UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Hadi Razzaghi Kouchaksaraei

Orchestrating Network Services Using Multi-domain, Heterogeneous Resources

Dissertation

submitted to the

Faculty of Electrical Engineering,
Computer Science, and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

Paderborn, August 2020

Referees:
Prof. Dr. Holger Karl, University of Paderborn, Germany
Asst. Prof. Dr. Panagiotis Papadimitriou, University of Macedonia, Greece

Additional committee members:
Prof. Dr. Stefan Böttcher, University of Paderborn, Germany
Prof. Dr. Christian Scheideler, University of Paderborn, Germany
Jun. Prof. Dr. Sevag Gharibian, University of Paderborn, Germany

Submission: August 2020
Examination: 28.09.2020

Abstract

Network Function Virtualization (NFV) has emerged to improve the flexibility and agility of networks and reduce the capital and operational expenditures of network providers. This is achieved by softwarising network functions and automating their management. To automate management, the European Telecommunication Standards Institute (ETSI) NFV group proposed a reference architecture of a framework called Management and Orchestration (MANO). A MANO framework is responsible for managing the NFV infrastructure and the lifecycle of network functions as well as orchestrating services as a whole.

To have a full-fledge MANO framework, however, we are facing multiple challenges. In this dissertation, I explore some of these challenges and introduce concepts and solutions to mitigate them. To this end, first, I investigate flexibility and programmability of MANO framework. I introduce a concept called Specific Management (SM) that allows to customise the MANO framework on the fly. Using a simulation-based approach, I evaluate the SM concept by comparing it with a rigid solution. Also, by experimental evaluation, I analyse the management overhead caused by employing the SM concept in a MANO framework.

The second challenge that I address in this thesis is the scalability and agility of MANO frameworks. In this regard, I introduce a benchmarking framework to quantify the performance of MANO frameworks in terms of resource consumption and service deployment time. I also quantify the effect of topological distance on service deployment time using Amazon Web Service (AWS) resources.

Supporting heterogeneous resources is the third challenge that I address. I introduce a multi-domain NFV MANO framework that supports heterogeneous resources by unifying resources from different domains and jointly orchestrating them.

The last challenge is the dynamic deployment of services over heterogeneous resources. I introduce multi-version services, a concept that realises services on multiple versions based on virtualization environments (virtual machine and container) and compute resources (General Purpose Processor (GPP), Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA)). I integrated this concept into a MANO framework and analysed different aspects of the concept.

With these contributions in place, the programmability of MANO frameworks is improved, scalable MANO framework can be realised, and network services can be dynamically provisioned over heterogeneous resources.

Zusammenfassung

Network Function Virtualization (NFV) wurde entwickelt, um die Flexibilität und Agilität von Netzen zu verbessern und die Investitionen und Betriebsausgaben von Netzanbietern zu reduzieren. Dies wird durch die "Softwarisierung" von Netzfunktionen und die Automatisierung ihrer Verwaltung erreicht. Für diese Automatisierung schlug die European Telecommunication Standards Institute (ETSI) NFV Gruppe eine Referenzarchitektur mit dem Namen Management and Orchestration (MANO) vor. Das darin beschriebene MANO-Framework ist für die Verwaltung der NFV Infrastruktur und des Lebenszyklus von Netzfunktionen sowie für die Orchestrierung von Diensten als Ganzes verantwortlich.

Für ein vollwertiges MANO-Framework müssen jedoch mehrere Herausforderungen gemeistert werden. In dieser Dissertation untersuche ich einige dieser Herausforderungen und stelle entsprechende Konzepte und Lösungen vor. Ich untersuche zunächst das Problem der Flexibilität und Programmierbarkeit des MANO-Frameworks. Ich stelle ein Konzept namens Specific Management (SM) vor, mit dem dasMANO-Framework im laufenden Betrieb angepasst werden kann. Mit einem simulationsbasierten Ansatz bewerte ich das SM-Konzept, indem ich es mit einer statischen Lösung vergleiche. Außerdem analysiere ich experimentell den Verwaltungsaufwand, der durch die Verwendung des SM-Konzepts in einem MANO-Framework verursacht wird.

Die zweite Herausforderung ist die Skalierbarkeit und Agilität von MANO-Frameworks. In diesem Zusammenhang stelle ich ein Benchmarking-Framework vor, mit dem die Leistung von MANO-Frameworks in Bezug auf Ressourcenverbrauch und Servicebereitstellungszeit quantifiziert wird. Ich quantifiziere auch den Effekt der topologischen Entfernung in der Servicebereitstellungszeit mithilfe von Amazon Web Service (AWS)-Ressourcen.

Die Unterstützung heterogener Ressourcen ist die dritte Herausforderung, die ich in dieser Arbeit anspreche. Zu diesem Zweck stelle ich ein Multi-Domain-NFV-MANO-Framework vor, das heterogene Ressourcen unterstützt, indem Ressourcen aus verschiedenen Domänen vereinheitlicht und gemeinsam orchestriert werden.

Die letzte Herausforderung in dieser Dissertation ist die dynamische Bereitstellung von Diensten über heterogene Ressourcen. In diesem Zusammenhang stelle ich Dienste mit mehreren Versionen vor; ein Konzept, das Dienste in mehreren Versionen basierend auf Virtualisierungsumgebungen (virtuelle Maschine und Container) und Rechenressourcen (General Purpose Processor (GPP), Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA)) realisiert. Ich habe dieses Konzept in ein MANO-Framework integriert und verschiedene Aspekte des Konzepts analysiert.

Mit diesen Beiträgen wird die Programmierbarkeit von MANO-Frameworks verbessert, ein skalierbares MANO-Framework realisiert und Netzwerkdienste können dynamisch über heterogene Ressourcen bereitgestellt werden.

Acknowledgements

I would like to express my very great appreciation to Prof. Dr. Holger Karl for his support throughout my research. Your valuable advice, constructive criticism, and encouragement made this dissertation possible. I would also like to extend my deepest gratitude to Prof. Dr. Panagiotis Papadimitrou for leading me to this journey. Without your support, this journey could not have started.

Many thanks to my dear colleagues in the computer network group for all the interesting discussions and close collaboration. In particular, I wish to thank Manuel, Sevil, Stefan, and Marvin for the joint work and fruitful discussions that have enriched my dissertation. I am also grateful to have the opportunity to be part of SONATA and 5G-PICTURE projects in which I met and worked with many bright colleagues.

My friends Maryam, Hassan & Maryam, and Mohammad: I cannot thank you enough for all the supports you have given to me during this time. Without having you on my back in my life's ups and downs, this step could not be achievable for me.

I am deeply indebted to my parents Baba and Maman, my siblings Komail and Hoda, and my sister-in-law Akram for believing in me and supporting all the decisions I made in my life. Your unconditional love and support is the main reason for my achievements.

Contents

List of Figures

List of Tables

1

Introduction

Cloud computing has revolutionised the Information Technology (IT) infrastructure by transforming the way resources are provided. Technologies that have been developed for cloud computing (e.g., virtualisation) allow companies to reduce the capital and operational costs of their IT infrastructure and reduce time-to-market of new services. Cloud computing also boosts innovation by allowing new ideas to be implemented with significantly less upfront cost. Leveraging this opportunity, applications such as WhatsApp and Telegram have emerged. These applications have cannibalised the classical communication services such as Short Message Service (SMS) and telephony. This has significantly hampered the conventional revenue flow of incumbent network providers, and, as a result, they are seeking new business opportunities and revenue streams.

One potential new revenue stream for network providers is supporting new verticals such as connected vehicles, Internet of Things (IoT), remote surgery, and smart manufacturing (industry 4.0). Realising such new verticals requires excellent network connectivity, which can be provided by network providers. Specifically, these verticals require network connectivity with ultra-high data rate, ultra-low latency, and high dependability. To fulfil these requirements, however, network providers need to improve the flexibility, programmability, and agility of their networks [46]. Adding a new network service in legacy networks is very time-consuming and inefficient. This is because today, network services are provided on proprietary appliances and deploying them involves manual operations.

To solve this issue, network providers followed the path of cloud computing. They opted to softwarise network elements, which allows network services to be offered on-demand. Network softwarization not only improves the flexibility of the network but also reduces the capital and operational expenditures of network providers.

Network softwarization is enabled using two main technologies, namely Network Function Virtualization (NFV) and Software-Defined Network (SDN). These two technologies transform the rigid and fully hardware-based legacy network to a flexible software-based network that is automated using a centralised control entity. The SDN and NFV technologies are explained in detail in Chapter 2.

Multiple stakeholders are involved in providing a network service. As it is shown in Fig. 1.1, a *service provider*, on the top, provides the network service artefacts that can be, for example, service descriptors and images. Using service artefacts, a *network provider* deploys and manages the network services over an infrastructure, that is provided by *infrastructure operator*. The network services are, then, provided to *service tenants*, which can be verticals. The services are eventually used by the end-users of the *service tenants* who want to access the tenant's services (e.g., to access a web service).

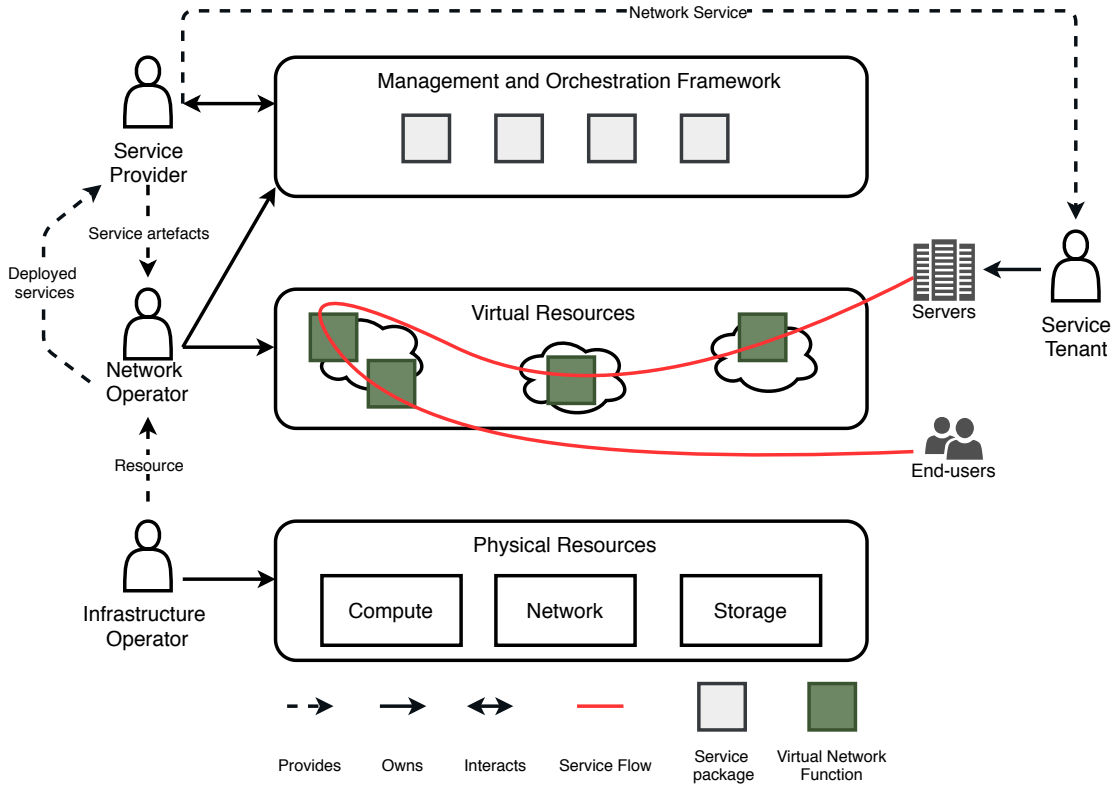


Figure 1.1: The stakeholders involved in providing network services

To automate the deployment and management of network services, the European Telecommunication Standards Institute (ETSI) NFV has introduced a reference software architecture called Management and Orchestration (MANO) Framework. A MANO framework manages the lifecycle of network services. The

lifecycle of a network service, usually, includes on-boarding, instantiation, configuration, start, stop, update, and termination of the service. This dissertation addresses some of the challenges concerning MANO frameworks, which are explained in the next section.

1.1 Challenges in NFV MANO frameworks

To have a full-fledged MANO framework, we need to overcome some challenges. Examples of these challenges that will be addressed by this dissertation are specified in the following.

1.1.1 Supporting Diverse Service Requirements

Supporting the wide variety of network services' management and orchestration requirements is one of the main challenges that NFV is dealing with. While general management requirements such as Virtualized Network Function (VNF) resource requirements can be specified by the service developers using service descriptors (See Chapter 2), specific management operations like VNF-specific configuration cannot be performed by these descriptors. On the other hand, it is inefficient and also very challenging for MANO frameworks to provide all specific-management operations for every individual network service and their constituent VNFs. I address this challenge in Chapter 3 by implementing and evaluating the Specific Management (SM) concept. SM allows customizing an NFV MANO framework to support service requirements that are not natively supported by such a framework.

1.1.2 Scalability and Agility of MANO frameworks

In some situations, MANO frameworks should handle a large number of service deployment requests. For example, in scenarios like IoT where MANO frameworks are used to deploy IoT-related NFV services, hundreds of service instantiation requests might be sent to a MANO framework over a short time frame [27].To handle such scenarios, MANO frameworks need to be scalable.

Moreover, in some scenarios, MANO frameworks need to deal with Network Function Virtual Infrastructure (NFVI)s that are spread across a large geographical area (e.g., country or continental). An example of these scenarios is the 5G network in which NFV plays a vital role [1]. The 5G network leverages highly-distributed NFVIs to meet the 1 ms round trip latency [4]. In these scenarios, the topological distance between the NFV MANO and NFVIs affects the NFV services deployment time and consequently, the agility of MANO frameworks in deploying new services.

Scaling out and scaling up of MANO frameworks are known scaling strategies that can be used to support the specific deployment time of an NFV service when the load suddenly increases. Also, placing MANO instances close to Virtualized Infrastructure Manager (VIM)s where the services will be running is a viable solution to improve the agility of service deployment. Although these solutions

sound promising in theory, there is no real-word analysis and data about the current MANO frameworks to be used when deciding how to scale a MANO framework's components. Chapter 4 addresses this problem and provide analysis on software-based limitation of MANO frameworks and the effect of topological distance on the service deployment time.

1.1.3 Supporting Heterogeneous Resources

NFV infrastructures consist of heterogeneous resources that are utilised for different purposes. For example, acceleration resources such as Field Programmable Gate Array (FPGA)s are used to improve the performance of VNFs [36]. Also, different virtualisation environments can be used to host VNFs. For example, Virtual Machine (VM)s are used to host security-critical VNFs, and Container (CN)s are used when performance and deployment agility is in a higher priority [45]. Therefore, a MANO framework must support heterogeneous resource (i.e., I consider virtualisation environments as resources as well).

In the NFV orchestrator, Cloud Management System (CMS)s such as OpenStack [1] and Kubernetes [2] are used as VIMs. In a single-domain (i.e., a domain here refers to a cluster of resources that is managed by a specific CMS) MANO framework, the MANO is limited to a particular type of resources that are supported by its CMS. For example, using OpenStack limits the system in VMs that can run on General Purpose Processor (GPP)s and Graphics Processing Unit (GPU), and no other resources can be supported. In another example, using Kubernetes, resources such as containers, GPPs, GPUs can be supported, but it is not possible to run VMs and deploy VNFs on FPGAs. Public cloud solutions such as Amazon Web Service (AWS) can provide a wider range of heterogeneous resources (including FPGAs), but, being closed-source, AWS is not suitable for managing in-house resources.

To support all resources mentioned above, in Chapter 5, I implemented and evaluated an NFV MANO that deploys services across multiple domains. The NFV MANO framework implemented in that chapter has been used as the main framework in the EU project 5G-PICTURE [3].

1.1.4 Dynamic Provisioning of Services over Heterogeneous Resources

To reduce the cost and enhance the flexibility of network functions, VNFs are deployed on Commercial Off-The-Shelf (COTS) resources instead of specialized hardware. This, however, downgrades the performance of network functions. To mitigate this issue, leveraging acceleration hardware such as GPU and FPGA has been suggested [36]. This is because accelerators provide better parallelization and pipelining for compute- and network-intensive VNFs. However, studies show

[1] `https://www.openstack.org/`, accessed June 2020
[2] `https://kubernetes.io/`, accessed June 2020
[3] `https://www.5g-picture-project.eu`, accessed June 2020

that using accelerators are not beneficial for all cases [24]. One of the reasons is that accelerators are expensive resources, possibly increasing the service cost. Consequently, the use of accelerators needs to be decided for each situation individually.

In Chapter 6, I address this problem by implementing and evaluating a framework that dynamically deploys services over heterogeneous resources. In this framework, a service is implemented in different versions based on the compute resource and the virtualization environment it is intended to run over. Having different versions of the service, the framework dynamically deploys the right version based on the service demand.

1.2 Contributions

This work has been done throughout my PhD study from May 2016 to April 2020. During this time, I contributed to the open-source project SONATA and also created an open-source project called Pishahang. I supervised three student project groups, three bachelor theses and one master thesis. At the time of writing this thesis, my publications have been cited 119 times. I published six papers as first author in peer-reviewed conferences and co-authored five other conference papers – all are listed in the following.

- H. R. Kouchaksaraei, A. P. S. Venkatesh, A. Churi, M. Illian, and H. Karl. Dynamic Provisioning of Network Services on Heterogeneous Resources. In *2020 European Conference on Networks and Communications (EuCNC)*, June 2020.

- H. R. Kouchaksaraei and Holger Karl. Quantitative Analysis of Dynamically Provisioned Heterogeneous Network Services. In *Proceedings of the 15th International Conference on Network and Service Management*, CNSM '19. IFIP, 2019.

- H. R. Kouchaksaraei and Holger Karl. Service Function Chaining Across OpenStack and Kubernetes Domains. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, DEBS '19, pages 240–243, New York, NY, USA, 2019. ACM.

- H. R. Kouchaksaraei, T. Dierich, and H. Karl. Pishahang: Joint Orchestration of Network Function Chains and Distributed Cloud Applications. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 344–346, June 2018.

- H. R. Kouchaksaraei, S. Dräxler, M. Peuster, and H. Karl. Programmable and Flexible Management and Orchestration of Virtualized Network Functions. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9, June 2018.

- H. R. Kouchaksaraei and Holger Karl. Joint Orchestration of Cloud-Based Microservices and Virtual Network Functions. In *The Ninth International Conference on Cloud Computing, GRIDs, and Virtualization CLOUD COMPUTING*, pages 153–154, February 2018.
- D. Camps-Mur, F. Canellas, A. Machwe, J. Paracuellos, K. Choumas, D. Giatsios, T. Korakis, and H. R. Kouchaksaraei. 5GOS: Demonstrating Multi-domain Orchestration of End-to-end Virtual RAN Services. In *2020 6th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2020.
- S. Dräxler, H. Karl, H. R. Kouchaksaraei, A. Machwe, C. Dent-Young, K. Katsalis, and K. Samdanis. 5G OS: Control and Orchestration of Services on Multi-Domain Heterogeneous 5G Infrastructures. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9, June 2018.
- S. Van Rossem, M. Peuster, L. Conceicao, H. R. Kouchaksaraei, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester. A Network Service Development Kit Supporting the End-to-end Lifecycle of NFV-based Telecom Services. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–2, Nov 2017.
- M. Peuster, S. Dräxler, H. R. Kouchaksaraei, S. v. Rossem, W. Tavernier, and H. Karl. A Flexible Multi-pop Infrastructure Emulator for Carrier-grade MANO Systems. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–3, July 2017.
- S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris. SONATA: Service Programming and Orchestration for Virtualized Software Networks. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 973–978, May 2017.

The content of this thesis is mostly based on the papers that I published as first author. Some of the figures and verbatim used in this thesis are copied from the corresponding publications. To ease the flow of reading, such copies from my own publications are not explicitly marked, but all sources are mentioned at the beginning of each chapter. Parts of these publications mentioned above, in which I did not have a significant contribution, are excluded from this dissertation.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows:

- **Chapter 2: Technical Background**
  In this chapter, I explain the technical background required for the following chapters.

- **Chapter 3: Programmable Management and Orchestration of Network Services**
  In this chapter, I introduce the concept of SM. SM improves the flexibility of MANO frameworks and allows the service developers to program the management and orchestration of their services. I implemented the SM platform and integrated it into the SONATA MANO framework. Using a simulation-based approach, I evaluated the SM concept by comparing it with rigid solutions. Also, by experimental evaluation, I analysed the management overhead caused by employing the SM concept in a MANO framework. This chapter is based on [22].

- **Chapter 4: Scalable and Agile Management and Orchestration of Network Services**
  The second NFV MANO challenge that I address in this dissertation is the agility and scalability of MANO frameworks. In this chapter, I introduce a benchmarking framework that is used to evaluate MANO frameworks. Using the benchmarking framework, I analysed the scalability and agility of MANO frameworks. The benchmarking framework has been developed by a student project group, called SCRAMBLE[4], under my supervision.

- **Chapter 5: Multi-domain Management and Orchestration of Network Services**
  Another challenge concerning the MANO frameworks is to support the wide variety of heterogeneous resources. To this end, I implemented an open-source multi-domain MANO, called Pishahang. Pishahang supports heterogeneous resources by a multi-domain approach where resources from domains are jointly orchestrated. In this chapter, I describe the design and implementation of Pishahang and also explain how it was used in the 5G-PICTURE project to support 5G services.

  This chapter is based on four papers, namely [23], [25], [21], and [8]. Tobias Dierich contributed to the development of Kubernetes adaptor as part of his Bachelor thesis [10]. Also, Dennis Meier contributed to the development of the AWS adaptor as part of his Bachelor thesis [33]. The project group ENTANGLE[5] also contributed to integrating the adaptors and also developing the OpenStack adaptor.

- **Chapter 6: Dynamic Management and Orchestration of Network Services**
  The last NFV MANO challenge that I address is the dynamic deployment of services over heterogeneous resources. In this regard, I introduce multi-version services and classify the services that can benefit from dynamic deployment. In this chapter, I analysed an example of multi-version service in terms of performance and cost. Also, I extended Pishahang MANO framework to support multi-version services. Using the extension, I analysed

[4] `https://github.com/CN-UPB/pg-scrambLe`, accessed July 2020
[5] `https://bit.ly/3jSI34q`, accessed July 2020

the management overhead that multi-version services impose to the MANO framework. This chapter is based on two papers, namely [24] and [26].

- **Chapter 7: Final Thoughts**
  In this final chapter, I conclude the work presented in previous chapters and give an outlook on further research directions.

2

Technical Background

In this chapter, I introduce concepts and technologies on which my work is based. Network Function Virtualization (NFV) and Software-Defined Network (SDN) are the two major concepts that are discussed in this chapter.

2.1 Network Function Virtualization

Telecommunication service providers had been long suffered from the rigidity of middleboxes. Middleboxes are hardware appliances that provide functionalities such as inspection, transformation, filtering, and manipulation of networking traffic. These functionalities are called Network Function (NF)s and, mainly, are used to improve network security and performance. Examples of NFs are Firewall, Network Address Translation (NAT), Deep Packet Inspection (DPI), and Load Balancers. In middleboxes, NFs are coupled to dedicated hardware. This causes multiple challenges such as the following:

- **High capital expenses:** Employing middleboxes increases the capital expenditure as they are expensive appliances.
- **High operational expenses:** Using middleboxes also increases the operational costs of networks. Examples of operational costs are power consumption and manpower to configure and manage middleboxes.
- **Rigid to scale up/down and upgrade:** Scaling up and down in middleboxes mean adding and removing new appliances, respectively. This is difficult to achieve as it is an expensive and time-consuming process. The same holds for upgrading the middleboxes.
- **Fixed location:** They are fixed to one location, and moving them around to, for example, keep them close to the service users is infeasible.
- **Long time-to-market of new NFs:** Launching a new NF is a tedious process that includes development, shipping, and adopting a new appliance. This is a long process and can take months until the NF becomes production-ready.
- **Limited innovation:** development and testing new NFs and features will be expensive as new appliances need to be manufactured. This increases the cost of innovation that requires continues development and testing.

Inspired by Could computing, NFV is proposed by the European Telecommunication Standards Institute (ETSI) to address the above-mentioned challenges. In NFV, using virtualization technologies, NFs are decoupled from their hardware and offered as software products running on Commercial Off-The-Shelf (COTS) devices. These software products are called Virtualized Network Function (VNF)s. Using NFV, the capital expenditures are reduced as VNFs run on COTS resources that are cheaper than proprietary hardware used in middleboxes. Virtualization provides better resource utilization as it enables resource sharing. This reduces operational expenditures, such as power consumption. As proved in cloud computing, virtualization also allows to scale up/down the NFs by simply allocating more/less virtual resources to VNFs, respectively. In NFV, VNFs are portable and can run on any location that virtual resources are provided (e.g., data centres).

NFV also reduces the time-to-market of VNFs as there is no shipping and adopting involved and development of new VNFs is significantly faster (i.e., enabled by DevOps). NFV also lowers the cost of innovation as deploying new innovative services as VNFs do not need to run on expensive resources.

In NFV, multiple VNFs are chained together to deliver an NFV service. To accelerate time-to-market of NFV services, deployment and lifecycle management of NFV services are automated by NFV Management and Orchestration (MANO) framework. In the next section, NFV MANO is described in detail.

2.2 Management and Orchestration Framework

NFV MANO is a framework consisting of multiple functional blocks that handle the tasks required to manage and orchestrate NFV services. ETSI NFV has introduced a reference architecture [15] for NFV MANO that is shown in Fig. 2.1. The architecture can be described in three parts: part 1) data repositories, part 2) functional blocks belonging to NFV MANO, and part 3) functional blocks interacting with NFV MANO. In the following, I describe these parts in detail.

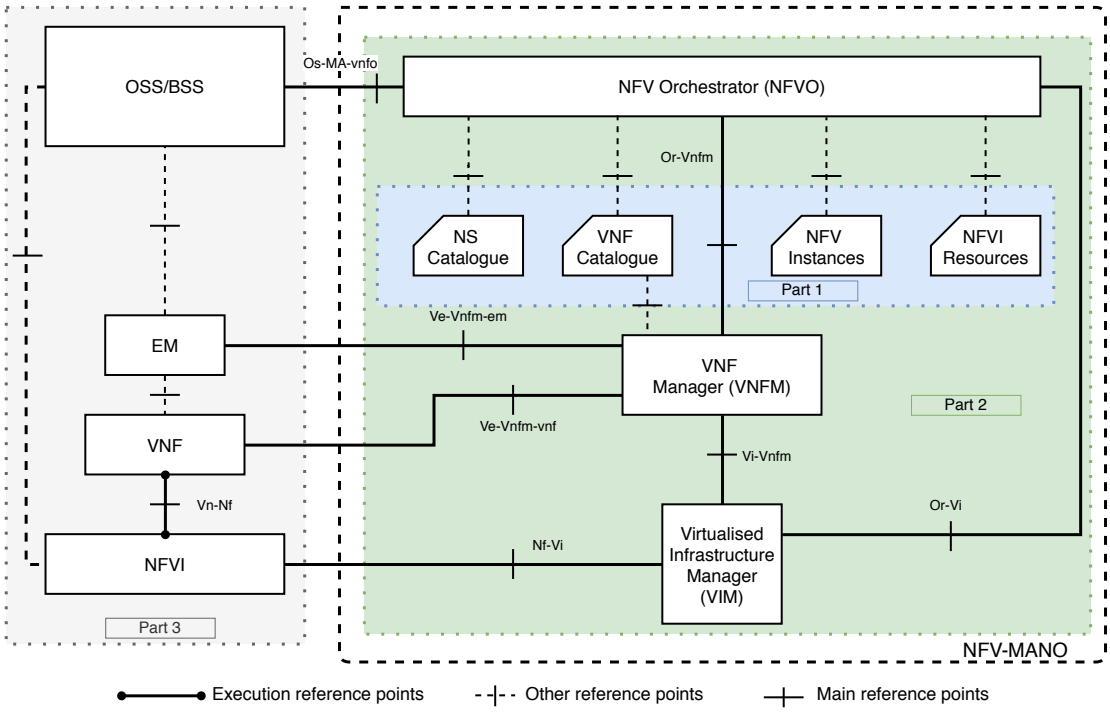


Figure 2.1: The ETSI NFV MANO reference architectural framework [15]

2.2.1 Data Repositories

This part of NFV MANO consists of repositories that are used to store metadata and artefacts. It includes four main repositories as follows.

- **Network Service (NS) Catalogue:** It stores all on-boarded network services and supports creation and management of Network Service Descriptor (NSD), Virtual Link Descriptor (VLD), and VNF Forwarding Graph Descriptor (VNFFGD).
- **VNF Catalogue:** It stores all on-boarded VNF packages and supports creation and management of the VNF package, which includes VNF artefacts such as Virtual Network Function Descriptor (VNFD), software images, and manifest files.
- **NFV Instances Repository:** It holds VNF and network service metadata. VNFs and network service instances are represented by VNF and network records, respectively. These records are updated during the lifecycle of instances and stored in the NFV instances repository.
- **Network Function Virtual Infrastructure (NFVI) Resources Repository:** It holds NFVI resource metadata. Information about available, reserved, and allocated NFVI resources are stored in NFV resource repository which is used, for example, for VNF placement.

2.2.2 Functional Blocks Belonging to NFV MANO

This is the main part of the MANO framework; it includes function blocks that are responsible for the management of VNFs and orchestration of NFV services. These functional blocks are as follows.

- **VNF Manager (VNFM):** It is responsible for managing the lifecycle of VNF instances. It may manage a single VNF instance or multiple instances of the same type. Some of the VNFM tasks are VNF instance instantiation, software update/upgrade, scaling up/down out/in, and termination. VNFM manages VNF instances based on the deployment and operational behaviour described in VNFD.
- **NFV Orchestrator (NFVO):** It has two main responsibility, namely orchestration of NFVI resources across multiple Virtualized Infrastructure Manager (VIM)s and lifecycle management of network services. Some of the NFVO tasks are network service instantiation and managing the lifecycle of network service instances, validation and authorisation of NFVI resources, managing the topology of network service instances such as creating, updating, and deletion of VNF forwarding graphs. The operation and deployment behaviour of NFVO is described in NSD.
- **VIM:** It manages NFV resources and provides northbound interfaces to be used by the NFVO. Some of the VIM's tasks are allocation, upgrade, release, and reclamation of resources, supporting the management of VNF forwarding graph, and supporting NFVO in retrieving available resources. Cloud Management System (CMS) are used as VIM to manage NFV infrastructure. In section 2.4, I will introduce some of the CMSs that are currently used as VIM in NFV.

2.2.3 Functional Blocks Interacting with NFV MANO

This part includes functional blocks that exchange information with NFV MANO. These functional blocks are as follows.

- **VNF:** It is a softwarised version of the middleboxes. VNFs run in virtualization environments such as Virtual Machine (VM)s and Container (CN)s.
- **Element Manager (EM):** It is responsible for fault, configuration, accounting, performance, and security (FCAPS) management of VNFs. The EM interacts with the VNF Manager of NFV MANO.
- **NFVI:** It consists of physical compute, network and storage resources that are managed by a VIM such as OpenStack.
- **Operations/Business Support System (OSS/BSS):** It provides operator's operations that is not realised in NFV MANO and need to exchange information with NFV-MANO. For example, OSS/BSS may provide management for legacy systems and interacts with MANO to provide an end-to-end orchestration.

2.3 Descriptors

NFV services have specific resource and operational requirements that need to be provided to the MANO framework. To express these requirements, the ETSI NFV has defined a set of descriptors, which are as the following.

- **VNFD:** Using a VNFD, service providers describe the requirements of a VNF in terms of deployment and operational behaviour. It includes information about the VNF resource, connectivity, interface and Key Performance Indicator (KPI) requirements.
- **VLD:** This descriptor is used to specify the resource requirements that are needed for connecting VNFs to one another and endpoints of the NS.
- **VNFFGD:** It is a deployment template used to express the topology of a NS. Using the VNFFGD, service providers can specify how to chain VNFs of a NS.
- **NSD:** It describes the service as a whole. NSD is a deployment template that references all other descriptors (e.g., VNFD, VNFFGD) involved in a NS.

These descriptors are created by the service providers and delivered to the MANO framework along with the other service artefacts (e.g., VNF images). The MANO framework uses the information specified in the descriptors to assign virtual resources and perform lifecycle management operations.

2.4 Virtualized Infrastructure Manager

VIMs work as NFVI operating systems and control a pool of compute, storage and networking resources. VIMs create virtual resources on top of the resource pool, which will be used to run NFV services. These virtual resources can be managed by northbound Application Programming Interface (API)s that are provided by VIMs. Using northbound APIs, virtual resources such as virtual machines and virtual links can be created, updated, scaled up/down or out/in, and deleted. As VIM operational requirements are similar to cloud management systems that manage cloud infrastructure, they have been used as a baseline for VIMs. Examples of management solutions used in NFV MANO are OpenStack, Kubernetes and Amazon Web Service (AWS), which are explained in the following.

2.4.1 OpenStack

OpenStack [1] is the most popular open-source software solution for building and managing both public and private cloud infrastructures. OpenStack is backed by major IT companies (e.g., AT&T, Intel, Red Hat) as well as individual community members. OpenStack provides functionalities such as service provisioning, lifecycle automation, billing, and orchestration. To virtualize resources, OpenStack leverages hypervisors like VMware vSphere [2] and KVM [3]. Fig. 2.2 shows the architectural framework of OpenStack; it consists of multiple components, each of which has been implemented for different purposes. The following are the NFV-relevant components of OpenStack.

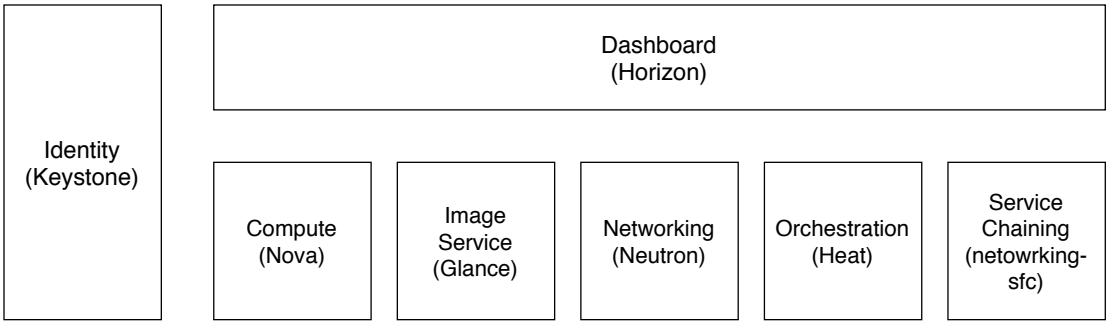


Figure 2.2: The OpenStack architectural framework

- **Nova:** It is the OpenStack compute resource manager. This is the component that spins up virtual machines to handle computing tasks. It provides on-demand compute resources on a large scale.
- **Neutron:** It is the OpenStack solution for providing network as a service. It is based on SDN network and provides connectivity between interface devices such as Virtual Network Interface Card (vNIC)s.

[1] `https://www.openstack.org/`, accessed July 2020
[2] `https://www.vmware.com/`, accessed July 2020
[3] `https://www.linux-kvm.org/`, accessed July 2020

- **Keystone:** It provides identity management. Keystone performs client authentication, service discovery and multi-tenant authorization.
- **Glance:** It is the OpenStack component responsible for storing and retrieving virtual machine images. Glance allows storing images in different locations like filesystems and object-storage systems.
- **Heat:** It creates an engine that orchestrates deployment of multi-component applications. The applications can be described using Heat Orchestration Template (HOT)s that are text-based templates based on the YAML Ain't Markup Language (YAML) format. NFV services that consist of multiple compute and networking components can be deployed by Heat.
- **Networking-sfc:** It is implemented for NFV services. Networking-sfc provides service chaining for VNFs running in OpenStack domain.

OpenStack is used in Chapter 5 as a virtual infrastructure manager to orchestrate VM-based VNFs.

2.4.2 Kubernetes

As mentioned before, VNFs can also be realised on containers. Containers are gaining momentum because of the remarkable benefits that they provide compared to virtual machines. Containers are lighter than VMs as they include only the components required to run an application and do not need a full Operating System (OS). This makes containers faster to spin up and deploy. Also, containers provide native performance (whereas VMs usually incur a performance penalty) and require less memory.

These benefits cannot be gained without a proper orchestration solution to come up with challenges such as service discovery, load balancing, storage orchestration, rollouts and rollbacks automation, and secret and configuration management. To address these challenges, Kubernetes has been introduced by Google; Kubernetes is an open-source container-orchestration system. There are three main components in a Kubernetes cluster:

- **Pod:** It is the smallest execution unit in Kubernetes that encapsulates an application container. A Pod can run a single or multiple container(s). Kubernetes manages the Pod rather than the containers directly.
- **Worker Node:** It is a physical or virtual machine that provides services to run pods. Worker nodes are managed and controlled by the Kubernetes control plane.
- **Master Node:** It runs the Kubernetes control plane. The control plane is responsible for making global decisions such as scheduling, healing, and scaling of the containers. The control plane is also composed of multiple components such as (i) kube-apiserver, the entry point of Kubernetes' control plane. It provides an API through which Kubernetes resources (e.g,

pod) can be managed. (ii) etcd, which stores all cluster data. It is a consistent and highly-available key/value storage. And last but not least, (iii) kube-scheduler, which assigns newly created Pods to Nodes. The assignments are based on different criteria, such as pod resource requirements.

Kubernetes is used in Chapter 5 as a virtual infrastructure manager to orchestrate container-based VNFs.

2.4.3 Amazon Web Service

AWS is the most popular public cloud provider, having 33 per cent of the market share in 2019 [4]. It provides the largest variety of services in the market, which is one of its key successes. AWS can also be used as a VIM in NFV. The wide distribution of AWS Point of Presence (PoP)s allows NFV MANO to place services at locations that are not supported by private cloud infrastructures. Also, AWS provides unique services that can be leveraged by NFV to improve the performance of VNFs. For example, AWS F1 provides Field Programmable Gate Array (FPGA) as services. Using this service, VNFs can be deployed on FPGA to get better performance.

AWS is used in Chapter 5 as a virtual infrastructure manager to orchestrate FPGA-based VNFs.

2.5 Service Function Chaining

As mentioned earlier, in NFV, usually, multiple VNFs are grouped to deliver a service. That means incoming traffic (e.g., user traffic) to the network will be redirected to a set of VNFs to be processed. The order of VNFs in the group is important as moving them around might change the behaviour of the service. To this end, VNFs are chained together, which means the incoming traffic will be traversed through a specific order of VNFs. Such chaining of VNFs is called Service Function Chaining (SFC). Fig. 2.3 shows an example of SFC, where the service is provided by three example VNFs. To realise an SFC like in the example, four concepts are involved, explained below.

- **Connection Point (CP):** CPs are the VNF external interfaces that are used to expose VNFs to send and receive traffic. CPs can be virtual or physical ports, virtual or physical Network Interface Card (NIC) addresses or the endpoint of an IP Virtual Private Network (VPN).
- **Virtual Link (VL):** VL connects CPs to one another to provide connectivity between VNFs. VLs can have different types, such as E-Line, E-LAN, or E-Tree.
- **Network Function Path (NFP):** NFP is a specific path in the service chain. It includes an order of CPs that are connected by VLs.

[4] `https://www.statista.com/`, accessed July 2020

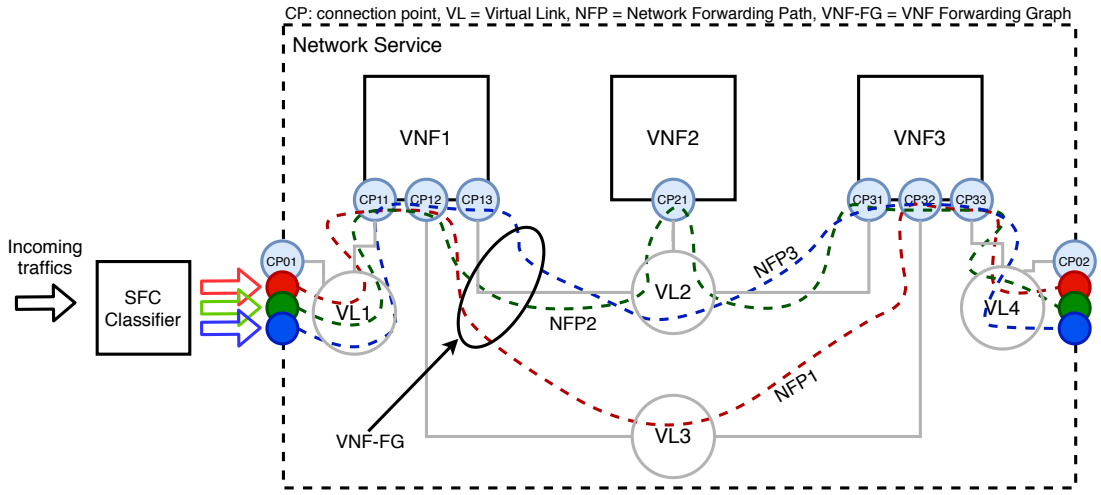


Figure 2.3: An example of service function chaining

- **VNF Forwarding Graph (VNF-FG):** It is used to describe the service chain. Using VNF-FGs, we can specify which VNFs are involved in the chain in what order. For each service chain, we can have multiple NFPs to steer the traffics through. This is because the same service chain can be used by multiple service users who might have different requirements (e.g., one user might want a fast path and another a cheap path).
- **SFC Classifier:** is used to classify the incoming network traffics. The classification is done usually based on the x-tuple and then assigned to different paths in the chain.

To apply SFC, we need to have control over the network that connects the VNFs to one another. A technology that provides such control over the network is SDN, which is described in the next section.

2.5.1 Software-defined Networking

The network control needs to be automated to support the fast-growing network usage and demand [44]. Automation of traditional networks is difficult as the decision-making engines are distributed and tied to switching and routing devices. To mitigate this challenge, SDN has been introduced. SDN decouples the control plane from hardware and delegates it to a centralised management system. In the SDN framework, routing devices work only as traffic forwarders and are configured externally by a controller through interfaces such as OpenFlow. Fig. 2.4 shows the SDN architectural framework. It consists of the following layers.

- **Applications Layer:** It consists of applications that define the decision making logic for the underlying network. These applications collect information from the SDN controller and construct an abstract view of the network. Using such network information, SDN applications instruct the SDN controller to (re)configure the SDN devices.

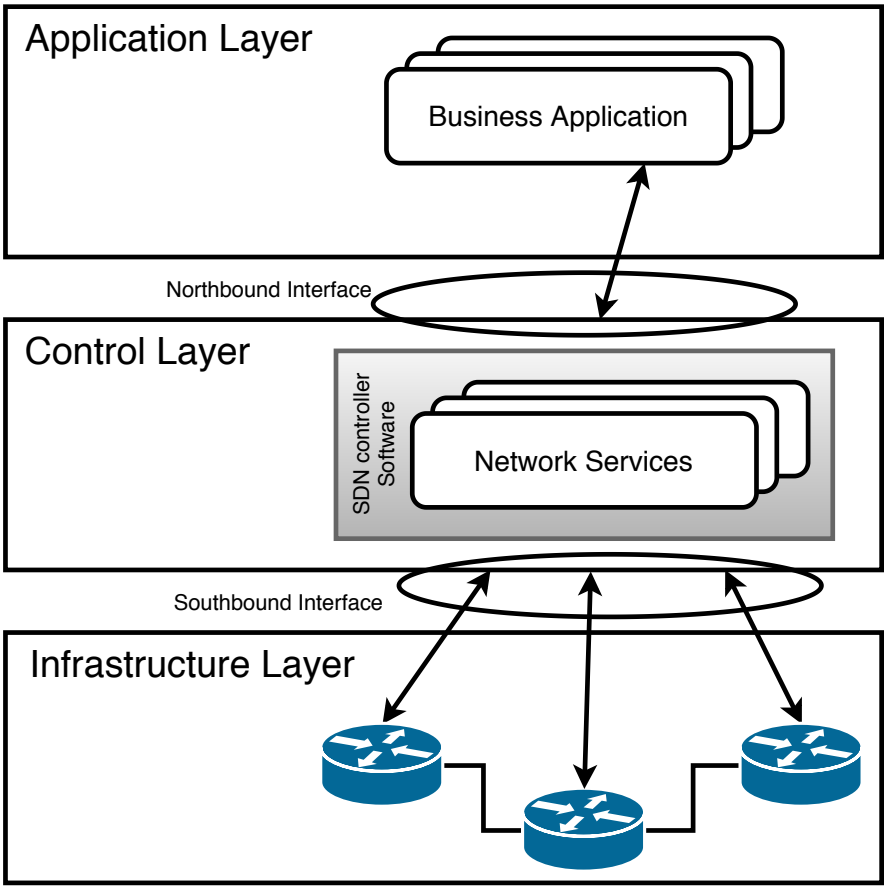


Figure 2.4: The architectural framework of Software-defined Networking

- **Control Layer:** It includes the SDN controller that works as an intermediate entity between SDN applications and devices. The controller translates the network instructions and requirements to forwarding rules and installs them on the SDN devices. The interface between the controller and devices can work based on different protocols such as OpenFlow [31] and NetConf [14].
- **Infrastructure Layer:** It consists of SDN devices such as OpenFlow switches. This layer is responsible for forwarding the packets based on the instructions sent by the upper layers.

The design of SDN controllers has been given a lot of attention from open-source communities from which many solutions have been implemented. Examples of these solutions are OpenDaylight [5] and Open Network Operating System (ONOS) [6], which are complex and designed to be used in large-scale production environments. Other solutions also exist such as POX [7] and Ryu [8], which are simple to use and are suitable for prototyping and development environments.

[5] `https://www.opendaylight.org/`, accessed July 2020
[6] `https://www.opennetworking.org/onos/`, accessed July 2020
[7] `https://github.com/noxrepo/pox`, accessed July 2020
[8] `https://ryu-sdn.org/`, accessed July 2020

SDN enables NFV to provide service chaining. With SDN, networks can be dynamically configured to add and remove service chains and steer the traffic through VNFs as intended by an NFV orchestrator. In this thesis, I used the SDN concept to provide service chaining for services deployed across different domains. I used Ryu as an SDN controller and physical OpenFlow switches in the infrastructure layer. This work is described in Chapter 5.

2.6 NFV Service Deployment Workflow

Now that we know about the fundamental concepts and technologies used to realise NFV, let's look into an example NFV service deployment and see how different components described above work together to deploy a service. Consider a scenario in which we have a service that consists of three VNFs; all to be deployed on one PoP. The workflow that is described next is a typical deployment workflow. It can change depending on the NFV MANO and SDN controller used for deployment.

First of all, we need to create descriptors. Depending on which NFV MANO is used, we need to create at least four descriptors; three VNF descriptors to describe individual regiments of each VNF and one NS descriptor to describe the requirements of the service as a whole. In the NS descriptor, we can specify the service name, constituent VNFs, virtual links between VNFs, and the forwarding graph. In the VNF descriptors, we specify requirements such as VNF name, deployment units, connection points, and VNF image. When all the descriptors are ready, we package the descriptors with other service artefacts such as VNF images and then, we on-board the package to NFV MANO. The package on-boarding is done usually using either a Graphical User Interface (GUI) or Command Line Interface (CLI) provided by the MANO framework.

Once the service package is on-boarded, we can deploy the service. Like the package on-boarding, this is also usually done through either a GUI or CLI. Initiating the deployment, NFVO receives the deployment request and contacts the VIMs to find a suitable location (i.e., a PoP) for the VNFs. Then, NFVO contacts the VNFM to deploy the VNFs separately. Once the VNFs are deployed, VNF records are collected by the VNFM and then stored in the repository. The VNFM also notifies the NFVO about the status of the deployment. If the VNF deployment succeeded, NFVO contacts the SDN controller to connect the VNFs.

In the SDN controller, using an or several SDN applications, forwarding graphs sent by the NFVO are translated to OpenFlow forwarding rules. These rules are, then, installed on the OpenFlow switches, which creates the chain between the three VNFs. The SDN controller then notifies the NFVO about the status of the service chain deployment. Receiving the deployment status, NFVO collects the service records and stores them in the repository.

2.7 SONATA

Multiple MANO frameworks have been implemented to support management and orchestration of NFV services. SONATA [13] is an example of these MANO frameworks. In this section, I describe the overall architecture and constituent microservices of the SONATA MANO framework. SONATA has been used in Chapter3 as a MANO framework to realise and evaluate the SM concept.

The SONATA MANO framework is based on the microservices architecture in which the MANO tasks (service deployment, management, placement, etc.) are implemented in container-based microservices. These microservices are called plugins. Fig 2.5 shows the high-level architecture of the SONATA MANO framework with the main plugins that are explained below.

- **Gate Keeper (GK)** is the main entry point of the MANO framework that secures the MANO framework and provides northbound APIs.
- **Function Lifecycle Management (FLM)** is the SONATA plugin for managing the lifecycle of individual VNFs.
- **Service Lifecycle Management (SLM)** manages and orchestrates services as a whole.
- **Placement Management (PLM)** places generic VNFs.
- **Scaling Management (SCM)** scales generic VNFs.
- **Repositories** is a group of databases to store metadata such as service, VNF, and request records.
- **Infrastructure Adaptor (IA)** enables SONATA to communicate with VIMs such as OpenStack.
- **Message Broker** is a RabbitMQ[9]-based message-queueing system used by MANO plugins to communicate with one another.

To instantiate a service using the SONATA MANO framework, first, the service descriptors will be uploaded into the repository through the APIs provided by the GK. Then, by triggering the service instantiation, GK sends an instantiation request to the SLM. The instantiation request contains the service and VNF descriptors. Receiving the descriptors, SLM sends a placement request to the PLM to find out where the VNFs of the service should be deployed. PLM maps the VNFs to the VIMs based on the service resource requirements and available resources on the VIMs. The decision is, then, sent to the SLM. Now that SLM knows where each VNF should be deployed, it sends a VNF deployment request to the FLM. FLM, then, communicates with IA and deploys the VNF on the corresponding VIM. Once the VNF is deployed, metadata gathered by the VIM is sent up to the FLM and SLM. Based on this metadata, FLM and SLM create

[9] `https://www.rabbitmq.com/`, accessed July 2020

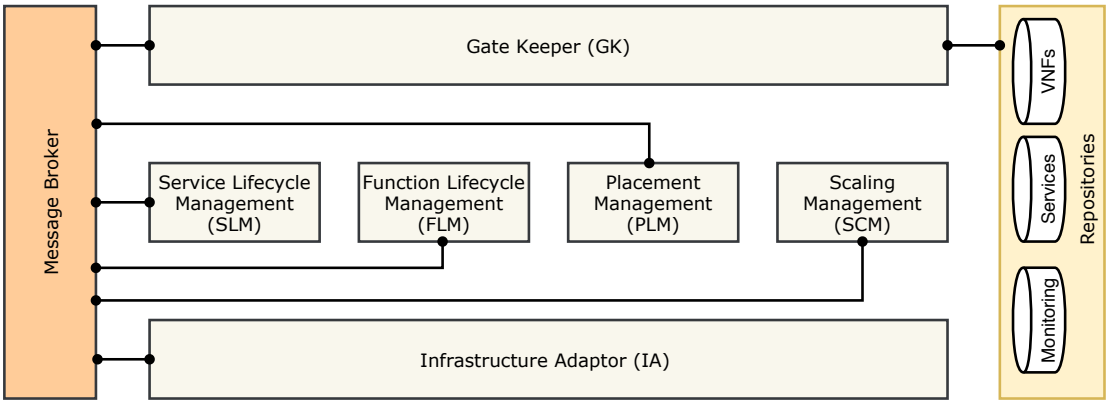


Figure 2.5: The high-level architecture of the SONATA MANO framework

the VNF and service records, accordingly, and store them in the corresponding databases. If service chaining or Wide Area Network (WAN) connectivity is also required for the service, then, SLM triggers the corresponding workflow; otherwise, the results of service instantiation requests are sent up to the GK.

SONATA plugins communicate with each other through a message bus in the publish/subscribe fashion. To this end, APIs are defined for each task, which can be subscribed by plugins to send and receive requests and response messages, respectively. For example, the service placement API is subscribed by PLM and SLM. The SLM plugin publishes placement requests to this API, and, on the other side, the PLM plugin, which is listening to this API, receives the request, reacts accordingly, and sends the response back to the same API to be received by SLM. This enables plugins to be loosely coupled, which provides a high level of flexibility.

3

Programmable Management and Orchestration of Network Services

This chapter is about a mechanism that improves the programmability of NFV MANO frameworks. The idea, *Specific Manager*, that is discussed in this chapter has been proposed by SONATA consortium. I evaluated the idea and published the results in [22]. Moreover, I implemented a platform to support Specific Managers in the MANO framework. This platform has been integrated into the SONATA MANO framework.

3.1 Introduction

Managing the lifecycle of network services and orchestrating them are critical challenges on the way to Network Function Virtualization (NFV). These challenges come from the wide diversity of operational and deployment requirements of network services, which need to be supported by network operators.

It is crucial to have comprehensive lifecycle management in NFV as it affects service performance and also resource consumption. Using a generic management process for managing all network services and ignoring their specific management requirements (e.g., service-specific placement) make the network services inefficient (see Section 3.4). Therefore, network operators need to manage services based on their specific requirements and that requires these operators to fully understand the specific requirements of all individual network services.

Service descriptors are used to describe the requirements of services, which are provided by the service providers. This communicates how to run the service from service providers to network operators. These descriptors are shipped, along with other service artefacts, to network operators. Using these descriptors, the network operator's Management and Orchestration (MANO) framework can manage and orchestrate network services considering individual network service requirements.

Descriptors can be used by service providers to specify general requirements of services such as the resource requirements of a Virtualized Network Function (VNF) or the service graph of a network service. Although these descriptors simplify network service management (see Section3.4), anything outside the predefined semantics of the descriptor cannot be realised; examples are service-specific management and orchestration operations like service placement with a specific optimisation goal.

In principle, it might be possible to have a very broad, comprehensive semantic of descriptors. However, supporting such descriptors is very challenging for the MANO framework. For example, it might be possible to extend the semantic of descriptors to specify service requirements like VNF-specific configuration. Still, a MANO framework cannot configure each VNF individually as each VNF might have a very specific configuration process that cannot be foreseen and pre-implemented in the MANO framework. Therefore, the orchestration process also needs to be programmable, just like the network itself has become programmable.

To realise such programmability for the management and orchestration process, the use of Specific Management (SM)s inside an orchestration framework is proposed. SMs are management programs that are developed by service providers and are transmitted along with other service artefacts (e.g., software containers

for the actual functions) to the network operators. Using these programs, service providers can customize the management and orchestration of network services and also extend the capability of MANO frameworks to support complicated management scenarios. Architecturally, this evolves so-far monolithic orchestration platforms into a microservice-based system design, reaping all the known benefits of this architecture pattern [51] over other concepts.

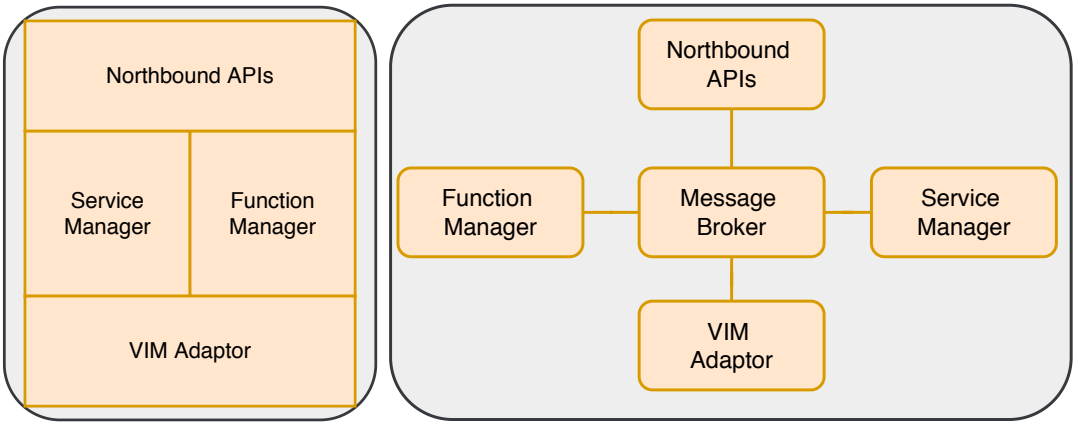
The chapter reminder is as follows. First, in Section 3.2, I discuss the SM concept in more detail. Then, in Section 3.3 I introduce the SM platform which has been implemented to support SMs in the SONATA MANO framework. In Section 3.4, first, using the network service chains placement as a test case, I showcase the benefits of programmable MANO frameworks, and then I evaluate the overhead imposed by SM concept to the MANO framework. In Section 3.5, I review the related work and, finally, in Section 3.6, I highlight the conclusions.

3.2 Specific Management

Two most popular software architectures that can be considered for implementing a MANO framework are monolithic and microservices. In monolithic architectures, all the functions are realised in one functional block that runs entirely on one machine (i.e., it can be a physical or virtual machine). Extending, scaling, and testing one single function in monolithic applications requires extending, scaling, and testing the entire application as all the functions are tied up in one functional block. This is not efficient, especially for large applications, since it makes the aforementioned operations tedious. Monolithic applications are also not resilient since the failure of one function can bring the whole application down. Another problem with monolithic applications is that they are rigid in their design and technology, meaning that the initial technology selected to develop the application cannot be upgraded without significant effort. If one single part of the application needed to be upgraded, the entire application should be redesigned and implemented from scratch. A flexible architecture is important for the MANO framework that needs to be programmable (see Section 3.4). As a result, monolithic architecture is not a suitable solution for implementing a MANO framework.

Microservices are proposed to mitigate the issues associated with monolithic applications. Unlike the monolithic architecture, in the microservice architecture, applications are broken down into multiple microservices. Microservices are loosely-coupled functional blocks that communicate with each other, for example, using message brokers. Microservices can run in different machines (physical or virtual machines, containers); therefore, a failure in one machine will not affect the operation of other microservices. This, in turn, increases the dependability of applications. The distributed nature of microservices increases the flexibility of an application. It makes it possible to add new features by simply adding and integrating new microservices. This improves costly and time-consuming testing and reconfiguring of an entire application. These benefits make microservices architecture a viable option for implementing the MANO framework. Fig. 3.1a and 3.1b show the difference between monolithic and microservices architecture of the

MANO framework. For simplicity, these figures contain only four primary functions of the MANO framework.



(a) Monolithic architecture (b) Microservices architecture

Figure 3.1: Monolithic vs. Microservices architectures for MANO frameworks

Although flexibility is provided by microservices, programmability is an important feature for MANO frameworks, that is not directly realised by microservices. To this end, we propose SMs that allow new features to be added to the MANO framework, even during the framework runtime. SMs deliver a specific design of microservices architecture that provides programmability for MANO frameworks. SMs are developed by service providers for a specific service or function. They are container-based programs that are shipped along with other service artefacts in a service package to the MANO framework. These programs are then integrated into the MANO framework and replace the MANO framework's generic and pre-implemented lifecycle managers. Using SMs, a MANO framework can support any specific management scenario that is not natively supported.

SMs are provided for two levels: function and service. SMs on function level are called Function-Specific Management (FSM) and service level are called Service-Specific Management (SSM)s – explained as follows.

3.2.1 Function-Specific Managers

FSMs are microservices that are developed to deal with specific requirements of individual VNFs. FSMs extend a MANO framework's VNF-specific lifecycle management. The following shows some use cases of FSMs.

- Configuration FSM: The VNF developer can provide specific configuration operations to be triggered during VNF runtime. For example, start or stop some VNF functionalities at a particular event (e.g., when it is required by the other VNF in the service chain or based on monitoring triggering) or run specific commands (e.g., installing rules in a virtualised firewall to drop some packets during the runtime) in the VNF.

- Monitoring FSM: It specifies monitoring metrics of a particular VNF. Monitoring FSM can communicate with the main monitoring component of the MANO framework, request particular monitoring data, and react accordingly.
- Scaling FSM: It defines how to scale a specific VNF. For example, a developer can specify how many instances of a particular VNF should be instantiated at a certain event during VNF runtime.
- Multi-versioning FSM: FSMs can also be used to change the deployment version of a VNF on the fly. For example, to optimise the cost and performance of a VNF, it might be offered in two versions, like Commercial Off-The-Shelf (COTS)-based and Acceleration-based. Depending on the service demand, an FSM can choose between the two versions on the fly (see Chapter 6).

3.2.2 Service-Specific Managers

SSMs are service-specific microservices that provide specific orchestration requirements of network services. SSMs improve a MANO framework's programmability for orchestrating network services as a whole. The followings are some of the use cases of SSMs.

- Placement SSM: It maps functions to resources. For example, a developer can provide a sophisticated placement algorithm with optimisation goals that are not supported by the MANO framework (see Section 3.4).
- Scaling SSM: It customises the scaling of network services. Scaling of a single VNF of a network service can impact the other VNFs involved in the same network service. To deal with consequences of scaling, a scaling SSM can be used. Scaling SSM can provide all the required service-specific orchestration functionalities caused by scaling one VNF (e.g., triggering the scaling of another VNF in the service chain).
- Service chaining SSM: Another use case of SSMs is to take care of changes in the service graph during the service runtime, e.g., compensating for the failure of a VNF in a service chaining SSM.
- Multi-versioning SSM: Multi-versioning can be provided on service levels as well. Using an SSM, MANO framework can change between different set of VNFs on the fly as the service demand and other parameters change (see Chapter 6).

3.3 Specific Manager Platform

In this section, I describe the requirements of integrating the SM concept into the SONATA MANO framework and, then, I explain the SM platform that has been implemented to fulfil the requirements.

3.3.1 Requirements

SMs are service-developer-made plugins. Integrating these plugins into the MANO framework can have unwanted consequences. For example, a developer can use the SMs to access sensitive data that are exchanged in the MANO framework. To mitigate this issue, I have defined six security requirements for SM integration into the MANO framework. These six requirements are the followings.

- **Req. 1:** SMs should not have direct access to the MANO framework message broker. This requirement prevents SMs to eavesdrop or manipulate messages exchanged in the MANO framework.
- **Req. 2:** SMs' messages belonging to a service must be isolated from other services. This requirement prevents SMs from manipulating messages of other services.
- **Req. 3:** SMs' identity should be confirmed by the SM platform, possibly using a credential. This requirement blocks the access of unauthorised SMs to the message broker.
- **Req. 4:** SMs must not be allowed to impose workflows that violate the MANO framework's global policies or requirements. This requirement prevents SMs to inject malicious operations to the MANO framework.
- **Req. 5:** SMs' resource consumptions should be limited and monitored periodically and SMs with unusual resource consumption should be stopped. This requirement reduces the risk of fork bomb [1] attack on the machine that runs MANO's crucial operational code.
- **Req. 6:** Sensitive MANO data must not be exposed to SMs. Examples of these data are the detail resource information and the topology of the underlying Network Function Virtual Infrastructure (NFVI)s.

I also defined four operational requirements that are described below.

- **Req. 7:** SMs have some specifications that need to be described and shipped to the MANO framework. These specifications are, for example, the name of the SM, the Uniform Resource Locator (URL) of the docker registry where the SM image is stored, and the type of operation it performs (e.g., placement, scaling).
- **Req. 8:** SMs are plugins that live as long as their corresponding services live; therefore, there should be a mechanism to manage the lifecycle of SMs. The SM lifecycle manager should be able to pull the SM container images into the MANO framework, start the containers, and terminate them whenever needed.

[1] It is a type of denial-of-service attack that continuously replicates a process to deplete available resources. This results in slowing down or crashing the system due to resources starvation.

- **Req. 9:** SMs should be able to access their corresponding Application Programming Interface (API)s in the MANO framework. For example, an SM that customizes the placement of a service should be able to access the placement-related APIs of the MANO framework.
- **Req. 10:** SMs must run isolated from one another. This is because, for example, they might desire to be written in different programming languages and have different runtime environment requirements.

3.3.2 Design and Implementation

There are two leading technologies that can be used to host SMs and fulfil Req. 10. These technologies are Virtual Machine (VM)s and containers. We have opted to use docker containers [34] to host SMs. This is because Docker containers provide a lightweight and agile runtime environment for SMs. Also, as SONATA plugins run on docker containers, using this type of containers for SMs eases the integration of SMs into the SONATA MANO framework.

To fulfil Req. 7, the schema of SONATA's Network Service Descriptor (NSD) and Virtual Network Function Descriptor (VNFD) has been extended. In the NSD and VNFD schema, a new element has been added that allows service developers to describe the specification of SSMs and FSMs, respectively. These specifications are: (i) id: a name assigned to the SM, (ii) description: a descriptions of what the SM does, (iii) image: the URL where the SM can be downloaded from, and (iv) type: this element determines which MANO framework task will be customised with the SM. The value of "type" can be placement, scaling, configuration, monitoring, start, stop, and lifecycle. Using placement and scaling SMs, the service developer can customise the placement and scaling of services. Using a configuration SM, the developer can run a specific script in the VNF to (re)configure the VNF after deployment. Using a monitoring SM, specific metrics can be monitored with different time intervals. With start and stop SMs, a specific operation inside theVNF can be started and stopped, respectively. Finally, by lifecycle SM, the lifecycle of services or VNFs can be customised. Using this type, the service developer can decide which SM should be triggered when. For example, the developer can define the start FSM to be triggered right after the service deployment or after the configuration FSM is done with the VNF configuration. It also allows to provide SM types that are not already supported. To do so, first, we create a lifecycle FSM to provide the unsupported type. Then, using a lifecycle SSM, we trigger the lifecycle FSM whenever needed. An example of SM descriptor is shown below.

```
function_specific_managers:
  - id: "sonfsmvcdnvcccss1"
    description: "FSM to do a first FSM test"
    image: "sonatanfv/vcdn-vcc-fsm-css"
    options:
      - key: "type"
        value: "start"
```

The rest of the requirements are provided by the SM platform shown in Fig. 3.2. In the SM platform, there are three main plugins as follows.

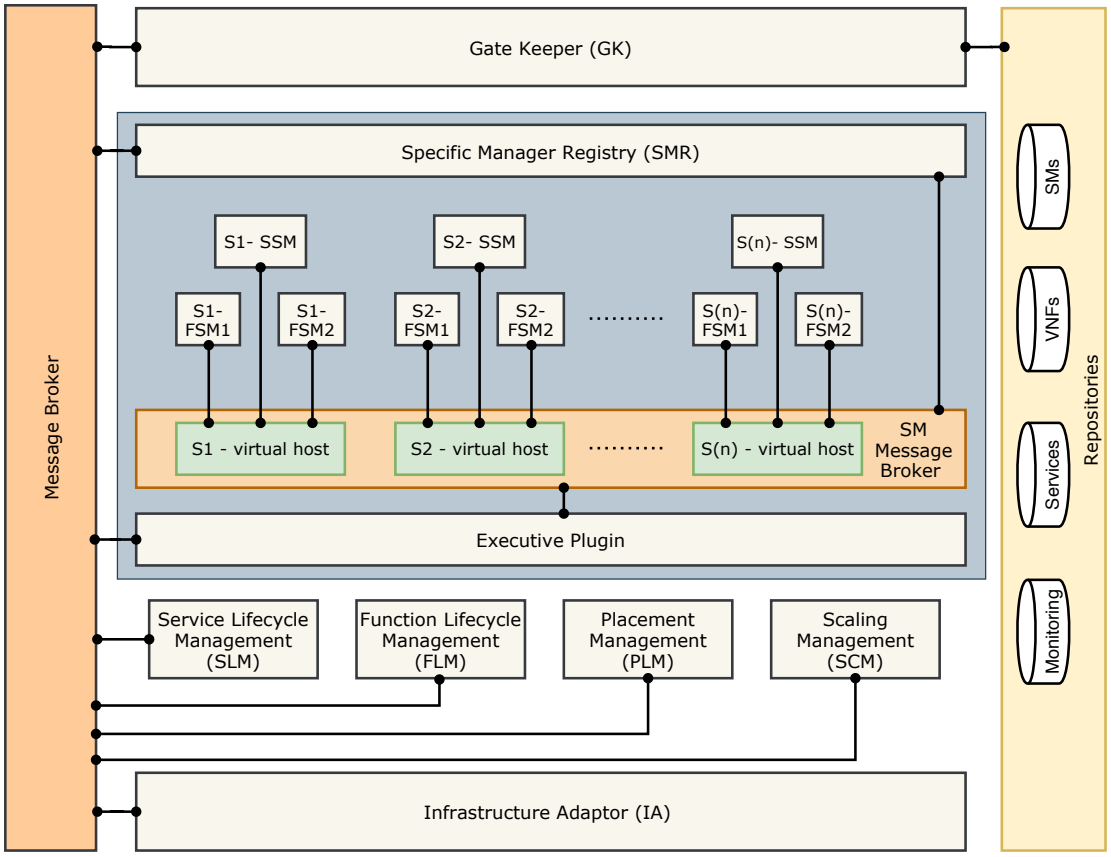


Figure 3.2: The Specific Manager infrastructure in the SONATA MANO framework

3.3.2.1 SM Message Broker

Message broker has been implemented to keep SM messages out of the MANO framework and satisfy Req 1.

3.3.2.2 Executive Plugin

It is the SM plugin connector to the MANO framework plugins. On the one side, Executive plugin subscribes to all the MANO framework APIs; on the other side, it subscribes to the SM APIs. For the SM to MANO framework direction, Executive plugin, first, detects to which MANO framework plugin API the message belongs to – this is done based on the SM API the message is received from – (fulfilling Req. 9). Then, it inspects the message and checks if the workflow complies with the MANO framework's global policies and requirements. To do so, the request is checked against a list of operations that are allowed to be enforced. Examples of these operations are getting available resources or starting a particular VNF. The requested operation is identified from the SM API that the message is sent from.

If the message satisfies the requirements, it will be forwarded to the respective API; otherwise, it will be dropped (fulfilling Req. 4). On the MANO framework plugins to SMs direction, Executive plugin transforms the data to the right level of abstraction (e.g., by removing sensitive information not related to the service) using the solution described in [47] and provides it to the target SMs (fulfilling Req. 6).

3.3.2.3 Specific Manager Registry (SMR)

This plugin is responsible for managing the lifecycle of SMs. It uploads SM images into the SM platform, starts the SM container and terminates them whenever requested, which fulfils Req. 8. SMR performs two other tasks as well. During the SM start phase, SMR, first, creates a virtual host in the message broker and generates a key that will be used by SMs to access the virtual host. Next, it starts the SM container and passes the virtual host name and the key to the container by environment variables. Using the host name and the key, SM can register itself to the SM platform and access the SM message broker (fulfilling Req. 3).

Based on the SM specification in the descriptor and the service ID, SMR creates a RbbitMQ topic (API) specific to the SM. This topic is, then, used by SM and the Executive plugin to exchange messages to get information or enforce an operation. The messages are based on JavaScript Object Notation (JSON) format and parameters included in the message are aligned with SONATA APIs that are intended to call, mentioned in [7]. The RabbitMQ topic has the following format:

```
sm.[type].[service id]
```

The field *"type"* is filled by SM type (e.g., placement, scaling). The *"service id"* is the id of the service that the SM belongs to. This topic is, then, communicated to the SM and Executive plugin, which enables them to communicate with each other. This isolates SM messages belonging to a service from other services (fulfilling Req 2). For each service, SMR creates a topic with the type *internal* which is used for internal communication of SMs belonging to a service. SMR also monitors the resource consumptions of SMs and stops them if they have unusual consumption to fulfil Req. 5.

3.3.3 Deployment Workflow

The SM deployment workflow and interaction of SM and MANO frameworks during the service deployment is as follows. It starts with Service Lifecycle Management (SLM) sending an SM on-boarding request to SMR. The request contains the descriptor which allows SMR to identify from where the SM images should be pulled (it will be extracted from image element of the descriptor, explained above). Once the images are successfully pulled, SLM will be notified. Then, SLM sends an SM instantiation request to SMR. SMR, then, starts the SM containers and handles the networking as explained before. By successful instantiation, SMR stores a record of SM into the SM repository. All SMs, belonging to a service, are

deployed during the service deployment at once and will be triggered later on by their associated events (e.g., when placement is triggered).

Using the SM platform, the SM concept has been integrated into the SONATA MANO framework. In the next section, first, I evaluate the benefit of SM concept compared to rigid solutions and, then, I evaluate the management and resources overhead caused by the SM platform.

3.4 Evaluation

In this section, I assess two aspects of the SM concept. First, I evaluate the main benefit of SMs – making MANO frameworks programmable – and in the second part, I evaluate the management overhead that it causes.

3.4.1 Programmability Improvement

To evaluate the benefits of programmability that SMs can bring to a MANO framework, I compare an SM-enabled MANO framework with others, using placing network service chains as a test case. Conventional MANO frameworks employ pre-implemented placement algorithms that cannot be modified or extended to support specific requirements of individual network services. Therefore, all services – no matter what requirements they have – are placed in the same way, with the same optimisation algorithms and objectives. For example, latency-sensitive network services cannot be efficiently deployed using a MANO framework that does not support VNF placements minimising total latencies.

To support this claim, we designed three MANO frameworks that place network services using different approaches. Utilising these MANO frameworks, we deployed two services with contradicting optimisation goals on a network with 12 nodes and 42 edges based on the Abilene network from SNDlib [37]. Services are manually created in the form of a deployment request as explained in [32] and deployed in different network background loads. The network load is the percentage of network nodes' and links' capacity that is being used. We chose services based on the European Telecommunication Standards Institute (ETSI) NFV use cases [15], including fixed access and mobile core networks. According to the requirements mentioned in [15], we defined the services' optimisation goals as follows.

- Fixed Access Network: minimising the number of used nodes as the primary objective and minimising total latencies as the secondary objective.
- Mobile Core Network: minimising total latencies as the primary objective and minimising the number of used nodes as the secondary objective.

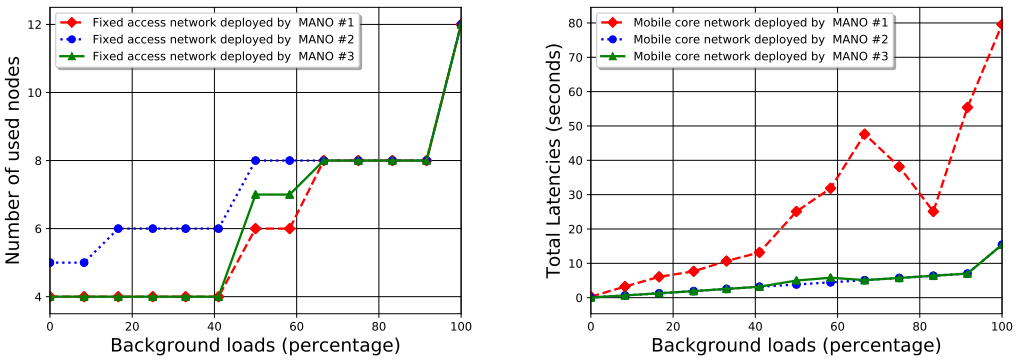
Having this scenario, we compared SM-enabled MANO frameworks with others as follows. Each test was performed five times with different service source/destination nodes, and the average results are shown in the figures.

3.4.1.1 Specific Managers vs. Single-Objective Placement

The placement approaches used in the first comparison for each MANO framework are as follows.

- MANO #1: minimises the number of used network nodes when placing services.
- MANO #2: minimises total latencies when placing services.
- MANO #3: minimises a service's desired objective (based on SM).

We evaluated the resulting service placements based on the two natural metrics, the number of used nodes to deploy the services and the total latency of the services. Fig. 3.3a and Fig. 3.3b show the number of used network nodes and total latencies that we obtained for each service, respectively. The results show that while MANO #1 can meet the primary optimisation requirement of the fixed access network the best in all network loads, it is inadequate for providing the secondary optimisation requirement. The same happens for the mobile core network service when MANO #2 is used for deployment. Considering the results obtained for both metrics, however, MANO #3 performs the best as services deployed by MANO #3 use a low number of network nodes plus having low latencies. This is because MANO #3 places the services based on their specific requirements.



(a) Number of used nodes by the fixed access network service

(b) Total latency of the mobile core network service

Figure 3.3: Results of service deployment using MANO frameworks with different placement approaches on different network background loads based on the first scenario

In high background load, the number of used nodes become almost the same for the fixed access network service deployed by all MANOs. This is to be expected as there are not many free nodes, and the placement algorithm does not have many choices to place the services. On the other hand, for the mobile core network service, MANO #1 results in the worst performance even in high background

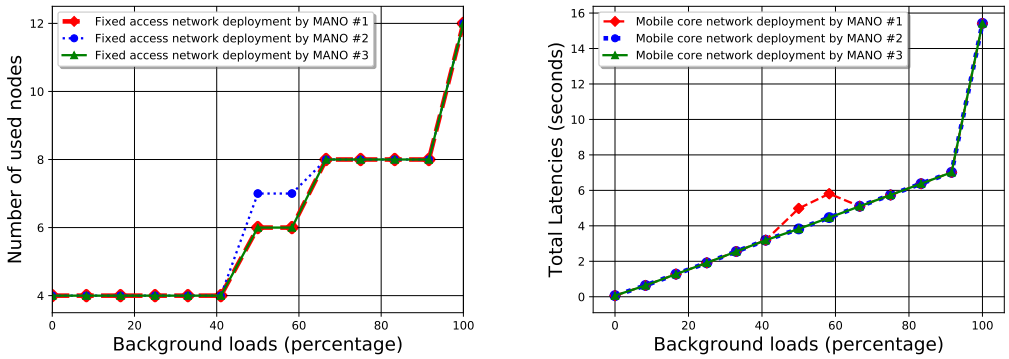
load. This is because MANO #1 does not consider the service latency and just tries to decrease the number of nodes in all network loads. This results in choosing the nodes that generate higher latencies.

3.4.1.2 Specific Managers vs. Multiple-Objective Placement

In the second comparison, we changed the optimisation objectives used in MANO frameworks number one and two as follows, making them smarter and closer in behaviour to our SM-based MANO system.

- MANO #1: minimises the number of used network nodes when placing services, use total latency as a tiebreaker.
- MANO #2: minimises total latencies when placing services, use number of used nodes as a tiebreaker.
- MANO #3: minimises a service's desired objective (based on SM) – as above.

Like the first comparison, we have two metrics, including the number of used network nodes to deploy services and the total latency of the services to evaluate the placement performed by different MANO frameworks. Comparing the results illustrated in Fig. 3.4a and 3.4b, we can see that the SM-enabled MANO framework is the only framework that gives the best results for both services in all background loads. Although MANO #1 performs the same as MANO #3 for deploying the fixed-access network service, it is the framework performing worst for deploying the mobile core network service. On the other hand, MANO #2 gives the worst results for deploying the fixed access network service. This is as expected, given MANO and service characteristics.



(a) Number of used nodes by the fixed access network service

(b) Total latency of the mobile core network service

Figure 3.4: Results of service deployment using MANO frameworks with different placement approaches on different network background loads based on the second scenario

A significant difference between the results from different MANO frameworks can be seen at medium network loads. This is because, in low network load, the placement algorithm has plenty of options to accurately map the services on the resources. Therefore, a good result can be achieved even without a sophisticated optimisation algorithm. On the other hand, when the network is highly loaded, the placement algorithm has little manoeuvring space. Therefore, the results of any placement algorithm will be the same. The use of a service-specific placement becomes advantageous when the network operates at medium background load. But as this is indeed a practically relevant operational regime (networks should be reasonably utilised but not overloaded to be commercially feasible), our SM-based approach shines when it matters.

3.4.2 Management and Resource Overhead

In this section, we evaluate and quantify the management and resource overhead that is caused by having the SM concept in the MANO framework. As explained in Section 3.3, I have implemented a platform that supports management of SMs in the SONATA MANO framework. This platform is evaluated by three metrics: (i) deployment time, (i) CPU utilization, and (ii) Memory utilization. The first metric shows how much management overhead is imposed by the SM platform and the other two metrics allow quantifying the resources overhead brought by SM platform.

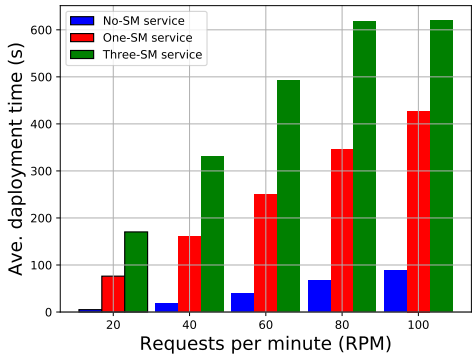
For this evaluation, I implemented three services: (i) *No-SM service*: it consists of one VNF with no SM, therefore, it doesn't use the SM platform for deployment, (ii) *One-SM service*: it consists of one VNF with one FSM and uses SM platform to deploy the FSM, and (iii) *Three-SM service*: this service includes one SSM and one VNF with two FSMs. Like the One-SM service, it also uses the SM platform during the service deployment. As for the parameter for the evaluation, we used RPM. Requests Per Minutes (RPM) is the number of service deployment requests that are sent in one minute.

The test-bed consists of two virtual machines. One of them is used to host SM and SONATA service platforms and the other hosts the Virtualized Infrastructure Manager (VIM). Both virtual machines are equipped with 8 CPU cores of Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 15 GB of RAM.

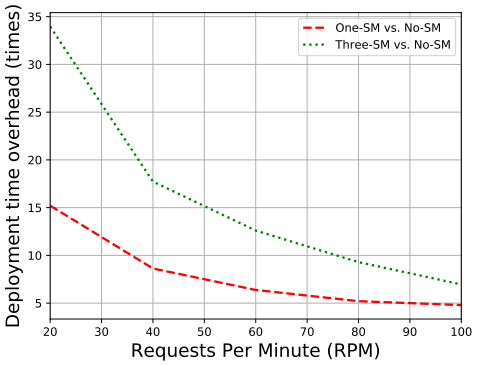
3.4.2.1 Management Overhead Analysis

As mentioned earlier, the deployment time was taken as a metric to analyse the management overhead of SM platform. Fig. 3.5 shows the results of the service deployment time evaluation. For this evaluation, only the times consumed by SM and SONATA service platforms are evaluated and the time that is consumed by VIM is excluded as it is not relevant to this evaluation. Fig. 3.5a shows the average deployment time of the three services over a range of RPM. As expected, the average deployment time increases as the number of RPM increases. Larger RPMs not only put more stress on SM and SONATA service platforms by sending requests in shorter time intervals but also increases the number of SMs to be

deployed. For example, in 100 RPM, for One-SM service, 100 services with 100 SMs are deployed. For Three-SM service, 100 services with 300 SMs are deployed. So, clearly, using SMs has an impact on service deployment time. However, the interesting results can be seen in Fig. 3.5b where the deployment time overhead of One-SM and Three-SM services compared to No-SM service is shown over a range of RPMs. As you can see, the deployment time overhead of services with SMs decreases as the number of RPM (consequently, the number of SMs) increases. For example, considering the overhead of the One-SM service over the No-SM service, we can see the deployment time overhead is more than 15 times for 20 RPM. However, this number decreases to less than five times for 100 RPM. The same happens in the deployment time overhead of the Three-SM service over the One-SM service. While Three-SM deployment time overhead is more than 33 times for 20 RPM, it is just less than seven times for 100 RPM. These results show the more load we have on the MANO framework, the less deployment time overhead we will get by having the SM concept in the MANO framework.



(a) Average deployment time of No-SM, On-SM, and Three-SM services over different RPMs



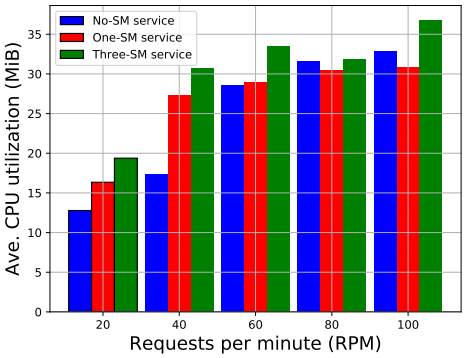
(b) Deployment time overhead of One-SM and Three-SM services compared to No-SM service

Figure 3.5: Results of the service deployment time evaluation

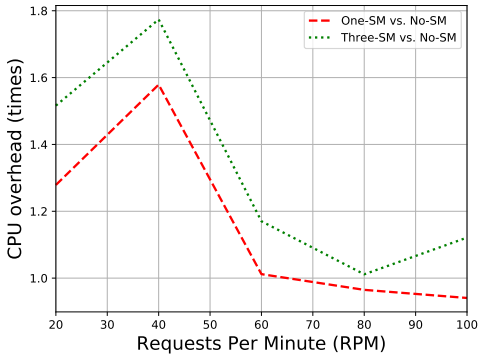
3.4.2.2 Resource Overhead Analysis

Fig. 3.6a and 3.7a shows the average CPU and memory utilization of the MANO framework during the deployment of services over a range of RPMs. Like the deployment time evaluation results, we can see that the CPU and memory utilization of services with SMs are more than services without SMs. The resource utilization also increases as the number of SMs grows. This is expected as higher number of SMs means higher number of containers, and that needs more resources to run. However, you can see in Fig. 3.6b and 3.7b that the resource overhead of the services with SMs are negligible. This is because containers are light-weight execution environments that do not impose a huge overhead on the host machine.

It is also interesting to see in Fig. 3.6b and 3.7b that the CPU overhead of One-SM and Three-SM services compared to No-SM service decrease as the number of



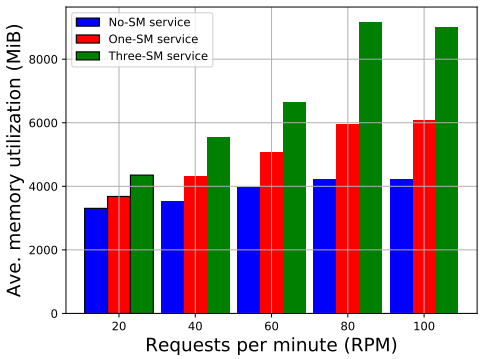
(a) Average CPU utilization of No-SM, On-SM, and Three-SM services services over different RPMs



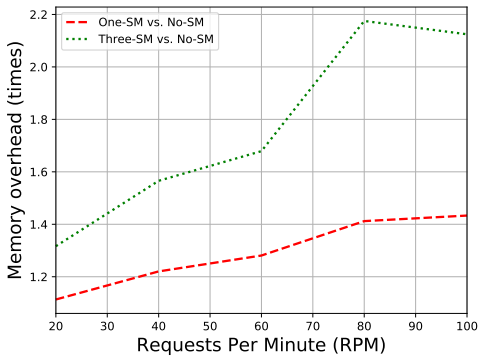
(b) CPU overhead of One-SM and Three SM services compared to No-SM service

Figure 3.6: Results of the CPU utilization evaluation

RPM increases. However, this is different for memory overhead, as the trend shows that the memory overhead increases by growing the number of RPM (consequently SMs).



(a) Average memory utilization of No-SM, On-SM, and Three-SM services services over different RPMs



(b) Memory overhead of One-SM and Three-SM services compared to No-SM service

Figure 3.7: Results of the memory utilization evaluation

3.5 Related Work

Following the definition of the ETSI architecture, various MANO frameworks have emerged for managing and orchestrating network services. In the following, we review some of these frameworks and discuss their capabilities to support per-VNF/service management and orchestration.

Tacker[2], the OpenStack NFV platform, has a policy-based approach for network service management and orchestration. For example, to scale and place network services, Tacker uses a monitoring component that triggers scaling based on strategies (e.g., CPU usage threshold) defined for each VNF or network service. On the other hand, each VNF/service has its specific policies (e.g., max/min number of instances for scaling out/in) specified in the VNF/service descriptor that describes how the scaling/placement should be performed. Although with this approach, the general VNF/service requirements can be considered, the pre-implemented lifecycle management modules (e.g., scaling and placement) are fixed and cannot be extended/programmed to support service-specific lifecycle management operations (e.g., VNF configuration during the runtime) of VNFs/services.

T-NOVA[3] is an NFV platform that provides a MANO framework to manage and orchestrate network services over virtualised infrastructures. T-NOVA's MANO framework employs a dedicated resource mapping module that is responsible for network service scaling and placement. This module contains some generic optimisation algorithms, which includes minimising mapping cost, maximising provider's revenue, and maximising the acceptance rate of NFV service requests. It is feasible for service operators to integrate new optimisation algorithms with different objectives into this module. However, service providers cannot influence the algorithms of the resource mapping module, and all network services are limited to the predefined resource mapping optimisation.

Open Network Automation Platform (ONAP)'s[4] solution for providing per-VNF/service management is POLICY. POLICY is the ONAP subsystem that maintains, distributes, and applies a set of rules to ONAP's underlying control, orchestration, and management functions. Service developers can create a set of policies for each VNF/service during service creation. These rules are then injected into the MANO framework to customise the management of network services. Although ONAP's POLICY can support per-service management to some extent, its expressiveness is limited to the given policy language and does not allow a service developer to actually program the MANO framework. POLICY uses the YAML Ain't Markup Language (YAML) language to describe the policies, which does not provide any programming capability to extend the MANO framework for supporting complicated management scenarios.

Open Source MANO (OSM)[5] employs Topology and Orchestration Specification for Cloud Applications (TOSCA)-based templates [52] for describing both service and VNF requirements. To perform per-VNF management, OSM uses Juju[6], which enables service developers to write scripts for configuring/reconfiguring their VNFs based on their specific needs. However, OSM misses a strategy for providing per-service orchestration on the service level, which makes customising orchestration tasks such as service placement or scaling rather challenging.

[2]https://wiki.openstack.org/wiki/Tacker, accessed Jan. 27, 2020
[3]http://www.t-nova.eu/, accessed Jan. 27, 2020
[4]https://www.onap.org/, accessed Jan. 27, 2020
[5]https://osm.etsi.org/, accessed Jan. 27, 2020
[6]https://jaas.ai/, accessed Jan. 27, 2020

Cloudify[7] uses the blueprint, a TOSCA-based template, to describe service requirements. In the blueprint, lifecycle events are described in a YAML format, which is inflexible as in the other platforms. In Cloudify, developers can use workflows to customise the management of VNFs. Workflows are described in Python and use dedicated API's to communicate with other components of the MANO framework. But workflows can only customise the management of VNFs and are unable to customise the orchestration of network services. Also, workflows have to be written in Python and cannot be implemented in any other programming language.

In another project, Unify [49], a framework has been implemented that aims at unifying cloud and career network resources. It uses a closed source MANO framework, called ESCAPE [48], to orchestrate services over multiple Point of Presence (PoP)s. In Unify, to support function-specific management requirements, a program, called Ctrl APP, is installed on VNFs. Ctrl APP provides a direct interface between VNFs and orchestrator [41] and handles tasks like Key Performance Indicator (KPI) monitoring, topology change triggering and VNFs configuration [42]. Like SMs, Ctrl Apps are also provided by the service providers who develop the VNFs. Unlike SMs, Ctrl Apps cannot customise the orchestration of services. This is because they are tied to VNFs and do not have a holistic view of all resources and VNFs.

3.6 Conclusion

In this chapter, I have presented an innovative method to perform per-service management and orchestration for NFV services. The proposed method is to inject microservices that can perform service-specific management and orchestration into a MANO framework. These microservices are instantiated by MANO frameworks in conjunction with their corresponding services and replace the generic lifecycle management components of MANO frameworks.

Evaluating this method, I showed that SM-enabled MANO frameworks are advantageous over rigid MANO frameworks for both service users and providers as it improves service performance, better utilizes resources, and consequently reduces the cost of NFV services.

I also evaluated the management and resource overhead of having such a concept in a MANO framework. The results show that SM concept can increase the overhead of the MANO framework. However, the overhead is not a game-changer, especially when the MANO framework is dealing with a higher number of requests per minute.

[7] https://osm.etsi.org/, accessed Jan. 27, 2020

4

Scalable and Agile Management and Orchestration of Network Services

In this chapter, I analyse the scalability and agility of Management and Orchestration (MANO) frameworks. To do the analysis, I used a tool called MANO Benchmarking[1] that has been implemented in a project group under my supervision, called SCRAMBLE[2].

4.1 Introduction

Network Function Virtualization (NFV) employs a MANO framework to manage and orchestrate services. In some situations, MANO frameworks should handle a large number of service deployment requests. For example, in scenarios like Internet of Things (IoT) where MANO frameworks are used to deploy IoT-related NFV services, hundreds of service instantiation requests might be sent to a MANO framework over a short time frame. An example of such services is the smart drive-assistant service which is an IoT/Machine to Machine Communication (M2M) service that provides safety and stress-free driving [27]. To provide this service, various control information needs to be fetched from sensors embedded in the car and roadside units. This information, then, will be analysed to make real-time decisions. The data analysis and decision making can be performed by Virtualized Network Function (VNF)s deployed on mobile edge servers. Considering the number of sensors embedded in a car (60–100 sensors), in a traffic jam or accident situation where hundreds of vehicles will be stuck in an area, the number of NFV services to be instantiated will be enormous. Therefore, the NFV MANO should be scalable to be able to instantiate hundreds of services that are required for real-time decision making.

Other than the large number of service requests, in some scenarios, MANO frameworks need to deal with Network Function Virtual Infrastructure (NFVI)s that are spread across a large geographical area (e.g., country or continental). An example of these scenarios is the 5G network in which NFV plays an important role[1]. The 5G network leverages highly-distributed NFVIs to meet the 1 ms round trip latency [4]. In these scenarios, the topological distance between the NFV MANO and NFVIs affects the NFV services deployment time and consequently, the agility of MANO frameworks in deploying new services.

Scaling out and scaling up of MANO frameworks are known scaling strategies that can be used to support the deployment of a large number of services. Also, placing MANO instances close to Virtual Infrastructure Managers (VIMs) where the services will be running is a viable solution to improve the agility of service deployment. Although these solutions sound promising in theory, there is no real-word analysis and data on performance of the current MANO frameworks at scale in terms of number of requests and topological distance to be used when deciding how to scale a MANO framework's components. To address this issue, in this chapter, I analyse the software-based scalability limitations of MANO frameworks using Open Source MANO (OSM) and Pishahang as case studies. Also, the idea of having MANO instances close to the VIM for increasing the

[1] `https://github.com/CN-UPB/MANO-Benchmarking-Framework`, accessed July 2020
[2] `https://github.com/CN-UPB/pg-scrambLe`, accessed July 2020

agility of service deployment is evaluated using Pishahang as a case study and Amazon Web Services (AWS) as real-word resources.

For this analysis, we implemented a tool called MANO benchmarking framework. The MANO benchmarking framework consists of multiple microservices in three tiers that facilitate the testing and analysis of MANO frameworks. It generates test services, monitors the resource utilisation of MANO frameworks, provides a MANO wrapper to access northbound Application Programming Interface (API)s of OSM and Pishahang MANO frameworks, provides VIM mock-up to handle a large number of service deployment without using actual resources, and data fetcher and plotter to gather and analyse the data.

Note that this work is focused on the scalability of the MANO framework itself, not the scalability network services. The remainder of this chapter is as follows. In Section 4.2, the benchmarking framework is explained in detail. In Section 4.3, the scalability of MANO frameworks is evaluated. The related work is reviewed in Section 4.4 and finally the chapter is concluded in Section 4.5.

4.2 MANO Benchmarking Framework

The MANO benchmarking framework has been implemented to support testing and benchmarking MANO frameworks. In this section, first, I describe the requirements of such a framework and then explain microservices that have been implemented to satisfy these requirements.

4.2.1 Requirements

To facilitate the evaluation of MANO frameworks, a benchmarking framework should fulfil the following requirements.

- **Req. 1:** In MANO frameworks, usually, a Graphical User Interface (GUI) or Command Line Interface (CLI) is used for deployment of services. Using GUI and CLI, a user can on-board and deploy services one by one, which is not the most efficient way especially for testing the scalability as multiple service deployment request should be sent to the MANO in a short period. Therefore, a benchmarking framework should provide a solution for sending a batch of deployment request with different time intervals.

- **Req. 2:** The most obvious requirement of a Benchmarking framework is to have means to monitor different performance metrics of MANO frameworks such as CPU or memory usage.

- **Req. 3:** To evaluate the scalability of MANO frameworks, a large number of services should be tested. Deploying a large number of services requires NFVIs to have a huge amount of resources available. Usually, it is not possible to provide that amount of resources for testing. Therefore, the benchmarking framework should provide a solution to test a large number of services without needing actual resources.

- **Req. 4:** The monitoring information might need to be fetched in different time intervals and for different resources. Therefore, the benchmarking framework should provide mechanisms for users to choose what monitoring information in what time intervals to be fetched. The data should also be stored in a database to be available for later usage.

- **Req. 5:** To analyse the data, the benchmarking framework should also have means to plot data in different types of graphs.

4.2.2 Design and Implementation

The MANO benchmarking framework has been implemented based on the requirements mentioned above. The high-level architecture of the MANO framework is shown in Fig. 4.1. It consists of four tiers. The first tier on the top is the Pre/post deployment, which consists of three microservices handling the pre-service deployment operations such as generating deployment requests and post service deployment operations such as fetching data and plotting them. The second tier abstracts northbound APIs of the MANO frameworks and allows pre/post deployment microservices to communicate with MANO frameworks. The third tier is the monitoring tier. This tier includes the actual MANO frameworks, currently supporting OSM and Pishahang MANO frameworks. Along with each MANO framework, a Netdata (see Section 4.2.2.3) instance is running that monitors the performance of the MANO frameworks. The last tier on the bottom is the Virtualized Infrastructure Manager (VIM) mock-up tier. This tier includes the VIM mock-up implemented for the two MANO frameworks. It allows deployment of services at scale with little resources – only to run the VIM mock-up.

Each tier can be used independently from the others. For example, VIM mock-up can be used just to test the deployment of a particular service without using the other two tiers. Also, the Pre/post deployment tire can be used independently just to send example services to a particular MANO that uses real infrastructure underneath. In the following, the microservices are explained in detail.

4.2.2.1 Request Generator

To address the problem with issuing a batch of deployment requests and fulfilling Req 1, we implemented *Request Generator*. Request Generator is responsible for on-boarding a large number of descriptors and generating service instantiation requests. There are multiple parameters that can be configured in Request Generator, which are listed in Table 4.1.

The first two parameters that must be configured are “NSD” and “VNFD”. Using these two parameters, we can import one or multiple Network Service Descriptor (NSD)(s) and Virtual Network Function Descriptor (VNFD)(s) into the MANO frameworks. Each descriptor is represented by a Uniform Resource Identifier (URI), pointing to where the descriptor can be accessed – the descriptor can be downloaded from the internet or imported from the local machine.

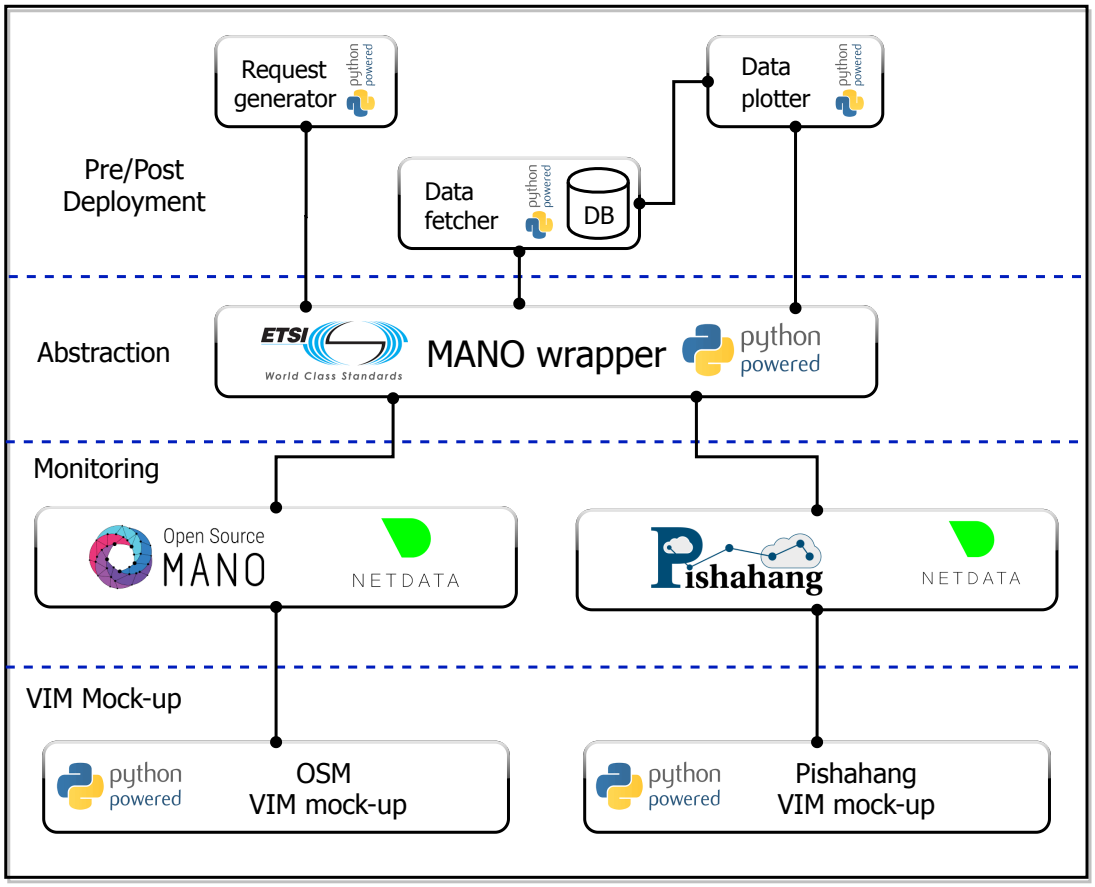


Figure 4.1: The high-level architecture of MANO benchmarking framework

It might be desired to include multiple services in one experiment. To address that, Request Generator provides the "service" parameter which specifies the list of services that should be deployed in one experiment. The services are identified by their Universal Unique Identifier (UUID); therefore, the list must contain a list of service UUIDs.

Request generator also allows to repeat an experiment multiple times. This can be configured by the "number of experiment" parameter. Also, the delay between experiments can be configured by the "Inter-experiment delay" parameter.

In case of errors in the service deployment, some times, the experiment goes to infinite loop and never ends. This can be prevented by setting the "skip experiment error" parameter to "TRUE".

Request Generator interacts with MANO frameworks using MANO wrapper that is explained in the next section.

4.2.2.2 MANO Wrapper

MANO Wrapper has been implemented to provide connectivity between the Pre/post deployment microservices and MANO frameworks. It wraps the north-bound APIs of OSM and Pishahang MANO frameworks in a Python library. MANO Wrapper not only allows Pre/post deployment microservices to communicate with MANO

Table 4.1: Request generator's configurable parameters

Parameter	Data type	Mandatory
NSD	List	Yes
VNFD	List	Yes
Services	List	Yes
Request per minute	Integer	Yes
Number of experiment	Integer	No
Inter-experiment delay	Decimal	No
Skip experiment error	Boolean	No

frameworks but also allows multiple instances of a MANO framework to communicate with each other (explained in detail in Section 4.3.2).

MANO Wrapper covers all MANO reference APIs standardised by the European Telecommunication Standards Institute (ETSI) [19]. It supports northbound APIs of OSM and Pishahang and consists of three groups of APIs: (1) APIs specific to Pishhang, (2) APIs specific to OSM, and (3) blueprint APIs. The first and second groups are the groups of the ETSI standardised APIs that are currently supported by Pishahang and OSM, respectively. The blueprint APIs, however, are those APIs that are specified in the ETSI document but not currently supported by any of these two MANO frameworks. These APIs are provided as blueprints based on python Abstract Class[3], enforcing ETSI standards in future extension to the wrapper. In the first two groups, functionalities such as start or terminate a service instance are supported. An example of blueprint APIs is starting a Physical Network Function (PNF).

4.2.2.3 MANO Frameworks

The benchmarking framework currently supports two MANO frameworks, namely Pishahang and OSM. The MANO framework runs in a dedicated machine (e.g., Virtual Machine (VM)) to avoid the effect of any non-MANO processes on the MANO framework. To monitor the performance of the MANO framework, Netdata[4] is used. Netdata allows real-time collecting of system and application performance metrics. Using Netdata, the benchmarking framework can, for example, monitor the CPU and memory utilisation of MANO frameworks which fulfils Req. 2. Netdata is a very lightweight monitoring tool with a maximum CPU usage of 8 % and RAM usage of 62 MB[5].

4.2.2.4 VIM Mock-up

To solve the resource issue mentioned in Req. 3, we implemented the *VIM mock-up*. VIM mock-up fakes all northbound APIs of OpenStack and generates an arbitrary

[3] `https://docs.python.org/3/library/abc.html`, accessed Huly 2020
[4] `https://www.netdata.cloud/`, accessed July 2020
[5] `shorturl.at/owMN6`, accessed July 2020

response to the requests. The VIM mock-up can cope with any number of services that a MANO can handle. Unlike other similar tools such as VIM-EMU [39], it replies immediately to all requests it receives from the MANO frameworks.

4.2.2.5 Data Fetcher

The data gathered by Netdata needs to be fetched and stored for further analysis. To do so, the *data fetcher* has been implemented to constantly fetch the monitoring data collected by the Netdata running in the MANO framework machine. This data is then stored in a database that will be used by the data plotter described next. The Data fetcher can be configured to fetch different metrics such as CPU and memory usage. It also allows fetching the data related to a specific period of time. Features provided in this microservice satisfies Req. 4.

4.2.2.6 Data Plotter

The final step in the benchmarking framework is to plot the results of the measurement. To this end, the *data plotter* has been implemented. Results Plotter is based on Python Matplotlib library[6] and automatically plots the dataset stored in the database. The results can be plotted in different types, which fulfils Req. 5.

4.3 Analysis

In this section, using experimental evaluation, I analyse scalability limitations of MANO frameworks. First, I evaluate the software-based limitations to find an acceptable load for a MANO framework, in terms of the number of requests per minute. Then, I quantify the effect of topological distance between a MANO framework and its NFVIs. With the later evaluation, I examine how much adding new instances of MANO frameworks close to the VIM can improve the deployment time and, consequently, agility of MANO frameworks in deploying new services.

4.3.1 Software-based Limitation Analysis

We analyse the software-based limitation of MANO frameworks to explore how much load a MANO framework can handle. This study helps to find out on which load level new MANO instances should be employed to fulfil a specific MANO performance requirement (e.g., to support a specific number of deployment requests or a specific deployment time for a service). Also, by evaluating individual components of the MANO frameworks, we explore which components are bottlenecks in the deployment of services.

We studied the performance of two MANO frameworks, namely Pishahang and OSM. The metrics evaluated in this experiment are CPU and memory utilisation, deployment time, and the number of lost requests. As for the parameter, we used

[6] `https://matplotlib.org/`, accessed July 2020

Requests Per Minutes (RPM), which is the number of service deployment requests sent to the MANO framework per minute.

We conducted the evaluation using the benchmarking framework. The test-bed shown in Fig. 4.2 is used to perform the evaluation. This test-bed consists of four virtual machines, each equipped with 16 CPU cores of Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 32 GB of RAM. On the Pre/post deployment VM, we on-boarded the service descriptors to the MANO frameworks. The network service that is used in the evaluation is a single-VNF service that is connected to a virtual network. The VNF used in the service is a mock-up VM-based VNF that is intended to run on OpenStack. During the evaluation, the request generator creates deployment requests and sent them to the MANO frameworks according to the RPM configured for each round of the experiment. For example, setting the RPM to ten, the request generator sends ten service deployment requests to the MANO framework in one minute. Then, the services will be deployed on the VIM mock-up by the MANO frameworks. An actual OpenStack instance has not been used for this evaluation because of two reasons: (i) it is not possible to provide resources (e.g., compute resources) to deploy that many services in our test-bed and (ii) we are not concerned with the VIM time in this evaluation as we want to analyse the scalability of MANO frameworks only.

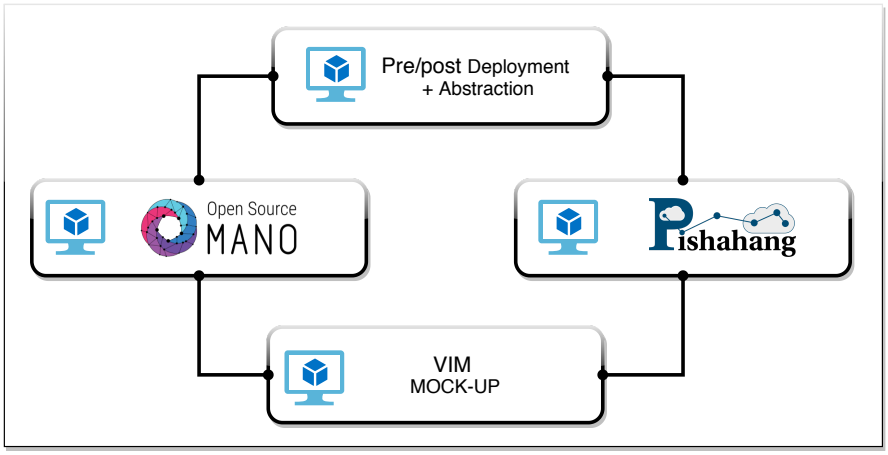
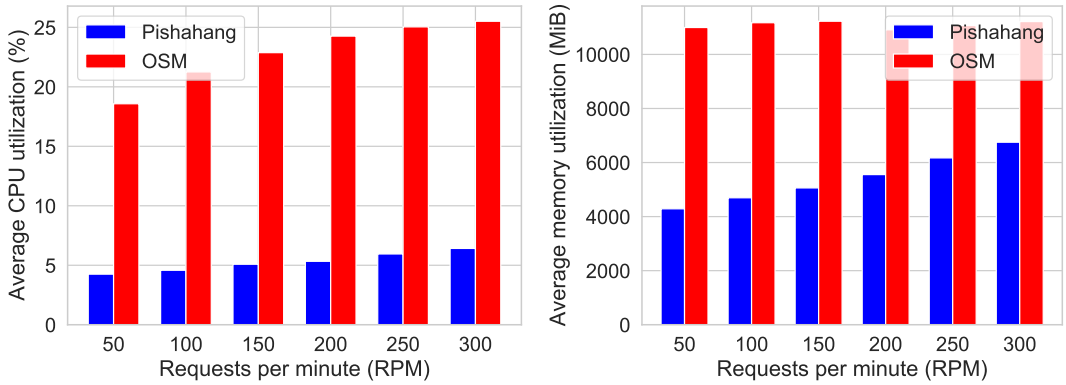


Figure 4.2: The test-bed setup used in software-based limitation analysis

Fig. 4.3 shows the average CPU and memory utilization of Pishahang and OSM MANO fretworks over a range of RPMs. The results show different resource utilization for the two MANO frameworks. For example, to support 300 RPM, Pishahang utilizes about 6 % of CPU capacity, which means one CPU core would be enough for Pishahang to handle 300 deployment requests initiated in one minute. This number for OSM is more than 25 % which means OSM needs more than 4 CPU cores to handle 300 RPM. For the Memory usage, while we see a gradual growth in Pishahang bars, OSM bars remain almost the same for all RPMs. While Pishahang needs between 4 to 6 GiB of memory, OSM needs more than 10 GiB for all different RPMs.

Other than resource utilization, we need to know the deployment time of services at different RPMs. This helps to fulfil MANO service deployment time require-



(a) The average CPU utilization (b) The average memory utilization

Figure 4.3: The average CPU and memory used by Pishahang and OSM MANO frameworks over a range of RPM

ments that might be requested for a particular service. Fig. 4.4 shows the average time spent in the MANO frameworks to deploy services on Pishahang and OSM MANO frameworks over 50 to 300 RPMs. This result shows that, for example, to keep the average deployment time under 200 seconds, no more than 150 RPMs should be sent to the OSM MANO framework. We can also see that Pishahang can support up to 200 RPM while keeping the average deployment time under 100 seconds. We also evaluated the number of lost requests over 50 to 300 RPMs, which is shown in Fig. 4.5. This result shows we need to keep the RPM under 100 for OSM and under 200 for Pishahang to prevent any loss of deployment requests.

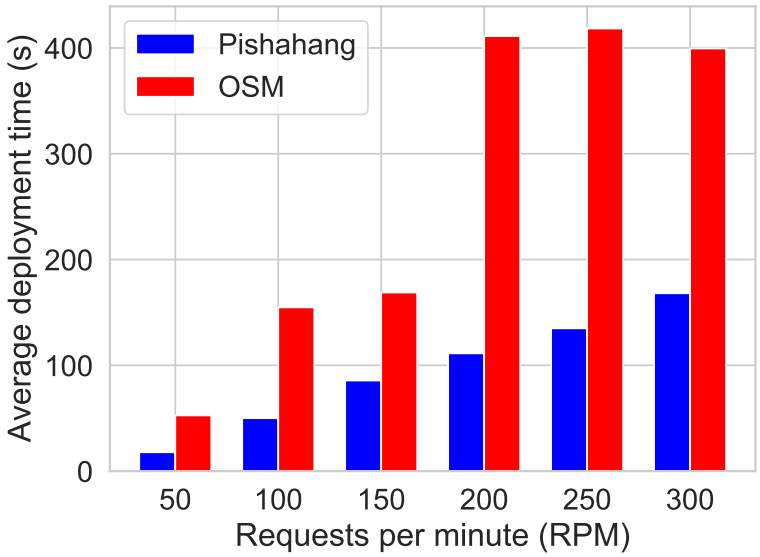


Figure 4.4: The average deployment time of services over a range of RPM

We also evaluated the CPU and memory utilisation of individual microservices of OSM and Pishahang, shown in Fig. 4.6 and 4.7, respectively. These results will help to understand which microservices consume the most resources and act as a

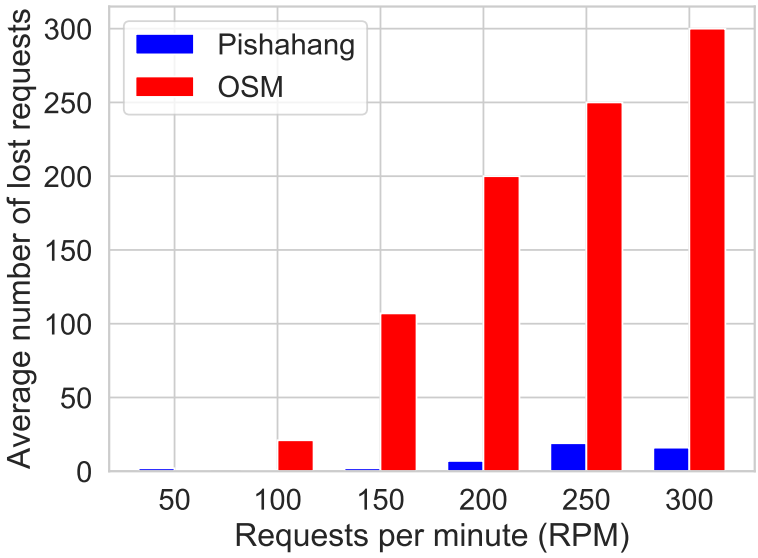
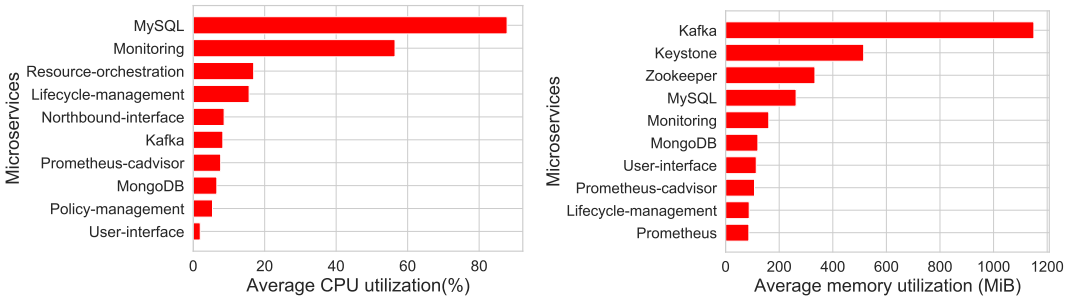


Figure 4.5: The average number of lost service deployment requests over a range of RPM

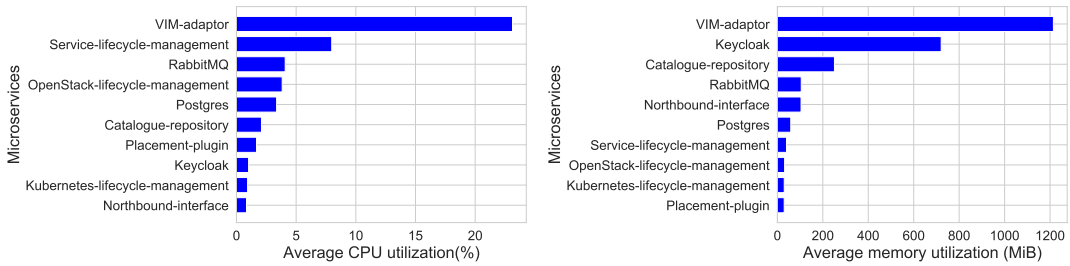
bottleneck in the service deployment process. For the OSM MANO framework, MySQL and monitoring microservices are the top two CPU consumers, which in total use more than 140 %. This means two CPU cores are used to handle these tasks; one is fully utilised, and the other is 40 % utilised. As for memory, Kafka (a message broker) consumes the most memory with about 1200 MiB. Looking at the results of Pishahang, we can see that VIM-adaptor is on top for both CPU and memory utilisation. Service lifecycle management, RabbitMQ (Pishahang's message broker), and OpenStack lifecycle management microservices follow the VIM-adaptor in using CPU. For the memory usage, Keyclock (authentication manager) and catalogue repository microservices are the second and third most memory consumers in the Pishahang MANO framework.



(a) The average CPU utilization

(b) The average memory utilization

Figure 4.6: The average CPU and Memory utilization of OSM individual microservices



(a) The average CPU utilization (b) The average memory utilization

Figure 4.7: The average CPU and Memory utilization of Pishahang individual microservices

4.3.2 Topological Distance Analysis

To analyse the impact of topological distance between MANO frameworks and their NFVIs on the performance of a MANO framework, particularly on the deployment time of services, we conducted an experimental evaluation using Amazon Web Service (AWS) resources. The case study MANO framework for this evaluation is Pishahang as it allows distributing MANO framework instances over different geographical locations and having a peer-to-peer or hierarchical management over the MANO instances.

One of the components that enables such management of MANO instances in Pishahang is the MANO wrapper that allows MANO instances to communicate with each other. Regardless of what management approach – in the case of having new instances – is taken (i.e., hierarchy or peer-to-peer), different instances need to communicate with each other. This is needed for outsourcing service requests to their child or peer instances and also to get some metadata about the services (e.g., service health) running under child or peer MANO instances. In the case of hierarchical management, MANOs need to communicate with child instances through their southbound interfaces and, in the case of peer-to-peer management, eastbound/westbound interfaces should handle such communication.

MANO wrapper supports both management approaches. In the case of hierarchical management, MANO wrapper works as a southbound interface enabling communication of parent MANO with child instances and, for the peer-to-peer model, it works as eastbound/westbound interface enabling the communication of a MANO with its peer MANOs.

Utilising the MANO wrapper, Pishahang framework can fully or partially outsource the management and orchestration of services to their child or peer instances. For example, a parent MANO can fully outsource the service management by uploading the service package into the child instance. Having the service package, the child MANO can, for example, start, stop, monitor, and scale the service and its constituent VNFs. For partial management, the parent MANO can outsource only the service or VNF descriptor to the child MANO and handle start and stop of the service while monitoring and scaling are taken care of by the child MANO.

For this evaluation, MANO wrapper is used next to the Kubernetes wrapper

which covers two cases: (i) deploying services directly into a Kubernetes cluster and (ii) deploying services through a hierarchy of Pishahang instances into a Kubernetes cluster.To generate deployment requests, the benchmarking framework is used. As for VIM, we used Amazon Web Service (AWS) Elastic Kubernetes Service (EKS) with a range of one to five worker nodes. The number of worker nodes is dynamically adjusted by the number of VNF deployment requests. We used AW"St3.medium" instances for the Kubernetes worker nodes and benchmarking frameworks. Also, AWS "m5.4xlarg" instances are used to host Pishahang instances. We also used two different AWS regions for this evaluation, namely the US East (N. Virginia) and Asia Pacific (Sydney). Fig. 4.8 shows the test-bed we set up for this evaluation. In this test-bed, we have two instances of Pishahang; one is placed in Virginia and the other in Sydney. The Kubernetes cluster is also placed in Virginia. There is another VM in the Sydney region that hosts the MANO benchmarking for generating and forwarding service deployment requests.

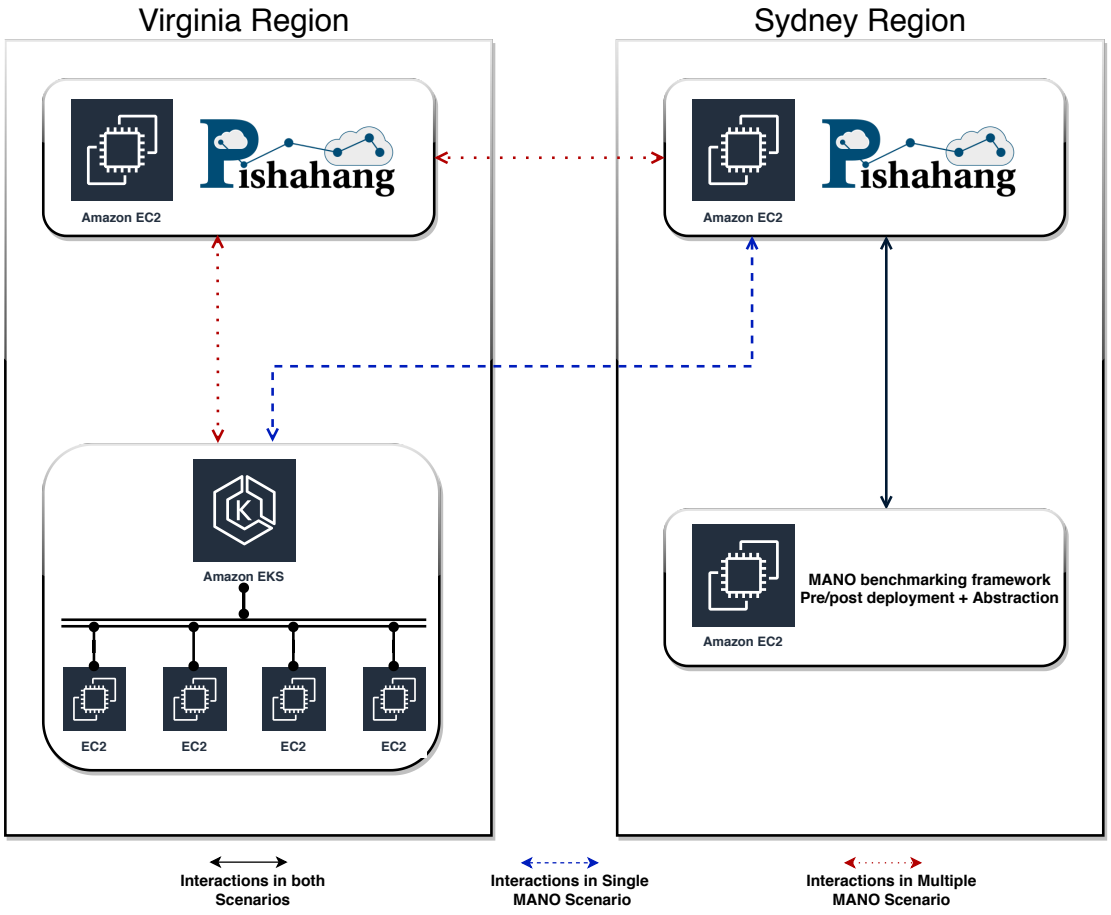


Figure 4.8: The test-bed setup used for the topological distance analysis

The number of RPM is again used as a parameter and service deployment time as a metric. The service deployment time has been evaluated in two scenarios as follows.

- Single MANO instance: In this scenario, the Pishahang instance placed in

Sydney directly communicates with the VIM placed in Virginia to deploy the services.

- Multiple MANO instances: In this scenario, the Pishahang instance of Sydney delegates some parts of the service deployment process to the Pishahang instance of Virginia. The MANO instance in Virginia takes care of any required communication with the VIM, handles the workflow locally, and send the result back to the Pishahang instance in Sydney.

The VNF used in this evaluation is a container-based VNF. This is because, as opposed to VMs, containers are lightweight virtualization solutions which allow us to test a higher number of VNFs with less cost in AWS.

During the deployment, after spinning up the containers/VMs, the VNFs needs to be configured to make the service available. To make sure that the configuration time is also included in our evaluation, we programmed the VNFs to expose an API after deployment. This API will be called by the MANO framework to run a series of VNF configuration scripts on the VNF container. Therefore, the deployment time includes the time that is consumed by EKS to spin up the container, add a load balancer to the VNF, and make the VNF API available for external accesses. This workflow is the same for both scenarios.

Fig. 4.9 shows the deployment time of services over a range of RPMs for both scenarios. The results show that using multiple instances of a MANO framework can reduce the deployment time of services. The time saving also becomes higher as the number of requests per minutes increases. At the highest RPM, the deployment time can be reduced 5.69 times by using an instance of MANO close to the VIM. This is a remarkable time saving which shows placing MANO instances close to the VIM is a valid approach with remarkable benefit.

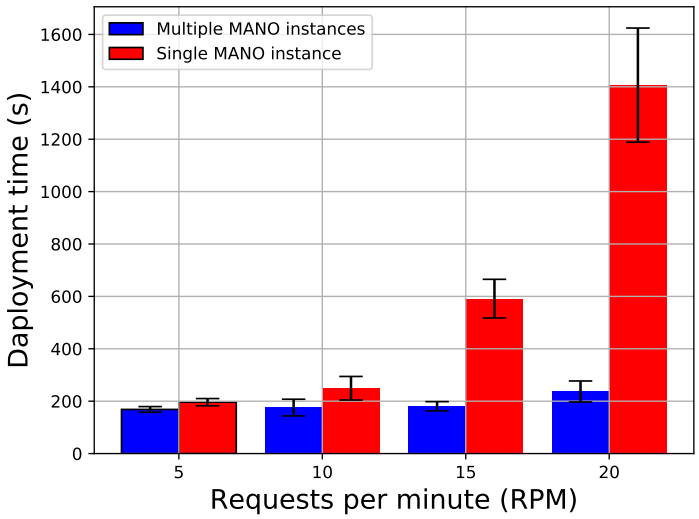


Figure 4.9: The deployment time of services over a range of RPM

4.4 Related Work

There are some papers that have been published in the context of scalable MANO frameworks and testing frameworks for MANOs. For example, Chen et al. [9] proposed a scalable, Kubernetes-enabled NFV MANO architecture. The authors leveraged Tacker as MANO framework and used OpenStack and Kubernetes as VIMs. In their architecture, they have a master node in the VNF management that monitors the CPU usage of the VNFs on OpenStack and Kubernetes domains to identify when services should be scaled. This work is mostly focussed on how to scale the services, not the MANO itself. Although it analyses the time required to scale out or in a service, it does not analyse scaling the MANO framework.

Sahhaf et al. [43] proposed a scalable orchestration architecture of network function chains. This work mostly focuses on the VNF placement problem and provides service models and algorithms to support VNF placement at scale. The proposed solution has been implemented in a closed source MANO called ESCAPE [48], which is not available for testing. Also, no evaluation has been performed in the context of service deployment time and resource utilisation.

In another work, Abu-Lebdeh et al. [2] proposed a solution for minimising the operational cost of running MANO framework without violating the performance requirements. The proposed solution uses a hierarchical approach to distribute the VNF management (one of the components of the MANO framework) across different regions. An algorithm is proposed in this study to find the best place for VNF management so that the operational cost will be reduced. This work is mostly focused on optimising the cost and does not provide analysis of the deployment time of services in large-scale MANO deployment scenarios.

Abu-lebadeh et al., in their other work [3], provided an integer linear programming formulation of the problem of placing NFVO and VNFM in large-scale and geo-distributed NFV infrastructure and proposed a two-step placement algorithm to solve it. Using the solution, they analysed the effect of the number of NFVOs on the communication delay between NFVO and VIMs and the number of VNFM on the communication delay between NFVO and VNFM as well as VNFM and VNFs. The solution provided in this work, however, is based on simulation and does not use any of the current MANO frameworks. This, consequently, does not quantify the benefit of using multiple instances of MANO frameworks in a real-word setup.

As for the testing framework, Peuster et al. [40] implemented an emulation-based testing framework that emulates OpenStack VIM and allows service to be deployed and tested as containers. Using this emulation framework, the authors analysed the deployment time of services that are deployed by different versions of OSM MANO framework across multiple PoPs. Using this testing framework, however, only the OSM MANO framework can be evaluated, and the performed evaluation includes resource utilisation evaluation neither for overall MANO components nor individual components of the MANO. In another work, Yilma et al. [54] defined MANO-specific Key Performance Indicators (KPIs) from which they compared OSM and ONAP MANO frameworks. Examples of the KPIs analysed in this work are the on-boarding process delay and the deployment process

delay. The evaluation in this work has been performed for a couple of services; however, it lacks evaluation of MANOs at scale.

4.5 Conclusion

In this chapter, we evaluated two MANO frameworks in terms of software-based scalability limitations. Also, we evaluated the effect of topological distance between a MANO framework and its NFVIs on the deployment time. The evaluation has been conducted using an open-source benchmarking framework that has been implemented in this study.

The results achieved in this work shows that, in general, OSM not only requires more resources to deploy and manage services but also requires a longer time to deploy services compared to Pishahang. Also, we realised that to prevent losing requests, the number of deployment requests sent to OSM should be less than 100 per minute and less than 150 for Pishahang (i.e., for the configuration used in the evaluation). These numbers can be used to decide under what load situation these two MANO frameworks should be scaled out or in.

We also found out that, in OSM, MySQL, monitoring, and resource orchestration microservices use more CPU resource compared to the other microservices. There is a different story, however, for memory usage as Kafka (the message broker) and Keystone (the authentication system) are the most memory consumers microservices in OSM. In Pishahang, VIM-adaptor – the counterpart of resource orchestration in OSM – along with service lifecycle management are the two most CPU consumers. For the memory usage, Keyclock – the counterpart of keystone in OSM – is one of the top two memory consumers. On the top, however, unlike in OSM, VIM-adaptor also uses the most memory among Pishhang's microservices. These results are useful for solutions that scale out individual microservices, which is superior to scaling out the entire MANO framework.

The topological distance analysis shows that using instances of MANO frameworks close to the NFVIs where the services are running is a valid approach as it can reduce the deployment time of services up to 5.69 times. This, consequently, improves the agility of MANO frameworks in deploying new services.

5

Multi-domain Management and Orchestration of Network Services

In this chapter, I describe my contribution to the realisation of management and orchestration of services across multiple domains (i.e., a domain refers to an Network Function Virtual Infrastructure (NFVI) managed by a specific Cloud Management System (CMS)). This work has been done in the context of 5G-PICTURE [1] project, which is an EU-founded project under Horizon 2020. The results of this work have been demonstrated in 5G-PICTURE review meetings in Jan. 2019 in Athens, Nov. 2019 in Barcelona, and March 2020 in Bristol. Two demo papers, [21] and [25], have been published out of this work in NetSoft 2018 and DEBS 2019, respectively, as well as a regular paper [8] in NetSoft 2020. Parts of the implementation that will be described in this chapter have been done by Tobias Dietrich [10] and Dennis Meier [33] in the context of bachelor theses under my supervision. The project group ENTANGLE[2] also contributed to this work under my supervision.

5.1 Introduction

Network Function Virtualization (NFV) infrastructures consist of heterogeneous resources that are used for different purposes. For example, utilising acceleration resources such as Graphics Processing Unit (GPU)s or Field Field Programmable Gate Array (FPGA)s is proposed to improve the performance of Virtualized Network Function (VNF)s [36]. Also, different virtualisation environments are suggested to host VNFs. For example, it is recommended to use Virtual Machine (VM)s when security is essential and Container (CN)s when performance and deployment agility is in a higher priority [45]. Therefore, it is crucial for a Management and Orchestration (MANO) framework to support heterogeneous resources.

In the NFV orchestrator, CMSs such as OpenStack and Kubernetes are used as Virtualized Infrastructure Manager (VIM)s. In a single-domain (i.e., a domain here refers to a cluster of resources that is managed by a specific CMS) MANO framework, the MANO is limited to a particular type of resources that are supported by its CMS. For example, using OpenStack limits the system to VMs that can run on General Purpose Processor (GPP)s and GPUs (i.e., GPUs are not fully supported), and no other resources can be supported. In another example, using Kubernetes, resources such as containers, GPPs, GPUs can be supported, but it is not possible to run VMs and deploy VNFs on FPGAs. Public cloud solutions such as Amazon Web Service (AWS) can provide most of the resources mentioned above – including FPGAs – however, begin closed-source, AWS is not suitable for managing in-house resources.

To solve this issue, I propose leveraging multiple domains in the MANO framework. Utilising multiple domains allows a MANO framework to support a wider range of resources. For example, leveraging Kubernetes, OpenStack, and AWS in a MANO framework allows to support VMs, CNs, GPPs, GPUs, and FPGAs.

[1] `https://www.5g-picture-project.eu/`, accessed June 2020
[2] `https://bit.ly/3jSI34q`, accessed July 2020

To realise this idea, we implemented a MANO framework called Pishahang. It supports a wide range of heterogeneous resources by leveraging multiple CMSs as VIMs. Pishahang can deploy services across different domains and provides inter-domain service chaining. Pishahang is used as the main orchestrator in the 5G-PICTURE project. 5G-PICTURE is an EU-founded project aiming at converging disaggregated network and compute resources.

The remainder of this chapter is as follows. In Section 5.2, first, the requirements of a multi-domain orchestrator are defined, and then, the design and implementation of Pishahang are described. In Section 5.3, I describe how Pishahang has been used in the 5G-PICTURE project and explain the evaluation results. Section 5.4 reviews the related work and Section 5.5 concludes the chapter.

5.2 Pishahang

In this section, I describe how the management and orchestration of services across multiple clouds and Software-Defined Network (SDN) domains are supported by Pishahang. To this end, first, I describe the requirements of such a system and then discuss the design and implementation of the solution.

5.2.1 Requirements

As mention in Section 5.1, this work is mostly focused on supporting heterogeneous resources by a multi-domain orchestrator. Having that as the main objective, I have defined the following requirements.

- **Req. 1:** The system should support multiple compute resources such as GPP, GPU, and FPGA.

  While acceleration resources such as GPUs and FPGAs can improve the performance of VNFs [36], GPPs are more cost-efficient. Being able to choose from these resources, we can optimise the cost and performance of the VNFs.

- **Req. 2:** The system must support VM-based and CN-based domains.

  Although containers are better solutions than VMs in terms of performance and deployment agility, VMs provide more secure environments [29]. Therefore, to utilise the benefits of both virtualisation solutions in a system, both VM- and CN-based solutions must be supported.

- **Req. 3:** The system must enable usage of resources from different domains in the same service.

  This is an important requirement as, without such support, the system will still be limited to one domain and unable to deploy services over heterogeneous resources.

- **Req. 4:** The system must provide chaining across different domains.

  In such multi-domain infrastructures, as the VNFs are distributed across different domains, the system is required to provide not just intra-domain (i.e., usually provided by domain manger) but also inter-domain chaining.

- **Req. 5:** The system must be able to support running containers on bare metal [20] via its VIMs. Running containers on bare metal gives a better performance compared to running them on virtual machines (i.e., all public clouds such as AWS and Google Cloud Platform (GCP) run their Kubernetes cluster and containers on virtual machines).

5.2.2 Design and Implementation

I have tried to use existing solutions where it was possible to design the Pishahang framework. This avoids reinventing the wheel and also accelerates implementation. To this end, I based Pishahang on the SONATA MANO framework introduced in Chapter 3. SONATA has been selected as the base MANO framework over its competitors (e.g., Open Source MANO (OSM)) mainly because of two reasons: (i) SONATA follows a microservice-based architecture that allows new functionalities to be added simply by creating and integrating a new microservice (i.e., this makes extending the MANO system much easier) and (ii) it also allows network services to bring their own orchestration code along with other service artefacts by the concept of Service-Specific Management (i.e., introduced in Chapter 3), which increases the flexibility of the MANO framework to meet the service requirements that are not natively supported.

To support the container orchestration, I opted to use Kubernetes over its major competitor Docker Swarm. I chose Kubernetes as it provides an easy service organisation with pods, it is open-sourced and modular, and most importantly, it is battle-tested as it has been used in production environments already for some years. Kubernetes is used as a VIM in the Pishahang MANO framework. Pishahang is also designed to use Kubernetes clusters that are based on both bare metal and virtual machines. In other words, Pishahnag can work with Kubernetes clusters running on a public cloud such as AWS and GCP and also Kubernetes clusters that run on bare-metal, in-house resources. To run Kubernetes on bare metal, Pishahang leverages third-party solutions such as Weave[3] to provide networking for Kubernetes cluster and Metallb[4] to provide load balancer for Kubernetes services.

For the VM orchestration, OpenStack is used as VIM in Pishahang. The major competitor of OpenStack is VMware. OpenStack has been chosen, mainly, because it is open-source and can be extended or modified as needed. While everything is free for OpenStack, VMware has a license and maintenance fee. Both OpenStack and Kubernetes support managing GPPs and to some extend GPUs. However, none of them supports the management of FPGAs. To support FPGAs, Pishahang

[3] `https://www.weave.works/oss/net/`, accessed July 2020
[4] `https://metallb.universe.tf/`, accessed July 2020

uses AWS as a VIM. AWS F1 service provides FPGA as a service which allows running VNFs on FPGA resources.

Although currently, Pishahang supports these three cloud domains, the framework is designed to be easily extendable. This is realised using Terraform, which provides infrastructure as code and supports all major cloud management tools. More detail on this comes in Section 5.2.2.2.

Fig. 5.1 shows the Pishahang architecture with its main components. As mentioned earlier, it has been built on top of SONATA. Service Lifecycle Management, Placement Plugin, Message broker, and repositories are the SONATA components that have been extended, and the rest of the components are built entirely for Pishahang (i.e., SONATA's base libraries have been used to implement Pishahang's plugins). Like any other NFV orchestrator, Pishahang uses descriptors to specify service and VNF-specific requirements. Pishahang descriptors are the extended version of SONATA descriptors (discussed in detail in Sections 5.2.2.1). OpenStack Lifecycle Management (OLM), Kubernetes Lifecycle Management (KLM), and AWS Lifecycle Management (ALM) have been implemented in Pishahang to manage the lifecycle of VNFs that run in OpenStack, Kubernetes, and AWS, respectively. They are responsible for tasks such as instantiation, starting, stopping and termination of VNFs running on their respective domains. The workflow of these plugins can also be customised using the Specific Management that has been introduced in Chapter 3. Service Function Chaining Management (SFCM) is responsible for providing inter-domain service chaining, explained in detail in Section 5.2.2.3. To provide the multi-domain orchestration, I have redesigned the SONATA Infrastructure Adaptor (IA). IA in SONATA only supports OpenStack, and it uses Heat templates to deploy services, which is not a good solution for multi-domain orchestration – more detail on this comes in Section 5.2.2.2. In the following, I discuss implementation details of Pishahang's major components.

5.2.2.1 Service Descriptors

There are two levels of descriptors, namely service and VNF descriptors – shown in Fig. 5.2. The service descriptors allow describing the service requirements as a whole. In the service descriptor, we can, for example, describe what VNFs are included in the services, to which cloud domain (OpenStack, Kubernetes, AWS) the VNFs belong to, and how the VNFs should be chained. On the VNF levels, VNF-specific requirements are described. Examples are type, image, and Virtual Deployment Unit (VDU)s. In Pishahang, there are three types of VNF descriptors: (i) OpenStack descriptor, (ii) Kubernetes descriptor, and (iii) AWS descriptor. As VNFs running in different cloud domains have different requirements, we designed different schema for them. An example of such domain-specific requirements is the service type in Kubernetes-based VNFs. The services in Kubernetes can be described as ClusterIP, NodePort, and LoadBalancer. This is a requirement that does not exist in other domains.

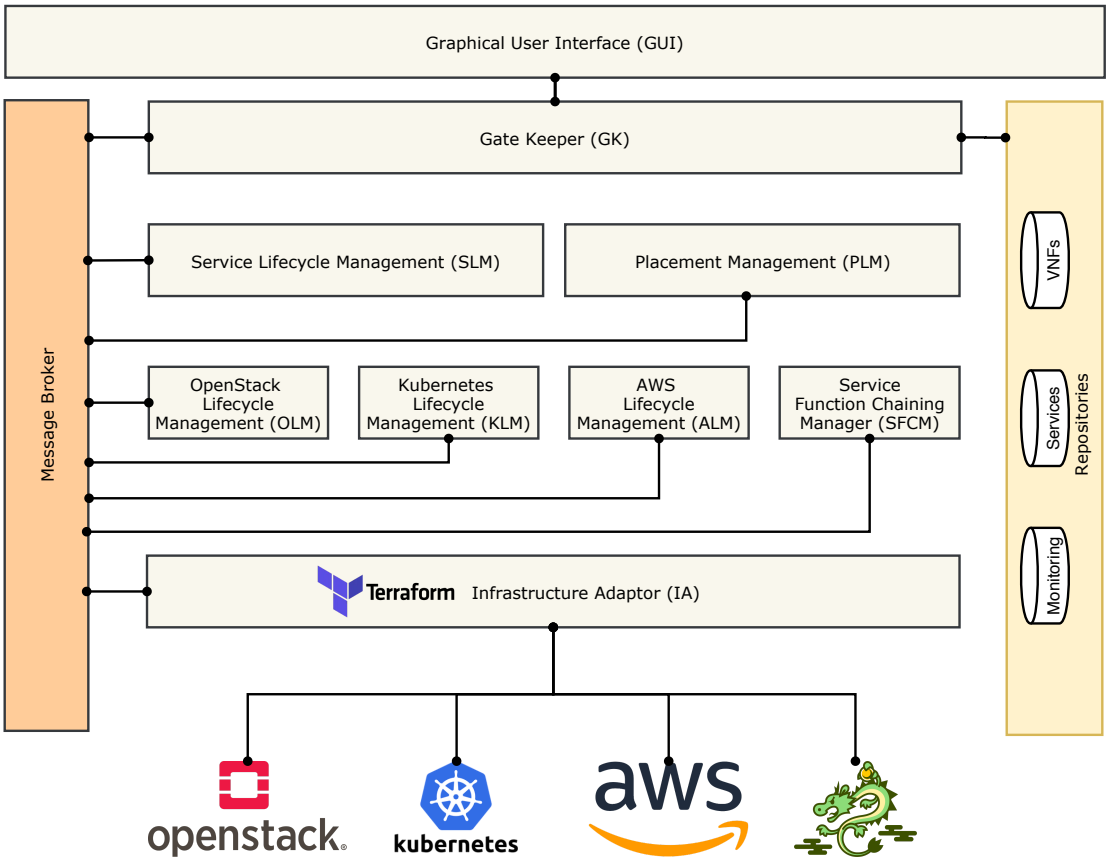


Figure 5.1: The high-level architecture of Pishahang

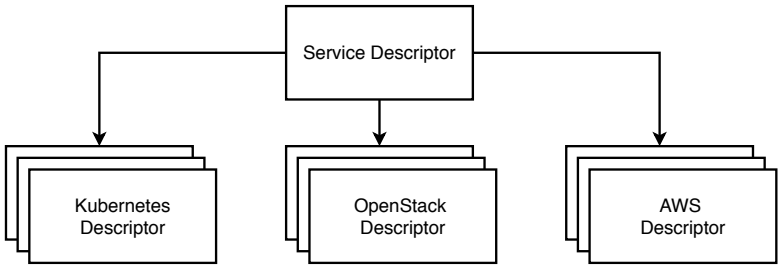


Figure 5.2: The descriptor model in Pishahang

5.2.2.2 Infrastructure Adaptor

IA is the key plugin in Pishahang for creating complex infrastructures to run Pishahang's services. Complex infrastructure here means a group of resources unified from different compute domains to run a service. For example, consider a service composed of a VM-, a CN- and an FPGA-based VNF. For such a service, we need a VM from OpenStack to run the VM-based VNF, a Pod from Kubernetes to run the CN-based VNF, and an AWS F1 to run the FPGA-based VNF. Such a group of resources, which I call *complex infrastructure*, is created by the IA plugin in Pishahang.

There are multiple options for creating such complex infrastructures. For exam-

ple, Heat[5], the OpenStack plugin for service orchestration, allows infrastructures to be described in code. In OpenStack Heat, infrastructures can be described using YAML Ain't Markup Language (YAML)-based Heat templates. The problem with a Heat template is that it is specific to the OpenStack domain and cannot be used to create infrastructure in other domains. The counterpart of Heat in AWS is CloudFormation, which works the same as Heat but is specific to AWS and cannot be used for other domains. For the Kubernetes domain, Kubernetes Manifest can be used to describe the resources needed to run an application. Like OpenStack Heat and AWS CloudFormation, Kubernetes manifest is also specific to a single domain.

One solution here would be to integrate Heat, Manifest, and CloudFormation into an IA and manage each domain separately. Employing this solution, three plugins need to be implemented to translate service descriptors to heat and CloudFormtation templates and Kubernetes manifest. The state of the created resources/infrastructure then needs to be managed by each plugin separately. However, in Pishahahng, a better solution, called Terraform[6], has been used. Terraform allows infrastructure to be described as code in a Terraform template. It supports all compute domains and also has a built-in state management solution. In Terraform, infrastructures are created in three stages, namely *init*, *plan*, and *apply*. In the *init* stage, a workspace is created and then configured using the information provided in the template (e.g., Terraform providers, credential). In the *plan* stage, an execution plan is prompted that shows what changes will be made on the infrastructure (e.g., what resources are created, deleted, or updated) compared to the last state. Finally, at the *apply* stage, changes specified in the Terraform templates will be applied to the actual infrastructure.

We integrated Terraform into IA. IA translates service descriptors to Terraform templates, which then will be used by Terraform to create the infrastructure. IA creates two levels of Terraform templates, namely complex infrastructure and domain levels. The complex infrastructure template consists of information about the constituent domains. For example, it specifies Terraform providers (AWS, OpenStack, Kubernetes), hostname/IP addresses of the domains, and credentials to access domain's Application Programming Interface (API)s (i.e., credentials can also be retrieved from secret management tools such as Vault[7]). Fig. 5.3 shows an example of a complex infrastructure template. The domain template, however, has three types of templates, one each for OpenStack, Kubernetes, and AWS. They are used to describe the resources that are needed to run VNFs in a specific domain. To some extent, the domain template is the translation of VNF descriptors to Terraform templates. The main difference is that VNF descriptors describe the specification of only one VNF in a domain, but domain templates describe the requirements of all VNFs of a domain.

[5] `https://wiki.openstack.org/wiki/Heat`, accessed July 2020
[6] `https://www.terraform.io/`, accessed July 2020
[7] `https://www.vaultproject.io/`, accessed July 2020

```
terraform {
  required_version = "= 0.12.26"
}

provider "kubernetes" {
  host = "{{ endpoint }}"
  token = "{{ token }}"
  client_certificate = "{{ certificate }}"
}

provider "aws" {
        region = "{{ region }}"
        access_key = "{{ access_key }}"
        secret_key = "{{ secret_key }}"

}

provider "openstack" {
  user_name   = "{{ admin }}"
  tenant_name = "{{ tenant }}"
  password    = "{{ pwd }}"
  auth_url    = "{{ url }}"
}
```

Figure 5.3: An example of complex infrastructure template

5.2.2.3 Cross-domain Service Chaining

Now that the deployment of services across different compute domains is provided, we need to chain these VNFs. To support such chaining, I extended the Pishahang MANO framework with a new plugin called SFCM. SFCM plugin works as an adaptor and enables the communication between the Pishahang MANO framework and the SDN controller. It translates forwarding graphs, described in the Network Service Descriptor (NSD), to a sequence of MAC/IP addresses and forwards it to the SDN controller along with a Virtual LAN (VLAN) id. The VLAN id is used to classify the flows belonging to a forwarding path.

To create the sequence, SFCM plugin, first extracts the VNF Connection Point (CP)s, that should be included in the inter-domain chain, from the NSD. Then, it retrieves the MAC/IP addresses of these CPs from their corresponding VIMs. The SFCM plugin retrieves the MAC and IP addresses after the successful deployment of the VNFs. On the OpenStack domain, the SDN plugin retrieves the floating IP address associated with the VNF and, on the AWS domain, the public IP address of the VNF is retrieved. On the Kubernetes domain, however, it is more complicated as Kubernetes does not provide a fixed IP address for Pods. The IP addresses of Pods change when pods get restarted, for example, due to a failure. To solve this issue, MetalLB[8] is used in the Kubernetes cluster. MetalLB is a Google project that provides a network load balancer implementation for bare-metal clusters. Its services (like address allocation) allow providing fixed IP addresses for Pods. IP addresses allocated by MetalLB can also be accessed

[8] `https://metallb.universe.tf/`, accessed June 24, 2020

externally. This completes the last puzzle in chaining VNFs across these three domains. SFCM plugin retrieves the IP and MAC addresses allocated by MetalLb to the VNFs in Kubernetes domains. The MAC/IP sequence is, then, created based on the sequence specified the forwarding path. Finally, SFCM generates a unique id which will be assigned to the path by the SDN controller as VLAN id.

As for the SDN controller, in Pishahang, we use Ryu[9] because of its simplicity. The Ryu controller is responsible for translating the MAC/IP sequence to forwarding rules and installing them on OpenFlow switches. The SDN controller creates two types of forwarding rules, classifier and forwarder. The classifier rules embed a VLAN tag to the packets and then forward them to the next hop. The forwarder rules, however, only forward the packets. Forwarding the packets is based the MAC/IP, ingress port number and the VLAN id.

5.3 Pishahang in 5G-PICTURE

Pishahang has been widely used in the 5G-PICTURE[10] project for different purposes. In this section, I describe how Pishahang is used in the context of the 5G-PICTURE project and for what purposes.

5.3.1 5G Operating System

One of the main goals of the 5G-PICTURE project was to design and implement a 5G Operating System (OS) [12]. 5G OS works as an umbrella over different management frameworks of 5G resources. It covers functionalities such as NFV orchestrators, SDN controllers, End To End (E2E) slice management, Business Support System (BSS), and Operations Support Systems (OSS). Integrating these functionalities, 5G OS aims at automating the deployment and lifecycle management of 5G services that run over a multi-domain, heterogeneous 5G infrastructure. To this end, the 5G OS architecture, shown in Fig. 5.4, has been designed; it consists of boxes representing different functionalities provided by the 5G OS and the high-level interfaces among them. In this context, Pishahang has been used as an NFV MANO that manages and orchestrates VM- and CN-based VNFs.

5.3.2 Pishahang in 5G Operating System

5G OS has been validated and evaluated in the project, leveraging a test-bed spread across Germany, Spain, Greece, and the UK. The high-level architecture of the test-bed is shown in Fig. 5.5. It consists of multiple technological domains, namely compute, Radio Access Network (RAN), and transport network domains. All these domains are managed by a centralised management entity called Multi Domain Orchestrator (MDO) that has been implemented in the 5G-PICTURE project. Each domain, then, uses different tools to manage the resources. For

[9] `https://ryu-sdn.org/`, accessed July 2020

[10] `https://www.5g-picture-project.eu/`, accessed June 17, 2020

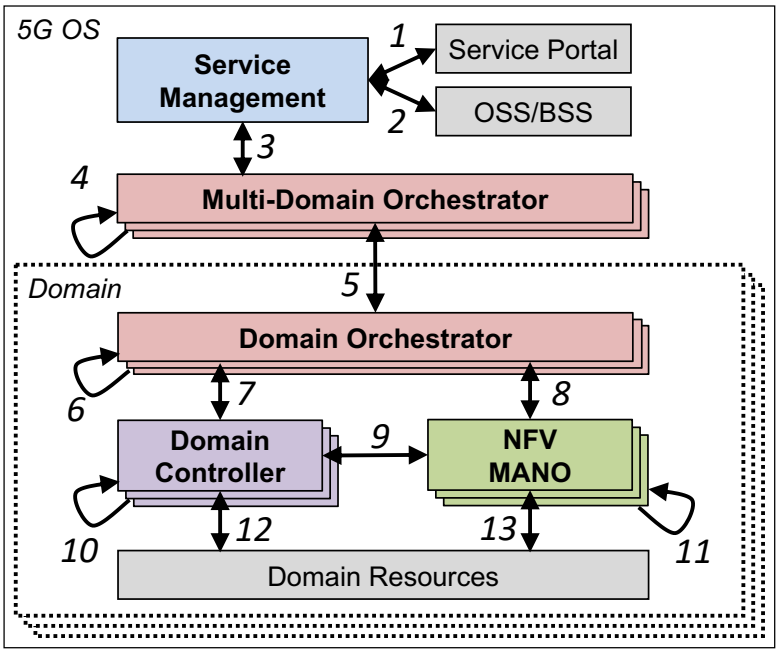


Figure 5.4: The high-level architecture of 5G Operating System [12]

example, the domain at Paderborn University (UPB) has been used as a core network for Wireless Fidelity (WiFi) and Long Term Evolution (LTE). The UPB domain employs Pishahang to manage services over compute resources. In this section, I only describe the UPB domain, where Pishahang has been evaluated and validated. The results of this work have been published in NetSoft 2020 [8]. The paper includes a full description of the conducted evaluation.

In the evaluation, the service provisioning time has been taken as a metric and two services, namely LTE and WiFi, have been used as case studies. For the LTE service, a virtual OpenAirInterface (OAI) is deployed on the core network managed by Pishahang. The virtual OAI is a VM-based VNF that runs on an OpenStack cluster. For the WiFi service, however, a Dynamic Host Configuration Protocol (DHCP) server is deployed on the core network using Pishahang. The DHCP server is a container-based VNF that runs on Kubernetes cluster.

The Pishahang cluster shown in Fig. 5.6 consists of two compute nodes and a VM. The VM is used to run the Pishahang orchestrator and equipped with 16 CPU cores of Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30 GHz and 32 GB of RAM. The single-node-cluster Kubernetes is provided in a workstation equipped with eight cores of Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz, 16 GB of RAM. The OpenStack cluster also includes a single node, which is a workstation equipped with 20 cores of Intel(R) Xeon(R) W-2155 CPU @ 3.30GHz and 64 GB of RAM.

Fig. 5.7a and 5.7b shows the Cumulative Distribution Function (CDF) of virtual OAI and DHCP server provisioning time, respectively. As we can see in the figures, the virtual OAI requires up to 2 minutes to prevision. On the other hand, the DHCP server can be provisioned in less than 5 seconds. This is as expected because the OAI is based on a virtual machine that is slow to provision compared to the container that is used to run the DHCP server.

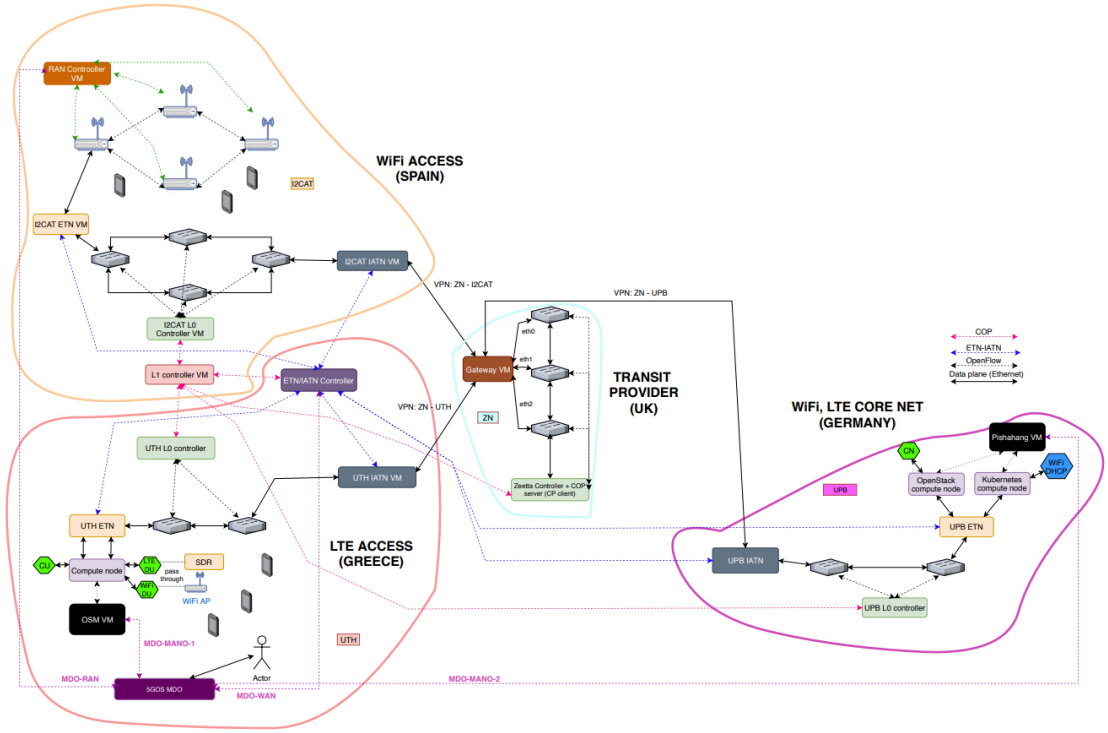


Figure 5.5: The high-level architecture of the test-bed used for 5G OS validation and evaluation [8]

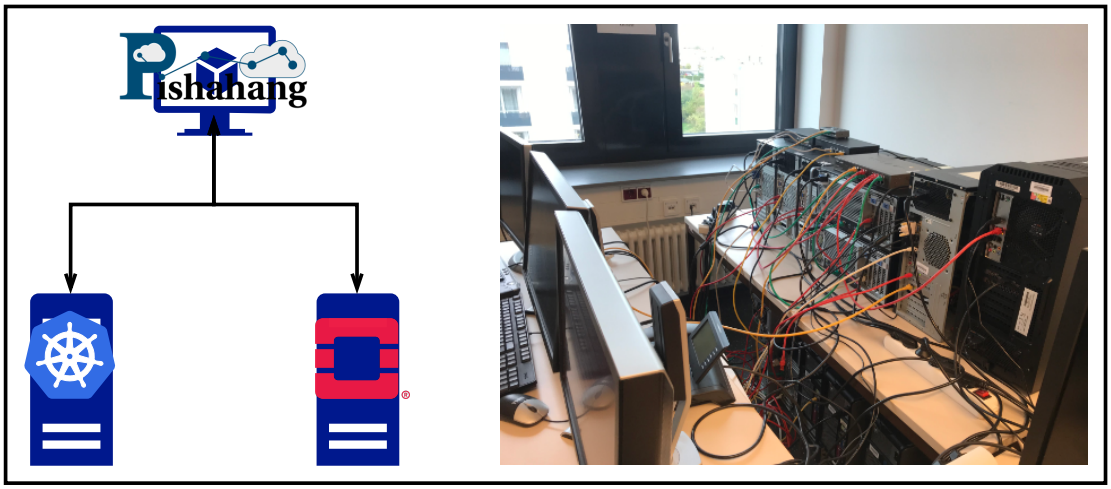
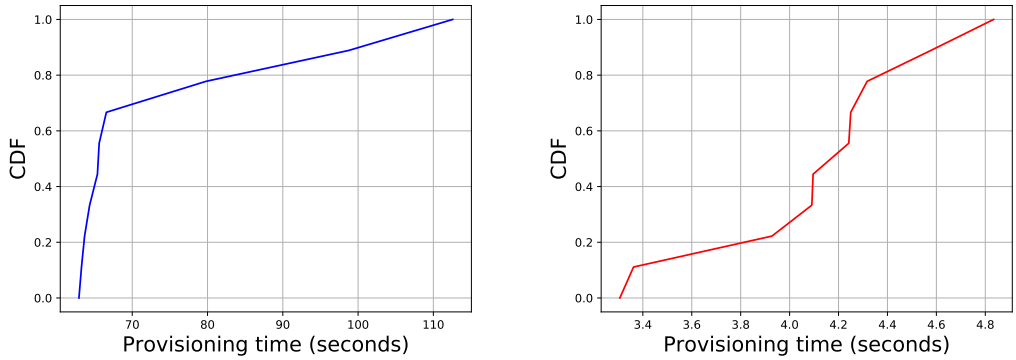


Figure 5.6: The test-bed facilities used for evaluation of 5G Operating System

5.3.3 Pishahang in 5G-PICTURE Demonstration

Pishahang has also been used in the 5G-PICTURE final demonstration. The aim of the demonstration was to show the capability of the 5G OS in managing network slices and deploying connectivity services dynamically over a stadium infrastructure. This has been shown by creating two network slices: (i) low-priority slice to be used as a baseline and (ii) high-priority slice to be used by spectators to share videos during a match in the stadium. In this context, I have



(a) Provisioning time of the VM-based virtual OAI)

(b) Provisioning time of the CN-based DHCP server

Figure 5.7: The provisioning time of LTE and WiFi services deployed by Pishahang

implemented a network service that can detect the need to handover between different slices to improve the Quality of Experience (QoE). This network service runs on the edge compute resources (i.e., located at the stadium), which are managed by Pishahang. Like Section 5.3.2, this section also focuses on the part of the demonstration that Pishahang has been leveraged. In the following, I describe the network service that has been implemented for this demonstration and the evaluation results.

5.3.3.1 Network Service

Fig. 5.8 shows the architecture of the network service. It follows the microservice-based architecture and consists of four microservices: (i) packet sniffer to inspect the traffic, (ii) database to store the extracted data from traffic for further analysis, (iii) endpoint that allows accessing the data stored in the database, and (iv) message broker that allows microservices to exchange messages with one another.

As mentioned before, this network service allows 5G OS to detect the need to switch to the high-priority slice. This is done by inspecting the incoming traffic, identifying the Watchity[11] application (i.e., the application is used to share videos) traffic, extracting the IP and MAC addresses of these flows and storing them along with a timestamp in a database. The IP and MAC are then used to identify which user should be given the higher-priority slice to improve QoE. The microservices are realised on containers and run under the Kubernetes cluster. Pishahang on top of Kubernetes manages the lifecycle of this network service.

[11] `https://www.watchity.com/`, accessed June 23, 2020

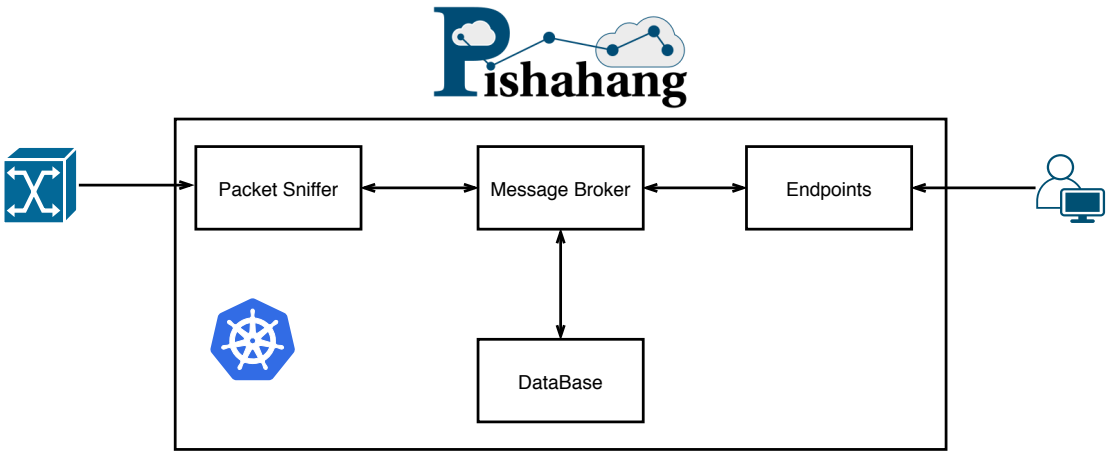
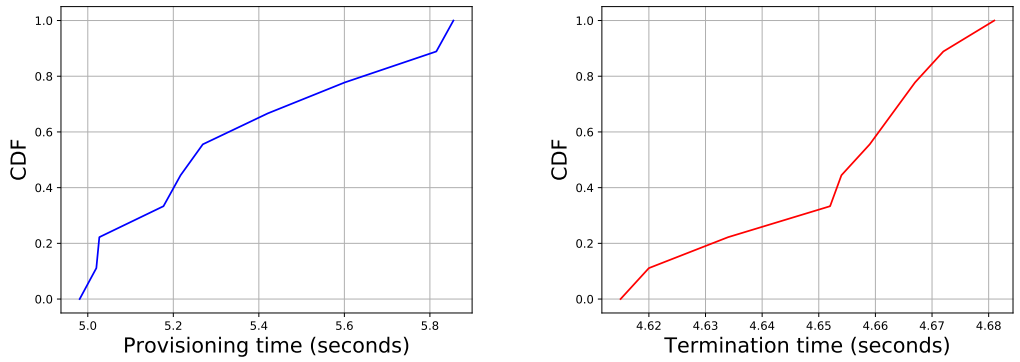


Figure 5.8: The high-level architecture of the network service

5.3.3.2 Evaluation Results

The provisioning and termination time of the network are measured and are shown in Fig. 5.9. As shown in the figure, the provisioning time of 4 microservices involved in this service takes less than 6 seconds. Also, to terminate this service, Pishahang requires less than 5 seconds.



(a) Provisioning time (b) Termination time

Figure 5.9: The provisioning and termination time of CN-based services deployed by Pishahang

5.4 Related Work

The European Telecommunication Standards Institute (ETSI) NFV Group has proposed a specification of the NFV MANO framework to manage the lifecycle of VNFs and orchestrate network services. Based on these specifications, MANO frameworks such as OSM and SONATA have been developed. Irrespective of all their pros and cons, none of them can manage or orchestrate the wide range of

resources that are supported by Pishahang. In the following, we mention some of these MANO frameworks and discuss their capabilities to manage heterogeneous resources.

OSM[12], an NFV MANO framework, supports separate infrastructure adaptors for OpenVIM, OpenStack, VMware vCloud, AWS, and Kubernetes. The infrastructure adaptors use these platforms' APIs to provision and manage services. It uses infrastructure-platform-independent descriptors to describe service requirements. Although OSM supports multiple CMSs, it does not support the deployment of services across multiple domains. Moreover, OSM AWS adaptor does not support AWS F1 services, which provide FPGA as a service.

Cloudify[13], a native cloud management system, utilises ARIA[14] to orchestrate VNFs and provides a plugin that allows operators to deploy services on an existing Kubernetes cluster. It also supports the orchestration of VM- and CN-based VNFs by allowing service developers to use multiple infrastructure provider plugins per service definition (so-called blueprints). However, it is not possible to chain VNFs across different domains. Moreover, Cloudify's blueprints depend on the specific infrastructure platform (AWS, Azure, etc.), thus forcing developers to re-write their service blueprints when switching infrastructure platforms.

SONATA[15] supports Kubernetes in its latest version. However, like OSM, SONATA does not support the orchestration of services across different domains. SONATA is also limited to OpenStack and Kubernetes and does not support any other CMSs.

Open Network Automation Platform (ONAP)[16] is another MANO framework that supports a wide range of CMSs and also chaining across different domains. However, ONAP does not support AWS F1 service, missing the support for FPGAs.

Pishahang is superior to all aforementioned MANO frameworks in term of supporting heterogeneous resources as it supports the largest variety of resources.

5.5 Conclusion

In this chapter, supporting heterogeneous resources using an NFV MANO framework has been addressed. To this end, Pishahang MANO framework has been implemented, which orchestrates services across domains that are managed by different CMSs. Having multiple domains as such allows supporting a wide range of heterogeneous resources.

Pishahang creates logical complex infrastructures out of the underlying domains. Complex infrastructures are composed of heterogeneous resources from different domains which can be used to host services that need to run on heterogeneous resources. The complex infrastructures are realised by employing Terraform

[12] `https://osm.etsi.org/`, accessed July 2020
[13] `https://cloudify.co/`, accessed July 2020
[14] `https://ariatosca.incubator.apache.org/`, accessed July 2020
[15] `https://www.sonata-nfv.eu/,accessedJuly2020`
[16] `https://www.onap.org/`, accessed July 2020

as VIM adaptor. Terraform supports a long list of CMSs[17] which also significantly simplifies the integration of new domains into Pishahang.

Pishahang also chains VNFs across different domains. Intra-domain chaining, however, is still not fully supported, specifically for Kubernetes and AWS domains. This is basically because these two domains do not provide production-ready solutions for service chaining.

Pishahang has been validated and tested in the 5G-PICTURE project. Using Pishahang, we deployed VM- and CN-based services across different domains on clusters equipped with GPPs and GPUs. Pishahang has also been benchmarked in 5G-PICTURE and the provisioning time of different services have been presented in this chapter.

[17] `https://www.terraform.io/docs/providers/index.html`, accessed July 2020

6

Dynamic Management and Orchestration of Network Services

This chapter is about dynamically managing and orchestrating network services over heterogeneous resources. This work has been done in the context of 5G-PICTURE[1] project. The results of this work have been demonstrated in 5G-PICTURE review meetings in Nov. 2019 in Barcelona. Two regular papers, [24] and [26], have been published out of this work in CNSM 2019 and EuCNC 2020, respectively.

6.1 Introduction

Network Function Virtualization (NFV) services have a variety of requirements that can change during the lifecycle of services. These requirements can be, for example, different data rates, latencies, and cost. To meet these requirements, various hardware resources or software structures have been suggested [36] [16] [35]. As usual, there are trade-offs between different hardware resources or software structures to realise a service, and there is no one single solution that can meet all requirements and optimise all metrics. For example, to improve the performance of compute- and network-intensive Network Function (NF)s, using acceleration hardware such as Graphics Processing Unit (GPU) or Field Programmable Gate Array (FPGA) is proposed [50] [36]. Nevertheless, as acceleration hardware is expensive, they increase the cost of services; this might not be desirable for a particular service user or a load level that can be handled by cheaper resources. This problem can be solved by dynamically provisioning network services: we switch to a different service implementation on the fly as service requirements change. This can provide service performance most suitable to the given requirements. For example, consider two versions of a Deep Packet Inspection (DPI) NF: v1 is a CPU-based, v2 an FPGA-based implementation. On the one hand, deploying v1 would perform well and be cheap when the input data rate is low, but it would not perform well for high data rates. On the other hand, v2 performs better at high data rate but is too expensive at low data rates [36]. To decide between these two DPI versions, a service package consisting of both versions can be used that allows us to deploy the right version and then switch between them on the fly as the input data rate changes. This type of services is called *multi-version services*, a concept that is proposed in [11]. While the idea looks appealing in theory, an experimental evaluation of such a service provisioning approach is still missing.

In this chapter, I describe my contribution to the dynamic provisioning of network services that are realised on heterogeneous resources based on different requirements. To this end, first, I classify multi-version services based on types of resources (i.e., compute and virtualisation resources) and management levels (i.e., function and service levels). Then, I quantify the trade-off between cost and performance of an example multi-version service. I used a virtual Transcoder (vTC) as a case study and implemented a Commercial Off-The-Shelf (COTS)-based and a GPU-assisted virtual transcoder. I evaluated these vTC versions by metrics such as resource utilisation and processing time. Based on this evaluation, I analysed

[1] `https://www.5g-picture-project.eu/`, accessed June 2020

the cost of vTCs running on cloud-based COTS and acceleration resources. I, further, extended Pishahang Management and Orchestration (MANO) framework (introduced in Chapter 5) to support the orchestration of multi-version services. Using Pishahang, I then evaluated the management overhead of dynamic service deployment.

The remainder of this chapter is structured as follows. In Section 6.2, I elaborate on the concept of multi-version services. In Section 6.3, I present an analysis of multi-version services in terms of performance and cost. An orchestration solution for multi-version services is described and evaluated in section 6.4. Section 6.5 highlights the related work and Section 6.6 concludes the chapter.

6.2 Multi-version Services

For dynamic, situation-based provisioning of Virtualized Network Function (VNF)s over heterogeneous resources, VNFs need to be provided in different versions. This is because, for example, a VNF implemented to run on a GPU cannot be used to run on an FPGA. Versioning can be provided on two levels: (1) on the level of individual VNFs, which I call Multi-version Network Function (MVNF)s and (2) on the level of network services, which I call Multi-version Network Service (MVNS)s. In multi-version VNFs, each version is an implementation of a VNF for specific hardware (e.g., GPU, FPGA) or virtualization environment (e.g., Virtual Machine (VM), Container (CN)). In the multi-version services, each version includes a specific combination of VNFs that run on heterogeneous resources. I elaborate on these two levels of versioning in the following.

6.2.1 Multi-Version Network Function (MVNF)

MVNF is a function package consisting of multiple implementation versions of a network function. As illustrated in Fig. 6.1, MVNFs can be categorised into two types: (1) MVNFs with different virtualization techniques, and (2) MVNFs with different hardware implementations. Below, I explain each of these types in detail.

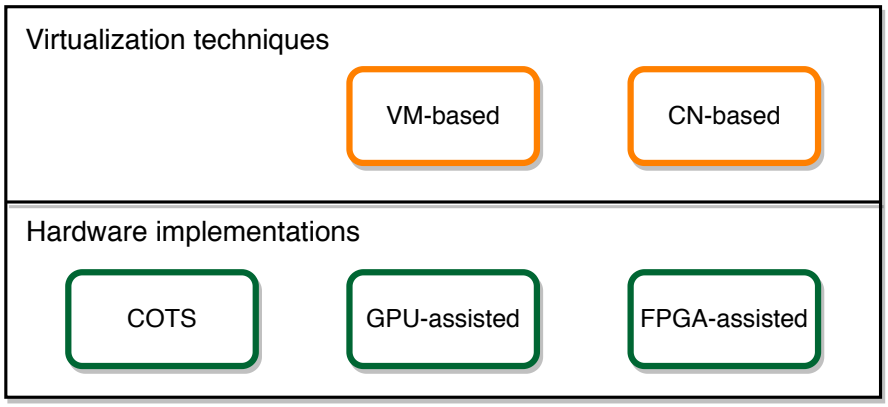


Figure 6.1: Different types of multi-version network functions (MVNFs)

6.2.1.1 MVNFs Available for Multiple Virtualization Techniques

MVNF implementations can be categorised based on virtualisation techniques. As mentioned in Chapter 5, there are two main virtualisation techniques, namely VM and CN. While VMs provide better isolation and security, CNs provide better resources utilisation (i.e., consequently cheaper technology) and higher agility [6][17]. To balance these requirements (e.g., security and cost), VNFs can be provided in CN- and VM-based versions. This allows to alternate them on the fly as requirements change.

6.2.1.2 MVNFs with Multiple Hardware Implementations

As mentioned before, when it comes to using acceleration resource (e.g., GPU, FPGA), we need to consider the trade-off between cost and performance. This is because acceleration hardware can increase the cost in some scenarios without improving the performance (i.e., this claim is evaluated in Section 6.3 for an example VNF). To balance between the cost and performance of MVNFs, we can provide them with multiple hardware implementations. This enables us to switch between different versions on the fly according to service requirements.

6.2.2 Multi-Version Network Service (MVNS)

MVNS is a network service that consists of multiple deployment versions of a network service chain. Each deployment version can have different VNF types. As an example, shown in Fig. 6.2, a chain consisting of DPI, Firewall, and Network Address Translation (NAT) VNFs is used in the data centre between the data centre router and the servers to provide value-added services [30]. This chain can be offered in multiple versions to provide different levels of performance, cost and security. All the three VNFs involved in this chain can be versioned based on hardware implementation (i.e., FPGA and COTS) and virtualisation technique (i.e., VM and CN). This gives us at least four versions that can be alternate depending on service requirements. For example, we can deploy version 1 when we want the service to be cheap and secure. Version 2 can be used when we want the service to be cheap and performant. Version 3 can be used when the service is required to be performant and secure. Finally, version 4 can be deployed when the service needs to be performant and can tolerate a lower level of security compared to version 3 to be agile and cheaper.

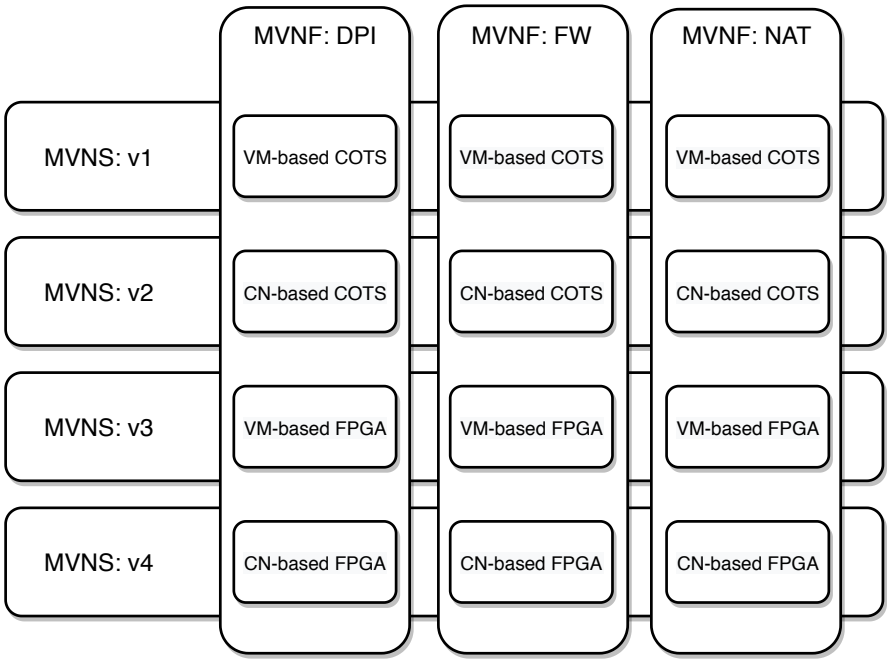


Figure 6.2: An example of multi-version network services (MVNSs)

6.3 Multi-version Services Analysis

The use of multi-version services will be beneficial if there are trade-offs between using different versions of a service. Without such trade-offs, there would not be any need to have multiple versions of a service, but simply scaling a single version of service up and down would be all that is needed for optimal performance. To find out if there are such trade-offs, I have conducted an experimental evaluation and quantified the cost and performance of different versions of an example service to see if there is any trade-off between different versions.

In this evaluation, vTC has been used as a case study as it consists of compute-intensive processes that can be offloaded to acceleration hardware. A transcoder provides functionalities such as converting video encoding format and spatial resolution, video transposing, and video transcasting. These functionalities are provided to adjust the original video to the viewer's network data rate, device resolution, frame rate and so on. Transcoder is used in both Video On Demand (VOD) (e.g., Youtube, Netflix) and live-streaming services to provide high-quality video streaming experience for the viewers [28]. Employing vTC as an example, I have analysed performance and cost of multi-version services, which is discussed in the following.

6.3.1 Performance Analysis

To analyse performance, I have implemented two versions of a vTC: (v1) a COTS-based vTC that is designed to only utilise CPU for video processing and (v2) a GPU-assisted vTC that offloads compute-intensive processes to GPU. Both

versions are based on FFmpeg[2], which is a software-based transcoder providing a wide range of transcoding functionalities. To emulate the NFV environment, I have deployed vTCs on KVM-based virtual machines. Fig. 6.3 shows the test-bed set-up used for the evaluation. It consists of four VMs: VM #1 hosts a packet generator that breaks down videos to Group of Pictures (GOP) [28] and embeds them into Real-Time Transport Protocol (RTP) packet payloads to be sent to vTCs, VM #2 runs the COTS-based vTC, VM #3 hosts the GPU-assisted vTC, and VM #4 receives the transcoded videos from vTCs and display them using a video player. All VMs were running on a server equipped with an Intel(R) Xeon(R) W-2123 CPU at 3.60GHz (8 Processors), 8 GB DDR4 RAM, and an Nvidia GeForce RTX 2080 GPU.

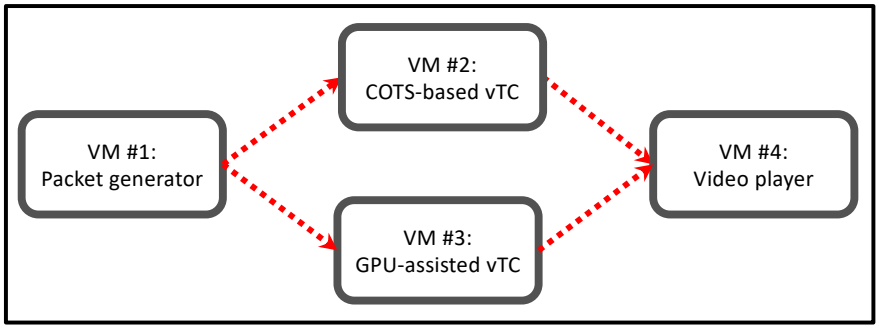


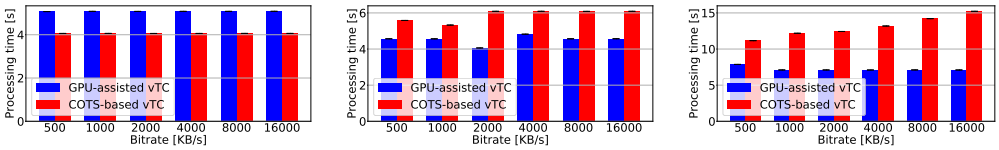
Figure 6.3: The test-bed set-up used for the evaluation

We have considered two metrics, namely the video transcoding processing time for the entire video and CPU/GPU utilization. These two metrics are considered as processing time can affect the total latency of video streaming services and CPU/GPU utilization is an indicator of the service cost. Parameters of the evaluation are the video bitrate and resolution. For a range of video resolutions, we have measured the performance of vTCs. For each video resolution, we ran the test for a range of input bitrates. Both transcoders convert the incoming video format to H.264 format. Also, the Big Buck Bunny[3] video has been used as the input video.

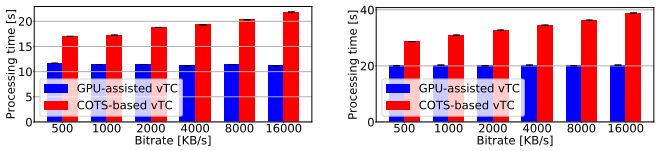
The video processing time results, illustrated in Fig. 6.4, show that the GPU-assisted vTC processes the videos much faster than the COTS-based vTC when the video has a high resolution and bitrate. Fig. 6.4e shows that the processing time difference can reach up to 40 seconds for videos with 1080p resolution and 1.6 MB/s Bitrate, which is indeed a remarkable difference. However, the processing time difference decreases as the video resolution gets lower. For example, looking at processing times for videos with 240p resolution (Fig. 6.4b), we see that the processing time difference goes down up to 1 second and, for 120p resolution videos (Fig. 6.4a), the COTS-based vTC even outperforms the GPU-assisted vTC. This is because, in the case of GPU-assisted vTC, there is an extra CPU and GPU communication overhead that does not exist in the COTS-based vTC. This

[2] `https://ffmpeg.org/`, accessed May 18, 2019
[3] `http://bbb3d.renderfarming.net/download.html`, accessed May 27, 2019

increases the total processing time of the GPU-assisted vTCs; however, as this overhead is very low, it has no significant impact on the processing time of high-resolution videos.



(a) Input video with 120p resolution (b) Input video with 240p resolution (c) Input video with 480p resolution



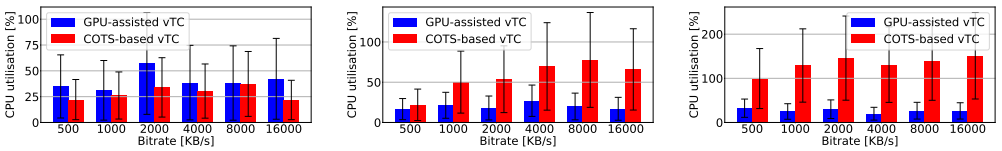
(d) Input video with 720p resolution (e) Input video with 1080p resolution

Figure 6.4: Transcoding processing time for videos with different resolutions (with 95 % confidence interval - error bars are too small)

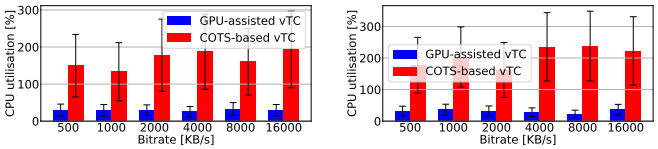
The CPU utilization evaluation results are depicted in Fig. 6.5. As expected, the trend is the same as what we got in the video processing time evaluation. While the GPU-assisted vTCs utilizes less CPU for *high-resolution videos* (Fig. 6.5e), COTS-based vTC uses less CPU to process *low-resolution videos*. This is because there is no processing associated with exchanging data between CPU and GPU in the COTS-based vTC. Looking at the results, the CPU utilization of GPU-assisted vTC remains between 30 % to 50 % for all video resolutions; however, COTS-based vTC utilization varies from 20 % for the 120p resolution (Fig. 6.5a) to 250 % for 1080p resolution (Fig. 6.5e). 250 % CPU utilization means at least 3 CPU cores are needed for the process in which 2 of them are fully utilized and the other is 50 % utilized.

We have also measured the GPU memory utilisation of GPU-assisted vTC. The results are shown in Fig. 6.6. Performing this evaluation, we observed that changing the input bitrate does not change the GPU utilisation, and it always remains constant. However, the GPU utilisation increases, as the input video resolution gets larger.

Based on the processing time and resource usage evaluation, we observe that there is a trade-off between using COTS-based and GPU-assisted vTCs. Although using GPUs can improve the processing time of vTC in some cases, in some other cases, it is not a suitable implementation option as it increases the resource usage while not providing any performance improvement.



(a) Input video with 120p resolution (b) Input video with 240p resolution (c) Input video with 480p resolution



(d) Input video with 720p resolution (e) Input video with 1080p resolution

Figure 6.5: Transcoding CPU utilization for videos with different resolutions (with 95 % confidence interval)

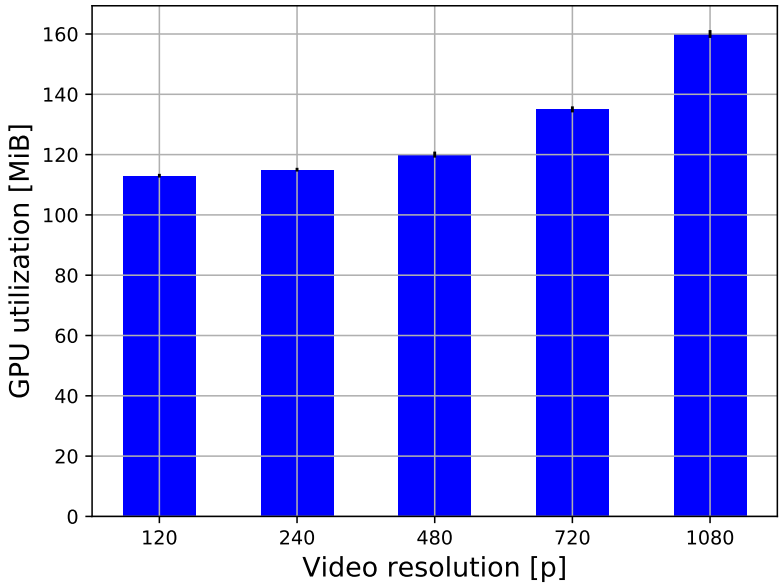


Figure 6.6: Memory usage of GPU-assisted vTC for videos with different resolutions (with 95 % confidence interval - error bars are too small)

6.3.2 Cost Analysis

We have also analysed the cost of running COTS-based and GPU-assisted vTCs. In the performance analysis, we observed how much CPU core and GPU memory are needed to run COTS-based and GPU-assisted vTCs, respectively. Having these data, we have looked at Amazon's EC2 price list[4] to see how much it costs

[4] `https://aws.amazon.com/ec2/pricing/on-demand/`, accessed May 20, 2019

to provide these resources in a virtualized cloud environment. In our analysis, Amazon's general-purpose "t2" instances are considered as possible instances to provide required resources for vTCs. For the GPU case, we consider the price of elastic graphics instances "eg1"[5] that allows GPU to be attached to EC2 instances.

The results of the cost analysis are illustrated in Fig. 6.6. It shows the cost of providing resources for different versions of the vTC for one hour. While the costs of GPU-assisted vTC remains constant for all video resolutions, the cost of COTS-based vTC shows an increasing trend as the video resolution increases. These results show that the use of COTS-based vTC for video with 1080p resolutions is inefficient both performance-wise and cost-wise. The same holds true for using GPU-assisted vTC to process videos with 120p and 240p resolutions. For the video with 480p and 720p resolutions, although GPU-based vTC performs better, it costs more compared to COTS-based vTC.

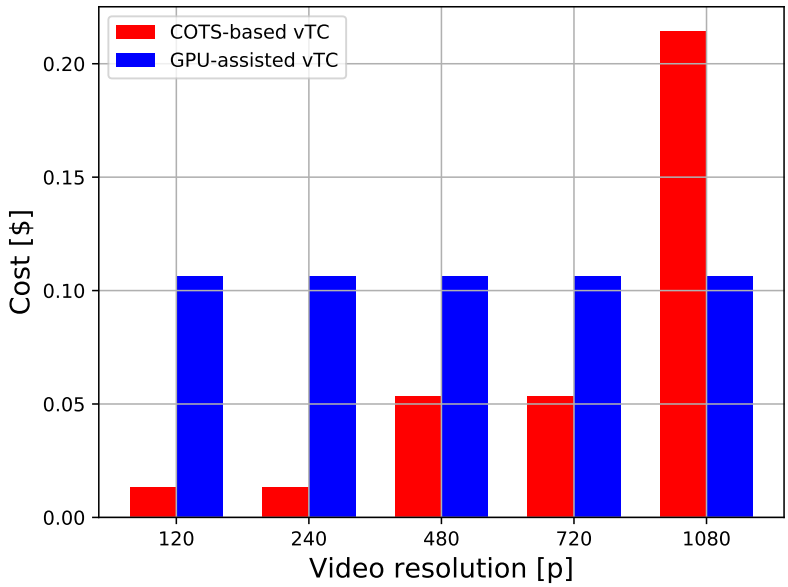


Figure 6.7: Cost of running different versions of vTC on Amazon Web Service (AWS) resources for one hour

From these results, we observe that GPUs are beneficial both performance-wise and cost-wise only when high-resolution or high-bitrate videos should be processed. When the input videos to the transcoder have low resolution or low bitrate, using GPUs is beneficial neither performance-wise nor cost-wise. Consequently, the use of GPU needs to be decided for each situation individually. This backs up the idea of multi-version services and dynamical service provisioning as using the right vTC version for the given input video can significantly improve performance and reduce cost.

Now that we know there are services that can benefit from multi-versioning, we need to see how we can orchestrate such services in an NFV environment and

[5] `https://aws.amazon.com/ec2/elastic-graphics/`, accessed May 20, 2019

what is the orchestration overhead of supporting multi-version services in NFV. In the next section, I address these points.

6.4 Multi-version Services Orchestration

To support management and orchestration of multi-version services, I extended Pishahang [21], a multi-domain NFV MANO framework, introduced in Chapter 5. Pishahang is based on a microservice architecture in which all functionalities that should be handled by the MANO framework are implemented using a set of loosely coupled microservices that are realised in containers. This simplifies extending Pishahang with new functionalities; this can be achieved by adding and integrating a new microservice (container) into the existing framework [13]. As mentioned in Chapter 5, Pishahang supports a wide range of heterogeneous resources on different domains. This is a valuable feature for multi-version services as it allows services to be offered in a large variety of versions. Although provisioning multiple versions of a VNF is already supported by Pishahang, it still lacks support for on-the-fly multi-version services. To be exact, it cannot switch to different versions of a service/VNF on the fly, while the service/VNF is in use. Therefore, Pishahang needs to be extended to support the management of multi-version services.

6.4.1 Requirements

The following requirements need to be met to manage and orchestrate multi-version services in Pishahang.

- **Req. 1:** The descriptor schema needs to be extended to support specific requirements of multi-version services. These requirements are, for example, constituent versions of a VNF, metrics to be monitored for a specific version, metrics thresholds to trigger version switching and the default version of a VNF or service.

- **Req. 2:** An algorithm is required to decide which version is best to deploy according to available resources, version requirements, and services demand (e.g., data rate).

- **Req. 3:** A monitoring system to gather metrics from running VNFs. Using these metrics (e.g., data rate), the service demand is measured.

- **Req. 4:** A new plugin needs to be implemented to manage multi-version services. This plugin should handle the communication between the monitoring system, the version selector algorithm, and other Pishahang plugins.

- **Req. 5:** Pishahang Service Lifecycle Management (SLM) needs (i.e., introduced in Chapter 3) to be extended to identify multi-version services and redirect respective requests to multi-version services manager.

6.4.2 Design and Implementation

To add orchestration support for multi-version services to Pishahang, I used existing solutions as much as possible. For example, to satisfy Req. 2, I employed the algorithm implemented by Dräxler et al. in [11]. They have implemented both the optimisation and the heuristic approaches as Python programs that optimise the cost and performance of multi-version services. I embedded the algorithm into the Multi-version Service Management (MVSM) plugin (see Section 6.4.2.2). Also, to meet Req. 3, I used Netdata [6]. Netdata is an open-source, real-time monitoring system that can be used to monitor the performance of systems and applications. It can be used to monitor physical and virtual servers plus containers. Netdata is fast and efficient and does not disrupt the performance of the core application. Compared to other solutions such as Prometheus [7] and Grafana [8], Netdata provides system metrics at a higher sampling rate, which allows to react faster to the changes in service demand. In the followings, I describe the extensions provided to Pishahang to support multi-version services.

6.4.2.1 Multi-version Service Descriptors

To meet Req. 1, I extended the descriptor model of Pishahang. As shown in Fig. 6.8, there are three main descriptors in the multi-version service descriptor model: (1) Multi-version Service (MVS) descriptor, (2) VNF descriptors and (3) Version descriptor. The MVS descriptor includes the high-level information of multi-version services such as the constituent VNFs, constituent service versions, and forwarding graph. Each service version in the MVS is represented by a forwarding graph. On lower layer, VNF descriptors are used to describe the high-level requirements of each VNF such as constituent VNF versions, their Virtualized Infrastructure Manager (VIM) type (OpenStack, Kubernetes, and AWS), and version switching information (e.g., default version, threshold values). Each version is then described using a version descriptor that includes the VNF requirements such as Version types (COTS, accelerated), deployment units, resources, and images.

6.4.2.2 Multi-version Service Manager

To meet Req. 4, I extended Pishahang with a new plugin, called MVSM plugin. This microservice decides when a service/VNF should be switched to which version. As it is illustrated in Fig. 6.10, the MVSM plugin consists of three main modules, including message handler, version selector, and monitoring. The message handler module takes care of the communication of MVSM plugin modules with other Pishahang microservices such as SLM. It processes the incoming messages to the MVSM plugin and hands them over to the respective module. The version selector module includes an algorithm taken from [11] that determines

[6] `https://www.netdata.cloud/`, accessed Jan. 22, 2020
[7] `https://prometheus.io/`, accessed July 2020
[8] `https://grafana.com/`, accessed July 2020

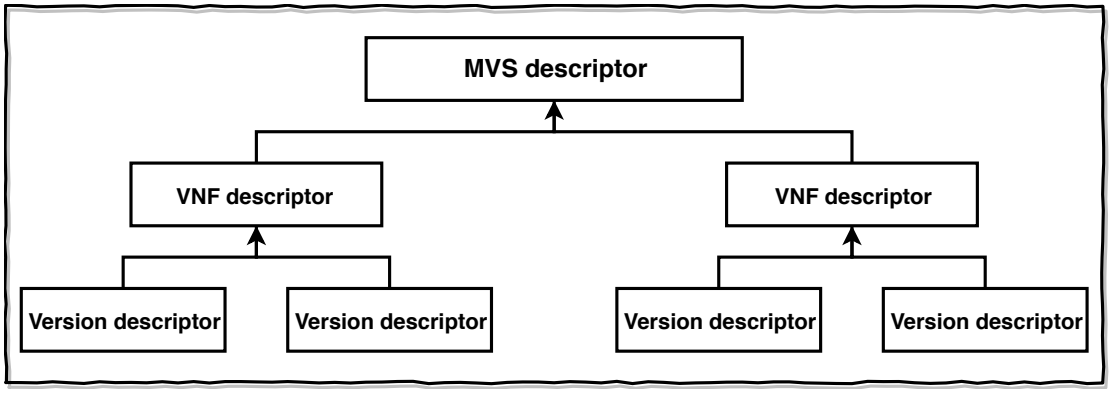


Figure 6.8: The multi-version services descriptor model

which version of the service/VNF should be used based on the service requirements, available resources, and service demand. Last but not least, the monitoring module, which is based on Netdata [9], monitors the performance of deployed VNFs along with the service demand.

Fig. 6.10 shows the workflow of the MVSM plugin and its interactions with other microservices in the Pishahang framework. The initial multi-version service request is sent to the MVSM plugin by SLM. The payload of this request includes the service descriptors (Network Service (NS) and VNF descriptors) and the available resources (e.g., CPU, GPU, Memory). The message is received by the message handler and then forwarded to the version selector. The version selector decides which version should be initially deployed based on the information given in the descriptors and available resources. Then, the selected version will be communicated with SLM by MVSM. SLM deploys the selected version and informs MVSM about the result of the deployment. Once the selected version is successfully deployed, the message handler sends a request to the monitoring module containing the VNF records (e.g., name, IP address), monitoring metrics (e.g., data rate, CPU), and the metrics threshold. Then, the monitoring module starts fetching monitoring data and sends an alert to the message handler whenever a threshold is reached. The message handler, then, forwards the new monitoring data to the version selector and notifies SLM if the version selector decides to switch the version. If the decision is to switch the service version, SLM deploys the new version, redirects the packets to the new version, and removes the old version.

SLM has been extended to identify requests related to multi-version services (meeting Req. 5). SLM identifies multi-version services based on the type of the descriptors included in the service requests and redirects the requests to MVSM. The orchestration of service requests containing regular service (i.e., regular services are single-version services) descriptors is taken care of by the SLM itself. This separates the orchestration of multi-version services from regular services. With this separation of orchestration, the regular services will still be fully supported as before.

[9] `https://www.netdata.cloud/`, accessed Jan. 22, 2020

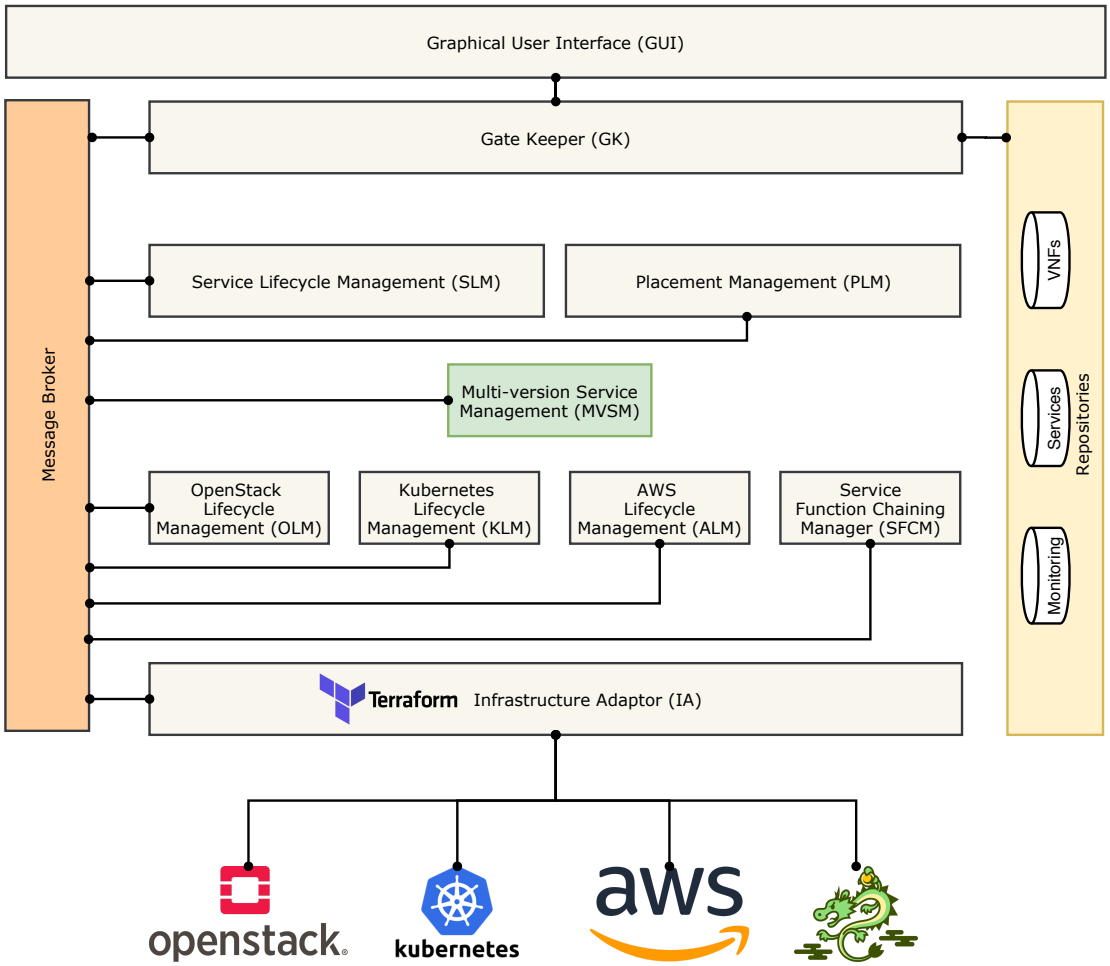


Figure 6.9: The high-level architecture of Pishahang containing the MVSM plugin to support multi-version services

6.4.3 Evaluation

I experimentally evaluated the switching time of different versions of multi-version services. Fig. 6.11 shows the set-up of our testbed. In this set-up, the extended version of Pishahang was running on a virtual machine equipped with 16 CPU cores of Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 32 GB of RAM. A single-node-cluster Kubernetes was used for CN-based NFs. This cluster includes a workstation equipped with eight cores of Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz, 16 GB of RAM, and a GeForce RTX 2080 GPU. The OpenStack cluster also includes a single node, which is a workstation equipped with 20 cores of Intel(R) Xeon(R) W-2155 CPU @ 3.30GHz and 64 GB of RAM. Moreover, a Ryu controller [10] along with a Zodiac FX OpenFlow switch was used to redirect the traffic to the intended service version on the fly.

For this evaluation, I also implemented an example multi-version service. This service consists of three versions of a vTC. This is an extended version of the vTC introduced in 6.3. The three versions of the vTC consist of two COTS

[10] `https://osrg.github.io/ryu/`, accessed Jan. 22, 2020

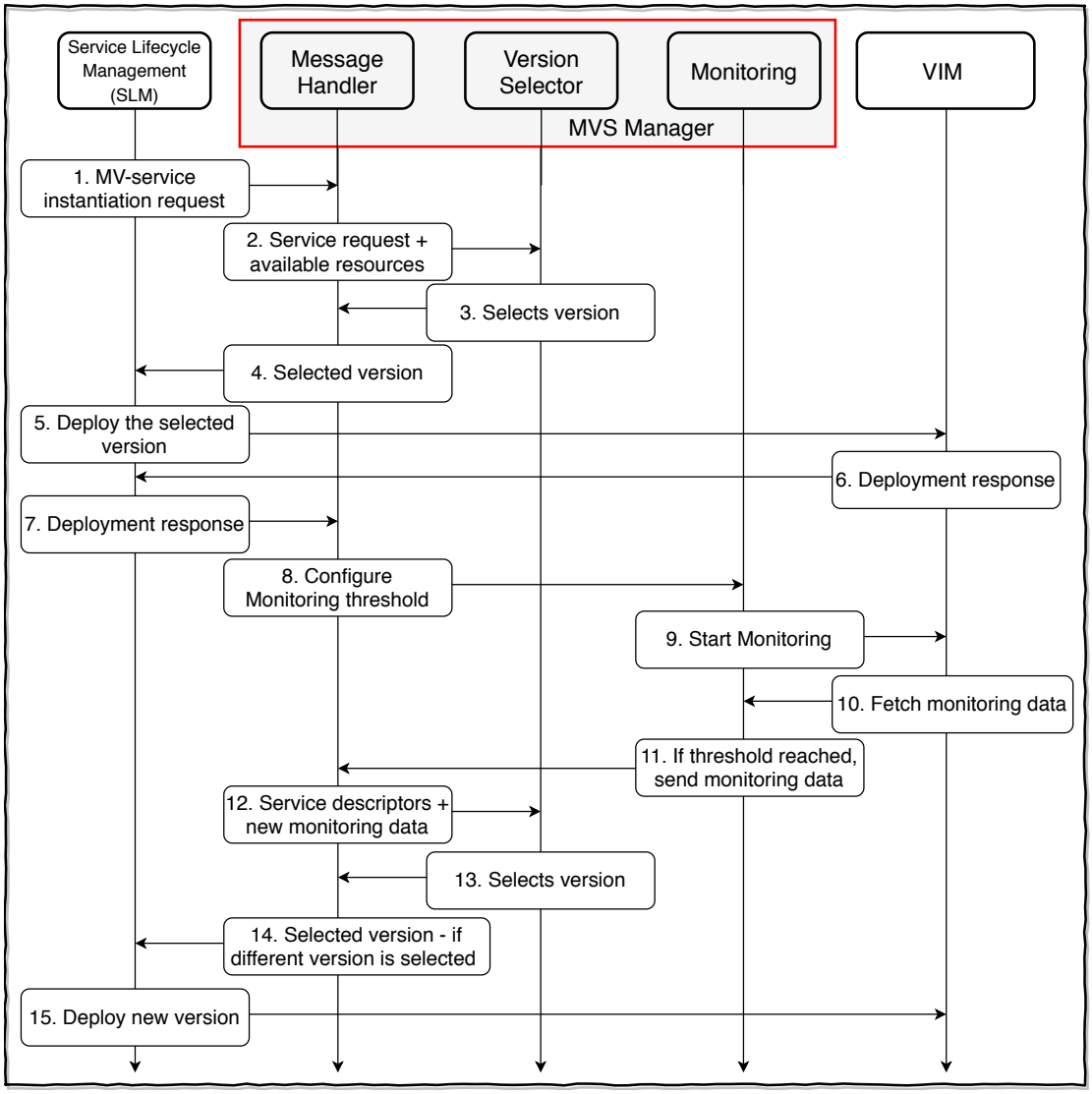


Figure 6.10: The workflow of the MVSM plugin and its interactions with other microservices in the Pishahang framework

versions, namely a VM-based COTS (VMC) vTC and a CN-based COTS (CNC) vTC. These two versions only use CPUs for all processes. The third version is an accelerated vTC that runs on a CN and offloads the compute-intensive processes to the GPU. This version is called CN-based Accelerated (CNA) vTC.

Fig. 6.12 depicts the high-level architecture of the vTC. It is composed of three FFmpeg [11] components. FFmpeg is an open-source solution for recording, converting and streaming audio and video. The first component on the left is Testsrc, which generates test pattern video frames to mimic a live streaming source. Testsrc runs on CPU on all vTC versions and provides raw videos with different bitrates and resolutions for the second component, the encoder. The encoder converts the video format to H.264. This is the main component of the vTC. For

[11] `https://www.ffmpeg.org/`, accessed Jan. 22, 2020

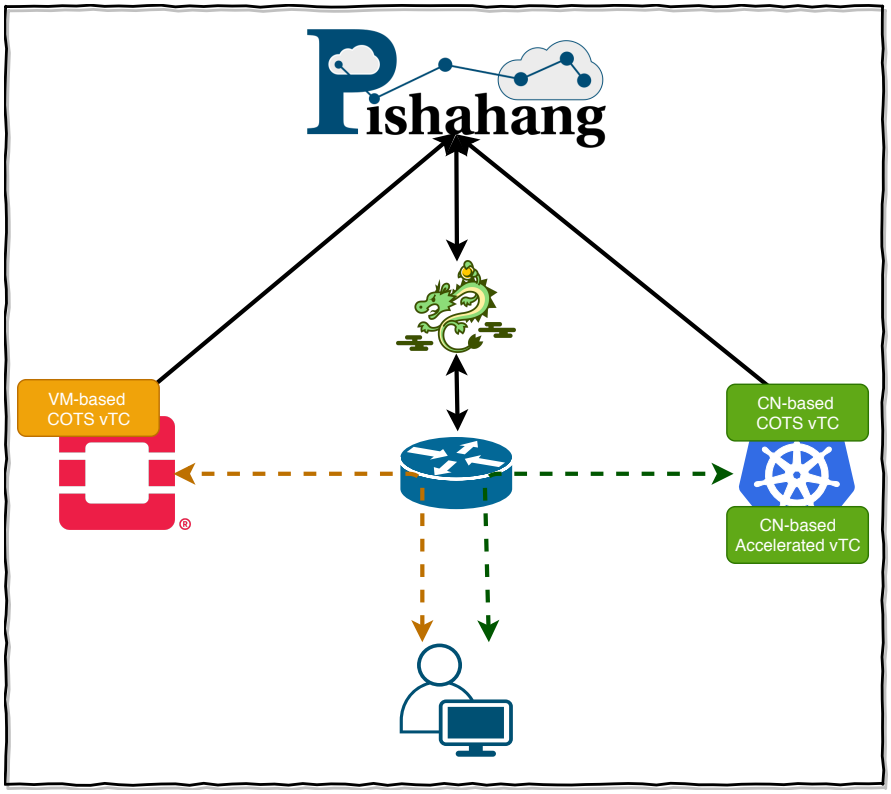


Figure 6.11: The test-bed set-up used for the evaluation

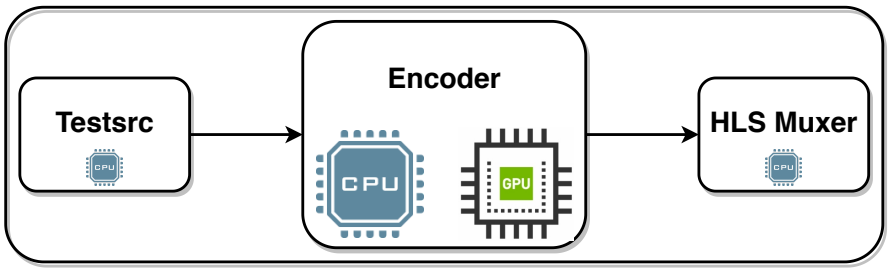


Figure 6.12: The high-level architecture of virtual transcoder

the COTS vTC, no matter VM or CN, the encoder runs entirely on CPU; for the accelerated vTC, however, the encoder offloads its compute-intensive processes to GPU. The final component is the HTTP Live Streaming (HLS) Muxer that includes the output of encoder into the Moving Picture Experts Group (MPEG) transport stream to be transmitted over the network. Like Testsrc, HLS Muxer also runs only on CPU in all vTC versions.

In the test-bed, I also used a virtual machine as a vTC user. This user sends requests to the vTC for different video resolutions and bitrates, which are the main metrics used by version selector in MVSM to switch between different versions. These two metrics have been chosen based on the results in Section 6.3: the accelerator-based vTC not only processes high resolution/bitrate videos faster than COTS-based vTC but also cheaper as it requires fewer resources. On the other hand, the CPU-based vTC is not only cheaper to process the resolu-

tion/bitrate videos but, in this case, also faster compared to an accelerator-based vTC. In this evaluation, I measured the time required to switch from one version to another. Having three versions of vTC, we had 6 cases of switching time, each of which was measured a hundred times. The switching time is the time between initiating the switching request in the version selector and receiving the switching response by the message handler in MVSM. This time includes the deployment of the new version, starting monitoring the new version, and termination of the old version.

Fig. 6.13 shows the Cumulative Distribution Function (CDF) of the time to switch from VMC to CNA, CNC to CNA, CNA to CNC, and VMC to CNC versions of vTC. These results show that the switching to CN-based VNFs takes mostly between 3.2 and 4 seconds. The switching time from VM-based to CN-based vTCs is longer than CN-based to CN-based vTCs. This is because the termination of VM-based vTCs takes longer than CN-based vTC.

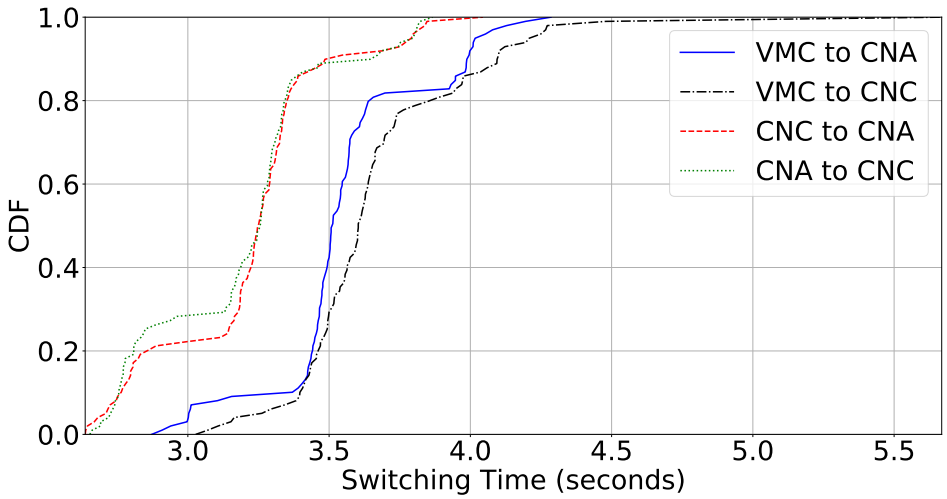


Figure 6.13: The time required to switch between CNA to CNA, CNA to CNC, and VMC to CNA versions of the virtual transcoder

Fig. 6.14 shows the CDF of switching time from CNA to VMC and CNC to VMC versions of vTC. These results show that the switching to VM-based VNFs takes mostly between 64 to 100 seconds, which is much longer than switching to CN-based vTCs. Most of the VM switching time is consumed for the deployment of the VMs in OpenStack, which is significantly higher compared to the deployment of CNs in Kubernetes. From Fig. 6.13 and 6.14, we observe that the deployment time of CNAs is slightly shorter than CNCs. Breaking down the results, I noticed that the difference comes from the time required by Kubernetes to deploy CNAs and CNCs. Surprisingly, Kubernetes handles the deployment of CNAs faster than CNCs.

Another action involved in version switching is the redirection of the traffic to the new version. To this end, I have measured the time required by Pishahang to generate a new forwarding graph and redirect vTC incoming traffic to the new

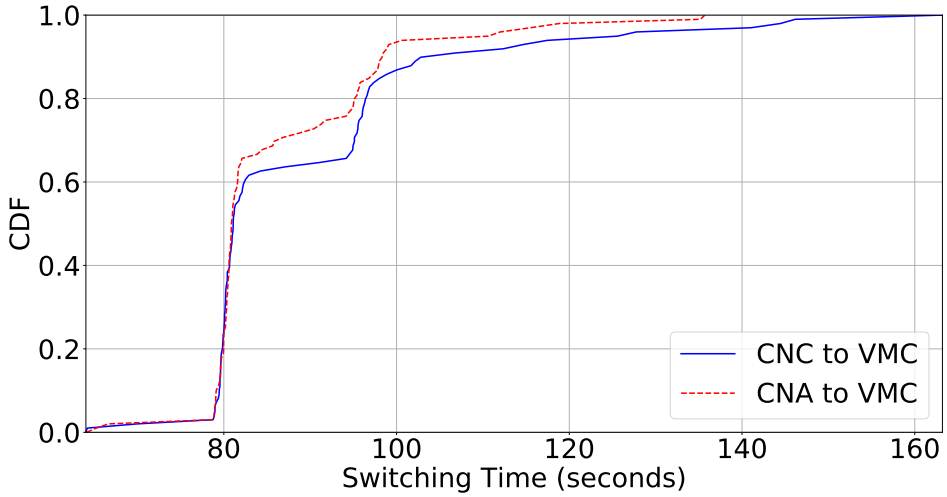


Figure 6.14: The time required to switch between CNA to VMC and CNC to VMC versions of the virtual transcoder

version. Fig. 6.15 illustrates the results of this evaluation. From these results, we observe that the packet redirection time also plays an important role in the case of switching to CN-based vTCs, which should not be overlooked, as it increases the switching time by up to 0.49%.

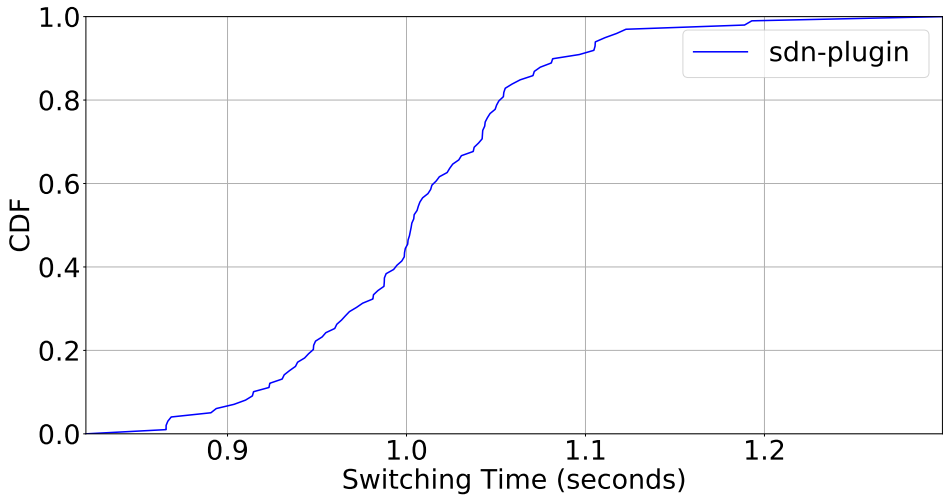


Figure 6.15: The time required by the Service Function Chaining Management (SFCM) to generate a new forwarding graph for redirecting the traffic to the new version

6.5 Related Work

Studies in the context of dynamic service provisioning and multi-version services can be categorised into (i) studies related to evaluating VNFs realised with different implementation options and (ii) studies related to analysing the dynamic deployment of multi-version services. The former category supports the idea of the multi-version services as it can prove that there are trade-offs between using different implementation options of VNFs. The latter category analyses the overhead generated by dynamically service provisioning, which is more associated with the management and orchestration of these services.

There are multiple studies in the first category. For example, Araújo et al. [5] studied the use of GPUs to assist packet processing in DPIs. To this end, they implemented and evaluated a CPU-based and a GPU-assisted DPI. Both DPIs are based on the Snort Intrusion Detection System (IDS)[12], with the GPU-assisted DPI adapted to execute on a GPU. They evaluated the DPIs based on metrics such as processing time, resource utilisation. The results presented in this study shows that GPU-assisted DPI performs well in high-traffic conditions and processes the packets up to 19 times faster than CPU-based DPI. On the other hand, CPU-based DPI has significantly less overall processing time in low-traffic conditions. This shows that dynamically using these two versions based on the traffic condition can improve the performance of DPIs. What is missing in this study is a cost analysis which is an important factor when employing acceleration resources.

In another work, Han et al. proposed PacketShader[18], a framework that enables offloading computation and memory-intensive operations to GPUs. In this study, a softwarised GPU-assisted OpenFlow switch has been implemented and tested against a softwarised COTS-based switch to quantify the gain of using GPU as an accelerator for packet processing. The OpenFlow switch implemented in this study captures the packets and extracts the flow keys from the packet header and matches them against the exact-match entries in the hash table. The COTS-based OpenFlow switch runs all tasks including the flow key extraction, hash value calculation and lookup for exact entries, linear search for wildcard matching and follow-up actions in CPU. In the GPU-assisted switch, however, hash value calculation and the wildcard matching tasks are offloaded to a GPU. For the evaluation, the throughput of switches was measured with the flow table size as a parameter. The results show that a GPU-assisted OpenFlow switch can have up to 10 times better throughput than the COTS-based one. Their evaluation results also show that the highest throughput improvement is achieved when the number of flow entries is high (1 M of exact entries and 1 k of wildcard entries), but in the opposite situation, when the number of flow entries is low, the throughput improvement is negligible. This indicates that, in the case of low number of flow entries, COTS-based switch can keep up well and provide cheaper switches than the GPU-assisted switch. Like [5], this work also lacks quantifying the cost of using these two implementation options based on different workload.

Nobach et al. [36] studied the performance of COTS-based DPI versus FPGA-

[12] `https://www.snort.org/` accessed, May 20, 2019

based DPI. In this work, they implemented a DPI that uses COTS resources for all its tasks and another one that offloads network-intensive tasks onto FPGA resources. They have evaluated the throughput and latency of these DPIs having the size of input packets as a parameter. The results of their evaluation show that FPGA-based DPI can improve the throughput up to 124 times for packet sizes up to 512 bytes. Also, it can improve the latency up to 6 times. However, in the case that packet sizes are bigger than 512 bytes, the performance improvement becomes negligible. With that, we can observe that the use of FPGA-based DPI is not a proper solution for large packets as it costs more than COTS-based DPI without having a significant performance improvement. Therefore, dynamical employment of these two implementation options based on the packet size can balance the cost and performance of DPIs. This paper also provides cost analysis; however, it is based on some assumptions and does not represent real-world cost ranges.

For the second category, however, there have not been many studies. Dräxler et al., [11] have evaluated dynamically deployment of multi-version services using simulation. While the results presented in the study validates the idea of multi-version services in theory, an experimental evaluation of such a service provisioning approach is still missing.

This chapter extends the literature on the first category by quantifying the performance and cost of deploying network function over heterogeneous resources and the second category by an experimental analysis on the management overhead of multi-version services.

About the orchestration framework, as mentioned in Section 5.4, multiple MANO frameworks support the orchestration of heterogeneous resources; however, none of them supports the orchestration of multi-version services.

6.6 Conclusion

In this chapter, we analysed the dynamic provisioning of NFV services over heterogeneous resources. To this end, we realised services in multiple versions based on hardware and virtualisation technologies and called them multi-version services. By alternating between different versions on the fly based on the service requirements (e.g., data rate), we can trade-off different metrics such as cost and performance.

To validate the concept of multi-version services, we quantified the trade-off between using different versions of an example service by experimental evaluation. In this evaluation, we used vTC as a case study and quantified the resource utilisation and processing time of COTS-based and GPU-assisted vTCs. Using the resource utilisation data, we calculated the cost of running vTCs in AWS cloud infrastructure. By this evaluation, we observe that using GPUs can accelerate the performance of vTC for high-quality videos while low-quality videos can be handled better by COTS-based vTCs and be cheaper at the same time.

To support the orchestration of multi-version services, we extended the Pishahang MANO framework with a new plugin called MVSM. This plugin orchestrates multi-version services based on a reactive approach and handles tasks such as de-

ploying multi-version services, identifying the need to switch to a different version, and switching to the right version of the service on the fly. Using this extension, we analysed the management overhead due to such dynamic usage of accelerators. From the results, we observed that the switching time to CN-based versions is significantly less than the switching time to VM-based versions. Thus, the virtualisation environment type can have a great impact on the management overhead of multi-version services.

Overall, the results presented in this chapter validates that multi-version services are beneficial in practice as they can provide performant services while guaranteeing the lowest price. Using CNs as virtualisation environments makes the multi-version services significantly more agile to respond to changes in service requirements (e.g., data rate), compared to using VMs as virtualisation environments. However, this does not rule out employing VMs in multi-version service. This problem can be mitigated by using a proactive approach for switching to different versions and starting the deployment of VM-based versions ahead of time to have them ready to serve at the time they are needed.

7

Final Thoughts

In this final chapter, I, first, summarise the key contributions of my thesis and, then, draw the conclusions. Finally, I end my thesis with future research directions.

7.1 Summary

In this thesis, I investigated different challenges concerning Network Function Virtualization (NFV) Management and Orchestration (MANO) frameworks. First, in Chapter 3, I introduced the concept of Specific Management (SM). SM improves the flexibility of MANO frameworks and allows the service developers to program the management and orchestration of their services. I implemented the SM platform and integrated it into the SONATA MANO framework. Using a simulation-based approach, I evaluated the SM concept by comparing it with rigid solutions. Also, by experimental evaluation, I analysed the management overhead caused by employing the SM concept in a MANO framework.

In Chapter 4, I investigated the scalability and agility of MANO frameworks. To this end, I introduced a benchmarking framework for MANO frameworks. This framework is used to quantify the resource consumption and deployment time of two MANO frameworks, namely Pishahang and Open Source MANO (OSM). Furthermore, using the benchmarking framework, I quantified the effect of topological distance in service deployment time.

In Chapter 5, I addressed the challenge of supporting heterogeneous resources in the NFV MANO framework. To this end, I implemented an open-source multi-domain MANO, called Pishahang. Pishahang supports heterogeneous resources by a multi-domain approach where resources from multiple domains are jointly orchestrated. In this chapter, I described the design and implementation of Pisha-

hang and also explained how it was used in the 5G-PICTURE project to support 5G services.

I also investigated the dynamic deployment of NFV services over heterogeneous resources. In this regard, I introduced multi-version services and classified the services that can benefit from dynamic deployment. I analysed an example of multi-version service in terms of performance and cost. Also, I extended the Pishahang MANO framework to support multi-version services. Using the extension, I analysed the management overhead that multi-version services impose on the MANO framework.

7.2 Conclusion

The work presented in Chapter 3 is the first effort at improving the programmability of MANO frameworks. The concept is not specific to MANO frameworks and can be used for other applications that need to be customised on the fly. The SM platform that I implemented for SONATA MANO framework has been employed to manage NFV services in the SONATA [1] project's final demonstration in Gent, Belgium. The SM platform has also been used and extended in the 5G-Tango [2] project, which shows the platform has been accepted by the community.

The results presented in Chapter 4 can already provide MANO frameworks that are fully scalable. Both Pishahang and OSM realise their functionalities in individual containers. Being container-based, they can run in a Kubernetes cluster which has native scaling solutions for containers. To scale out/in the containers, Kubernetes needs the resource requirement of the containers. My work provided this data. Assuming that there is no coordination overhead between containers, by running Pishahang and OSM under Kubernetes, we can make sure the MANO framework is fully scalable and can keep up with any sudden increase of load.

Existing NFV orchestrators were limited in supporting heterogeneous resources. At the time of Pishahang's first release in Jan. 2018, there were no other NFV MANO frameworks supporting the orchestration of container-based Virtualized Network Function (VNF)s. Pishahang is not only the first NFV MANO that supports container-based VNFs, but also the only NFV MANO that supports Field Programmable Gate Array (FPGA)-based VNFs. Also, Pishahang has been widely used in the 5G-PICTURE project as main NFV MANO, which shows that it has been accepted by the community.

Finally, I took multi-version services from theory to practice. While there had been some theoretical work on multi-version services [11], I evaluated the concept in practice and implemented mechanisms to manage and orchestrate them. I not only published research papers out of this work [24] [26], but I also demonstrated the multi-version services in the 5G-PICTURE review meeting in Barcelona.

[1] `https://project.sonata-nfv.eu/`, accessed July 2020
[2] `https://www.5gtango.eu/`, accessed July 2020

7.3 Future research

Specific Managements (SMs) aggregation: As the results in Chapter 3 show, SMs impose extra overhead on the MANO framework in terms of resource usage and service deployment time. The overhead increases as the number of containers used to run SMs increases. To mitigate this issue, multiple SMs can be aggregated to run in a single container, instead of running each SM on a different container. This would decrease resource consumption and also the deployment time of the services. On the other hand, this solution might increase the complexity of the management of SMs and degrades dependability of individual SMs. As future work, such solution can be investigated.

Coordination overhead analysis: The study presented in Chapter 4 assumes that there is no coordination overhead in scaling the MANO microservices. This, however, may not be true. There may be extra overheads in adding/removing new replicas of a microservice as the load needs to be constantly distributed among multiple replicas. An investigation of such coordination overhead of MANO microservices can complete the last puzzle in the scalability of MANO frameworks and allow to design fully scalable MANO frameworks.

Intra-domain service chaining: Although Pishahang provides inter-domain service chaining, intra-domain chaining is not fully supported. For example, in a Kubernetes domain, although Pods can chain with VNFs running outside of their cluster, they cannot be chained with other Pods inside the Kubernetes cluster as required for NFV services. A possible solution for this issue is to integrate Network Service Mesh (NSM) [3] into Pishahang. NSM provides service function chaining for containers running in Kubernetes cluster.

Proactive solution for on-the-fly version switching: The solution provided in this thesis uses a reactive approach to identify the need for switching to a different version, which is slow to react to changes compared to a proactive solution. A proactive approach provides a better solution as it allows the right version to be up exactly when it is needed, not with some delay like in a reactive solution. Such proactive solution can be realised using a machine learning algorithm to predict the service load.

[3] `https://networkservicemesh.io/`, accessed July 2020

Acronyms

IT	Information Technology
MANO	Management and Orchestration
NFV	Network Function Virtualization
VNF	Virtualized Network Function
SM	Specific Management
SSM	Service-Specific Management
FSM	Function-Specific Management
COTS	Commercial Off-The-Shelf
RPM	Requests Per Minutes
VIM	Virtualized Infrastructure Manager
ETSI	European Telecommunication Standards Institute
TOSCA	Topology and Orchestration Specification for Cloud Applications
OSM	Open Source MANO
YAML	YAML Ain't Markup Language
ONAP	Open Network Automation Platform
KPI	Key Performance Indicator
PoP	Point of Presence
IoT	Internet of Things
M2M	Machine to Machine Communication
SMS	Short Message Service
NSM	Network Service Mesh
GK	Gate Keeper
FLM	Function Lifecycle Management

SLM	Service Lifecycle Management
PLM	Placement Management
SCM	Scaling Management
IA	Infrastructure Adaptor
VM	Virtual Machine
NSD	Network Service Descriptor
VNFD	Virtual Network Function Descriptor
WAN	Wide Area Network
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
SMR	Specific Manager Registry
NFVI	Network Function Virtual Infrastructure
CLI	Command Line Interface
GUI	Graphical User Interface
PNF	Physical Network Function
URI	Uniform Resource Identifier
UUID	Universal Unique Identifier
AWS	Amazon Web Service
EKS	Elastic Kubernetes Service
CMS	Cloud Management System
GPP	General Purpose Processor
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
CN	Container
GCP	Google Cloud Platform
OLM	OpenStack Lifecycle Management
KLM	Kubernetes Lifecycle Management
ALM	AWS Lifecycle Management

VDU	Virtual Deployment Unit
OS	Operating System
E2E	End To End
BSS	Business Support System
OSS	Operations Support Systems
RAN	Radio Access Network
MDO	Multi Domain Orchestrator
UPB	Paderborn University
WiFi	Wireless Fidelity
OAI	OpenAirInterface
DHCP	Dynamic Host Configuration Protocol
CDF	Cumulative Distribution Function
SFCM	Service Function Chaining Management
CP	Connection Point
VLAN	Virtual LAN
vTC	virtual Transcoder
NF	Network Function
MVS	Multi-version Service
MVNS	Multi-version Network Service
MVNF	Multi-version Network Function
MVSM	Multi-version Service Management
NAT	Network Address Translation
NS	Network Service
IDS	Intrusion Detection System
VOD	Video On Demand
GOP	Group of Pictures
RTP	Real-Time Transport Protocol
VMC	VM-based COTS

CNC	CN-based COTS
CNA	CN-based Accelerated
MPEG	Moving Picture Experts Group
SFC	Service Function Chaining
VPN	Virtual Private Network
VL	Virtual Link
NFP	Network Function Path
VNF-FG	VNF Forwarding Graph
ONOS	Open Network Operating System
VNFM	VNF Manager
NFVO	NFV Orchestrator
NIC	Network Interface Card
vNIC	Virtual Network Interface Card
HOT	Heat Orchestration Template
OSS/BSS	Operations/Business Support System
EM	Element Manager
VLD	Virtual Link Descriptor
VNFFGD	VNF Forwarding Graph Descriptor
API	Application Programming Interface
DPI	Deep Packet Inspection
HLS	HTTP Live Streaming
LTE	Long Term Evolution
QoE	Quality of Experience
SDN	Software-Defined Network
URL	Uniform Resource Locator

Bibliography

[1] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati. Network Function Virtualization in 5G. *IEEE Communications Magazine*, 54(4):84–91, 2016.

[2] M. Abu-Lebdeh, D. Naboulsi, R. Glitho, and C. Tchouati. On the placement of vnf managers in large-scale and distributed nfv systems. *IEEE Transactions on Network and Service Management*, 14(4):875–889, 2017.

[3] M. Abu-Lebdeh, D. Naboulsi, R. Glitho, and C. W. Tchouati. NFV Orchestrator Placement for Geo-distributed Systems. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–5. IEEE, 2017.

[4] P. K. Agyapong, M. Iwamura, D. Staehle, W. Kiess, and A. Benjebbour. Design Considerations for a 5G Network Architecture. *IEEE Communications Magazine*, 52(11):65–75, 2014.

[5] I. M. Araújo, C. Natalino, Á. Santana, and D. L. Cardoso. Accelerating vnf-based deep packet inspection with the use of gpus. In *2018 20th International Conference on Transparent Optical Networks (ICTON)*, pages 1–4. IEEE, 2018.

[6] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose. Performance Analysis of Virtual Machines and Containers in Cloud Computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1204–1210, 2016.

[7] J. Bonnet and et al. D4. 3. service platform first operational release and documentation. *SONATA Project Deliverable*, 7, 2017.

[8] D. Camps-Mur, F. Canellas, A. Machwe, J. Paracuellos, K. Choumas, D. Giatsios, T. Korakis, and H. R. Kouchaksaraei. 5GOS: Demonstrating Multi-domain Orchestration of End-to-end Virtual RAN Services. In *2020 6th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2020.

[9] H. Chen and F. J. Lin. Scalable iot/m2m platforms based on kubernetes-enabled nfv mano architecture. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1106–1111, 2019.

[10] Tobias Dierich. Joint Orchestration of Network Function Chains and General-Purpose Cloud Services. *Bachelor thesis*, 2018.

[11] S. Dräxler and H. Karl. Spring: Scaling, placement, and routing of heterogeneous services with flexible structures. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 115–123, June 2019.

[12] S. Dräxler, H. Karl, H. R. Kouchaksaraei, A. Machwe, C. Dent-Young, K. Katsalis, and K. Samdanis. 5G OS: Control and Orchestration of Services on Multi-Domain Heterogeneous 5G Infrastructures. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9, June 2018.

[13] S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris. SONATA: Service Programming and Orchestration for Virtualized Software Networks. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 973–978, May 2017.

[14] Rob Enns. NETCONF Configuration Protocol. RFC 4741, December 2006.

[15] ETSI NFV ISG. Network Functions Virtualisation (NFV): Architectural Framework. Group Specification ETSI GS NFV-MAN 001 V1.1.1, ETSI, Oct. 2013.

[16] N. Ghrada, M. F. Zhani, and Y. Elkhatib. Price and performance of cloud-hosted virtual network functions: Analysis and future challenges. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 482–487, June 2018.

[17] A. S. Gowri and P. Shanthi Bala. Impact of Virtualization Technologies in the Development and Management of Cloud Applications. *International Journal of Intelligent Systems and Applications in Engineering*, 7(2):104–110, 2019.

[18] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[19] A. Kojukhov and et al. Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; VNF Package Specification. GS NFV-SOL 004 V2. 3.1. In *ETSI, Group Specification*, 2017.

[20] C. G. Kominos, N. Seyvet, and K. Vandikas. Bare-metal, virtual machines and containers in openstack. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 36–43, 2017.

[21] H. R. Kouchaksaraei, T. Dierich, and H. Karl. Pishahang: Joint Orchestration of Network Function Chains and Distributed Cloud Applications. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 344–346, June 2018.

[22] H. R. Kouchaksaraei, S. Dräxler, M. Peuster, and H. Karl. Programmable and Flexible Management and Orchestration of Virtualized Network Functions. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–9, June 2018.

[23] H. R. Kouchaksaraei and Holger Karl. Joint Orchestration of Cloud-Based Microservices and Virtual Network Functions. In *The Ninth International Conference on Cloud Computing, GRIDs, and Virtualization CLOUD COMPUTING*, pages 153–154, February 2018.

[24] H. R. Kouchaksaraei and Holger Karl. Quantitative Analysis of Dynamically Provisioned Heterogeneous Network Services. In *Proceedings of the 15th International Conference on Network and Service Management*, CNSM '19. IFIP, 2019.

[25] H. R. Kouchaksaraei and Holger Karl. Service Function Chaining Across OpenStack and Kubernetes Domains. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, DEBS '19, pages 240–243, New York, NY, USA, 2019. ACM.

[26] H. R. Kouchaksaraei, A. P. S. Venkatesh, A. Churi, M. Illian, and H. Karl. Dynamic Provisioning of Network Services on Heterogeneous Resources. In *2020 European Conference on Networks and Communications (EuCNC)*, June 2020.

[27] S. Kukliński and et al. D4.1 – scalability-driven management system. *5G!PAGODA Project Deliverable*, 2018.

[28] X. Li, Mohsen A. Salehi, Y. Joshi, M. Darwich, B. Landreneau, and M. Bayoumi. Performance Analysis and Modeling of Video Transcoding Using Heterogeneous Cloud Services. *CoRR*, abs/1809.06529, 2018.

[29] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962, 2017.

[30] W Liu, H Li, O Huang, M Boucadair, N Leymann, Q Fu, Q Sun, C Pham, C Huang, J Zhu, et al. Service function chaining (sfc) general use cases. *Internet Request for Comments (RFC) Working Draft, IETF Secretariat, Internet-Draft draft-liu-sfc-use-cases-08*, 2014.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[32] S. Mehraghdam, M. Keller, and H. Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 7–13, Oct 2014.

[33] Dennis Meier. Management and Orchestration of FPGA-based network functions. *Bachelor thesis*, 2020.

[34] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[35] L. Nobach and D. Hausheer. Open, elastic provisioning of hardware acceleration in nfv environments. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, pages 1–5, March 2015.

[36] L. Nobach, B. Rudolph, and D. Hausheer. Benefits of Conditional FPGA Provisioning for Virtualized Network Functions. In *2017 International Conference on Networked Systems (NetSys)*, pages 1–6, March 2017.

[37] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. Sndlib 1.0—survivable network design library. *Networks: An International Journal*, 55(3):276–286, 2010.

[38] M. Peuster, S. Dräxler, H. R. Kouchaksaraei, S. v. Rossem, W. Tavernier, and H. Karl. A Flexible Multi-pop Infrastructure Emulator for Carrier-grade MANO Systems. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–3, July 2017.

[39] M. Peuster, H. Karl, and S. van Rossem. MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153, 2016.

[40] M. Peuster, M. Marchetti, de G. Blas, and H. Karl. Automated testing of nfv orchestrators against carrier-grade multi-pop scenarios using emulation-based smoke testing. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):172, 2019.

[41] S. Van Rossem, X. Cai, I. Cerrato, P. Danielsson, F. Németh, B. Pechenot, I. Pelle, F. Risso, S .Sharma, and P. Sköldström. NFV Service Dynamicity with a DevOps Approach: Insights from a Use-case Realization. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 674–679. IEEE, 2017.

[42] S. Van Rossem, W. Tavernier, B. Sonkoly, D. Colle, J. Czentye, M. Pickavet, and P. Demeester. Deploying Elastic Routing Capability in an SDN/NFV-enabled Environment. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 22–24. IEEE, 2015.

[43] S. Sahhaf, W. Tavernier, J. Czentye, B. Sonkoly, P. Sköldström, D. Jocha, and J. Garay. Scalable Architecture for Service Function Chain Orchestration. In *2015 Fourth European Workshop on Software Defined Networks*, pages 19–24. IEEE, 2015.

[44] V. Shamugam, L. Murray, J. Leong, and A. S. Sidhu. Software defined networking challenges and future direction: A case study of implementing sdn features on openstack private cloud. In *IOP Conference Series: Materials Science and Engineering*, volume 121, page 012003, 2016.

[45] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*, page 1. ACM, 2016.

[46] T. Shimizu, A. Nakao, and K. Satoh. Network Softwarization View of 5 G Networks. *5G Networks: Fundamental Requirements, Enabling Technologies, and Operations Management*, pages 499–518, 2018.

[47] T. Soenen, S. Sahhaf, W. Tavernier, P. Sköldström, D. Colle, and M. Pickavet. A Model to Select the Right Infrastructure Abstraction for Service Function Chaining. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 233–239. IEEE, 2016.

[48] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso. Multi-domain service orchestration over networks and clouds: A unified approach. *ACM SIGCOMM Computer Communication Review*, 45(4):377–378, 2015.

[49] B. Sonkoly, R. Szabo, D. Jocha, J. Czentye, M. Kind, and F. Westphal. UNIFYing Cloud and Carrier Network Resources: An Architectural View. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2015.

[50] DRVLB Thambawita, Roshan Ragel, and Dhammika Elkaduwe. To use or not to use: Graphics processing units (gpus) for pattern matching algorithms. In *7th International Conference on Information and Automation for Sustainability*, pages 1–4. IEEE, 2014.

[51] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.

[52] TOSCA. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. Technical report, 2013.

[53] S. Van Rossem, M. Peuster, L. Conceicao, H. R. Kouchaksaraei, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester. A Network Service Development Kit Supporting the End-to-end Lifecycle of NFV-based Telecom Services. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–2, Nov 2017.

[54] G. Yilma, F. Yousaf, V. Sciancalepore, and X. Costa-Perez. On the challenges and kpis for benchmarking open-source nfv mano systems: Osm vs onap. *arXiv preprint arXiv:1904.10697*, 2019.