-of-concept implementation of a semi-supervised learning algorithm to monitor and detect malfunctioning VNFs. The authors planned to optimize the algorithm and extend their approach to more complex environments and malfunctions for further validation.

Performance benchmarking: Several platforms are dedicated to performance characterization. As such, *online monitoring* capabilities are often built-in and provided as a way to quickly detect performance anomalies at runtime. **NFV-vital** [52] interprets deployment and workload configurations to setup VNFs and generate workload. It also receives runtime statistics for posttest analysis. This design pattern is also adopted by [53] and [54]. **ConMon** [55] is a distributed framework to monitor the performance of containerized VNFs. **KOMon** [56] is a kernel-based online monitoring

tool to measure packet processing times imposed by the target VNF. Instead of merely focusing on individual VNFs, some platforms characterize performance for SFCs. For example, **SFCPerf** [57] uses a control module to parse service descriptions and deploy the corresponding SFCs in NFVI. The control module also collects critical statistics for data analysis and visualization. **NFVPerf** [58] detects performance bottlenecks for a SFC by monitoring inter-VNF communication. **Perfsight** [59] aggregates runtime information from data path to diagnose performance issues. **VBaaS** [60] supports SFC performance profiling on distributed NFVI. In [61], the authors proposed **NFV-Bench**, a benchmarking solution capable of performing dependability and performance evaluations for NFV solutions, and presented a case study on two state-of-the-art virtualization techniques. However, none of the aforementioned work is suitable to measure the performance of high-speed VNFs running at multi-gigabit rates. **OPNFV Barometer** [62] is designed to monitor the performance of DPDK-accelerated VNFs. It can be attached to the target VNF as a secondary process to gather shared processing information. **NFV-VIPP** [63] can be integrated into the DPDK-accelerated data plane to collect execution metrics and demonstrate the internals of an NFVI node. **BOLT** [64] defines the concept of performance contract, which expresses the expected VNF or SFC performance as a function of critical parameters (e.g., execution instructions, CPU cycles, memory accesses). **DeepDiag** [65] monitors the runtime queuing statistics for each VNF and constructs an online impact graph to diagnose the cause of performance degradation. **CASTAN** [66] adopts symbolic execution to identify the worst code path and a CPU cache model to determine memory access patterns that cause cache invalidation.

C. Deployment

The deployment phase of VNFs or SFCs within NFV platforms requires a careful planning which can be divided into three subcomponents: (i) *resource allocation*, (ii) *service composition* and (iii) *scaling and migrations*. During the first stage, the NFV platform should allocate the necessary resources on top of the COTS server (e.g., bandwidth, CPUs, storage, ...). The second stage consists of the creation of the actual service that is being deployed, which requires traffic classification and steering methods as well as per-flow operations that will allow packets to traverse the required VNFs. Finally, resources can be updated at runtime, by scaling (up or down) when needed; if the COTS resources are being exhausted and no more scaling is possible, migrations can occur that will move the VNFs or the SFCs to a new physical or virtual infrastructure, without any service disruption and with guaranteed performance limits.

Resource allocation: Some NFV platforms formulate and solve an optimization problem to obtain the optimal allocation of VNFs on the underlying infrastructure. **SDNFV** [67] proposes a heuristic algorithm to jointly deploy the VNFs to COTS servers and configure traffic routes for each traffic class through its per-node managers. Each manager maintains an OpenFlow table to be configured by a remote SDN controller and local VNFs to realize a flexible, multi-level control

paradigm. **CoMb** [68] advocates consolidated development of network functions at execution and management level. It parses service policies and infrastructure descriptions and solves an optimization model to decide the optimal deployment strategy. Other platforms can adopt a heuristic approach to provide a sub-optimal allocation policy. For instance, **UNO** [69] uses a heuristic to place VNFs to SmartNICs based on local system status and exposes a single-switch abstraction to manage traffic steering rules across host and SmartNIC switch. **NetFATE** [70] implements a holistic solution to deploy service chains on the edge based on the required traffic load and Quality of Experience (QoE). **ESFC** [71] implements a controller to monitor VNF status and enforce resource allocation policies using an asynchronous notification mechanism for flexible SFC resource management. A hash algorithm is devised for load balancing with flow affinity. Finally, **SCC** [72] collects execution statistics to identify causes of SFC delays, which are addressed by SCC runtime by adjusting the batch size, scheduling policies, priorities, and time slices of SFCs.

Service composition and scheduling: NFV platforms provide the required mechanisms for packets belonging to different users to be served according to their specific user policy. This is usually done by providing advanced traffic classification and steering techniques, and by organizing the VNFs or SFCs in logically connected graphs, that is thus traversed by packets when they must be served. **Cloud4NFV** [73] is an ETSI-compliant platform. It provides an SFC model for fine-grained traffic classification and steering and relies on cloud management tools for service orchestration. **Metron** [74] decomposes SFC processing graph into stateless and stateful operations. The former is offloaded to in-path programmable switches, while the latter remains on COTS servers. Metron also leverages the in-path tagging to dispatch packets to the correct processing cores. An agent is also deployed on each server to conduct MANO operations. **Flurries** [75] is a container-based NFV platform with flexible service chaining and flow-level service customization. A combination of polling and interrupt I/O scheme is also adopted to consolidate per-flow service chains. **SNF** [76] uses graph composition and set theory for traffic classification, and synthesizes VNFs for each traffic class using a minimal number of elements. More recent approaches adopt machine learning techniques: for example, **DeepNFV** [77] is based on GNF but incorporates deep learning techniques for VNFs to learn hidden data patterns and provide enhanced services such as traffic classification, QoS optimization, link status analysis, and so on.

Careful scheduling of VNFs is also important to achieve resource efficiency. Some NFV platforms specifically tackle this issue. **NFVnice** [78] adopts rate-cost proportional fairness by adjusting CPU weight for each VNF based on the estimated traffic arrival rate and service time. The scheduling is done by tuning the OS scheduler via Linux cgroups. At runtime, NFVnice monitors workload and employs a back-pressure mechanism to early-drop packets for congested SFCs to spare resources. **EdgeMiner** [79] spares CPU resources from co-located VNFs to execute other applications at the network edge, by employing a back-pressure scheme to detect SFC

overloads and puts upstream VNFs into sleep to harvest the otherwise wasted CPU cycles. **UNiS** [80] is tailored to schedule poll-mode VNFs. For each worker core, it measures intermediate buffer occupancies to make scheduling decisions. **SNF**¹ [81] dynamically traces the VNF workload and allocates compute resources at a fine granularity. A peer-to-peer in-memory store is deployed to proactively replicate the states and reduce packet processing latency. **ResQ** [82] profiles VNFs to construct profiles and computes a resource-efficient allocation using a greedy approach. ResQ also periodically solves an optimization model to maintain an optimal allocation and selectively applies it based on a constraint. **NetContainer** [83] builds an estimation model based on footprint theory to infer cache access overhead and model the cache mapping problem as a minimum-cost/maximum-flow problem to decide the optimal memory buffer mappings. **NFV-throttle** [84] spreads modules across NFVI to dynamically monitor system conditions and drop excessive packets to prevent VNFs from being overwhelmed. **Iron** [85] introduces an enforcement mechanism to account for the time spent by the VNFs in kernel space, and throttles or even drops packets for the aggressive VNFs through Linux scheduler or a hardware-based approach.

Scaling and migration: Several platforms strive for efficient VNF scaling and migration. **Split/Merge** [86] uses a centralized orchestrator and SDN controller to direct instance scaling and flow migration. It provides an API to split or merge flow states among VNF replicas. The system then migrates the relevant states and configures the network to direct flows to the correct replicas. **TFM** [87] performs migration through a centralized controller, which decouples flow and state migration processes with three modules: a state manager, a flow manager, and a forwarding manager. The state manager conducts state migration through southbound APIs. The forwarding manager interacts with the SDN controller to update the corresponding traffic steering rules. The flow manager distributes TFM boxes for packet classification and buffering during VNF migration. **OpenNF** [88] implements a controller that consists of an event-driven model to capture relevant packets, a southbound API to request the import/export of VNF states at different granularities, and a northbound API to control applications and instruct state migration, which is carefully crafted to avoid packet losses or out-of-order packets. It also performs state synchronization with strong or eventual consistency. **DiST** [89] and **U-HAUL** [90] follow similar procedures for state and flow migration without controller intervention. In particular, U-HAUL only identifies and migrates states for elephant flows while serving mice flows in original VNFs until expiration, reducing the number of migrated states. **SlIM** [91] proposes statelet, a compact packet data representation, to achieve bandwidth-efficient state migration. It also integrates a kernel-bypassing I/O technique to boost performance. **State-Alyzr** [92] is a non-intrusive framework that handles state clone and migration based on program analysis. **LEGO** [93] provides a set of mechanisms for traffic splitting, instance partitioning, and runtime management, to enable elastic scaling

of Artificial Neural Network (ANN)-based VNFs. **Lange et al.** [94] employs a machine learning approach to predict and scale the number of VNF instances based on recently monitored traffic. The authors also provided guidelines on the data features and parameters to render reliable predictions. **DeepMigration** [95] deduces the migration and scaling cost of existing VNF instances using a customized graph neural network, and dynamically decides the best flow migration policies through a trained reinforcement learning model.

D. Management and execution

The final phase of a network function life cycle is represented by the parallel activities of management and execution. The NFV platform is now deployed on the COTS, and all mechanisms for resource allocation and traffic steering are up and running. During this phase, the NFV platform has the double objective of (i) allowing the execution of VNFs or SFCs without failures (and exceptionally provide failure recovery mechanisms) as well as in untrusted environments and (ii) allowing simple and effective management of the VNFs at runtime, which is usually achieved by providing access to open APIs and a separated management and orchestration plane. Orthogonally, the performance of network functions can be improved by adopting a series of common software acceleration techniques, which can be implemented on pure software or by hardware offloading.

Fault tolerance and Secure runtime: Although at this stage the NFV platform should have already ensured correctness (see Sec. III-B), runtime errors can always occur due to hardware failures or unpredictable events. We have previously discussed the migration capabilities of existing platforms that can also be used in case of faults (see Sec. III-C). On the other hand, some platforms may target a resilient execution approach to bring fault-tolerance even further. For example, **NFVector** [96] employs the distributed actor model to support per-flow abstraction and provides an API to implement VNFs with resilience. **NetStar** [18] implements a flow-based asynchronous interface combined with a future/promise library for VNF development. Instead of spreading control logic across multiple callback functions, NetStar mimics sequential execution by chaining multiple future objects and continuation functions over a single function call. **FTMB** [97] adopts a rollback recovery scheme to conduct VNF failover operations.

Another group of platforms has been specifically devoted to developing secure VNFs for execution in untrusted environments. **vEPC-sec** [98] incorporates a variety of traffic encryption, validation, and monitoring schemes to safeguard cloud-based LTE VNFs. **SplitBox** [99] distributes VNF functionalities to multiple cloud VMs to obscure its internals from public cloud. **Embark** [100] allows VNFs to operate on encrypted data leveraging a special HTTPS encryption scheme. **BSec-NFVO** [101] introduces a blockchain-based architecture to protect NFV orchestration by auditing all the operations over the SFCs. Other platforms exploit Intel® Software Guard Extensions (SGX) [102] instruction codes to secure VNFs from memory reading attacks. In specific,

¹The SNF cited here is different from the one previously mentioned [76].

S-NFV [103] concentrates on the protection of VNF states by stashing them into the shielded SGX memory region (*enclave*) to prevent unauthorized access or snooping. **Trust-Click** [104] and **ShieldBox** [105] extend the Click modular router to secure packet processing within SGX enclave, and rely on SGX remote attestation to verify code correctness. **ShieldBox** additionally integrates DPDK for high-speed packet processing and ring buffers to support SFC deployment. Several platforms further protect VNF states. **SafeLib** [106] offers comprehensive protection, including user traffic, VNF code, policy, and execution states. It also integrates DPDK and libVNF to support TCP functionalities without compromising performance. **LightBox** [107] employs a virtual interface to protect enclave I/O, a state management scheme to cache states for active flows, and a space-efficient algorithm for flow classification. **Safebricks** [108] partitions VNF code to minimize the trusted computing base in enclaves and performs packet exchanges across trust boundaries through shared memory. It also supports deploying an entire SFC inside an enclave and leverages Rust primitives to isolate the VNFs.

APIs: Inspired by the control/data plane decoupling and the massive usage of APIs (as advocated by SDN approaches), most existing NFV platforms adopt some flavor of the open interface that can be accessed by the programmer to monitor or to perform some change. **StatelessNF** [109] provides an API for VNF development while it delegates state management tasks to an external data store. **Slick** [110] offers a python API for developers to specify network services based on traffic classes. Then a Slick “runtime” component parses the service specifications and makes placement decisions using several heuristics before placing the VNFs and configuring the forwarding routes. Slick also interacts with a per-node “shim” component for incremental optimization. **CoNFV** [111] divides SFCs between cloud infrastructure and end-hosts and exposes an API to port existing VNFs into CoNFV. **Virt-Phy** [112] is an NFV platform designed for edge data centers. It integrates server-centric topology, SDN techniques, and software switches to realize efficient VNF deployment and service function chaining. **GNF** [113] exposes an interface to specify services at the network edge and relies on a manager for MANO operations. The manager instructs per-device agents to deploy and manage VNFs in containers.

Management and orchestration: A set of platforms is purposed to provide NFV MANO solutions. Some of them strive for full-fledged, holistic MANO systems, while others tackle only a facet of the MANO problems, such as scaling, migration, failover, scheduling, performance benchmarking, and secure execution. **ETSO** [114] is an ETSI-compliant NFV MANO platform for heterogeneous cloud environments. It addresses key orchestration issues with shared service abstraction and optimized placement algorithms. **UNIFY** [115], [116] employs a layered graph abstraction to automatically map user-specified services into SFCs deployed to the underlying NFVI PoPs. It also models network and service altogether and provides joint optimizations for service management and orchestration. **ONAP** [117] is a cloud-native architecture that

provides a whole set of solutions to compose, deploy, and manage the complete life cycle of network services across the NFVI. **Open Source MANO (OSM)** [118] aims at implementing a production-grade MANO stack interoperable with third-party NFV components while it allows for efficient service provisioning. **OpenMANO** [119] consists of an orchestrator (openmano), a VIM (openvim), and a graphical user interface. Openvim manages NFVI resources and relies on a REST API to communicate with OpenMANO for MANO operations. **Open Baton** [120] is an extensible MANO framework for service orchestration across heterogeneous NFVIs. It manages a diverse set of VNFs running across NFVI-Nodes with different virtualization technologies and features network slicing for resource multiplexing. **T-NOVA** [121] leverages SDN and cloud management tools to design and implement a software NFV MANO stack for automated VNF deployment and management. **TeNOR** [122] is an NFV orchestrator based on a micro-services architecture. It proposes two approaches to address resource and service mapping and provides a marketplace to accommodate third-party VNFs. **vConductor** [123] automates service deployment with a resource scheduling algorithm to meet business requirements, and uses enhanced inventory management for fault isolation.

NFVI acceleration: Acceleration of the NFV platform can be performed directly within the COTS servers, or by leveraging hardware-assisted design that delegates some portion of the VNFs’ computational resources to external accelerators. Within the first class, **NetVM** [124] achieves line-rate processing through a shared memory mechanism and relies on a hypervisor switch to steer packets based on traffic or system conditions. It also defines multiple trust domains to limit memory access of untrusted VNFs. **OpenNetVM** [125] follows NetVM design, but adopts containers to wrap VNFs. It achieves more flexible traffic steering by enabling VNFs and management entities to make routing decisions. **ClickOS** [3] utilizes the Click Modular Router [39] to build a wide range of VNFs in Xen-based uni-kernel VMs. A set of optimizations is performed on the hypervisor data path to boost the performance. Similarly, **HyperNF** [126] advocates consolidating VNFs for resource efficiency and reduces synchronization overhead with hypervisor-based packet I/O. **CLIMBOS** [127] implements lightweight, isolated, and modular IoT backends based on ClickNF. **MVMP** [128] employs a virtual device abstraction to flexibly steer traffic between containerized VNFs and physical NICs. **NFF-Go** [129] is designed to build and deploy network functions in the cloud. It leverages Go language for concurrency and safety, and a scheduler to scale VNFs on demand. **IOVTee** [130] optimizes the VNF RX path by mapping the VM queues to hypervisor switch, eliminating the expensive copy operations while ensuring security. **HALO** [131] optimizes the flow classification process by exploiting hardware parallelism of the CPU caches and extending the CPU instruction set to scale flow-rule lookups.

Within the hardware-assisted class of NFVI accelerated platform we may cite **P4SC** [132] and **P4NFV** [133], both exploring P4 language to accelerate SFC processing. P4SC parses specified service policies and converts them into a

P4 program, which is subsequently deployed onto the P4-compatible hardware. P4NFV is designed for both hardware and software targets and supports runtime reconfiguration without violating state consistency. Albeit augmented with various software acceleration techniques, CPU cores might still fall short of performance. As a result, several platforms explore other hardware components for processing acceleration. **OpenANFV** [134] aims at supporting VNF acceleration in the cloud by delegating a subset of tasks to PCIe-based FPGA card. Similarly, the work in [135] proposes to integrate OpenNetVM with SmartNICs to offload VNF processing and enforce memory isolation. **ClickNP** [136] augments COTS servers with FPGA acceleration and exposes a modular abstraction to implement VNFs. **NICA** [137] exploits F-NICs to accelerate inline processing. It implements an API to grant direct control over F-NIC accelerators and an I/O path virtualization for multiple VMs to share F-NICs with security and fairness. Some platforms construct CPU-GPU pipelines to expedite SFC processing. **NetML** [138] accelerates data transfer to GPU by optimizing the data path. **FlowShader** [139] leverages kernel stack for traffic classification and exposes an API to develop compatible VNFs across CPU and GPU domains. It also employs a scheduling algorithm to balance the workload between GPU and CPU. **GPUNFV** [140] employs flow-level parallelism and runs an SFC to completion in a GPU thread. It exploits GPU for processing and devotes CPU for packet I/O and flow classification. **Gen** [141] features the dynamic scheduling of GPU threads for VNF scaling, and supports runtime SFC modification using CUDA API. **Grus** [142] reduces the processing latency through coordinated access of the PCI-E bus, fine-grained VNF scheduling, and dynamic batching. **G-NET** [143] manipulates GPU context to allow for spatial GPU sharing across manifold VNF kernels and leverages safe pointers to guarantee GPU memory isolation. A scheduling algorithm is also employed to calculate the per-SFC cost and optimize the GPU resource sharing.

IV. DESIGN SPACE

We now explore the design space and summarize the different design choices adopted by existing NFV platforms. This section focuses on the technological solution adopted by each platform, and it is complementary to Sec. III, whose taxonomy refers to the platform life cycle of VNFs. We begin our discussion from the MANO plane followed by the service plane and NFVI. The design choices of some existing representative NFV platforms are listed in Table I. For a more detailed description of the NFV design space, please refer to our technical report [144]. Note that in general there is no superior choice over the others, and a choice should be performed according to specific use scenarios and application contexts.

A. MANO plane

High-level API: Most of the existing NFV frameworks provide high-level APIs to specify service policies or smooth the process of VNF development. These APIs can be either

Domain-Specific Language (DSL) or **General-Purpose Language (GPL)**. GPLs such as C, C++, Java, and Python are mature programming languages capable of solving problems in multiple domains. They are shipped with multitudinous control primitives, miscellaneous data structures, and flexible operating patterns. Most of the existing NFV platforms adopt GPL. For example, OpenNF relies on a C++ API to develop control applications. NfVNice exposes the “libnf” C library to perform I/O operations asynchronously and to monitor the workload of each VNF. OpenBox exposes a Java API for operators to specify processing logic and subscribe to system events. Slick API allows developers to specify service policies in Python, while NetBricks achieves it with a Rust API. Compared to GPLs, DSLs provide higher-level optimized abstractions for specific problems, and they usually operate in an environment with limited operation patterns and restricted resource usage. For example, Flick language supports parallel execution and safe resource sharing. In addition to basic primitives such as event handling and common data types, Flick can deserialize input packets into application-specific data types and vice versa, bringing application semantics into VNF development. Service policies in Eden are specified in F# language for efficient safety checking. NFMorph also proposes a DSL that allows developers to express per-packet operations and compiler hints for dependency analysis and runtime optimization.

Placement: Service placement is achieved in two steps: **pre-processing** and **deployment**. In the pre-processing phase, input policies are optimized through graph merging and parallelization. In the deployment phase, network services are installed across NFVI-Nodes with pre-defined objectives. Existing platforms generally follow the same procedure. For instance, CoMb consolidates SFCs on a single NFVI-Node by solving an optimization model based on service and infrastructural description. OpenBox merges input processing graphs with correctness guarantees and deploys the related VNFs to the specified NFVI-Nodes. Slick employs an inflation heuristic to consolidate VNFs with minimum cost and uses a placement algorithm to deploy them while configuring traffic steering rules on the network switches. E2 also merges multiple service graphs to reduce processing redundancy. It models VNF placement as a graph partition problem over NFVI-Nodes and employs a heuristic placement algorithm to minimize the inter-server traffic. SDNFV formulates the placement problem as a MILP problem and designs a heuristic algorithm to maximize resource utilization. Metron leverages SNF to optimize the input graph and constructs a synthesized one, which is subsequently split into a stateful subgraph and a stateless subgraph. The stateful graph is deployed on COTS servers selected by the Metron server selection scheme. The stateless graph is offloaded to in-path network elements based on the locations of the stateful graph. MicroNF performs dependency analysis for VNF elements and reconstructs the service graph to reduce redundant processing and improve resource efficiency. It places the modularized SFCs to the COTS servers by solving an integer programming problem to minimize inter-VM overhead. The μ NF orchestrator constructs an optimal forwarding graph by consolidating VNFs performing similar processing, but the

TABLE I: Design choices for a subset of the existing NFV platforms.

Platform	MANO plane					Service plane				NFV infrastructure (NFVI)						
	High-level API		Placement engine	State coordination		Execution model		TCP function	VNF I/O		Packet I/O	VNF interconnect		Virtualization technique		
	GPL	DSL		SM	MA	RTC	Pipe.		Poll.	Intr.		VS	Cus.	VM	Co.	Pr.
OpenBox	✓		✓		✓	✓					Kernel			✓		
E2		✓	✓		✓	✓		✓			DPDK	✓				
SDNFV			✓			✓				✓	DPDK		✓	✓		
Slick	✓		✓			✓		✓			Kernel					✓
Eden		✓	✓					✓			Kernel		✓			✓
MicroNF			✓		✓		✓			✓	DPDK	✓		✓	✓	
μNF		✓			✓		✓			✓	DPDK		✓		✓	✓
Metron	✓		✓	✓		✓					DPDK	✓				✓
Flurries	✓							✓		✓	DPDK				✓	
MicroBoxes	✓							✓			DPDK				✓	
OpenNF				✓				✓			Kernel					
Split/Merge	✓			✓							Kernel	✓		✓		
ClickNF	✓					✓		✓		✓	DPDK					✓
Flick		✓			✓		✓	✓		✓	DPDK					✓
NetStar	✓				✓	✓		✓		✓	DPDK		✓			✓
StatelessNF	✓				✓			✓		✓	DPDK				✓	
NFMorph		✓					✓	✓		✓	DPDK		✓			✓
NetVM	✓					✓				✓	DPDK	✓		✓		
OpenNetVM	✓					✓				✓	DPDK	✓			✓	✓
NetBricks	✓					✓		✓		✓	DPDK	✓				✓
ClickOS	✓					✓				✓	netmap	✓		✓		
HyperNF											netmap	✓		✓		
IOVTee											DPDK		✓	✓	✓	✓
CHC					✓					✓	VMA		✓		✓	
NICA	✓							✓		✓	VMA				✓	
Polycube	✓							✓			eBPF				✓	
NFP							✓				DPDK		✓		✓	
ParaBox											DPDK	✓			✓	
NFVNice	✓							✓		✓	DPDK		✓		✓	
libVNF	✓				✓	✓	✓	✓		✓	DPDK netmap kernel	✓		✓		✓

authors did not indicate the placement approaches.

State coordination: With the proliferation of stateful VNFs, it is critical to timely coordinate the processing states upon the scaling, migration, and failover of VNFs. However, it is extremely challenging to simultaneously guarantee flow affinity, correct processing, and minimal service interruption. Existing NFV platforms resort to two strategies, **state migration** and **migration avoidance**, respectively. The former employs different approaches to migrate states. For instance, Split/Merge suspends traffic flows for all replicas and transfers the relevant states across them while configuring the related traffic routes. Metron divides states of the overloaded SFC into two groups and copies one group to new replicas. However, both approaches incur in-transit packet losses that might lead to state inconsistency. OpenNF uses a centralized controller and in-path OpenFlow switches to preserve flow affinity and packet processing order without in-transit losses. MicroNF, UNO, and OpenBox also advocate this solution for state coordination. Similarly, TFM deploys a “box” at each VNF instance to buffer incoming packets and feed them in the correct order to the new replica. In contrast, some platforms adopt a migration avoidance strategy to avoid state migration overhead. In particular, E2 splits the flow space and steers part

of the incoming flows to the new instance while keeps serving existing flows till termination. Another migration avoidance strategy is state externalization. For example, StatelessNF, CHC, NetStar, and libVNF keep VNF processing states in external data stores to avoid state migration costs.

B. Service plane

NFV platforms are required to consider several critical design choices in the service plane.

Execution model: In NFV domain, two VNF execution models are adopted: **run-to-completion (RTC)** and **pipeline**. In the RTC model, all the elements of a VNF run on a single thread, whereas in the pipeline model, each element is pinned to a separate thread, as illustrated in Fig. 4a. The performance of either model is highly dependent on processing complexity and input workload, which leads to different levels of cache and memory access patterns. In general, the RTC model presents better performance executing simple VNFs or short SFCs by eliminating inter-core transfer overhead [74]. It also requires fewer cores than the pipeline model. However, the pipeline model enables finer granularity scaling and incur less cache misses processing complex VNFs. Some platforms employ the RTC model to accommodate lightweight VNFs

or trimmed SFCs. For instance, CoMb, NetVM, NetBricks, ClickNF, ClickOS, NetStar, and SafeBricks execute VNFs in the RTC model to avoid the inter-core transfer and synchronization overhead; Metron offloads part of its SFCs to the in-path hardware devices, executing the trimmed tasks in RTC model on COTS servers. Other platforms employ the pipeline model. In particular, μ NF and MicroNF decompose VNFs or SFCs into loosely-coupled elements to be scaled individually.

TCP functionality: As stateful VNFs have become an important building block in the NFV ecosystem, it is worth pointing out existing platforms that implement or integrate TCP/IP stack to support stateful VNFs at layer 4 or beyond. ClickNF is equipped with a full-fledge modular TCP stack to facilitate the end-host application development. Microboxes comes with a modular, customizable TCP stack that can be shared among a group of VNFs to eliminate redundant processing. OpenANFV also designs a TCP stack in the userspace. NICA even implements a simplified TCP stack in SmartNICs to enrich its in-path processing features. xOMB stack only implements simple functions such as TCP connection termination. Instead of developing TCP functionalities from scratch, some platforms choose to directly incorporate third-party solutions. For example, Flick integrates the high-speed kernel-bypassing mTCP stack [145] to realize transport layer VNFs, NetStar directly employs a third-party user-space TCP stack with future/promise abstraction. libVNF is designed to be generic by integrating both the standard networking stack and mTCP. Besides, all the NFV platforms using the kernel TCP/IP stack and POSIX sockets are granted TCP functionalities by default. In particular, Polycube directly cooperates with the kernel TCP/IP stack to build complex SFCs.

VNF I/O: There are also two alternative means for VNFs to perform packet I/O, namely **polling mode** and **interrupt mode**. VNFs running in polling mode keep querying the NICs or upstream VNFs for data, which normally renders better performance at the cost of wasted CPU cycles and increased energy consumption due to idle waiting. Interrupt-based I/O usually does not entail wasted resources but suffers from performance losses due to interrupt propagation delay and cache line warm-up. Existing NFV platforms such as UNO, CHC, NetBricks, ClickNF, NetStar, and StatelessNF, execute VNFs in the polling mode to enhance performance. Other platforms such as Flick, NFVNice, xOMB, ClickOS, Flurries, and libVNF execute VNFs in interrupt mode.

Secure execution: As VNFs are increasingly delegated to untrusted environments (e.g., public cloud or third-party networks), traffic data and VNF internals are exposed to potential cyber-attacks. Existing NFV platforms secure the execution of VNFs with **encryption** and **shield execution**. Platforms adopting the former (e.g., vEPC-sec, Embark) leverage cryptographic schemes to enable VNFs to operate directly on encrypted traffic. Platforms adopting shield execution run VNFs in memory regions called enclaves whose contents are strictly protected from external accesses. For example, S-NFV, TrustedClick, ShieldBox, SafeLib, SafeBricks, and LightBox leverage Intel SGX to provide a shield execution environment for VNFs. Similarly, NetVM and OpenNetVM place VNFs inside trusted domains to ensure security. Compared to shield

execution, encryption approaches usually incur higher overhead imposed by the complex cryptographic operations and support a limited set of functionalities.

C. NFV Infrastructure (NFVI)

Packet I/O techniques: Existing NFV platforms adopt different I/O techniques to exchange packets with the outside network through physical NICs. When such NICs are managed by COTS servers, the two approaches used for I/O are **kernel-based** and **kernel-bypassing**, as shown in Fig. 4b. Traditional network applications rely on the feature-rich kernel stack for packet I/O, although the overhead imposed by kernel stack makes software solutions fail to sustain line-rate processing [146]. This bottleneck can be overcome by adopting kernel-bypassing techniques (e.g., DPDK [147], netmap [148]). Netmap partially bypasses the kernel, and it adopts system call based validation and interrupt-based packet reception. DPDK employs complete kernel-bypassing and poll-mode drivers to boost performance. They also expose APIs to simplify VNF development. As shown in Tab. I, most of platforms such as E2, Flick, ClickNF, IOVTee, NetStar, μ NF, Flurries, StatelessNF, NetVM, and OpenNetVM leverage DPDK for packet I/O. netmap is used by ClickOS and HyperNF. NICA and CHC adopt Mellanox Message Accelerator (VMA) [149], another kernel-bypassing technique with standard socket APIs and user-space library. Note that even though the traditional kernel-based approach fails to render comparable performance as kernel-bypassing techniques, it can still be useful when the VNFs are not I/O intensive or the cost to set up a kernel-bypassing stack is too high. The extended Berkeley Packet Filter (eBPF) [150] adopts an in-kernel virtual machine to run user-space programs that can be used to execute network functions. In conjunction with the XDP enhancements [151], it provides high-performance packet processing capabilities. eBPF is adopted by Polycube for packet I/O. Note that libVNF supports kernel, DPDK, and netmap to achieve generality.

VNF interconnects: Existing NFV platforms concatenate consolidated VNFs by integrating or implementing **software switches**. Software switches are widely used by existing platforms for efficient traffic steering. For instance, E2 augments BESS [152] as data plane, ClickOS and HyperNF extend VALE switch [153], CoMb customizes the Click Modular Router [39]. Metron, MiddleClick, SCC, and SplitBox leverage FastClick [146] to transfer packets between VNFs and the network. Split/Merge, TFM, and MicroNF employ Open vSwitch (OVS) [154] for VM networking. UNO extends OVS-DPDK [155] to steer packets at both host and SmartNIC level. NetBricks adopts both OVS-DPDK and BESS to interconnect VNFs. More details about the performance of these software switches can be found in [156]–[158]. Rather than adopting third-party software switches, some platforms implement customized solutions. For instance, G-NET uses a bespoke software switch to route packets between VNFs and physical NICs, NetVM implements a hypervisor software switch to enable state- and data-dependent forwarding.

Virtualization technique: As the central point of any NFV platform, existing implementations deploy network functions

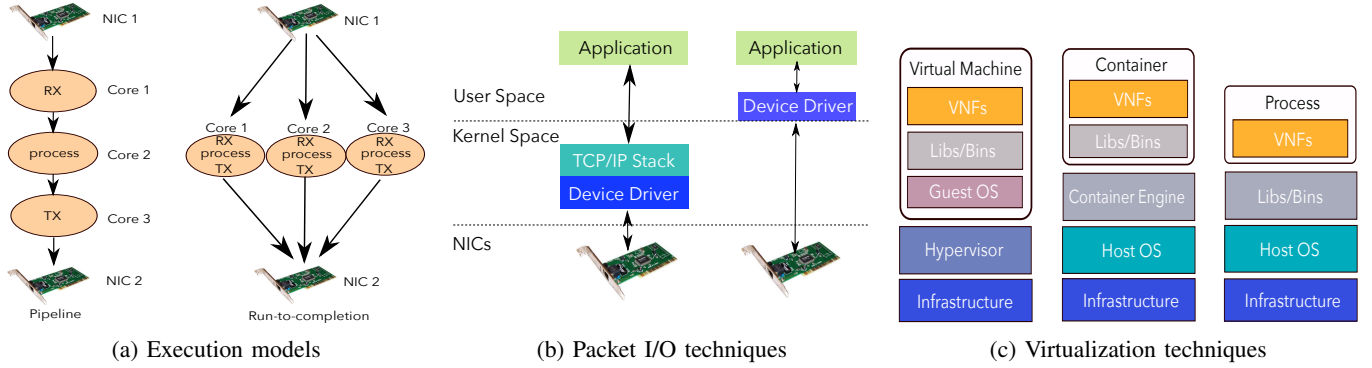


Fig. 4: Illustration of several key design choices

in Virtual Machines (VMs), Containers, and Processes.

As illustrated in Fig. 4c, virtualization is implemented at different layers in the commodity hardware, and therefore present different degrees of isolation and resource requirements. VM is a hardware-level virtualization technique that relies on the Virtual Machine Monitor (VMM) or hypervisor to accommodate VNFs². Hypervisors also manage the VMs for efficient resource sharing. Some platforms adopt different VM hypervisors to host VNFs. For example, NetVM, NICA, OpenANFV, IOVTee, and SDNFV adopt KVM; Split/Merge, HyperNF, and FlowOS adopt Xen. However, VMs commonly impose heavy resource demands, huge memory footprints, and high migration costs. To overcome these deficits, ClickOS and CLIMBOS adopt unikernel VMs that are small, agile, and fast-to-boot. The advent of containers, such as LXC and Docker, gives another option. Compared to VMs, containers are OS-level virtualization techniques with small memory footprint, short boot time, high deployment density, and low migration cost. The disadvantages of containers include weak isolation and degraded security. In terms of performance, they both can sustain line rate processing using tailored I/O paths. At present, many existing NFV platforms opt for containerized VNFs. For instance, OpenNetVM, Flurries, MicroBoxes, NFNice, MVMP, NFP, ParaBox, MVMP, statelessNF, and GNFC employ Docker, while CHC and Iron adopt LXC. Aside from VMs and containers, some platforms deploy VNFs as processes to trade isolation for performance. For example, NetBricks, ClickNF, libVNF, and SafeLib execute VNFs as processes and use different means to guarantee isolation. GPUNFV, Gen, FlowShader, Grus, and G-NET execute VNFs or SFCs as GPU threads. Note that some platforms feature multiple techniques. For example, OpenNetVM and μ NF adopt both process and container. MicroNF runs containers inside VMs probably to improve security.

²Note that there is another kind of hypervisor that operates at OS level (namely, hosted hypervisor) which was excluded from our discussion as rarely used by existing NFV platforms.

D. Other design choices

As discussed in [159], there is a large assortment of acceleration techniques for high-speed packet processing. Here, we choose the most commonly utilized techniques and enumerate their adoption by existing NFV platforms. The optimization knobs we consider including zero-copy, batching, memory pre-allocation, parallel execution, CPU cache optimization, and computation offloading. Although these optimizations are commonly applied by high-speed packet processing applications, we discuss them in the context of NFV.

Zero-copy: In the high-speed packet processing domain, runtime memory copy is an expensive operation that usually leads to unbearable overhead. For the sake of performance, many existing NFV platforms deliver packets across VNFs or memory boundaries in a zero-copy manner, by copying only their associated packet descriptors. For example, μ NF implements a zero-copy port abstraction that only exchanges packet addresses instead of copying full packets between VNFs. NetVM employs a shared memory mechanism to enable zero-copy packet delivery to and between the VNFs running in VMs. Instead of shared memory, IOVTee implements a safe zero-copy mechanism through memory mapping between hypervisor switch and VM. NetBricks adopts Unique Types to implement safe zero-copy packet delivery between NFs. The TCP stack of ClickNF exposes zero-copy interfaces to interact with user-space VNF. NICA leverages ring buffers for zero-copy message exchange between the F-NIC units and the user-space VNFs. GPUNFV achieves zero-copy packet delivery across CPU and GPU boundary through CUDA's page-lock memory. G-NET's switch also employs zero-copy design.

Batching: In high-speed packet processing frameworks, I/O batching is widely used to amortize the overhead of accessing the physical NIC over multiple packets. This technique is also employed by some NFV platforms to enhance performance. For example, NFNice and EdgeMiner batch the I/O interrupts to amortize VNF wakeup overhead. SCC handles the system calls of VNF I/O in dynamic batches to reduce the overhead of context switches. NFMorph advocates optimizing performance with batch tuning. The VNFs on μ NF platform perform packet I/O in batches through the intermediate ring buffers. The

TCP stack of ClickNF exchanges packets with the user-space VNFs in batches. StatelessNF aggregates multiple read/write requests to the data store into a single request to amortize the overhead of remote procedure call (RPC). SafeBricks implements an in-enclave module to perform batched packet I/O between the enclave and the host. LightBox adopts packet batching to amortize the system call overhead. GPUNFV, Grus, FlowShader, G-NET, and Gen deliver packets between CPU and GPU in dynamic batches.

Pre-allocation: Runtime memory allocation remains a heavy task. Barring pre-allocating packet buffers and descriptors for packet I/O with physical NICs, existing NFV platforms usually reserve memory regions to stage and reuse other relevant packet processing data structures. For example, libVNF pre-allocates memory pools for its per-core, persistent request objects, and lock-free packet buffers. Flick pre-allocates its task graphs and queues. S6 pre-allocates a pool of the cooperative, user-space, per-flow micro-threads to avoid the dynamic thread creation/deletion overhead. NetVM maintains a pool of idle VMs for prompt VNF migration. ShieldBox pre-allocates packet descriptor memory. LightBox pre-allocates state management data structures.

Parallel execution: To take advantage of the multicore CPUs, many platforms explore possibilities for parallelization. μ NF performs a dependency analysis on its forwarding graphs to identify parallelize VNFs. Consecutive VNFs are deemed parallelizable if they perform read-only operations or update disjoint packet regions. Then these VNFs are assigned independent CPU cores to process packets. A reference counter is attached as meta-data to avoid out-of-order operations from downstream VNFs. Likewise, SDNFV allows multiple VNFs to access a packet in parallel using a reference counter embedded in the packet descriptor. Eden exposes a concurrency model that creates consistent state copies for multiple VNFs to execute in parallel in the Eden enclave. CoMb allocates each SFC an independent shim layer to allow for parallel execution of multiple SFCs. Flick instantiates a new task graph for each new connection and schedules the tasks of these graphs onto multiple worker cores in parallel. NetBricks runs per-tenant service processing graphs in parallel in a multi-tenant environment. ClickNF utilizes the Receive-Side Scaling (RSS) of physical NICs to distribute incoming packets to multiple cores with flow-level affinity guaranteed. NetStar builds VNFs with a share-nothing thread model and distributes incoming packets to different threads for paralleled multicore processing. libVNF is built with multicore scalability of VNF and uses per-core data structures to avoid inter-core communication which can hamper the multicore scalability of a VNF.

Cache optimization: Modern CPUs are equipped with hierarchical caches between its cores and the main memory. Cache misses result in extra access to other cache levels or the main memory, which significantly slows down the processing speed. Many existing NFV platforms are aware of this issue and explore opportunities for cache optimization. NetContainer aims at exploiting cache locality at inter-flow and intra-flow levels for NFV workload and leverages page coloring techniques to aggregate buffer pages to separate cache regions to avoid cache contention. ResQ exploits Intel Cache Allocation

Technology with corresponding buffer sizing to eliminate last level cache invalidation while ensuring performance isolation. LightBox adapts cache line protection techniques to reduce the cache miss rate. μ NF and NFMorph perform data prefetching in batches to increase the cache hit rate. Some platforms also make their critical internal data structures cache-optimized. For example, the request objects of libVNF are cache-optimized, all the per-core data structures of ClickNF are cache-aligned.

Computation offloading: Computation offloading is widely adopted by existing NFV platforms to alleviate the pressure of COTS servers. Potential resources to offload computing tasks include GPU, smartNICs, and in-path network switches. E2 selectively offloads simple VNFs to adjacent hardware switches. Metron offloads stateless operations to the in-path programmable NICs and switches. OpenBox and Eden also implement their forwarding plane in hardware. OpenNetVM and OpenANFV incorporate programmable NICs or FPGAs for computation offloading. ClickNF explores common NIC features to perform TCP/IP checksum offloading, TCP segmentation offloading (TSO), and large receive offloading (LRO). GPUNFV, Gen, FlowShader, Grus, and G-NET employ GPU offloading to boost performance. SmartNICs are commonly equipped with programmable, multi-core processors and an integrated operating system, making them ideal to execute computation tasks. UNO exploits smartNICs to offload VNFs, forwarding rules, flow tables, and crypto/compression operations. NICA leverages the inline processing of FPGA on smartNICs to accelerate data plane processing.

V. OPEN ISSUES AND CHALLENGES

In this section, we envision some future directions for NFV platform design, including Artificial Intelligence (AI), network slicing, and Internet of Things (IoT), and discuss the potential challenges therein.

A. AI in NFV

With the proliferation of Artificial Intelligence, an increasing amount of effort has been devoted to driving networks using AI techniques without human intervention. Although Machine Learning (ML) and Deep Learning (DL) techniques have begun to be adopted by some NFV platforms for traffic prediction and runtime management [51], [77], [93], [94], AI is still far away from complete integration into NFV. Since AI techniques such as Neural Networks are intrinsically complex (if not impossible) to comprehend, a huge amount of domain-specific expertise is required to guarantee the correctness and reliability of the AI-integrated NFV solutions. Developers need to master knowledge in both NFV and AI domains to implement production-ready NFV platforms, making it more difficult for newbies to get started and expensive for companies to recruit talents. As the base of ML/DL, large sets of relevant data need to be collected and pre-processed. While plenty of datasets for legacy networks and applications are available, the patterns or features may be at odds with contemporary NFV and 5G networks. It is also time-consuming to extract correct features and train the models, not to mention the tedious verification process to avoid inaccuracy or overfitting. While

we believe AI will become commonplace in the NFV domain, it is still in its infancy in the telecommunication industry and several challenges yet need to be overcome to thoroughly unleash the potentials of AI in NFV.

B. Network slicing

As one of the key enabling technologies for 5G, network slicing promises to slice the physical network infrastructure into multiple self-contained, isolated, and programmable logical (or virtual) networks to indulge different genres of services demanded by assorted tenants [160]. Combined with other trending network softwareization technologies such as SDN and cloud/edge computing, NFV platforms are envisaged to realize network slicing over 5G infrastructure. We identify several fundamental challenges that need to be surmounted. Firstly, creating variable-sized network slices for varied tenants or business verticals entails effective management of an NFV platform's components, including dynamic slice instantiation/termination, cognitive resource scheduling among network slices, adaptive VNF placement and configuration for each slice, efficient intra-/inter-slice communication, and so on. Although some of the management issues have been individually resolved, few platforms come up with a joint solution to manage heterogeneous network slices, each of which calls for a particular set of resource and SLA requirements. As a result, it is of uttermost importance for NFV platforms to manage its services and resources tailored for each slice with guaranteed performance and efficient resource usage. Similarly, each network slice may also come with different levels of security concerns originated from diverse industrial verticals, NFV platforms are thus required to provide means to ensure customized security policies. At a minimum, attacks on one slice must be unconditionally isolated from the others. It is also challenging to enforce security policies for slices spanning across multiple administrative domains. Therefore, NFV platforms are required to implement a brand-new set of mechanisms to eliminate these security concerns.

C. Integration with IoT

In the foreseeable 5G era, the IoT ecosystem is expected to accommodate an unprecedented deluge of data traffic generated by hundreds of billions of heterogeneous interconnected devices. NFV platforms are envisioned to enable massive deployment and flexible management of IoT services [161]. However, most of the platforms reviewed in this paper are not specifically designed for IoT use cases. At present, only a few NFV platforms are tailored for IoT in terms of user privacy [127], QoS [24], deployment [25], and still, a vast design space must be explored in addition to proof-of-concept implementations. In particular, it is unclear whether NFV platforms can handle enormous data traffic with ultra-low latency and high-throughput. While many existing NFV platforms manage to sustain 40/100 Gbps links using high-speed I/O techniques, their performances have not been tested under IoT configurations. The NFV platforms may also face scalability issues due to the immense number of connections from

densely distributed IoT devices, making service provisioning and scheduling even more critical. Energy consumption is yet another crucial challenge faced by IoT systems, especially under the progressively heavy and fluctuating traffic load. Albeit a large collection of scientific research has been devoted to energy efficiency, few NFV platforms include this as their design goals. Also, as IoT devices are increasingly deployed on third-party environments, it is equally important to ensure the security of IoT applications. Despite NFV platforms usually leverage data encryption and shield execution to secure VNFs, their applicability in 5G-IoT environments is yet to be comprehensively validated.

VI. CONCLUSION

As a novel paradigm to shift network management and service provisioning, NFV is expected to revolutionize the next-generation telecommunication networks. To accelerate the innovation and commercial adoption of NFV, a large spectrum of platforms have been implemented in the last eight years. While sharing the ultimate objective of promoting NFV, they usually tackle divergent problems in the NFV eco-system and embrace disparate design choices to achieve different performance metrics or service layer agreements, and few works have been devoted to interpreting this huge collection of platform implementations. In this paper, we concentrate on existing NFV platforms and strive for comprehending their design. After introducing the NFV reference architecture, we present our taxonomy on existing NFV platforms based on their focus on the life cycle of a network function. Then we explore the design space and investigate the various choices individual NFV platforms opt for to tackle different implementation challenges. Last but not least, we envision future research directions for NFV platforms concerning AI, network slicing, and IoT, and discuss the variety of challenges to overcome. We believe that our work presents a first-hand guideline for both network operators and developers to choose or design NFV solutions according to their respective requirements.

REFERENCES

- [1] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM CCR*, 2012.
- [2] V. Cisco, "Cisco visual networking index: Forecast and trends, 2017–2022," *White Paper*, 2018.
- [3] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *USENIX NSDI*, 2014.
- [4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM CCR*, 2008.
- [5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, 2015.
- [6] "The European Telecommunications Standards Institute Industry Specification Group on Network Function Virtualization (ETSI ISG NFV)," <https://www.etsi.org/committee/nfv>, 2019.
- [7] X. Li and C. Qian, "A survey of network function placement," in *IEEE CCNC*, 2016.

- [8] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE TNSM*, 2016.
- [9] S. Cherrared, S. Imadali, E. Fabre, G. Gössler, and I. G. B. Yahia, "A survey of fault management in network virtualization environments: Challenges and solutions," *IEEE TNSM*, 2019.
- [10] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Elsevier JNCA*, 2016.
- [11] W. Yang and C. Fung, "A survey on security in network functions virtualization," in *IEEE NetSoft*, 2016.
- [12] R. Mijumbi, J. Serrat, J.-L. Gorricho, S. Latré, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, 2016.
- [13] N. F. S. de Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network service orchestration: A survey," *Computer Communications*, 2019.
- [14] B. Yi, X. Wang, K. Li, M. Huang *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, 2018.
- [15] G. ETSI, "Network Functions Virtualisation (NFV): Architectural framework," *ETSI Gs NFV*, 2013.
- [16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, 2014.
- [17] P. Naik, A. Kanase, T. Patel, and M. Vutukuru, "libVNF: Building virtual network functions made easy," in *ACM SOCC*, 2018.
- [18] J. Duan, X. Yi, J. Wang, C. Wu, and F. Le, "NetStar: A future/promise framework for asynchronous network functions," *IEEE JSAC*, 2019.
- [19] "CloudBand: Adopt lean operations and increase business agility," <https://www.nokia.com/networks/solutions/cloudband/>, 2019.
- [20] "CloudNFV," <https://www.cloudfnv.com/>, 2020.
- [21] "SONATA NFV: Home," <http://sonatanfv.org/>.
- [22] C. Price, S. Rivera *et al.*, "OPNFV: An open platform to accelerate NFV," *White Paper: A Linux Foundation Collaborative Project*, 2012.
- [23] A. Bremner-Barr, Y. Harchol, and D. Hay, "OpenBox: a software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM*, 2016.
- [24] C. A. Ouedraogo, S. Medjah, C. Chassot, and J. Aguilar, "Flyweight network functions for network slicing in IoT," in *IEEE SaCoNet*, 2018.
- [25] Y.-Y. Shih, H.-P. Lin, A.-C. Pang, C.-C. Chuang, and C.-T. Chou, "An NFV-based service framework for IoT applications in edge computing environments," *IEEE TNSM*, 2019.
- [26] Z. Meng, J. Bi, H. Wang, C. Sun, and H. Hu, "MicroNF: An efficient framework for enabling modularized service chains in NFV," *IEEE JSAC*, 2019.
- [27] S. R. Chowdhury, Anthony, H. Bian, T. Bai, and R. Boutaba, "μNF: A disaggregated packet processing architecture," in *IEEE NetSoft*, 2019.
- [28] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible open middleboxes with commodity servers," in *ACM/IEEE ANCS*, 2012.
- [29] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *ACM SIGCOMM CCR*, 2015.
- [30] M. Bezahaf, A. Alim, and L. Mathy, "FlowOS: A flow-based platform for middleboxes," in *ACM HotMiddlebox '13*, 2013.
- [31] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *USENIX OSDI*, 2016.
- [32] R. Riggio, I. G. B. Yahia, S. Latré, and T. Rasheed, "Scylla: A language for virtual network functions orchestration in enterprise WLANs," in *IEEE/IFIP NOMS*, 2016.
- [33] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft *et al.*, "FLICK: Developing and running application-specific network services," in *USENIX ATC*, 2016.
- [34] O. Alipourfard and M. Yu, "Decoupling algorithms and optimizations in network functions," in *ACM HotNets*, 2018.
- [35] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "Microboxes: high performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *ACM SIGCOMM '18*, 2018.
- [36] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, J. Pi, and A. Shaikh, "A service-agnostic software framework for fast and efficient in-kernel network services," in *ACM/IEEE ANCS*, 2019.
- [37] Y. Jiang, Y. Cui, W. Wu, Z. Xu, J. Gu, K. Ramakrishnan, Y. He, and X. Qian, "SpeedyBox: Low-latency NFV service chains with cross-NF runtime consolidation," in *IEEE ICDSCS*, 2019.
- [38] M. Gallo and R. Laufer, "ClickNF: A modular stack for custom network functions," in *USENIX ATC 18*, 2018.
- [39] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM TOCS*, 2000.
- [40] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *USENIX NSDI*, 2018.
- [41] Y. Hu and T. Li, "Enabling efficient network service function chain deployment on heterogeneous server platform," in *IEEE HPCA*, 2018.
- [42] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *ACM SOSR*, 2017.
- [43] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "NFP: Enabling network function parallelism in NFV," in *ACM SIGCOMM '17*, 2017.
- [44] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for NFV applications," in *ACM SOSP '15*, 2015.
- [45] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *USENIX NSDI*, 2019.
- [46] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang, "SFC-Checker: Checking the correct forwarding behavior of service function chaining," in *IEEE NFV-SDN*, 2016.
- [47] F. Rath, J. Krude, J. Rüth, D. Schemmel, O. Hohlfeld, J. Á. Bitsch, and K. Wehrle, "Symperf: Predicting network function performance," in *ACM SIGCOMM Posters and Demos*, 2017.
- [48] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with Dysco," in *ACM SIGCOMM*, 2017.
- [49] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy, "Building a chain of high-speed VNFs in no time," in *IEEE HPSR*, 2018.
- [50] R. Bonafiglia, S. Miano, S. Nuccio, F. Risso, and A. Sapia, "Enabling NFV services on resource-constrained CPEs," in *IEEE Cloudnet*, 2016.
- [51] J. Ahrens, M. Strufe, L. Ahrens, and H. D. Schotten, "An AI-driven malfunction detection concept for NFV instances in 5G," *arXiv preprint arXiv:1804.05796*, 2018.
- [52] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-VITAL: A framework for characterizing the performance of virtual network functions," in *IEEE NFV-SDN*, 2015.
- [53] R. V. Rosa, C. Bertoldo, and C. E. Rothenberg, "Take your VNF to the gym: A testing framework for automated NFV performance benchmarking," *IEEE Communications Magazine*, 2017.
- [54] Q. Du, Z. Ni, R. Zhu, M. Xu, K. Guo, W. You, R. Huang, K. Yin, and Q. Zheng, "A service-based testing framework for NFV platform performance evaluation," in *IEEE ICRMS*, 2018.
- [55] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "Common: An automated container based network performance monitoring system," in *IFIP/IEEE IM*, 2017.
- [56] S. Geissler, S. Lange, F. Wamser, T. Zinner, and T. Hofffeld, "KOMon - Kernel-based online monitoring of VNF packet processing times," in *IEEE NetSys*, 2019.
- [57] I. J. Sanz, D. M. F. Mattos, and O. C. M. B. Duarte, "SFCPerf: An automatic performance evaluation framework for service function chaining," in *IEEE/IFIP NOMS*, 2018.

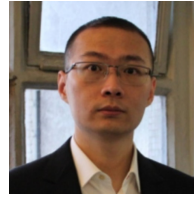
- [58] P. Naik, D. K. Shaw, and M. Vutukuru, "NFVPerf: Online performance monitoring and bottleneck detection for NFV," in *IEEE NFV-SDN*, 2016.
- [59] W. Wu, K. He, and A. Akella, "Perfsight: Performance diagnosis for software dataplanes," in *ACM IMC*, 2015.
- [60] R. V. Rosa, C. E. Rothenberg, and R. Szabo, "VBaaS: VNF benchmark-as-a-service," in *IEEE EWSDN '15*, 2015.
- [61] D. Cotroneo, L. De Simone, and R. Natella, "NFV-Bench: A dependability benchmark for network function virtualization systems," *IEEE TNSM*, 2017.
- [62] "Barometer Home, OPNFV wiki," <https://wiki.opnfv.org/display/fastpath/Barometer+Home>.
- [63] M. Dodare, Y. Taguchi, R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "NFV-VIPP: Catching internal figures of packet processing for accelerating development and operations of NFV-nodes," in *IFIP CNSM*, 2019.
- [64] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *USENIX NSDI*, 2019.
- [65] J. Gong, Y. Li, B. Anwer, A. Shaikh, and M. Yu, "DeepDiag: Detailed nfv performance diagnosis," in *ACM SIGCOMM 2019 Conference Posters and Demos*, 2019.
- [66] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *ACM SIGCOMM '18*, 2018.
- [67] W. Zhang, G. Liu, A. Mohammadkhan, J. Hwang, K. Ramakrishnan, and T. Wood, "SDNFV: Flexible and dynamic software defined control of an application-and flow-aware data plane," in *ACM Middleware Industry '16*, 2016.
- [68] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *USENIX NSDI*, 2012.
- [69] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. Lakshman, "UNO: Unifying host and smart NIC offload for flexible packet processing," in *ACM SoCC '17*, 2017.
- [70] A. Lombardo, A. Manzalini, G. Schembra, G. Faraci, C. Rametta, and V. Riccobene, "An open framework to enable NetFATE (Network Functions at the edge)," in *IEEE NetSoft*, 2015.
- [71] Z. Shen and Y. Zhang, "An NFV framework for supporting elastic scaling of service function chain," in *IEEE ICC*, 2018.
- [72] G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "Profiling and accelerating commodity NFV service chains with SCC," *Journal of Systems and Software*, 2017.
- [73] J. Soares, C. Gonçalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento, "Toward a telco cloud environment for service functions," *IEEE Communications Magazine*, 2015.
- [74] G. P. Katsikas, T. Barbette, D. Kostic, R. Steinert, and G. Q. Maguire Jr, "Metron: NFV Service Chains at the true speed of the underlying hardware," in *USENIX NSDI*, 2018.
- [75] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFs for flexible per-flow customization," in *ACM CoNEXT*, 2016.
- [76] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "SNF: Synthesizing high performance NFV service chains," *PeerJ Computer Science*, 2016.
- [77] L. Li, K. Ota, and M. Dong, "DeepNFV: A lightweight framework for intelligent edge network functions virtualization," *IEEE Network*, 2018.
- [78] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumathurai, and X. Fu, "NFVNice: Dynamic backpressure and scheduling for NFV service chains," in *ACM SIGCOMM '17*, 2017.
- [79] L. Zhang, C. Li, P. Wang, Y. Liu, Y. Hu, Q. Chen, and M. Guo, "Characterizing and orchestrating NFV-ready servers for efficient edge data processing," in *ACM IWQoS '19*, 2019.
- [80] S. R. Chowdhury, T. Bai, R. Boutaba, J. François *et al.*, "UNIS: A user-space non-intrusive workflow-aware virtual network function scheduler," in *IEEE CNSM*, 2018.
- [81] A. Singhvi, J. Khalid, A. Akella, and S. Banerjee, "SNF: Serverless network functions," *arXiv preprint arXiv:1910.07700*, 2019.
- [82] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling slos in network function virtualization," in *USENIX NSDI*, 2018.
- [83] Y. Hu, M. Song, and T. Li, "Towards full containerization in containerized network function virtualization," *ACM SIGOPS Operating Systems Review*, 2017.
- [84] D. Cotroneo, R. Natella, and S. Rosiello, "NFV-throttle: An overload control framework for network function virtualization," *IEEE TNSM*, 2017.
- [85] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based CPU in container environments," in *USENIX NSDI*, 2018.
- [86] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *USENIX NSDI*, 2013.
- [87] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamati, "Transparent flow migration for NFV," in *IEEE ICNP*, 2016.
- [88] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM CCR*, 2014.
- [89] B. Kothandaraman, M. Du, and P. Sköldström, "Centrally controlled distributed VNF state management," in *ACM HotMiddlebox '15*, 2015.
- [90] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, "U-HAUL: Efficient state migration in NFV," in *ACM APSys*, 2016.
- [91] L. Nobach, I. Rimac, V. Hilt, and D. Hausheer, "Statelet-based efficient and seamless NFV state transfer," *IEEE TNSM*, 2017.
- [92] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr," in *USENIX NSDI*, 2016.
- [93] M. Zhang, J. Bai, G. Li, Z. Meng, H. Li, H. Hu, and M. Xu, "When NFV meets ANN: Rethinking elastic scaling for ANN-based NFs," in *IEEE ICNP*, 2019.
- [94] S. Lange, H.-G. Kim, S.-Y. Jeong, H. Choi, J.-H. Yoo, and J. W.-K. Hong, "Machine learning-based prediction of VNF deployment decisions in dynamic networks," in *IEEE APNOMS*, 2019.
- [95] P. Sun, J. Lan, J. Li, Z. Guo, Y. Hu, and T. Hu, "Efficient flow migration for nfv with graph-aware deep reinforcement learning," *Computer Networks*, p. 107575, 2020.
- [96] J. Duan, X. Yi, S. Zhao, C. Wu, H. Cui, and F. Le, "NFVactor: A resilient NFV system using the distributed actor model," *IEEE JSAC*, 2019.
- [97] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo *et al.*, "Rollback-recovery for middleboxes," in *ACM SIGCOMM CCR*, 2015.
- [98] M. T. Raza, S. Lu, and M. Gerla, "vEPC-sec: Securing LTE network functions virtualization on public cloud," *IEEE Transactions on Information Forensics and Security*, 2019.
- [99] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy, "Splitbox: Toward efficient private network function virtualization," in *ACM HotMiddlebox '16*, 2016.
- [100] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *USENIX NSDI*, 2016.
- [101] G. A. F. Rebello, I. D. Alvarenga, I. J. Sanz, and O. C. M. Duarte, "BSec-NFVO: A blockchain-based security for network function virtualization orchestration," in *IEEE ICC*, 2019.

- [102] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," *Hasp@ isca*, 2013.
- [103] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV states by using SGX," in *ACM SDN-NFV Security '16*, 2016.
- [104] M. Coughlin, E. Keller, and E. Wustrow, "Trusted click: Overcoming security issues of NFV in the cloud," in *ACM SDN-NFV Sec'17*, 2017.
- [105] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "ShieldBox: Secure middleboxes using shielded execution," in *ACM SOSR*, 2018.
- [106] E. Marku, G. Biczók, and C. Boyd, "Towards protected VNFs for multi-operator service delivery," in *IEEE NetSoft*, 2019.
- [107] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "LightBox: Full-stack protected stateful middlebox at lightning speed," in *ACM SIGSAC CCS*, 2019.
- [108] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding network functions in the cloud," in *USENIX NSDI*, 2018.
- [109] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *USENIX NSDI*, 2017.
- [110] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *ACM SOSR '15*, 2015.
- [111] Z. Xu, Y. Cui, and Y. Jiang, "CoNFV: An endhost-cloud collaborated network function virtualization framework," in *IEEE IMCEC*, 2018.
- [112] C. K. Dominicini, G. L. Vassoler, L. F. Meneses, R. S. Villaca, M. R. Ribeiro, and M. Martinello, "VirtPhy: Fully programmable NFV orchestration architecture for edge data centers," *IEEE TNSM*, 2017.
- [113] R. Cziva and D. P. Pazaros, "Container network functions: Bringing NFV to the network edge," *IEEE Communications Magazine*, 2017.
- [114] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, "NFV orchestration framework addressing SFC challenges," *IEEE Communications Magazine*, 2017.
- [115] A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, and F.-J. Westphal, "Unifying cloud and carrier network: EU FP7 project UNIFY," in *IEEE/ACM UCC*, 2013.
- [116] S. Van Rossem, X. Cai, I. Cerrato, P. Danielsson, F. Németh, B. Pechenot, I. Pelle, F. Rizzo, S. Sharma, P. Sköldström *et al.*, "NFV service dynamicity with a DevOps approach: Insights from a use-case realization," in *IFIP/IEEE IM*, 2017.
- [117] "ONAP - Home," <https://www.onap.org/>, 2020.
- [118] "Open Source Mano," <https://osm.etsi.org/>, 2020.
- [119] D. Lopez, "OpenMANO: The dataplane ready open source NFV MANO stack," in *IETF Meeting Proceedings, Dallas, Texas, USA*, 2015.
- [120] "OPEN BATON: An extensible and customizable NFV MANO-compliant framework," <https://openbaton.github.io/>, 2019.
- [121] G. Xilouris, M.-A. Kourtis, M. J. McGrath, V. Riccobene, G. Petralia, E. Markakis, E. Palis, A. Georgios, G. Gardikis, J. F. Riera *et al.*, "T-nova: Network functions as-a-service over virtualised infrastructures," in *IEEE NFV-SDN*, 2015.
- [122] J. F. Riera, J. Batallé, J. Bonnet, M. Días, M. McGrath, G. Petralia, F. Liberati, A. Giuseppi, A. Pietrabissa, A. Ceselli *et al.*, "TeNOR: Steps towards an orchestration platform for multi-PoP NFV deployment," in *IEEE NetSoft*, 2016.
- [123] W. Shen, M. Yoshida, K. Minato, and W. Imajuku, "vConductor: An enabler for achieving virtual network integration as a service," *IEEE Communications Magazine*, 2015.
- [124] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE TNSM*, 2015.
- [125] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *ACM HotMiddlebox'16*, 2016.
- [126] K. Yasukata, F. Huici, V. Maffione, G. Lettieri, and M. Honda, "HyperNF: Building a high performance, high utilization and fair NFV platform," in *ACM SOCC*, 2017.
- [127] M. Gallo, S. Ghamri-Doudane, and F. Pianese, "ClimBOS: A modular NFV cloud backend for the internet of things," in *IFIP NTMS*, 2018.
- [128] C. Zheng, Q. Lu, J. Li, Q. Liu, and B. Fang, "A flexible and efficient container-based NFV platform for middlebox networking," in *ACM SAC*, 2018.
- [129] "NFF-Go -Network Function Framework for GO (former YANFF)," <https://github.com/intel-go/nff-go>, 2019.
- [130] R. Kawashima and H. Matsuo, "IOVTee: A fast and pragmatic software-based zero-copy/pass-through mechanism for NFV-nodes," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–6.
- [131] Y. Yuan, Y. Wang, R. Wang, and J. Huang, "HALO: Accelerating flow classification for scalable packet processing in NFV," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.
- [132] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4SC: Towards high-performance service function chain implementation on the p4-capable device," in *IFIP/IEEE IM*, 2019.
- [133] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane reconfiguration," in *IEEE CNSM*, 2018.
- [134] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack," in *ACM SIGCOMM CCR*, 2014.
- [135] Z. Ni, G. Liu, D. Afanasev, T. Wood, and J. Hwang, "Advancing network function virtualization platforms with programmable NICs," in *IEEE LANMAN*, 2019.
- [136] B. Li, K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 1–14.
- [137] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *USENIX ATC*, 2019.
- [138] A. Dhakal and K. Ramakrishnan, "NetML: An NFV platform with efficient support for machine learning applications," in *IEEE NetSoft*, 2019.
- [139] X. Yi, J. Wang, J. Duan, W. Bai, C. Wu, Y. Xiong, and D. Han, "FlowShader: A generalized framework for GPU-accelerated VNF flow processing," in *IEEE ICNP*, 2019.
- [140] X. Yi, J. Duan, and C. Wu, "GPUNFV: a GPU-accelerated NFV system," in *ACM APNet'17*, 2017.
- [141] Z. Zheng, J. Bi, C. Sun, H. Yu, H. Hu, Z. Meng, S. Wang, K. Gao, and J. Wu, "Gen: A GPU-accelerated elastic framework for NFV," in *ACM APNet '18*, 2018.
- [142] Z. Zheng, J. Bi, H. Wang, C. Sun, H. Yu, H. Hu, K. Gao, and J. Wu, "Grus: Enabling latency SLOs for GPU-accelerated NFV systems," in *IEEE ICNP*, 2018.
- [143] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-NET: Effective GPU sharing in NFV systems," in *USENIX NSDI*, 2018.
- [144] T. Zhang, "NFV platform design: A survey," *arXiv: Networking and Internet Architecture*, 2020.
- [145] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *USENIX NSDI*, 2014.
- [146] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE ANCS*, 2015.
- [147] "Data Plane Development Kit," <https://www.dpdk.org/>, 2020.
- [148] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *USENIX Security 12*, 2012.

- [149] "Linux user space library for network socket acceleration based on RDMA compatible network adaptors," <https://github.com/Mellanox/libma>, 2019.
- [150] "eBPF - extended Berkeley Packet Filter," <https://prototype-kernel.readthedocs.io/en/latest/bpf/>, 2020.
- [151] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *ACM CoNEXT'18*, 2018, pp. 54–66.
- [152] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," *Tech. Rep. UCB/EECS-2015-155*, 2015.
- [153] L. Rizzo and G. Lettieri, "VALE, a switched ethernet for virtual machines," in *ACM CoNEXT*, 2012.
- [154] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of Open vSwitch," in *USENIX NSDI*, 2015.
- [155] "Open vSwitch with DPDK," <http://docs.openvswitch.org/en/latest/intro/install/dpdk/>, 2020.
- [156] T. Zhang, L. Linguaglossa, J. Roberts, L. Iannone, M. Gallo, and P. Giaccone, "A benchmarking methodology for evaluating software switch performance for NFV," in *IEEE NetSoft*, 2019.
- [157] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," in *ACM CoNEXT*, 2019.
- [158] R. Kawashima, H. Nakayama, T. Hayashi, and H. Matsuo, "Evaluation of forwarding efficiency in NFV-nodes toward predictable service chain performance," *IEEE TNSM*, 2017.
- [159] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proceedings of the IEEE*, 2019.
- [160] A. A. Barakabitze, A. Ahmad, R. Mijumbi, and A. Hines, "5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106984, 2020.
- [161] G. A. Akpakwu, B. J. Silva, G. P. Hancke, and A. M. Abu-Mahfouz, "A survey on 5G networks for the Internet of Things: Communication technologies and challenges," *IEEE Access*, 2017.



Tianzhu Zhang is a research engineer at Nokia Bell Labs. He received his B.S. degree from Huazhong University of Science and Technology, Wuhan, China, in 2012. Afterward, he received the M.S. degree in 2014, and the Ph.D. degree in 2017, both from Politecnico di Torino, Turin, Italy. From 2017 to 2019, he was a PostDoc researcher at Telecom ParisTech and LINCOS, under a research grant from Cisco Systems. He joined Nokia Bell Labs in August 2020. His research interests include Artificial Intelligence, Edge computing, Robotics, and Big Data.



computing.

Han Qiu (Member, IEEE) received the B.E. degree from the Beijing University of Posts and Telecommunications, Beijing, China, in 2011, the M.S. degree from Telecom-ParisTech (Institute Eurecom), Biot, France, in 2013, and the Ph.D. degree in computer science from the Department of Networks and Computer Science, Telecom-ParisTech, Paris, France, in 2017. He is currently a Researcher with the LINCOS Laboratory, Paris, and co-affiliated with Telecom Paris. His research interests include AI security, big data security, applied cryptography, and cloud



Leonardo Linguaglossa is an assistant professor at Telecom Paris (France). He received his master degree in telecommunication engineering at University of Catania (Italy) in 2012. He pursued a Ph.D. in Computer Networks in 2016 through a joint doctoral program with Alcatel-Lucent Bell Labs (nowadays Nokia), INRIA and University Paris 7. Leonardo's research interests focus on architecture, design and prototyping of systems for high-speed software packet processing, future Internet architecture and SDN.



conference proceedings. He serves/served as Series Editor for the IEEE Communications Magazine, Associate Editor for the IEEE Communications Letters, and Technical Program Co-Chair for IEEE-sponsored international workshops and conferences.

Walter Cerroni (Senior Member, IEEE) is an Associate Professor of communication networks at the University of Bologna, Italy. His recent research interests include software-defined networking, network function virtualization, service function chaining in cloud computing platforms, intent-based northbound interfaces for multi-domain/multi-technology virtualized infrastructure management, modeling and design of inter- and intra-data center networks. He co-authored more than 130 articles published in the most renowned international journals, magazines and



and cloud computing systems.

Paolo Giaccone (Senior Member, IEEE) received the Dr.Ing. and Ph.D. degrees in telecommunications engineering from the Politecnico di Torino, Torino, Italy, in 1998 and 2001, respectively. During the summer of 1998, he was with High Speed Networks Research Group, Lucent Technology-Bell Labs, Holmdel, NJ, USA. From 2000 to 2001 and in 2002, he was with Information Systems Networking Lab, Department of Electrical Engineering, Stanford University, Stanford, CA, USA. He is currently an Associate Professor with the Department of Elec-