Cloud Native Network Slice Orchestration in 5G and Beyond

Sagar Arora

Cloud Native Network Slice Orchestration in 5G and Beyond

Dissertation

submitted to

Sorbonne Université

in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Author:

Sagar Arora

Scheduled for defense on the 16th of October 2023, before a committee composed of:

Reviewers

Prof.	**Thierry Turletti**	INRIA, Sophia Antipolis, France
Prof.	**Lyes Khoukhi**	ENSICAEN, France

Jury President

Prof.	**Raymond Knopp**	EURECOM, France

Examiners

Dr.	**Amina Boubendir**	Airbus, France
Dr.	**Pantelis Frangoudis**	TU WIEN, Austria

Thesis Advisors

Prof.	**Adlen Ksentini**	EURECOM, France
Prof.	**Christian Bonnet**	EURECOM, France

Orchestration de tranches de réseau cloud natives dans la 5G et au-delà

Thèse

soumise à

Sorbonne Université

pour l'obtention du Grade de Docteur

présentée par:

Sagar Arora

Soutenance de thèse prévue le 16 Octobre 2023 devant le jury composé de:

Rapporteurs

Prof.	**Thierry Turletti**	INRIA, Sophia Antipolis, France
Prof.	**Lyes Khoukhi**	ENSICAEN, France

Président du Jury

Prof.	**Raymond Knopp**	EURECOM, France

Examinateurs

Dr.	**Amina Boubendir**	Airbus, France
Dr.	**Pantelis Frangoudis**	TU WIEN, Austria

Directeurs de thèse

Prof.	**Adlen Ksentini**	EURECOM, France
Prof.	**Christian Bonnet**	EURECOM, France

ABSTRACT

Network Function Virtualization (NFV) is the founding pillar of 5G Service Based Architecture. It has the potential to revolutionize the future mobile communication generations. NFV started long back in 2012 with Virtual-Machine (VM) based Virtual Network Functions (VNFs). The use of VMs raised multiple questions because of the compatibility issues between VM hypervisors and their high resource consumption. This made containers to be an alternative network function packaging technology. The lightweight design of containers improves their instantiation time and resource footprints. Apart from network functions, containerization can be a promising enabler for Multi-access Edge Computing (MEC) applications that provides a home to low-latency demanding services. Edge computing is one of the key technology of the last decade, enabling several emerging services beyond 5G (e.g., autonomous driving, robotic networks, Augmented Reality (AR)) requiring high availability and low latency communications. The resource scarcity at the edge of the network requires technologies that efficiently utilize computational, storage, and networking resources. Containers' low-resource footprints make them suitable for designing MEC applications.

Containerization is meant to be used in the framework of cloud-native application design fundamentals, loosely coupled microservices-based architecture, on-demand scalability, and high resilience. The flexibility and agility of containers can certainly benefit 5G Network Slicing that highly relies on NFV and MEC. The concept of Network slicing allows the creation of isolated logical networks on top of the same physical network. A network slice can have dedicated network functions or its network functions can be shared among multiple slices. Indeed, network slice orchestration requires interaction with multiple technological domain orchestrators, access, transport, core network, and edge computing. The paradigm shift of using cloud-native application design principles has created challenges for legacy orchestration systems and the ETSI NFV and MEC standards. They were designed for handling virtual machine-based network functions, restricting them in their approach to managing a cloud-native network function.

The thesis examines the existing standards of ETSI NFV, ETSI MEC, and network service/slice orchestrators. Aiming to overcome the challenges around multi-domain cloud-native network slice orchestration. To reach the goal, the thesis first proposes MEC Radio Network Information Service (RNIS) that can provide radio information at the subscriber level in an NFV environment. Second, it provides a Dynamic Resource Allocation and Placement (DRAP) algorithm to place cloud-native network services considering their cost and availability matrix. Third, by combining NFV, MEC, and Network Slicing, the thesis proposes a novel Lightweight edge Slice Orchestration (LeSO) framework to overcome the challenges around edge slice orchestration. Fourth, the proposed framework offers an edge slice deployment template that allows multiple possibilities for designing MEC applications. These possibilities were further studied to understand the impact of the microservice design architecture on application availability and latency.

Finally, all this work is combined to propose a novel Cloud-native Lightweight Slice Orchestration (CLiSO) framework extending the previously proposed LeSO framework. In

addition, the framework offers a technology-agnostic and deployment-oriented network slice template. The framework has been thoroughly evaluated via orchestrating OpenAirInterface container network functions on public and private cloud platforms. The experimental results show that the framework has lower resource footprints than existing orchestrators and takes less time to orchestrate network slices.

ABRÉGÉ

La virtualisation des fonctions réseau (NFV) est le pilier fondateur de l'architecture 5G basée sur les services. Elle a le potentiel de révolutionner les futures générations de communications mobiles. La NFV a débuté il y a longtemps, en 2012, avec les fonctions de réseau virtuelles (VNF) basées sur les machines virtuelles (VM). L'utilisation des machines virtuelles a soulevé de nombreuses questions en raison des problèmes de compatibilité entre les hyperviseurs de machines virtuelles et de leur consommation élevée de ressources. Les conteneurs sont donc devenus une technologie alternative de conditionnement intéressante pour la virtualisation des fonctions réseau. La technologie des conteneurs est légère en termes de consommation de ressources ce qui améliore leur temps d'instanciation. Outre les fonctions de réseau, la conteneurisation peut être un outil prometteur pour les applications multi-access edge computing (MEC) qui abritent des services exigeants à faible latence. MEC permet l'émergence de calcul à la périphérie du réseau c'est une des technologies clés de la dernière décennie,au-delà de la 5G (par exemple, la conduite autonome, les réseaux robotiques, la réalité augmentée (AR)), nécessitant une haute disponibilité et des communications à faible latence. La rareté des ressources à la périphérie du réseau exige des technologies qui utilisent efficacement les ressources de calcul, de stockage et de mise en réseau. La faible empreinte des conteneurs sur les ressources les rend adaptés à la conception d'applications MEC.

La conteneurisation est censée être utilisée dans le cadre des principes fondamentaux de la conception d'applications cloud-native, une architecture basée sur des microservices à couplage lâche, d'une évolutivité à la demande et d'une résilience élevée. La flexibilité et l'agilité des conteneurs peuvent certainement profiter au découpage du réseau 5G en tranches,ces derniers reposent fortement sur NFV et MEC. Le concept de découpage du réseau permet de créer des réseaux logiques isolés au-dessus du même réseau physique. Une tranche de réseau peut avoir des fonctions de réseau dédiées pouvant être partagées entre plusieurs tranches. En effet, l'orchestration des tranches de réseau nécessite une interaction avec de multiples orchestrateurs de domaines technologiques: l'accès radio, le transport, le réseau central et l'informatique périphérique. Le changement de paradigme consistant à utiliser des principes de conception d'applications cloud-natives a créé des défis pour les systèmes d'orchestration existants et les normes NFV et MEC de l'ETSI. Ces derniers ont été conçus pour gérer des fonctions de réseau basées sur des machines virtuelles. Ils sont donc limités dans leur approche de la gestion d'une fonction de réseau cloud-native.

Par le présent manuscrit, nous examinons les normes existantes de l'ETSI NFV, de l'ETSI MEC et des orchestrateurs de services/tranches de réseau,et nous proposons de résoudre les défis liés à l'orchestration de tranches de réseau multi-domaines cloud-native. Pour atteindre cet objectif, nous proposons tout d'abord un service d'information sur le réseau radio (RNIS) MEC qui a la capacité de fournir des informations radio au niveau de l'abonné dans un environnement NFV. Deuxièmement, nous fournissons un algorithme d'allocation et de placement dynamique des ressources (DRAP) pour placer les services réseau cloud-native en tenant compte de leur matrice de coût et de disponibilité. Troisièmement, en combinant NFV, MEC et Network Slicing, nous proposons un

nouveau mécanisme d'orchestration de tranches MEC (LeSO) pour surmonter les défis liés à l'orchestration de tranches MEC. Quatrièmement, le mecanisme proposé offre un modèle de déploiement de tranches de réseau qui permet de multiples possibilités de conception d'applications MEC. Ces possibilités ont été étudiées plus en détails pour comprendre l'impact de l'architecture de conception microservice sur la disponibilité et la latence de l'application.

Enfin, tous ces travaux sont combinés pour proposer une nouvelle approche d'orchestration de tranches légères Cloud-native (CLiSO) étendant le précédant mécanisme d'orchestration de tranches légères de bord (LeSO). En outre, cette nouvelle approche offre un modèle de tranche de réseau agnostique sur le plan technologique et orienté déploiement. La solution a été évaluée de manière approfondie en orchestrant les fonctions réseau du conteneur OpenAirInterface sur des plates-formes de cloud public et privé. Les résultats expérimentaux montrent que la solution proposée a des empreintes de ressources plus faibles que les orchestrateurs existants et prend moins de temps pour orchestrer les tranches de réseau.

Acknowledgements

Education for me was always tough. I was never good at studies or hardworking. I was lazy in my childhood and had no ambitions. I never wanted to go to school or study. I started excelling in my studies when I was 15 years old. Since then I have never stopped studying and working hard towards my dreams. What I am today is because of my mother and father. They provided me with everything I wanted and did the best they could do for me. I have immense gratitude for my parents. My brother played an equally important role in my life. I am blessed to have such a caring family.

Since my master's I am working with my thesis advisors Adlen Ksentini and Christian Bonnet. I have learned so much from their experience that I would have never been able to learn myself. Long discussions and brainstorming sessions with Adlen nurtured my technical ideas and without his trust and support, I would have never finished this journey. I sincerely thank my thesis advisors.

I am thankful to Irfan Gauri and Raymond Knopp for showing their trust in me and giving me the opportunity to contribute to the OpenAirInterface code base.

I started my Ph.D. in October 2019 and in March 2020 there was a lockdown in France due to the COVID pandemic. At that time I was living alone, running on my balcony, talking to my friends and family on the phone or with the cashier in the supermarkets like most people in the world. After the lockdown, I moved into a colocation with Jonas, Marine, and Julien for two years. If they would have not been there I would have never survived the lockdown.

During my Ph.D. journey, I fell in love with Marie. She always kept me motivated to finish my Ph.D. I am thankful to have her love and support in my life. Thank you so much for being with me.

I would like to thank all my lovely friends for being a part of my life. Jose, Emanuele, Lorenzo, Placido, Tryfonas, Alison, Thomas, Lara, Alberto, Raquel, Ali, Marina, Robert and Berkay. The list will never be completed without mentioning my life's constants Sagar, Anshal, and Jaivardhan.

Kanika is the first person in my life who helped me understand my life challenges and gave me a hand to overcome my problems. I am grateful to her. I would also like to thank the Soka Gakkai organization.

Last but not least, I would like to thank all my colleagues at Eurecom and OpenAir-Interface. Karim, Tien-Thinh, Emmanuel, Jean-Christophe, Michaël, Valerie, Tom, Camille, Rohan and Raphael.

Thank you all!

Contents

ABSTRACT . . . . . **ii**

ABRÉGÉ . . . . . **iv**

Acknowledgements . . . . . **vi**

Table of Contents . . . . . **x**

List of Figures . . . . . **xi**

List of Tables . . . . . **xiii**

Acronyms . . . . . **xiv**

Glossary . . . . . **xvii**

1.Introduction **1**
1.1 The 5G Journey 1
1.2 Network Slicing, MEC and NFV 1
1.3 Multi-domain Network Slicing 3
1.4 Challenges in Orchestrating a Multi-domain Network Slice 4
1.4.1 Challenges related to Availability 5
1.5 Thesis Motivation 5
1.6 Thesis Outline 6

2. Background **10**
2.1 Network Function Virtualization 10
2.1.1 Virtualization and Containerization 11
2.1.2 Cloud Native Network Functions 12
2.1.3 ETSI NFV Management and Orchestration 13
2.1.4 Kubernetes: CISM 14
2.2 3GPP 5G Core Network SBA and Network Slicing 15
2.2.1 3GPP Approach Towards Managing Network Slices 16
2.2.2 ETSI NFV-MANO and 3GPP Network Slicing 17
2.3 ETSI Multi-access Edge Computing 17
2.4 Network Service Orchestrators 19

3. Radio network information in a MEC-in-NFV environment **21**
3.1 Introduction 21
3.2 RNIS as a Service (RNISaaS) 22
3.2.1 Implementing a MEP on top of OpenAirInterface 22
3.2.2 OAI-based RNIS Implementation 24
3.3 RNIS message broker implementation 25
3.3.1 RabbitMQ 25
3.3.2 Apache Kafka 25
3.3.3 Distinctive characteristics 25
3.4 Performance evaluation 26
3.4.1 Increasing numbers of subscribers 27
3.4.2 Resource utilization 28
3.4.3 Analysis 28
3.5 Summary 30

4. Placement of Cloud Native Network Services **31**
4.1 Introduction 31
4.2 Placement of Virtual Network Functions 32
4.3 Simple Cloud Native Network Service 32
4.3.1 Modeling a Simple Cloud Native Network Service 32
4.3.2 Deploying a Simple Cloud Native Network Service 33
4.4 Dynamic Resource Allocation and Placement Model 34
4.4.1 Preliminary 34
4.4.2 Cost Model 34
4.4.3 Availability Model 35
4.4.4 Constraints 36
4.4.5 Problem Formulation 37
4.5 Heuristic Approach 38
4.6 Performance Evaluation 39
4.7 Summary 43

5. Lightweight edge Slice Orchestration Framework **44**
5.1 Introduction 44
5.2 Lightweight edge Slice Orchestration Framework 45
5.2.1 Global Architecture 45
5.2.2 Edge Sub-Slice 46
5.2.3 LeSO Design 48
5.2.4 Isolation between slices 49

5.2.5 Working . . . 50
5.3 Performance Evaluation . . . 51
5.4 Summary . . . 54

6. Availability and Latency Aware Deployment of Cloud Native edge Slices 55
6.1 Introduction . . . 55
6.2 Deployment Models for edge Slice . . . 56
6.2.1 Orchestration and Management . . . 57
6.2.2 Availability . . . 58
6.2.3 Latency . . . 58
6.3 Modeling Availability . . . 59
6.4 Performance Evaluation . . . 62
6.5 Summary . . . 65

7. Cloud Native Lightweight Slice Orchestration Framework . . . 66
7.1 Introduction . . . 66
7.2 Cloud-native Lightweight Slice Orchestration (CLiSO) Framework 68
7.2.1 CLiSO Framework Architecture . . . 68
7.2.2 Domain Specific Handler (DSH) . . . 71
7.2.3 Network Slice Template . . . 73
7.2.4 Isolation between slices . . . 76
7.2.5 Monitoring and Logging . . . 76
7.2.6 Working of the Framework . . . 77
7.3 Performance Evaluation . . . 78
7.3.1 Compatibility Testing . . . 78
7.3.2 Configuration Testing . . . 80
7.3.3 Scalability Testing . . . 81
7.4 Summary . . . 83

8. Concluding Remarks and Perspectives . . . 84
8.1 Perspectives . . . 85

9. Résumé . . . 87
9.1 The 5G Journey . . . 87
9.2 Network Slicing, MEC and NFV . . . 87
9.3 Multi-domain Network Slicing . . . 90
9.4 Challenges in Orchestrating a Multi-domain Network Slice . . . 90
9.4.1 Challenges related to Availability . . . 91
9.5 Motivation de la thèse . . . 91

9.6 Résumé des chapitres 92

References **96**

List of Figures

1.1 A Multi-domain Network Slice 4

1.2 Visual Representation of Thesis Outline 6

2.1 Timeline of ETSI-NFV Standards Evolution 10

2.2 Cloud native Network Function (CNF) ETSI-NFV-IFA 029 sec.6.2.4 . 12

2.3 NFV-MANO Architectural Framework with Support for Containers . 13

2.4 Kubernetes Pod 15

2.5 5G Core Network Service Based Architecture 16

2.6 Mapping between NSI and Network Service 17

2.7 Multi-access edge system reference architecture variant for MEC in NFV 18

3.1 Architecture of the MEC Platform. 23

3.2 Operation of the RNIS provided by our OAI-based MEC platform. . . 24

3.3 Effects on E2E latency with increasing numbers of subscribers. The message rate is set to 10/s; each subscriber has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 10,000 * Number of subscribers. 27

3.4 Apache Kafka and RabbitMQ CPU utilization vs. E2E latency 29

4.1 Simple cloud-native network service S with K replicas of CNF A . . . 33

4.2 Cloud native network service deployment flow diagram 33

4.3 Number of nodes and pods required by a cloud-native network service 40

4.4 Execution time as a function of cluster size 41

4.5 Comparing two availability models 42

5.1 Placement of the LeSO Framework in Global NS LCM 46

5.2 Edge Sub Slice Skeleton 46
5.3 Skeleton of Edge Sub-Slice Template 47
5.4 Lightweight edge Slice Orchestration Framework 48
5.5 Isolation between slices 49
5.6 LCM of ESST 50
5.7 Computational Resource Consumption for 4 Edge Slices 52
5.8 Parallel Slice Creation Time 53

6.1 1 to 1 Mapping of MEC Applications and Microservices 56
6.2 1 to *N* Mapping of MEC Application and Micro-services 57
6.3 Markov chain for 1 to N deployment model 59
6.4 Markov Chain for 1 to 1 deployment model 60
6.5 Inter Microservice RTT (μS) for different models and testbeds 62
6.6 Slice Creation Time (S) for different models and Testbeds 63
6.7 Pod Recreation Time for different models and Testbeds 64

7.1 Proposed Cloud-native Lightweight Slice Orchestration Framework . 69
7.2 Proposed CISM Descriptor 71
7.3 Proposed MEP Descriptor 72
7.4 Proposed Network Slice Template 73
7.5 Proposed VNF and PNF Descriptor 75
7.6 Ran and Core Slice Deployment 80
7.7 Slice in Three Technological Domains 82
7.8 Slice Creation time vs CPU Consumed 82

9.1 A Multi-domain Network Slice 90
9.2 Représentation visuelle du plan de la thèse 93

List of Tables

1.1 Publication and their contributions to the chapters 9

4.1 Summary of Notations 34

5.1 Time spent (in seconds) in LCM of 4 slices 51

5.2 Time spent (in seconds) in each stage of ESST LCM 51

5.3 Resource Requirement Comparison 53

6.1 Availability of a Slice with 10 and 50 Microservices 64

7.1 CISM Details for Compatibility Testing 79

7.2 Core Network Slice Life cycle On Various Cloud Platforms via CLiSO and OSM 79

7.3 Ran and Core Network Slice Life cycle Time (seconds) 81

7.4 Resource Requirement OSM and ONAP 83

9.1 Publication and their contributions to the chapters 95

Acronyms

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
AppD	MEC Application Descriptor
AR	Augmented Reality
BSS	Business Support System
CCM	CIS Cluster Management
CIR	Container Image Registry
CIS	Container Infrastructure Service
CISM	Container Infrastructure Service Management
CNCF	Cloud Native Computing Foundation
CNF	Cloud-native or Container Network Function
DNS	Domain Name Server
eMBB	enhanced Mobile Broadband
ETSI	European Telecommunications Standards Institute
GRPC	Google Remote Procedure Call
HMTC	High-Performance Machine Type Communication
ICMP	Internet Control Message Protocol
IPC	Inter Process Calls
ISG	Industry Standard Group
JSON	JavaScript Object Notation
KPI	Key Performance Index
MEC	Multi-access Edge Computing

MEO	MEC Orchestrator
MEP	Multi-access Edge Computing Platform
MEPM	MEP Manager
MIoT	Massive Internet of Things
NFVI	NFV Infrastructure
NFVO	Network Function Virtualization Orchestrator
NSD	Network Service Descriptor
NSI	Network Slice Instance
NSSAI	Network Slice Selection Assistance Information
NSSI	Network Slice Subnet Instances
NST	Network Slice Template
OAI	OpenAirInterface
OCI	Open Container Initiative
ONAP	Open Network Automation Platform
OS	Operating Systems
OSM	Open Source MANO
OSS	Operational Support System
PNF	Physical Network Function
PNFD	Physical Network Function Descriptor
QCI	QoS Class Identifier
QoE	Quality of User Experience
QoS	Quality of Service
RAN	Radio Access Network
RNIS	Radio Network Information Service
SBA	Service Based Architecture
SD	Service Differentiator
SLA	Service Level Agreement
SLO	Service Level Objective
SST	Slice Service Type

TOSCA	Topology and Orchestration Specification for Cloud Applications
UE	User Equipment
uRLLC	ultra-Reliable Low Latency
V2X	Vehicular to Everything
vCPU	Virtual Central Processing Unit
VM	Virtual Machine
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VNFD	Virtual Network Function Descriptor
VR	Virtual Reality
YAML	Yet Another Markup Language
ZSM	Zero-touch Service Management

Glossary

AMF Access and Mobility Mobility Function is the 5G control plane component responsible for providing connection and mobility to UE.

AUSF Authentication Server Function is responsible for authenticating the UE on the network.

CNI Container Network Interface is responsible for inserting network interfaces in container network namespace

ESS AN Edge Sub Slice contains multiple MEC Applications or CNF or both, not connected or connected to each other. MEC Applications and CNFs consume the services of one another via their own service discovery mechanism if needed.

ESST An Edge Sub Slice Template is used to define an ESS.

NRF Network Registry Function's role is to provide service registry and discovery service to other network functions. It has a prime role in enabling the cloud-native deployment of 5G core network

SMF Session Management Function is the control plane component responsible for treating the data network session request for a UE.

SO Slice Orchestrator manages the lifecycle of a NSI, which is composed of four steps: preparation, instantiation, configuration, activation, and decommissioning

UDM Unified Data Management is responsible to providing the authentication vectors to AUSF or session management information to SMF. It gathers this information from UDR

UDR Unified Data Repository is the 5G centralized repository to store the 5G subscriber information

UDSF Unstructured Data Storage Function stores the state of the stateless network function. Such has UE context information or timer-related information

Chapter 1

Introduction

1.1 The 5G Journey

The 2nd generation of the mobile network is the catalyst behind making mobile phones a necessity in our daily lives. It was the 1st generation of mobile networks that was present across the globe. It allowed sending text and multimedia messages which were unimaginable at that time. The 3rd generation provided the possibility of using mobile phones for using websites and watching videos. The 4th generation truly revolutionized our society by allowing an always-on data connection at more affordable prices, which was impossible with other generations. It is the reason why the number of mobile phone users has increased tremendously in the past 10 years. Every generation is trying to efficiently use the limited spectrum to serve the ever-growing demands of mobile communication users.

4G created an opportunity for many new businesses, that relied on mobile phone internet connectivity for reaching their customers. The heavy usage of media streaming and broadcasting, online gaming, and smart connected devices challenged the 4G one-size-fits-all design. 4G was not designed to provide every application with tailored Quality of Service (QoS), especially for applications requiring ultra-reliable low latency. This shortcoming was well addressed by the 5th generation of mobile networks. That is designed to provide tailored QoS as per application requirements via the concept of Network Slicing [1]. Between every generation, there is a transition period due to high capital investment in the successor. In the case of 5G, some telecommunication operators preferred to deploy 5G non-standalone, using 5G New Radio with a 4G core network to provide certain features of 5G to mobile phone users.

1.2 Network Slicing, MEC and NFV

3rd Generation Partnership Project (3GPP) introduced Network Slicing to logically slice the network to handle different traffic patterns. All the logical slices are hosted on the same physical infrastructure, increasing the efficiency of the underlying infrastructure. 3GPP release 17 [2] identifies five different types of slices based on application traffic patterns,

- enhanced Mobile Broadband (eMBB): This slice is suitable for the handling of high downlink throughput demanding applications. For example, Ultra High Definition (UHD) media streaming.
- ultra-Reliable Low Latency (uRLLC) Demanding Applications: This slice is suitable for applications that require ultra-reliable and low-latency communication. For exam-

ple, controlling drones, remote medical surgeries, and playing musical instruments for a remote concert.

- Massive Internet of Things (MIoT): This slice is suitable for the handling of massive machine-to-machine type communication for Industry 4.0 [3].
- Vehicular to Everything (V2X): This slice is suitable for handling autonomous driving and truck platooning scenarios.
- High-Performance Machine Type Communication (HMTC): This slice is suitable for handling IoT devices that require high bandwidth communication.

3GPP provides the possibility to define custom slices because all the above slices can not handle every traffic pattern. Emerging 5G applications such as mission-critical services, Augmented Reality (AR), and Virtual Reality (VR) require the capabilities of both eMBB and URLLC. Such services can be enabled by including Multi-access Edge Computing (MEC) in 5G infrastructure. Edge cloud or MEC [4] [5] provides a home to low latency demanding applications. The close placement of MEC to its end users makes it suitable to deploy emerging 5G services.

European Telecommunications Standards Institute (ETSI) MEC Industry Standard Group (ISG) was formed to create standards for orchestrating applications at the edge of the network. The group proposed an Application Descriptor (AppD) to describe the requirements of the MEC applications. One such requirement for the applications hosted at the MEC is traffic redirection or DNS-based redirection rules to steer traffic to MEC Applications instead of the Internet. As the traffic redirection needs to be done dynamically at the instantiation of the MEC application, the traffic redirection is defined as a rule in the AppD. Besides, the traffic redirection is enforced by the MEC Platform (MEP) element that acts as an interface between the MEC and 5G Radio Access Network(RAN) and Core Network domains. MEP hosts a Radio Network Information Service (RNIS) that can be used by applications to get radio information for their subscribers. RNIS allows tailoring the Quality of Experience (QoE) at the subscriber level. Making MEC an essential entity in 5G and beyond networks.

The architectural complexity of mobile networks has been increasing from one generation to another. Each generation requires different deployment and monitoring methodologies. Network Functions Virtualization (NFV), introduced in 2012, proposed an alternative way of deploying network functions rather than Physical Network Functions (PNF). NFV proposed to decouple the software stack from the hardware of PNF and deploy the software in a virtualized environment in the cloud. The aim of NFV is to bring a paradigm shift from using purpose-built hardware to Commercial off-The-Shelf (COTS) hardware for deploying network functions. This allows faster upgrades of the network. ETSI NFV ISG, formed in 2012, is responsible for publishing standards for managing the life cycle of VNFs.

Virtual Machines (VMs) have a proven track record in the field of software virtualization. The first NFV architecture, the VNF instantiation descriptors, and orchestration methodology depended a lot on VM-based VNFs. Though VMs have always been hard to

manage, the interoperability issues between hypervisors, long instantiation time, and high resource demands are some of the main problems behind using VMs. Whereas containers have a quicker instantiation time, smaller resource footprint, and enable microservices architecture for software design. Microservices architecture allows breaking a monolithic software application into an easily manageable software stack to facilitate software management and upgrade.

Containerization existed for a long time, but IT industries started using containers when docker [6] was introduced in 2013. NFV group started considering containers as an alternative to VMs after ETSI NFV's, "Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS" which was published [7] in 2019. The term cloud-native as coined by Cloud Native Computing Foundation (CNCF) [1] is a software approach for developing and managing applications that are native to all kinds of clouds, Public, Private, and Hybrid clouds. Well-known network services orchestrators such as ETSI Open-Source MANO (OSM), Open Network Automation Platform (ONAP) [8] and orchestrators presented by authors in [9], [10] are capable of orchestrating container-based VNFs or Container or Cloud-native Network Functions (CNF). A network service is a functionality provided by the VNF, PNF, or a connected combination of both. Cloud-native and NFV have played a big role in influencing the 5G Core Network's Service Based Architecture (SBA) and Network Slicing. The whole 5G core network can be considered a network service, making Cloud-native NFV the backbone for 5G and beyond networks.

Containerization is one of the basic requirements for a cloud-native application. There are other requirements as stated by CNCF [2]:

- Loosely coupled and API-based design: Applications should follow the microservices-based architecture and should communicate using remote Application Programming Interface (API) calls. 5G Core Network SBA fulfills this criterion.
- Manageable: The application configuration should be outside of the application. Allowing easy changes in application functionality.
- Observability: The application should expose metrics to observe its health and functioning.
- Scalability: CNCF does not explicitly mention scalability but in the cloud environments applications are designed to be sized and re-sized dynamically to balance the cost and availability ratio. The applications should support either vertical, horizontal, or both types of scalability.

1.3 Multi-domain Network Slicing

A Network Slice can spread across multiple technological domains, i.e., Radio Access Network (RAN), Transport Network, Core Network (CN), and Edge computing. The Core Network domain acts as the centralized cloud to host the components of the 5G

[1] https://www.cncf.io/

[2] https://github.com/cncf/toc/blob/main/DEFINITION.md

core network. RAN domain is designed for hosting either monolithic or disaggregated RAN components such as Distributed Units (DU), and Centralize Units (control or user plane) (CU-C/U). In some cases, the control plane unit of CU can be hosted in the central cloud and the user plane unit of CU and 5G core network in the edge cloud. The edge computing domain consists of MEC, which was proposed by the authors of [11]. The transport domain is spread across all three domains and it is responsible for interconnecting the domains. Figure 1.1 depicts a Network Slice consisting of all the technological domains. In the figure edge computing domain is placed in between the Core and RAN domain to consider the scenario where RAN and 5G core user plane components will be placed in the edge cloud. Otherwise, if the edge computing domain only consists of MEC applications then it can be placed after the core network domain.

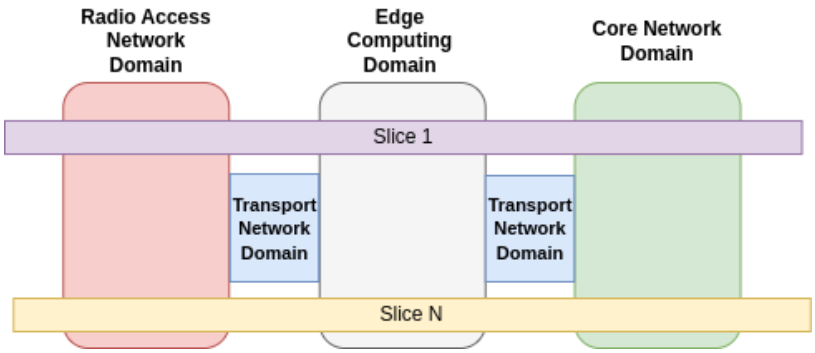


Figure 1.1: A Multi-domain Network Slice

In 5G the network functions require Network Slice Selection Assistance Information (NSSAI) that they can support. This allows performing network slicing at the network function level i.e., using a single network function to support multiple slices. Whereas, a network slice can also have dedicated network functions supporting only a single slice. It is also possible to have a combination of both scenarios.

1.4 Challenges in Orchestrating a Multi-domain Network Slice

Orchestrating all these different scenarios of network slicing can be challenging, especially when the slice is spread across multiple domains. Indeed, the current state-of-the-art does not focus on realizing multi-domain cloud-native network slice orchestration. Below are some of the challenges of the existing orchestrators:

- No native support for network slice orchestration: They were designed to orchestrate only network services, limiting their ability to support end-to-end network slice orchestration.
- Legacy VM-based Design: Their architectural design is heavily based on VM-based VNFs and they were recently upgraded to support container-based VNFs.

- No support for Orchestrating MEC Applications: The orchestrators can manage network functions at the edge but none of the orchestrators provide MEP capabilities such as traffic redirection that are needed by MEC applications.

- Infrastructure and platform depends: Most of the orchestrators use Kubernetes[3], a container orchestration engine to orchestrate container-based VNFs. Their architecture and network slice or network service descriptors are highly dependent on the community distribution of Kubernetes. Whereas there are multiple distributions of Kubernetes and in the future, there can be other orchestration engines. This adds an obligation to be infrastructure aware.

- Complex architecture and high resource footprints: The support for containers on top of legacy VM-based infrastructure increased the architectural complexity. Orchestrators such as ONAP require high amounts of computational and storage resources, making it unsuitable to operate in resource-constrained environments.

1.4.1 Challenges related to Availability

Apart from these challenges, the existing literature on cloud-native network function orchestration does not discuss their availability aspect. Cloud-native network functions have to abide by the telco grade 99.999% availability.

- Challenge I: The cloud-native way to achieve this availability is to have multiple replicas of the network function. This might lead to over-provisioning of infrastructural resources, increasing the deployment and management costs. Hence, a decision problem arises; How many replicas of each network function should be deployed? Without over-provisioning computational resources to avoid the high cost and provide service availability as promised in the Service Level Agreement (SLA)?

- Challenge II: Network functions deployed at the edge are latency-sensitive. The microservices architecture of cloud-native applications can affect their latency. Tightly coupled deployment of such applications will have a better latency but lower availability. Whereas, loosely coupled deployment will have better availability but higher latency. What should be the trade-off?

1.5 Thesis Motivation

Following these challenges, this thesis is motivated by the absence of a:

1. lean easy to use lightweight multi-domain slicing orchestration framework. A framework that can simultaneously orchestrate ran, edge, and core domain slices.

2. framework that follows cloud-native principles to orchestrate and manage CNFs on public, private, and hybrid clouds.

[3]https://kubernetes.io/

3. framework that abstracts the platform and infrastructure-related information from the network slice owner.

4. placement algorithm that provides cost and availability-aware deployment of cloud-native network functions.

5. methodology around the deployment of cloud-native network function at the edge of the network that requires low-latency and telco-grade 99.999% availability.

The end goal of this thesis is to realize a multi-domain slice orchestration framework to fill gaps among the standards and existing orchestrators.

1.6 Thesis Outline

The thesis starts by providing a background of the relevant terminologies. Chapter 2 provides a brief background on NFV, MEC, Network Slicing, 5G Core Network Service Based Architecture, and state-of-the-art network slice and service orchestrators. Figure 1.2 represents the visual outline of the thesis.

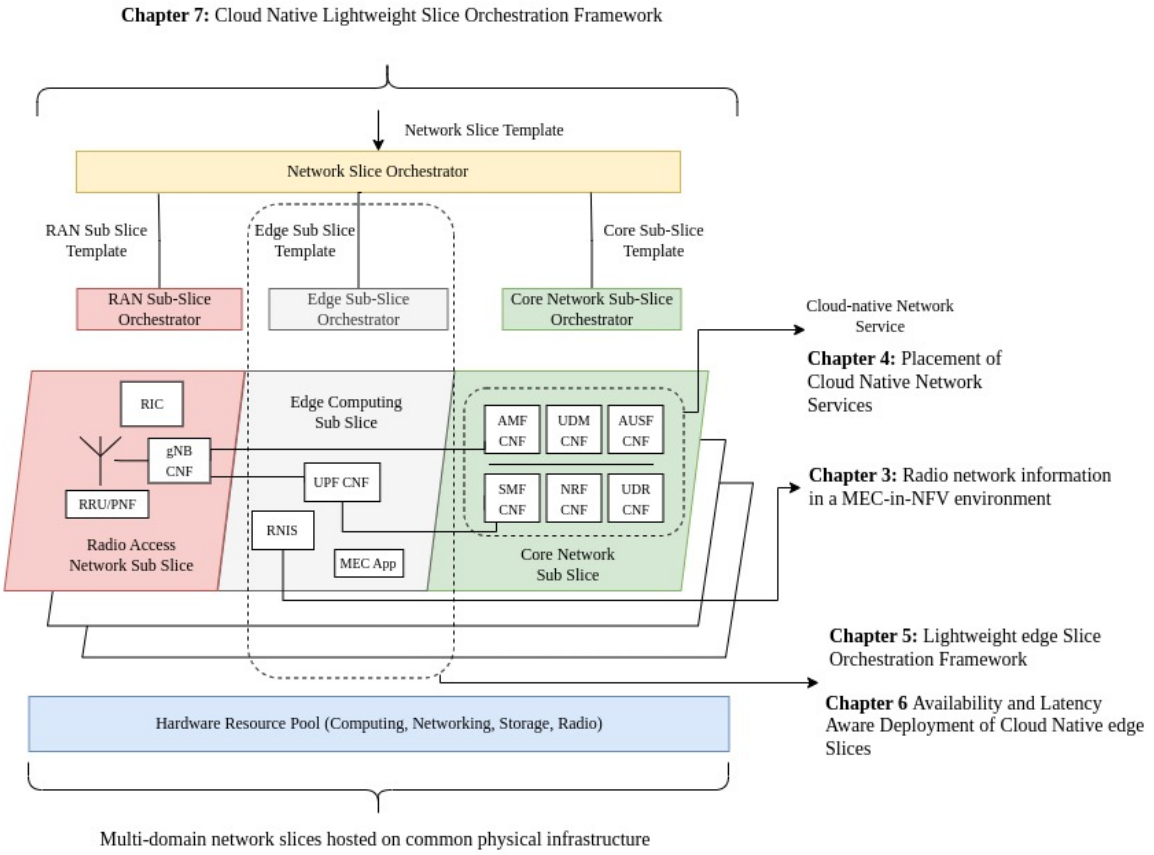


Figure 1.2: Visual Representation of Thesis Outline

- **Chapter 3**: The Radio Network Information Service (RNIS) is one of the key services provided by a MEP. It is responsible for interacting with the RAN, collecting RAN-level information about User Equipment (UE), and exposing it to MEC applications. In turn, this can be used by the latter to adjust their behavior to optimally match the RAN conditions. This chapter presents a standard-compliant RNIS implementation based on OpenAirInterface and studies critical performance aspects for its provision as a VNF for *MEC-in-NFV* environment. Since the RNIS design and operation follow the publish-subscribe model, the chapter investigates alternative implementations using different message brokering technologies (RabbitMQ and Apache Kafka) and compares their use and performance in an effort to evaluate their suitability for providing RNIS in an *as-a-service* manner.

- **Chapter 4**: This chapter proposes a cloud-native approach to provide resilience to simple cloud-native network services. A simple cloud-native network service comprises a single CNF. Considering that a CNF alone is capable of providing service to other CNFs, CNF as a service. The chapter presents a dynamic resource allocation and placement algorithm for modeling and placing a simple cloud-native network service. The algorithm aims to minimize infrastructural resource utilization under the constraint of abiding by service availability mentioned in the SLA.

- **Chapter 5**: This chapter proposes a Lightweight edge Slice Orchestration (LeSO) framework, a cloud-native orchestrator that orchestrates and manages the deployment of microservices as sub-slices at the edge. Whilst the existing orchestration frameworks are greedy for computing resource consumption and fail to integrate with the Multi-access Edge Computing (MEC) domain, LeSO by design is very lightweight and integrates a MEC platform-like component to guarantee traffic steering to automate edge slice deployment. Experiment results show that LeSO necessitates a small amount of CPU and memory, even when a high number of edge slices are deployed.

- **Chapter 6**: In the cloud-native paradigm, highly embraced by cloud providers, network functions and applications are decomposed into microservices that run in a container. Defacto container orchestration engine, namely Kubernetes, deploys multiple containers inside a Pod. The mapping between microservices and Pod highly affects the availability and latency of deployed microservices and hence the running application. This chapter proposes novel availability and latency-aware deployment models for an edge service composed of multiple applications designed as multiple microservices. The two considered deployments are analyzed and evaluated using experimentation and an analytical model, considering critical performance criteria for edge-oriented services, like availability and latency requirements.

- **Chapter 7**: To overcome the challenges of orchestrating a multi-domain network slice, this chapter proposes a novel Cloud-native Lightweight Slice Orchestration (CLiSO) framework extending the Lightweight edge Slice Orchestration (LeSO) framework proposed in chapter 5. In addition, the chapter presents a technology-agnostic and deployment-oriented network slice template. To allow zero-touch management of network slices, the framework provides a concept of Domain Specific Handlers. The

framework has been thoroughly evaluated via orchestrating OpenAirInterface[12] container network functions on public and private cloud platforms.

The last Chapter 8 concludes the work presented in this thesis and discusses future works. Table 1.1 contains a list of the publications made during the thesis work.

Type	Reference	Ch. 4	Ch. 5	Ch. 6	Ch. 7	Ch. 8
C	S. Arora, P. A. Frangoudis and A. Ksentini, "Exposing radio network information in a MEC-in-NFV environment: the RNISaaS concept," *2019 IEEE Conference on Network Softwarization (NetSoft)*, Paris, France, 2019, pp. 306-310	**X**				
C	S. Arora and A. Ksentini, "Dynamic Resource Allocation and Placement of Cloud Native Network Services," *ICC 2021-IEEE International Conference on Communications*, Montreal, QC, Canada, 2021, pp. 1-6		**X**			
C	S. Arora, A. Ksentini and C. Bonnet, "Lightweight edge Slice Orchestration Framework," *ICC 2022-IEEE International Conference on Communications*, Seoul, Korea, Republic of, 2022, pp. 865-870			**X**		
M	S. Arora, K. Boutiba, M. Mekki and A. Ksentini, "A 5G Facility for Trialing and Testing Vertical Services and Applications," in *IEEE Internet of Things Magazine*, December 2022 vol. 5, no. 4, pp. 150-155			**X**		
C	S. Arora, A. Ksentini and C. Bonnet, "Availability and Latency Aware Deployment of Cloud Native Edge Slices," *GLOBECOM 2022-2022 IEEE Global Communications Conference, Rio de Janeiro, Brazil*, 2022, pp. 3441-3446				**X**	
J	S. Arora, A. Ksentini and C. Bonnet, "Cloud Native Lightweight Slice Orchestration (CLiSO) Framework," *Computer Communications*					**X**

C = Conference, J = Journal, M = Magazine

Table 1.1: Publication and their contributions to the chapters

Chapter 2

Background

2.1 Network Function Virtualization

Network functions are the functional building blocks of any network infrastructure. In 2nd Generation of mobile networks, all the network functions use purpose-built hardware known as Physical Network Function (PNF). The concept of Network function virtualization (NFV) proposed in 2012 has transformed the Telecom industry today. NFV's purpose is to decouple the software stack from the PNF hardware and use commodity-off-the-shelf (COTS) hardware for the software stack. These software-only network functions are known as Virtual Network Functions (VNF), and they are packaged in VMs or Containers to be hosted on general-purpose or slightly customized hardware.

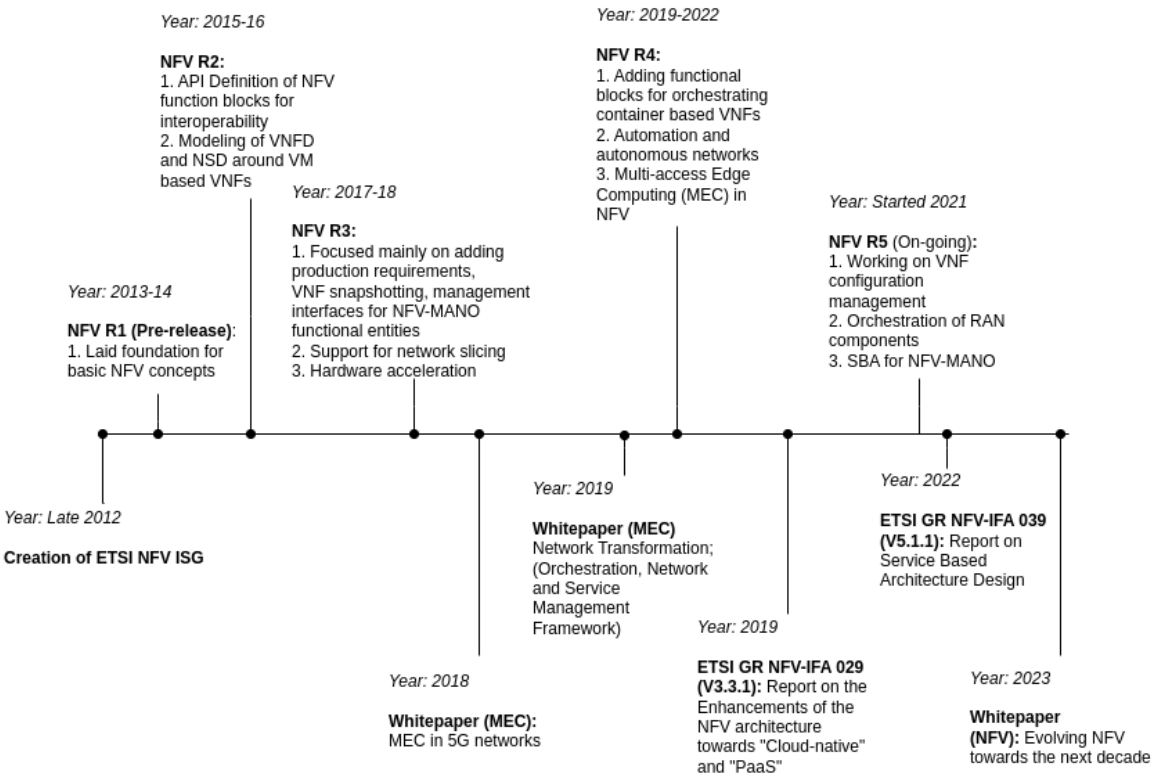


Figure 2.1: Timeline of ETSI-NFV Standards Evolution

NFV standards are driven by ETSI NFV ISG. It was created in 2012 to facilitate the adoption of NFV in the telecommunication industry. The first release of ETSI GS-NFV

laid the foundation by providing NFV use cases and an architectural framework for VNF orchestration and management, ETSI NFV-MANO. According to ETSI NFV, VNFs are designed to be deployed individually or with other VNFs and PNF to provide a network service. In other terms, a network service comprises one or more PNF or VNFs. The life cycle of this network service is managed by ETSI NFV Orchestrator (NFVO).

Figure 2.1 summarizes different releases of ETSI-NFV and their relevant milestones which are important for this thesis. The subsequent sections will explain different terminologies and technologies which form the functional building block of ETSI NFV standards.

2.1.1 Virtualization and Containerization

Virtualization allows sharing of the physical hardware of a computer among multiple virtual computers, known as Virtual Machines (VM). Virtualization creates an abstraction layer on top of the computer hardware to allow sharing of CPU, RAM, Storage, and Networking with VMs. The virtualization software is known as Hypervisor. There are multiple types of Hypervisors,

- Type I, Bare-Metal Hypervisors: These types of hypervisors replace the physical computers OS. They interact directly with the hardware to create VMs. They are mostly present in commercial servers.
- Type II, Operating System Hypervisors: These types of hypervisors are present on top of the physical computer's OS. They are mostly used on personal computers.

Virtual machines are like physical computers. They virtualize the underlying hardware and run a complete OS known as guest OS. Their performance depends on the underlying hardware. They provide good computational and security-based isolation to run different applications on the same hardware. The ability to run VMs of different guest OS than the host OS allows running multiple different applications which require different OS. This optimizes hardware utilization. Whereas if all the applications require the same OS, then packaging applications in VMs will add overhead.

To overcome this containerization took another approach of OS-level virtualization, packaging applications with their required libraries and some OS packages. This allows container-based applications to have a quicker instantiation time than VMs, as VMs have to mimic the actual hardware. Containers do not require a complete OS which reduces their resource footprints, enabling microservices architecture for designing applications in granular components. The foundation and basics of containerization technology have been around for a long time, but their usage started with the release of dockers[1].

There are multiple container orchestration tools, but all of them use a similar format for container images, similar to a VM image. This allows container images built with different tools to be managed using different orchestrators. Whereas VMs built for different hypervisors are not interoperable. The authors of [13] provide a performance comparison between a KVM[2] based VM and a docker container. The performance of a docker container

[1] https://www.docker.com/

[2] https://www.linux-kvm.org/page/Main_Page

is similar to or better than the KVM VM. Containers today do not provide the same level of isolation as VMs and they are not suitable if the application running inside the container requires a different Linux Kernel than the host machine. For I/O intensive applications both VM and container host machines require necessary customization.

The term cloud-native, as coined by CNCF is a software approach for developing and managing applications that are native to the cloud. Containers play an important role in designing and packaging such applications that are hosted in the modern container-based cloud. The industrial switch of using containers rather than VMs for running multiple applications on the same physical hardware pushed the NFV community to re-design the standards.

The foundation of NFV standards is based on VMs and for a long time till release 3, the standards were entirely focused on VM-based VNFs. The ETSI group report NFV-IFA 029 [7] provided the starting point for instantiating container-based VNFs. To keep in harmony with previous NFV standards, ETSI uses the term VNF for both VM-based and container-based VNFs. Though container-based VNFs are sometimes referred to as Cloud-native or Container Network Functions (CNF).

2.1.2 Cloud Native Network Functions

Cloud-native applications are designed on microservices-based architecture with statelessness in consideration. A stateless application stores its consumer's data or state in a database from which it can be retrieved when needed or it requests the consumer to store the data. This allows for securing the application data in case of any failure. In contrast, the traditional VM-based network functions are monolithic and stateful. Containerizing these monolithic applications will not receive the complete benefits of an agile cloud-native application. One of the possible solutions is to re-engineer these traditional VM-based VNFs to adapt to cloud-native principles.

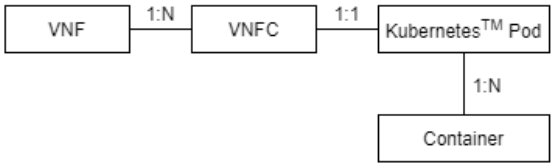


Figure 2.2: Cloud native Network Function (CNF) ETSI-NFV-IFA 029 sec.6.2.4

Figure 2.2 depicts a Kubernetes-based cloud-native VNF. Every VNF can have multiple Virtualized Network Function Components (VNFC) and each component is mapped to a Kubernetes pod. These Kubernetes pods can have multiple containers depending on the design of VNFC. A VNF is a logical entity, while VNFCs are functional blocks. That should follow cloud-native design principles. In a traditional VM-based VNF, the VNFC is mapped to a VM. This is one of the design possibilities of the CNF. There can be multiple others that are discussed in [7].

2.1.3 ETSI NFV Management and Orchestration

ETSI NFV Management and Orchestration (MANO) is a framework to orchestrate and manage the life cycle of network services comprised of VNFs and PNFs. The legacy NFVO architecture was not designed to manage modern container-based VNFs. It was only designed to orchestrate VM-based VNFs. ETSI GS NFV 006 release v4.4.1 proposes new functional blocks for managing the life cycle of container-based VNFs. Figure 2.3 shows the new ETSI NFV MANO framework supporting VM and Container-based VNFs.

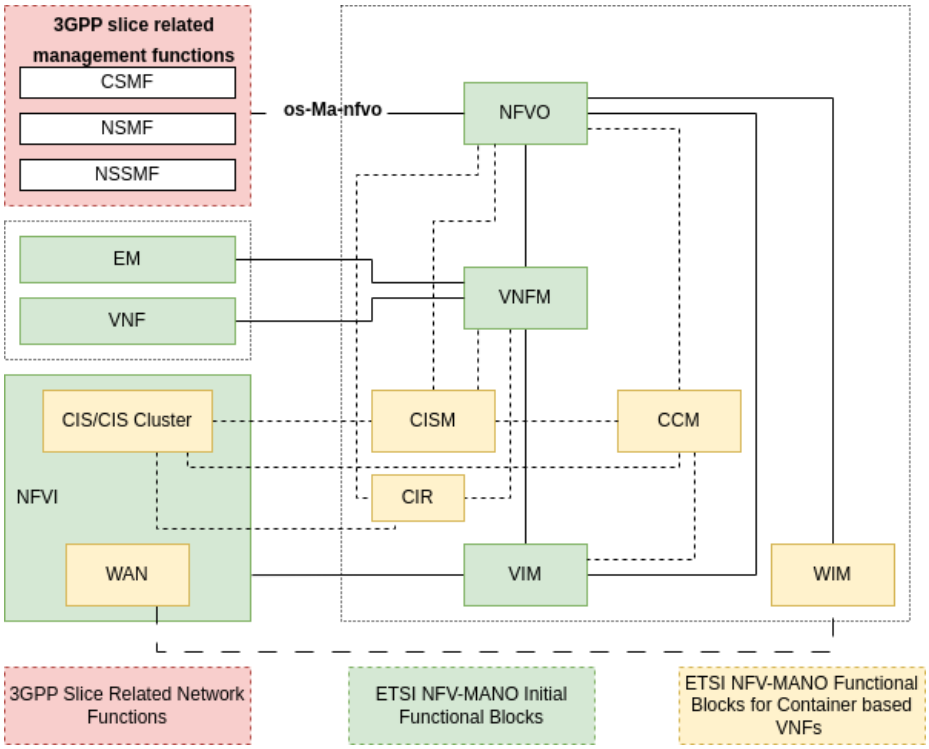


Figure 2.3: NFV-MANO Architectural Framework with Support for Containers

For simplicity, the newly added blocks for container management are in yellow and dotted lines to highlight the connections. The interface names are not present to avoid confusion. The role of different components is defined below:

- Network Function Virtualization Orchestrator (NFVO): Responsible for managing the life cycle of network services.
- Virtual Network Function Manager (VNFM): Responsible for managing the life cycle of VNFs, their configuration, performance, and fault management.
- Virtualized Infrastructure Manager (VIM): Responsible for controlling and managing the virtual resources of NFVI to provide them to VNFs. In short, it is responsible for the life cycle management of VMs.
- Element Manager (EM): These are designed to monitor and configure the VNFs. They can be used in some aspects of VNF's life cycle management. For example, fault management.

- Container Infrastructure Service (CIS): Container Infrastructure Service (CIS) provides run-time infrastructural dependencies, computational, storage, and networking resources for one or more containerization technologies. It can be considered the cloud-native equivalent of a virtual-machine hypervisor. Hypervisors provide infrastructure to host virtual machines.
- Container Infrastructure Service Management (CISM): manages containers executed by CIS. It is responsible for container deployment, monitoring, and life cycle management.
- CIS Cluster Management (CCM): is responsible for managing the life cycle of CISM.
- Wide Area Network (WAN): is the transport network used to connect multiple NFVI sites.
- WAN Infrastructure Manager (WIM): provides management of Multi-Site Connectivity Services.
- Container Image Registry (CIR): stores all the container images of the container-based VNFs.
- NFV Infrastructure (NFVI): The hardware infrastructure on which virtual machines or containers will be hosted.

In the figure 2.3 *os-Ma-nfvo* interface can be used to connect an Operational Support System (OSS) for example, Openslice [14].

Network services are described using a Network Service Descriptor (NSD). A NSD contains multiple Virtual Network Function Descriptors (VNFDs) and Physical Network Function Descriptors (PNFDs). VNFDs are VM centric and were designed for VM-based VNFs. The new specification contains special fields such as *osContainerDesc* to include container description. Each VNFD inside a NSD can have multiple containers described via the *osContainerDesc* field, one for each container. The number of containers depends on the VNFD provider. However, the VNFD template still lacks some fields that are needed to configure the security, networking, and configuration of network functions in a cloud-native environment. It should be noted that the current specification has a tight coupling with Kubernetes, a CISM. Apart from NSD, ETSI NFVO requires other packages that are needed for orchestrating network functions. These packages rely on helm-charts[3], which are tightly coupled with Kubernetes.

2.1.4 Kubernetes: CISM

Kubernetes is an industrial de-facto standard for container orchestration, inspired by Google's Borg platform[15]. ETSI NFV-MANO standard mentions Kubernetes as one of the possible container orchestration platforms, but, other service orchestrators may use Kubernetes. Today Kubernetes has an important role in realizing a network slice built on top of CNFs. Kubernetes is capable of managing multiple pods spread across a cluster

[3] https://helm.sh/docs/topics/charts/

of nodes. A Kubernetes Pod is a group of co-located containers. It is the smallest entity that can be scheduled by the Kubernetes scheduler. The containers inside a Pod share the same network namespace, separate computational resources, and can have common storage. Pods are designed to run multiple co-located processes with a degree of isolation. Whereas, containers are designed to host a single process. The containers inside a pod communicate with each other using the local network (loopback interface), shared memory IPC (inter-process calls), and shared volumes if the volumes are common. Figure 2.4 shows the design of a Kubernetes Pod.

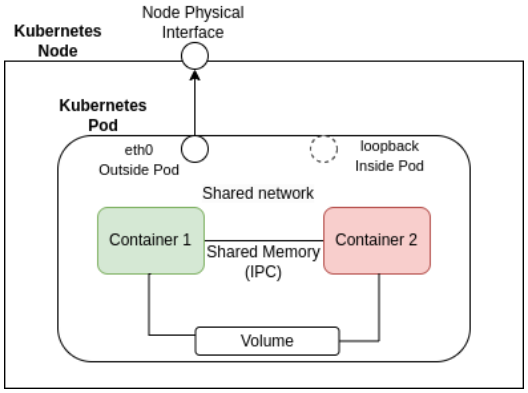


Figure 2.4: Kubernetes Pod

It should be noted Kubernetes is an example of CISM, but there can be alternatives such as Apache Mesos[4], Docker Swarm [5] and HashiCorp Nomad [6]. Functioning and comparison of these alternatives are out of the scope of this thesis.

2.2 3GPP 5G Core Network SBA and Network Slicing

In ETSI TS 123.501, 3GPP proposed two architectures for a 5G Core Network, Reference Point and Service Based Architecture (SBA). The SBA was highly influenced by NFV and was designed to be deployed in a modern container-based cloud. Figure 2.5 shows 5G Core Network SBA. Network functions like Network Registry Function (NRF) and Unstructured Data Storage Function (UDSF) facilitate service discovery and registry and storage of the state of the network functions to make them stateless, respectively. These two features are much needed for an application to be cloud-native. Today big telecommunication companies[7] offer containerized 5G control plane and user plane functions.

Network slicing [1] is one of the key features of 3GPP release 15. It allows the creation of logically isolated networks on top of common physical network infrastructure to support service customization, isolation, and multi-tenancy. Network slicing challenges the 4G one-size-fits policy by tailoring the network according to network service needs. NFV provides the right infrastructure for realizing network slicing.

[4]https://mesos.apache.org/

[5]https://docs.docker.com/engine/swarm/swarm-tutorial/

[6]https://www.nomadproject.io/

[7]https://www.ericsson.com/en/news/2023/3/swisscom-ericsson-and-aws-collaborate-on-5g-core-for-hybrid-cloud

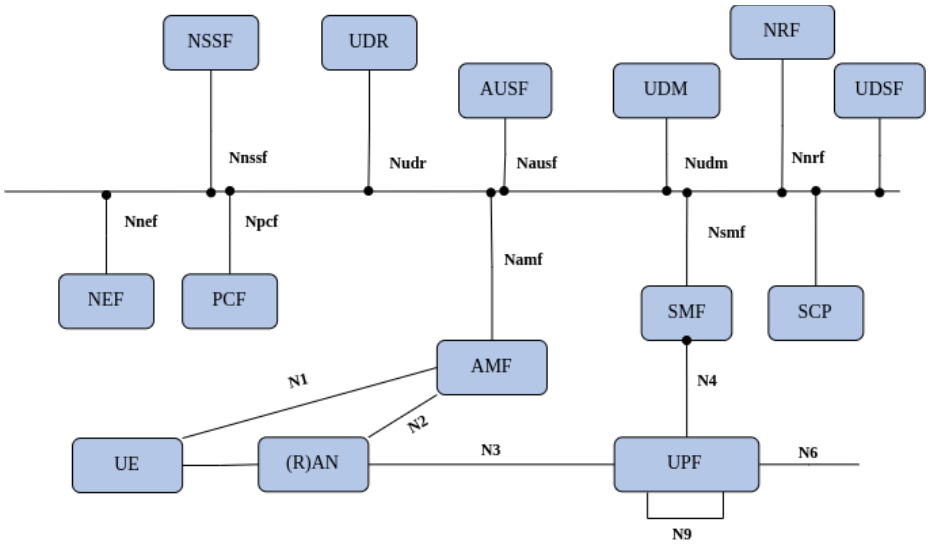


Figure 2.5: 5G Core Network Service Based Architecture

2.2.1 3GPP Approach Towards Managing Network Slices

Network slicing depends on resource virtualization while the orchestration mechanism depends on whether the network functions are realized as physical machines, virtual machines, containers, or a combination of both. 3GPP SA5 group in Technical Specification 28.53X (X in 0,1,2,3) defines the network slice management and automation mechanism. The 3GPP approach has two key concepts Network Slice Instance (NSI) and Network Slice Subnet Instance (NSSI). A NSSI is analogous to a network sub-slice. The 3GPP defines the following management functions related to NSI management, listed below in the order corresponding to their hierarchy:

- Communication Service Management Function (CSMF): Responsible for translating the communication service-related requirement(s) to network slice-related requirement(s). It forwards the request to the Network Slice Management Function (NSMF) to manage the life cycle of a NSI.
- Network Slice Management Function (NSMF): Manages the life cycle and monitors the KPIs of NSIs. It derives network slice subnet-related requirements from the network slice-related requirements. NSMF communicates with several NSSMFs, one for each domain, to manage the life cycle of the Network Slice Subnet Instance (NSSI).
- Network Slice Subnet Management Function (NSSMF): Manages the life cycle and monitors the KPIs of NSSIs. There can be a dedicated NSSMF for each functional domain, access network, transport network, edge network, and core network.

A NSI has four phases preparation, commissioning, operation, and decommissioning. In the preparation phase, NSI does not exist. The phase includes calculating instance

requirements, onboarding needed entities, reserving resources, and preparing the environment. NSI's life cycle stage includes the commissioning, operation, and decommissioning phase. In these phases, NSI goes through different stages creation, activation, modification, de-activation, and termination. A NSI is described using a Network Slice Template (NST). GSMA defines a Generic network Slice Template (GST) [16] to describe slice behavior, required quality of service, network functions required by the slice, etc. It does not focus on the deployment aspect of a NSI, the hardware requirement of network functions, their configuration, and links between them, etc. In [17] authors have proposed to use ETSI NFV Orchestrator and ETSI MEO as NSSMF for core and edge domains, respectively.

2.2.2 ETSI NFV-MANO and 3GPP Network Slicing

ETSI group report NFV-EVE 012 V3.1.1, explains the relationship between 3GPP proposed NSI and a network service. A network service is a resource-centric view of a network slice if a NSI contains at least one VNF. A NSI can contain one or more VNF, PNF, or NSSI. Similarly, NSSI can contain VNFs or PNF. Figure 2.6 explains the mapping between a communication service, NSI, NSSI and network service.

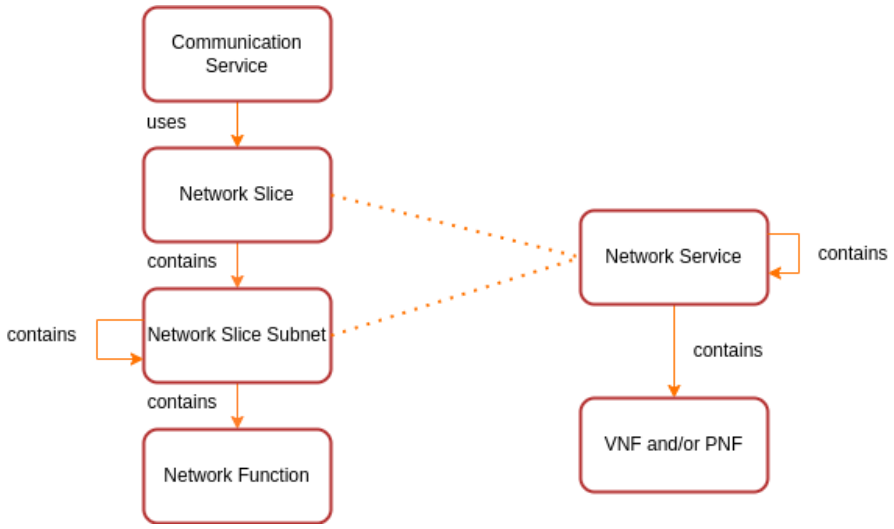


Figure 2.6: Mapping between NSI and Network Service

The group report highlights that the 3GPP slice management functions can be integrated with ETSI NFVO using *os-Ma-nfvo* interface as shown in figure 2.3. The NSMF and/or NSSMF is responsible for determining the type of network service or set of network services, VNF, and PNF that can support the resource requirements for a NSI or NSSI. It determines if there is a need to create new network services, VNFs, and the connectivity to the PNFs or existing instances can be re-used.

2.3 ETSI Multi-access Edge Computing

Multi-access Edge Computing provides a home to low-latency demanding applications. The early standards of MEC were introduced during 4G mobile communication

systems and the white paper [11] describes how MEC can play a vital role in enabling URLLC applications.

The first released document of ETSI MEC covers the reference architecture [18], specifying the different necessary components and their interfaces. It introduces four entities: (i) MEC platform (MEP) that acts as an interface between the mobile network and the MEC applications, (ii) MEC host that may host both the MEC framework and MEC applications, or only MEC applications, by providing a virtualization environment, (iii) MEC application, that is executed on top of the Network Functions Virtualization Infrastructure (NFVI) of the MEC host, with a MEP Manager (MEP-M) component in charge of MEP configuration and MEC application lifecycle management (LCM), and (iv) Mobile Edge Orchestrator (MEO) which is in charge of the lifecycle management of MEC applications, acting as the interface between the MEC host and the operator's OSS/BSS. ETSI MEC also introduced the concept of MEC services, which are either provided natively by the MEC platform, such as the RNIS [19], or are provided by MEC applications, e.g., video transcoding. MEC applications can discover MEC services available at the MEC host and register their own via the Mp1 reference point.

The MEC architecture is defined to run independently of the NFV environment. However, specific ETSI MEC activities have focused on the integration of MEC in NFV [20]. This involves running the MEP/MEP-M as a VNF and delegating MEC application instantiation to a regular ETSI MANO NFVO, and LCM to a VNFM, via standard ETSI NFV interfaces. Figure 2.7 depicts MEC in NFV architecture. ETSI MEC in NFV architecture has not been revised yet for container-based environment.

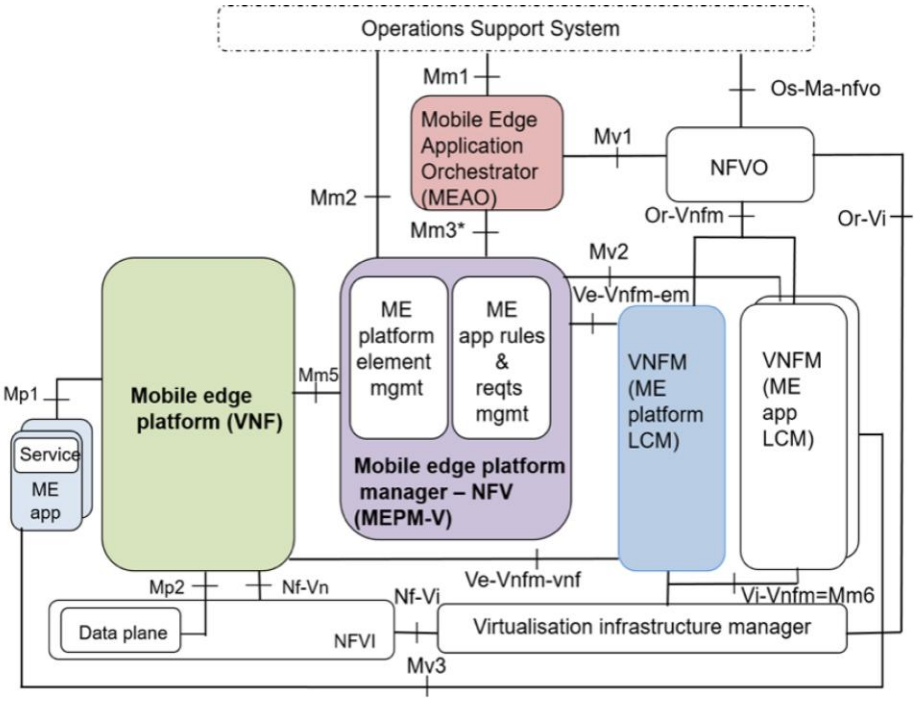


Figure 2.7: Multi-access edge system reference architecture variant for MEC in NFV

A MEC application is designed using the MEC Application descriptor (AppD) as defined in ETSI MEC 010-2 standard. The foundation of AppD is highly influenced by VMs

same for VNFDs. But the recent updates of the standards briefly mention the possibility of describing a container-based MEC application. At the time of writing this thesis, the standards lack a lot of clarity around the containerization of MEC applications.

2.4 Network Service Orchestrators

The existing service orchestrators and their network service descriptors were designed to orchestrate VM-based VNFs. Later they adapted their architecture and service descriptors to orchestrate container-based VNFs or CNFs. The service descriptors of the orchestrators are either based on standard ETSI NSD or a tailored NSD. The orchestrators are capable of placing the applications at the edge, but they do not provide the capabilities of MEP to edge applications. Below are some of the known service orchestrators,

- Open Source MANO (OSM): A network service orchestrator proposed by ETSI as a reference design based on the ETSI NFV standard. Designed to orchestrate VM-based network services, it supports container-based network functions. OSM can deploy network functions at the edge cloud but not MEC Applications, which require MEP support. For supporting MEC applications the ETSI MEC framework proposes an interface with NFVO. Starting OSM version 5, network slices can be orchestrated using the OSM network slice template. The network slice template contains several ETSI NSDs. The template does not clearly allow resource isolation. Apart from NSD, the orchestrator requires helm-charts or juju-charms[8] to describe the CNF configuration, required software images, and computation resources.

- Open Network Automation Platform (ONAP): Designed to manage VM-based VNFs, and with the recent release, it follows cloud-native principles to orchestrate CNFs. ONAP's complicated architecture with a large number of components results in high resource consumption. The Frankfurt release of ONAP introduced the functionality of network slice orchestration using the Topology and Orchestration Specification for Cloud Applications (TOSCA) template. The network slice template is convoluted and it is difficult to define resource isolation. The complex realization of ONAP demands the involvement of a big team in taking care of service orchestration. The high resource consumption makes it unsuitable to deploy in a resource constraint environment.

- 5G Transformer Service Orchestrator (5GT-SO)[9]: 5G transformer framework proposed a vertical slicer as the gateway for different verticals to instantiate their network services. The 5GT-SO is built on top of OSM and Cloudify but only supports VNFs.

- 5G SONATA[10]: A service orchestrator capable of orchestrating VNFs and CNFs. However, it does not orchestrate MEC applications. Making it unsuitable to orchestrate an end-to-end network slice.

ONAP and OSM both have their own tailored NSDs and NSTs; apart from that, they need VIM-specific packages like helm-charts and juju-charms. This forces the network

[8] https://charmhub.io/

function provider to be infrastructure and platform aware. OSM has a tight coupling with Juju-charms and Ubuntu-based Linux. Besides these orchestrators, Google proposed Nephio[9]. It is integrated with Kubernetes and it requires the users to be aware of Kubernetes. The project is at a very early stage and there is nothing mentioned about network slicing. The authors of [21] proposed a MEC application slicing concept via a proof of concept framework. They proposed a MEC Application Slice Subnet Descriptor (MAPSSD) to describe container-based MEC applications of a MEC slice. Their proposed descriptor and framework highly rely on helm-charts to describe application deployment/configuration-related information. Such an approach requires application providers to be infrastructure aware.

The core and edge domains both will have container-based network functions or applications. In fact, according to Open RAN Alliance (O-RAN)[10] architecture, it is possible that some of the access network functions for example, Central Unit-Control Plane (CU-CP) and Central Unit-User Plane (CU-UP) can be deployed as CNFs. ONAP addresses the challenges of orchestrating RAN network functions. However, its high resource consumption restricts its usage to a limited group of users.

[9]https://nephio.org/

[10]https://www.o-ran.org/

Chapter 3

Radio network information in a MEC-in-NFV environment

3.1 Introduction

RAN awareness can be proven beneficial for a wide range of applications in 5G and beyond context. A wealth of useful information is constantly generated at the RAN level, such as events pertaining to the network control and data planes, UE status and capabilities, mobility events, location updates, and information on the radio conditions at the user end. These data were traditionally available only to the network operator via the mobile network equipment's vendor-specific monitoring and management interfaces. However, with the advent of Multi-access Edge Computing (MEC), this situation is expected to change.

As per the ETSI MEC standard, a MEC platform (MEP) provides a set of services to application instances running at the mobile edge, among which is the *Radio Network Information Service (RNIS)* [19]. This service allows authorized MEC applications to consume RAN-level information, such as UE channel quality indications and location updates, which they can utilize to offer enhanced services and optimize performance. This creates space for innovative, RAN-aware third-party applications deployed at the mobile edge, in areas ranging from network troubleshooting and network resource management [22] to QoE optimized service delivery [23; 24].

From the perspective of the network operator, harnessing the potential of these data requires dealing with significant challenges. At the MEP level, handling such amounts of data and efficiently delivering them to MEC applications is already non-trivial. RAN-level data are generated at high volumes and have to be treated at the edge, where storage, processing, and memory resources are typically scarce. Scalability challenges thus emerge as the number of mobile terminals generating data and the number of MEC-hosted applications consuming the RNIS grow.

This gets more pronounced in a *MEC-in-NFV* environment [20] and as Network Slicing finds its way toward edge computing. In this environment, the MEP and its services, including the RNIS, are instantiated on demand over an edge cloud as virtual instances and as parts of a network slice instance. MEC orchestration components thus need to appropriately allocate compute resources to multiple RNIS instances corresponding to multiple MEC tenants.

This chapter contributes towards a better understanding of the performance requirements of offering *RNIS-as-a-Service (RNISaaS)* in a MEC-in-NFV environment. This chapter presents:

1. Design and implementation of an RNIS featuring a *standards-compliant* publish-subscribe API.

2. Experimental comparison between two candidate solutions for implementing publish-subscribe model, (RabbitMQ[1] vs. Apache Kafka[2]) for the provisioning of RNISaaS.

To the best of my knowledge, although existing works focuses on potential applications of the RNIS, this is the first work that addresses the design and implementation of the RNIS component itself, its internal workings, and their performance implications, particularly towards MEC-in-NFV. The experimental testbed and implementation of RNIS presented in this chapter are based on a 4G/5G non-standalone environment. Early RNIS ETSI MEC data models were designed for 4G and recently [25] the standards are adapted to support 5G new radio-related information. This chapter is the first step towards discovering MEC, VM-based NFV during the early 5G period. The chapter is structured as follows:

- Section 3.2 briefly presents the design principles, architecture, and implementation of the MEC platform built on top of OAI [12], with more details on how the RNIS delivers information to applications following the publish-subscribe model.

- Section 3.3 delves into the details of how message broker is implemented internally using two different candidate technologies.

- Section 3.4 shows experimental evaluation.

- Section 3.5 summarises the chapter.

3.2 RNIS as a Service (RNISaaS)

The RNIS is a key MEC service, allowing third-party applications to access contextual information on the UE end. Once the MEP is envisioned to be executed as a VNF, it is important to assess its performance, particularly the performance of the RNIS service in a virtualized environment. Given the volume of data handled by the RNIS and the potentially stringent delay requirements for their delivery to interested applications, the results of such a study can be important for the MEC operator to appropriately dimension the resources to allocate to each RNIS virtual instance and to set up the management mechanisms for their automatic scaling to meet the performance requirements of the MEC tenants. At the same time, such results can provide insight into the choice of suitable technologies for the implementation of specific internal RNIS mechanisms.

This section starts with a brief introduction to the MEP implementation based on OAI. A detailed description of the implementation of RNIS-as-a-Service (RNISaaS) follows.

3.2.1 Implementing a MEP on top of OpenAirInterface

OpenAirInterface (OAI) [12] is an open-source implementation of a complete, 3GPP-compliant 4G and 5G mobile systems. At the time of the MEP implementation, 5G was still

[1] https://www.rabbitmq.com/

[2] https://kafka.apache.org/

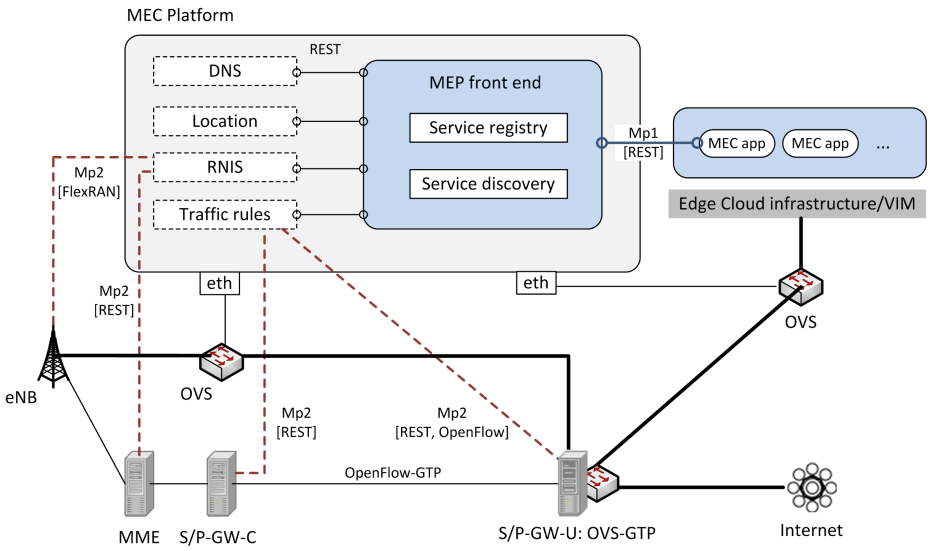


Figure 3.1: Architecture of the MEC Platform.

under development. Hence, this work uses OAI 4G RAN and 4G/5G non-standalone core network. The MEP is the MEC element that interfaces directly with radio and core network components via the Mp2 interface. MEP uses these interfaces to manage traffic steering towards MEC applications and to gather RAN-level information on the UE's environment and context, to expose it via the RNIS API (and/or other standardized interfaces such as the location API [26]).

To implement the traffic steering for 4G/5G non-standalone core network 3GPP proposed Control and User Plane Separation (CUPS)[27] provides a possibility to interact with the user plane. CUPS proposes to split the Service and Packet Gateway (SPGW) into two entities: SPGW-C and SPGW-U (C for the control plane; U for the user plane). The former is in charge of managing the signaling control to create the user-data plane, whereas the latter is in charge of forwarding the user-plane data. The SPGW-U is connected to the Internet and the edge virtualization platform. In response to a request from a MEC application (or when requested by the MEO via the MEPM), the MEP installs traffic rules on the SPGW-U to offload traffic to the MEC application. MEP performs this traffic redirection using OpenFlow protocol via Mp2 reference point. In the proposed OAI-oriented MEC platform, the SPGW-U is based on a patched version of the OpenVSwitch (OVS) software, which adds support for matching GTP packets.

On the other hand, the FlexRAN [28] protocol is used to implement the Mp2 interface towards the RAN to obtain radio statistics and expose them via the RNIS API. FlexRAN is a flexible and programmable software-defined RAN platform designed for controlling the 4G RAN. In 5G FlexRIC [29] supersedes FlexRAN to provide an Open-RAN (O-RAN[3]) standard compliant RAN Intelligent Controller (RIC) to manage the 5G RAN components and gather statistics.

[3]https://www.o-ran.org/

There are other functions that the MEP can provide (service registration and discovery, DNS, etc.) Figure 3.1 provides a global overview of the MEC platform's architecture and its interfaces with MEC applications and network elements.

3.2.2 OAI-based RNIS Implementation

The RNIS is exposed by the MEC platform via the Mp1 reference point. This service provides up-to-date radio network-related information to authorized mobile edge applications. This information can be provided with different granularity, using the IMSI, IPv4, or IPv6 address as UE identifiers. For example, the RNIS can provide RAN information per UE, for all the UEs under a specific cell coverage, by QCI value, and using various other combinations. The proposed RNIS implementation offers two methods for fetching this information:

- First, it provides a simple request-response mechanism where applications can access the RNIS using a REST HTTP interface.
- Second, it exposes a publish-subscribe interface, where an application can subscribe to a set of notifications to get updates on a range of parameters.

The latter provides more up-to-date, near-real-time information on radio conditions and gives the opportunity to applications to receive notifications across a rich set of criteria and their combinations. In the ETSI MEC 012 specification [19], these notifications have been divided into eight categories: cell change, UE measurement report, Radio Access Bearer (RAB) establishment, RAB modification, RAB release, UE timing advance, UE carrier aggregation reconfiguration, and S1-U bearer information. The operation of the OAI-MEP publish-subscribe mechanism is illustrated in Figure 3.2.

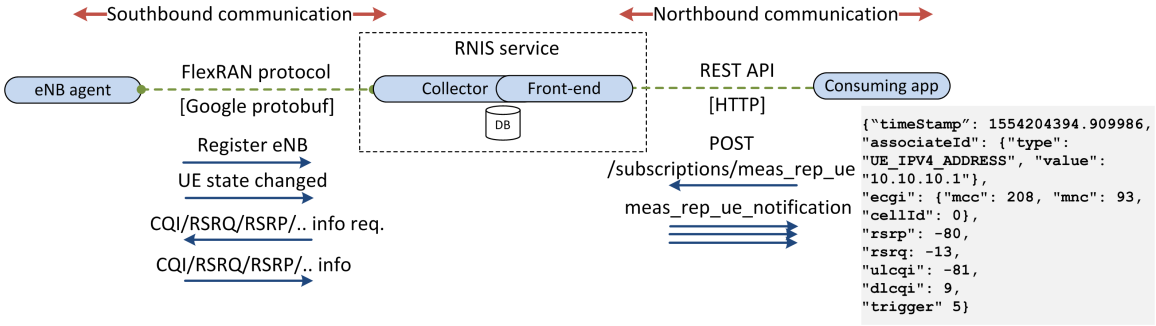


Figure 3.2: Operation of the RNIS provided by our OAI-based MEC platform.

At the southbound interface (between the eNodeB and the MEP), the FlexRAN agent of the eNodeB sends messages in a raw format including several pieces of information on the UE radio context, e.g. CQI, RSRP, and RSRQ. These messages need to be treated and formatted in a JSON format as specified in [19]. OAI-MEP provides a RNIS service to subscribe to all eight different types of notifications, as described above.

The proposed RNIS implementation includes two components, as shown in Figure 3.2. The first component is the *collector*, which receives, parses, and stores the messages

coming from the eNodeBs it manages, and formats them appropriately in JSON as specified in [19]. Every notification has a different message structure. The formatted messages are forwarded to the second component, i.e., the *broker/publisher*, which classifies the messages according to the different filtering criteria and notifies the subscribed MEC applications. The messages can be filtered on a per eNodeB (cell) or on a per UE basis. That is, a MEC application can subscribe to notifications related to an entire cell or a set of UEs. Section 3.3 covers the implementation of the broker/publisher in more detail.

3.3 RNIS message broker implementation

As described in Section 3.2, the RNIS comprises a collector and a broker/publisher component. RNIS broker implementation is done using two different message brokers Apache Kafka and RabbitMQ. The reason for choosing two candidate technologies is their fundamental design difference and implementation, which is reflected in their performance, as it shall be explained in Section 3.4.

3.3.1 RabbitMQ

This is a traditional message queuing system that implements the Advanced Message Queuing Protocol(AMQP)[4] and is built in Erlang. It follows the standards for AMQP 0.9.1 and can also support AMQP 1.0 via a plugin. In RabbitMQ, all the messages first arrive at an *exchange*, which distributes them to different queues based on a routing key or message header value. Once a message arrives in a queue, the RabbitMQ server pushes it to the consumer(s) listening to the queue.

3.3.2 Apache Kafka

This is a distributed streaming platform designed around a distributed commit log. It supports consumer clusters, i.e., running multiple consumer instances in a consumer group. In Kafka, the messages are published according to topics and each topic has multiple partitions. A copy of the message is stored in each partition. (Depending on the replication factor there can be more copies in other Kafka clusters.) Once the messages have arrived in the partition, they can be pulled by the consumer groups, if the latter have subscribed to the particular topic.

3.3.3 Distinctive characteristics

The two candidate technologies have some distinctive differences:

- **Routing Capability**: RabbitMQ provides various exchanges (direct, fan-out, headers, topic) and extensive capabilities for routing the messages (pattern matching, header matching), whereas in Kafka the messages can only be routed according to topics.

[4] https://www.amqp.org/

- **Message Storage**: In Kafka, messages are available even after consumption (depending on the message retention period), which is not the case with RabbitMQ, where messages can only be consumed once.
- **Multiple Consumers**: Kafka supports multiple consumer groups subscribing to the same topic. On the contrary, in RabbitMQ if there are multiple consumers listening to the same queue, then the messages they have subscribed for will be pushed in a round-robin manner.

RabbitMQ is implemented using the header exchange: a message is routed according to its header, which acts like a routing argument. Every subscriber has its own dedicated queue and these queues have new header values for every new subscription. The higher the number of subscriptions, the higher will be the number of routing arguments. Every subscriber has one RabbitMQ consumer instance running locally, on the same machine where the RNIS application is running. As per the ETSI RNIS specification, the messages have to be delivered to the subscribers via the HTTP protocol. A consumer instance sends an HTTP post request to the callback URL of the subscriber, which is provided by the latter at subscription time, together with the rest of the subscription information.

In Kafka, this implementation is slightly different, Messages are routed according to topics. For each subscription, there is one topic and one consumer instance (running locally, as with RabbitMQ) that listens to these topic partitions. This consumer instance belongs to a consumer group (one consumer group for one subscriber). Kafka provides the facility for consumer groups to subscribe to the same topic. Every consumer group maintains an offset value which helps to fetch the messages sequentially or in a random manner. This feature provides the ability to merge similar topics, having similar filtering criteria chosen by subscribers. This reduces the number of topics.

In conclusion, in RabbitMQ there is a single dedicated consumer instance posting the notifications to the subscriber, while in Kafka there are a lot of consumer instances (belonging to the same consumer group) posting the notifications to the subscriber. Here are the final remarks regarding the implementation:

1. Message batching is not considered for any implementation. Messages are posted as soon as they are produced. This provides a near real-time view of the network to the notification subscriber.
2. Both message brokers are used in unacknowledged mode. The producer is not waiting for an acknowledgment from the broker. This is done to improve end-to-end (E2E) latency. It should be noted, that both brokers were found to be reliable in the tests, with no message loss.

3.4 Performance evaluation

The experiments were performed on a host with a 4-core Intel (i5-3470 @3.2 GHZ) CPU, 500 GB hard disk, and 16 GB RAM, running Ubuntu 16.04 LTS. The application was written for Python 2.7.12; pika 0.11.2 and confluent-kafka 0.11.4 are the respective

Python libraries of RabbitMQ (v3.77 with Erlang 21.0.6; standard settings) and Apache Kafka (v2.0.0, Java 1.8.0_181; Java VM settings provided by confluent [30]).

There is one cluster of Apache Kafka and, similarly, only one RabbitMQ broker. The replication and clustering capabilities of Kafka or RabbitMQ were not explored. All applications (RNIS application, broker, subscribers, and message producer) were executed on the same host to avoid the effects of network delays on the results of the measurements. Also, the brokers were given the highest priority on the CPU(s) they were running using the *nice* Linux utility.

To get insight into which broker is more suitable to implement the RNIS, a set of experiments was performed. Experiment results are presented in subsequent sub-sections.

3.4.1 Increasing numbers of subscribers

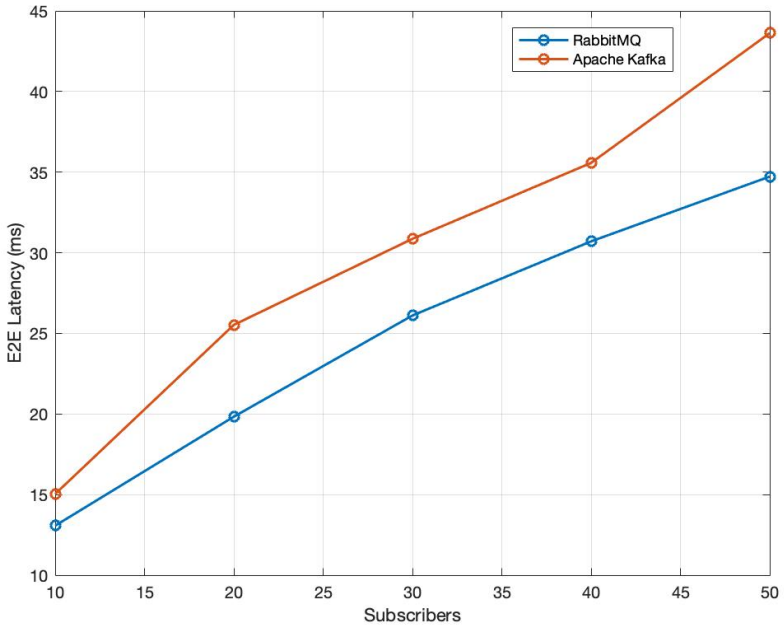


Figure 3.3: Effects on E2E latency with increasing numbers of subscribers. The message rate is set to 10/s; each subscriber has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 10,000 * Number of subscribers.

Considering that every subscriber has subscribed to eight different notifications, this experiment measured the effects of increasing the number of subscribers on E2E latency for both message brokers. E2E latency is defined as the interval from the time instance when specific data to which an application has subscribed are received by the RNIS from the eNodeB over the FlexRAN-based southbound interface (thus generating a publication), until the moment they have been successfully delivered to the consuming application.

Figure 3.3 illustrates the effects on E2E latency with increasing number of subscribers. The average E2E latency for both brokers is less than 50 ms, which indicates that both are suitable for near real-time applications. When the number of subscribers

increases, in RabbitMQ the number of queues starts increasing, which results in increasing workload (replicating messages for every subscriber) for the exchange. The number of routing headers is the same for every subscriber. The number of RabbitMQ consumers posting messages to MEC applications is the same as the number of subscribers. All the subscribers are subscribing to similar eight notifications. In Kafka, this results in eight topics for all the subscribers, and when the number of subscribers increases the consumer groups linearly increase. This results in a growing number of consumer groups on topic partitions. There are eight consumer instances in each consumer group posting the notifications to the subscribers. Therefore, the increased number of consumer instances in Kafka in comparison with RabbitMQ results in lower E2E latency for the latter.

3.4.2 Resource utilization

This sub-section showcases the result of the set of experiments performed to measure resource utilization for the same message production rate, number of subscribers, and number of subscriptions per subscriber[5] and for increasing CPU resources allocated to the broker.

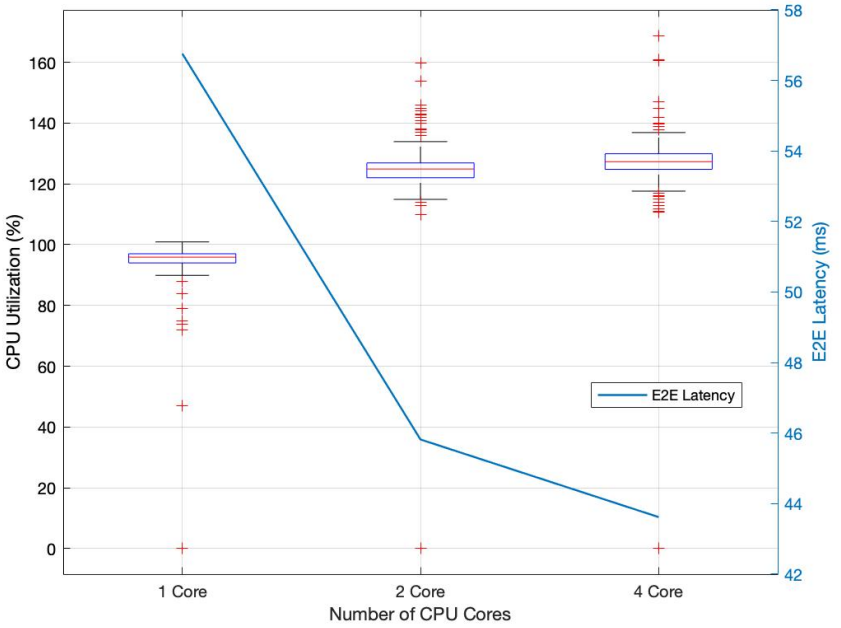
In particular, the number of CPU cores assigned to the broker application was gradually increased from 1 to 3, using the *taskset* Linux utility to pin the broker process to a specific set of CPU cores. As shown in Figure 3.4, Kafka utilizes more resources than RabbitMQ, in part due to the use of Java versus Erlang (for example, how Java handles garbage collection). Second, the message production rate in our experiments is in general kept low; for higher message production rates, it is possible that the curves can deviate. For the experimental settings studied in this work, which are considered realistic, RabbitMQ shows better performance in terms of resource utilization.

It should be further noted that, as expected, E2E latency consistently reduces with an increase in the CPU resources allocated. This result is important for the operator of the RNIS in an NFV environment: Given a specific workload in terms of the number of UEs (which translates to a specific rate/volume of generated RAN level information) and subscribing MEC applications, and under specific E2E latency requirements, the MEC application orchestrator may appropriately (re-)dimension the resources assigned to a virtualized RNIS instance. This way, it can dynamically scale the number of virtual CPUs allocated to an RNIS instance to match the service workload and ensure that it is adequately provisioned to deliver notifications to the subscribed MEC applications in a timely manner, without "overspending" CPU resources.

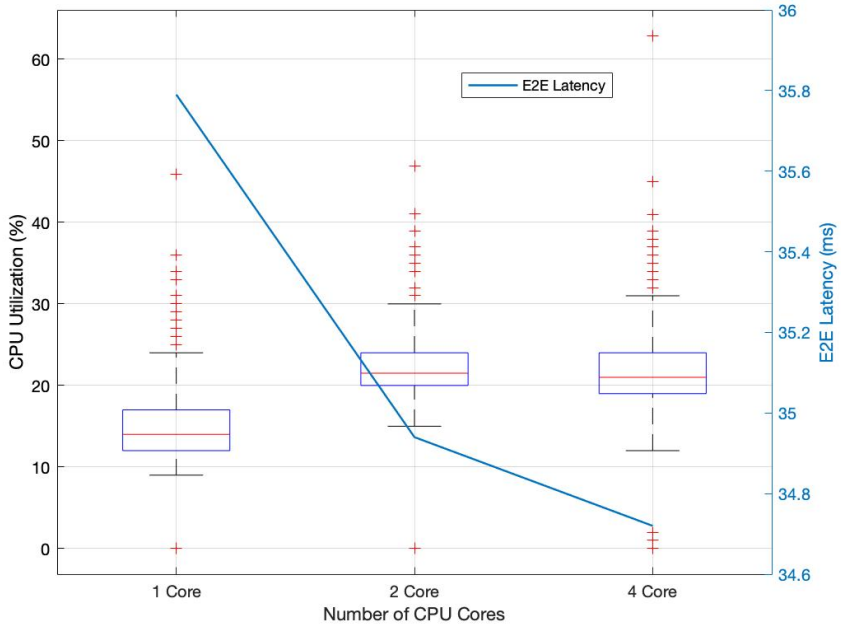
3.4.3 Analysis

For the given message generation rate (1 message/100 ms), the latency of both brokers was below 50 ms, which makes them appropriate for some real-time applications. However, the number of Kafka consumer instances per subscriber is increased compared with RabbitMQ, where there is one consumer instance per subscriber. This is due to the fact that, for Kafka, if a single consumer instance listens to multiple topics at the same time, then the consumption of messages will be very slow. To compensate for that, for

[5]Message rate: 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.



(a) Apache Kafka CPU utilization vs. E2E latency



(b) RabbitMQ CPU utilization vs. E2E latency

Figure 3.4: Apache Kafka and RabbitMQ CPU utilization vs. E2E latency

every subscription there is a single instance that improves on latency at the expense of CPU and memory utilization. Furthermore, a Kafka consumer follows the *pull* model, while in RabbitMQ it is based on the *push* model. Therefore, if many consumer instances are polling simultaneously, the broker has to maintain offsets for all consumers. This increases the processing load on the broker. In summary, RabbitMQ appears to be a better option in our settings.

3.5 Summary

This chapter presented the implementation of a standards-compliant RNIS service on top of an OAI-based MEC platform and experimental results on its latency and resource consumption performance. The chapter targetted a MEC-in-NFV environment, where virtualized MEC platform components, including the RNIS, are to be executed on top of an edge NFVI, without excluding the case for multiple coexisting virtual RNIS instances, belonging to different tenants and authorized to expose different subsets of RAN-level information. The presented results can be used to gain insight into the performance characteristics of the RNIS as a function of the underlying technologies used to implement information delivery, and, importantly, towards dynamically allocating resources to RNIS virtual instances for efficiently providing the RNIS in an on-demand, "as-a-service" manner, satisfying the requirements for timely RAN-level information delivery. Furthermore, the chapter presented a comparison between the performance of two well-known message brokers (i.e., RabbitMQ and Kafka) for publish-subscribe RNIS message delivery. The results advocate for the use of RabbitMQ, being more lightweight and thus appropriate for a MEC context, where compute resources are typically more scarce.

In the next chapter, the ETSI NFV framework will be discussed in detail to understand how container-based VNFs/CNF or MEC applications can be placed on the virtualized infrastructure. Taking into account their cost and availability matrix.

Chapter 4

Placement of Cloud Native Network Services

4.1 Introduction

Container-based applications do not require a full operating system like virtual machines. Making them lightweight and reducing their deployment time. They can be assigned vCPU at a granularity of 1milli CPU, where 1000m CPU is equivalent to 1vCPU. Whereas in VMs, the minimum vCPU that can be assigned is 1vCPU. CNFs can use this finer vCPU assignment to have improved infrastructural resource utilization in comparison to traditional VM-based VNFs.

Network functions can be used alone or along with other network functions to provide network services. The latter are offered by service providers who either owe the infrastructure or lease it from infrastructure providers. While providing a service, they have to abide by service availability which is an important attribute of the Service Level Agreement (SLA). The ETSI NFV group has published specifications and guidelines on the resilience and availability of network functions and network services [31]. They acknowledge higher availability is subjected to higher deployment and management costs. Making it important to find cost-availability trade-offs. This trade-off has always been a challenge for service providers, in maximizing their profits.

A cloud-native network service comprises single or multiple CNFs. To achieve the telco grade 99.999% availability with only single instances of each CNF, as it was for VNF or PNF can be challenging. The cloud-native way to achieve this availability is to have multiple replicas of the CNFs composing a network service. This might lead to over-provisioning of infrastructural resources, which increases the deployment and management costs. Hence, a decision problem arises; How many replicas of each CNF a cloud-native network service needs? Without over-provisioning computational resources to avoid the high cost and provide service availability as promised in the SLA.

This chapter proposes a solution for this decision problem from the perspective of service providers. The existing work focuses on VM-based VNF placement. It does not consider CNFs or address the above problem or consider cost and availability together as an attribute while placing CNFs on cloud infrastructure. To fill this gap, this chapter provides an algorithm to model a simple cloud-native network service.

A simple cloud-native network service requires a single CNF to provide the network service functionality. The proposed algorithm considers two assumptions: first, the service provider knows in advance the computational resources, and importantly vCPU required by the service to serve the user demand; second, the nodes on which CNF replicas will be placed have enough storage and memory resources. To summarize, the chapter will present:

1. Cost and availability model for simple cloud-native network service

2. Dynamic Resource Allocation and Placement (DRAP) algorithm for design and placement of a simple cloud-native network service. DRAP provides a dictionary that contains the number of CNF replicas, placement on the infrastructure node, and vCPU allocation for each replica. It abides by service availability as a constraint.

4.2 Placement of Virtual Network Functions

Most of the existing related work focuses on the resource allocation and placement of traditional VM-based virtual network functions. Authors of [32] have proposed a model for joint vCPU to VM allocation and VM placement considering a simple CDN network service. They have used this model to highlight and address the cost and availability trade-off. The service availability model presented in our thesis is inspired by this work. Authors of [33] provided a queuing theory-based approach to solve traditional VM-based VNF placement and resource assignment problems for a 5G network service. Their presented model considers a service function chain that can be useful for simple as well as complex network services.

Both of these works consider hypervisor-based VNFs, whereas the primary aim of the work presented in this chapter is to use a cloud-native approach for placing network services. Other works related to CNF have focused on different problems. Authors of [34] presented CNF design principles and use cases, and in [35], the authors have presented network service function chaining based on service mesh.

4.3 Simple Cloud Native Network Service

NFVO manages the lifecycle of a network service using a Network Service Descriptor (NSD), which is received from northbound entities such as OSS/BSS or a slice orchestrator in case of Network slicing[1]. The NSD contains details about virtualized and physical network functions, virtual and physical links between them, etc. The NSD is used for modeling, placing, and scaling the instances of VNFs. Based on this, it is considered along with NSD, NFVO receives the maximum vCPU required by the service and the required service availability. The calculation for the maximum vCPU required by a service is out of the scope of the proposed algorithm.

This section presents the model of the proposed simple cloud-native network service and the position of the proposed Dynamic Resource Allocation and Placement (DRAP) algorithm in NFV-MANO architecture.

4.3.1 Modeling a Simple Cloud Native Network Service

In a traditional VM-based network service, the network functions are connected via the Service Function Chain (SFC) concept [36]. Chapter 2 highlights that a network service can have multiple different types of VNFs and PNFs. These VNFs may have replica instances depending on the design of the network service. The ETSI specification for

cloud-native VNF implementation [37] acknowledges redundancy as a possible solution to provide resiliency.

Based on this theory, a cloud-native network service can be modeled as multiple replicas of the same CNF, and the maximum vCPU that the service requires is divided among these replicas. The maximum vCPU consumed by the replicas will not exceed the maximum vCPU required by the service. The service load is distributed between replicas, which provides resilience to the service without over-provisioning the computational resources. This will be discussed in detail in section 4.4.3.

This service is named *Simple Cloud Native Network Service* because there are multiple replicas of the "same kind of CNF". This CNF has one VNFC in a Kubernetes pod. This pod can have single or multiple containers.

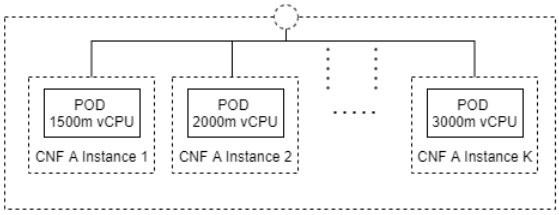


Figure 4.1: Simple cloud-native network service *S* with *K* replicas of CNF *A*

Figure 4.1 depicts a simple cloud-native network service *S* comprising *K* replica instances of CNF *A* with different vCPU allocation. The allocation is performed by the proposed algorithm based on the placement of these replicas on NFVI. It should be noted that a CISM schedules pods based on the resource requirements, whereas the proposed algorithm performs simultaneous scheduling and vCPU allocation.

4.3.2 Deploying a Simple Cloud Native Network Service

In Figure 4.2, the NFVO receives a deployment request for a simple cloud-native network service. The request contains NSD, maximum vCPU required by the service (R_{vCPU}), and required service availability (*RA*). NFVO starts with onboarding CNF images and then the Dynamic Resource Allocation and Placement (DRAP) algorithm will generate the Placement And vCPU Allocation (PACA) dictionary. It sends the dictionary and *RA* to CISM.

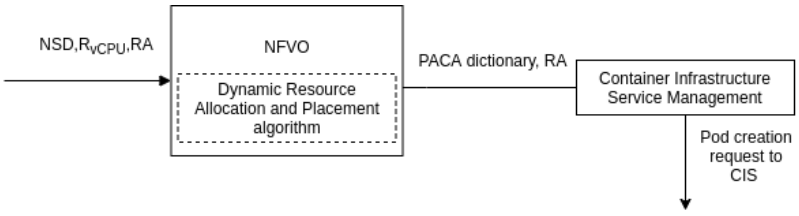


Figure 4.2: Cloud native network service deployment flow diagram

CISM places CNF pod(s) on NFVI and allocates the vCPU as mentioned in the dictionary. Pod placement on NFVI is based on the dictionary. It should be noted, that

generally, CISM performs the initial placement of pods. Here, NFVO is performing the initial placement. To maintain required service availability, CISM can reassign pods to different nodes, and scale horizontally or vertically depending on the information received from the monitoring system of CISM.

4.4 Dynamic Resource Allocation and Placement Model

4.4.1 Preliminary

Now onward the notations highlighted in Table 4.1 will be used. The notations are specific for a simple cloud-native network service S. A pod refers to a CNF instance as depicted in Figure 4.1. Besides the table, $X = (x_{ij})$ is the pod placement matrix, x_{ij} is 1 if pod i is placed on node j otherwise 0. $W = (w_{ij})$ is the vCPU allocation matrix and w_{ij} denotes the vCPU allocated to pod i placed on node j. $\forall i \in [1,K]$ and $\forall j \in [1,N]$.

Table 4.1: Summary of Notations

N	Number of nodes available in NFVI (CIS) for pod placement
R_{vCPU}	Maximum vCPU in milli units required by the service
P_{max}, P_{min}	Maximum and minimum number of pods which can host the service
K	Actual number of pods hosting the service S
min_{vCPU}	Minimum vCPU in milli units which can be assigned to a pod
max_{vCPU}	Maximum vCPU in milli units which can be assigned to a pod
$C(j)$	vCPU capacity of node j in milli units, $\forall j \in [1,N]$
RA	Service availability required by the service
h_j	Failure probability of node j, $\forall j \in [1,N]$
g_i	Failure probability of pod i, $\forall i \in [1,K]$

4.4.2 Cost Model

Consider an array $Y = (y_j)$ and $\forall j \in [1,N]$, where y_j denotes the status of a NFVI node j on which pods can be placed. y_j is 1 when node j is hosting at least one pod. Otherwise, the node is not hosting any pod then 0.

NFVI is a cluster of physical machines providing infrastructural resources (computational, storage, and networking resources). The container infrastructure service instance can be present on VMs or bare-metal. This work considers:

- Bare-metal deployment of CIS
- Each node of the cluster has a fixed cost L when it is hosting pod(s), otherwise, the cost is zero. In a Kubernetes node, this fixed cost can be calculated by computing the computational resources consumed by operating system processes, container runtime engine, and Kubernetes fix components. The cost model can be formulated as shown in Eq. 4.1.

$$D = L * \sum_{j=1}^{N} y_j + R_{vCPU} \tag{4.1}$$

D denotes the deployment cost for a simple cloud-native network service. Generally, in Kubernetes, the CPU is always requested as an absolute quantity[38]. The unit of D is in milli vCPU.

4.4.3 Availability Model

Two types of service availability models are considered depending on the QoS perceived by each user of the service. First, *relax availability* or minimal service model, i.e., at any time, at least one pod is accessible. In this model, the QoS perceived by each user can be degraded and might not be the same as requested in the SLA. This might happen due to the unavailability of some instances and their load being shared among available instances.

Second, the *strict availability model*, which maintains the QoS perceived by each user as requested in the SLA. To achieve this, all the pod(s) and the node(s) hosting the respective pod(s) should be accessible. The following assumptions are applicable for both the models,

- A pod i can fail with probability g_i, independent of the other pod(s) and node(s), irrespective of the load imposed on the pod, and resources allocated to the pod.
- A node j can fail with probability h_j, independent of the other node(s) and pod(s) running on it.

The above probabilities are already known to the service provider as a result of measurement studies or prior experience. A pod may be inaccessible due to its failure or node failure, which is hosting the pod. Pod failures can be correlated due to their dependence on the underlying node(s). This results in defining a correlated group of pods as the pods instantiated on the same node. The availability of a correlated group is subjected to the below models,

4.4.3.1 Relaxed availability model

The availability of a correlated group is subjected to the availability of,

- The node on which the group is hosted,
- At least one pod of the correlated group should be available

The probability that a correlated group j hosted on node j will be available is,

$$a_j = (1 - h_j) * (1 - \prod_{i \in [1,K] | x_{ij}=1} g_i) \tag{4.2}$$

For the service to be available, at least one correlated group should be available. Considering that correlated groups fail independently, the service availability is defined as,

$$\begin{aligned} A(X) &= 1 - \Pr\{\text{All correlated group fail}\} \\ &= (1 - \prod_{j \in [1,N] | \sum_{i=1}^{K} x_{ij}=1} (1 - a_j)) \end{aligned} \tag{4.3}$$

4.4.3.2 Strict availability model

The availability of a correlated group is subjected to the availability of,

- The node on which the group is hosted,
- All the pods of the correlated group should be available

The probability that a correlated group j hosted on node j will be available is,

$$a_j = (1 - h_j) * [\prod_{i \in [1,K] | x_{ij}=1} (1 - g_i)] \tag{4.4}$$

To deliver per-user-perceived QoS as mentioned in the SLA, all the correlated groups should be available. Considering that correlated groups fail independently the service availability is defined as,

$$\begin{aligned} A(X) &= \Pr\{\text{All correlated group are available}\} \\ &= \prod_{j \in [1,N] | \sum_{i=1}^{K} x_{ij}=1} a_j \end{aligned} \tag{4.5}$$

4.4.4 Constraints

Resource allocation and placement are subjected to constraints. These constraints can be categorized as infrastructural level or service level.

4.4.4.1 Infrastructural level constraints

Eq. 4.6 defines the capacity constraint, where the pod(s) hosted on node j can not exceed the available vCPU resources,

$$\sum_{i=1}^{K} w_{ij} \leqslant C(j) * y_j \tag{4.6}$$

Eq. 4.7 defines the pod hosting constraint, where a pod i can only be hosted on one node,

$$\sum_{j=1}^{N} x_{ij} = 1 \tag{4.7}$$

Eq. 4.8 defines vCPU limiting constraints,

$$\begin{aligned} & w_{ij} \leqslant max_{vCPU} * x_{ij} \\ & w_{ij} \geqslant min_{vCPU} * x_{ij} \\ & \forall i \in [1,K], j \in [1,N] \end{aligned} \tag{4.8}$$

4.4.4.2 Service level constraints

Eq. 4.9 defines the provisioning constraint to avoid over-provisioning. Here pod overheads [39] are not considered.

$$\sum_{j=1}^{N} \sum_{i=1}^{K} w_{ij} = R_{vCPU} \tag{4.9}$$

Eq. 4.10 defines the service availability constraint, where the service availability achieved by the placement algorithm should be equal to or higher than the required availability.

$$A(X) \geqslant RA \tag{4.10}$$

4.4.5 Problem Formulation

Service providers aim to minimize the service deployment cost as defined in eq. 4.1 and maintain service availability as per SLA. The variable component of that cost is the number of nodes hosting the service pods. Based on this, the problem can be formulated as Integer Linear Programming (ILP) with an objective to minimize the number of nodes hosting the service pods while considering service availability constraint eq. 4.10.

$$Min \sum_{j=1}^{N} Y_j \tag{4.11}$$

The objective function expressed in eq. 4.11 also considers other constraints mentioned in eq. 4.6, 4.7, 4.8, 4.9. The value of the minimum vCPU that can be allocated to a pod is fixed due to design reasons of the VNFC application running inside a pod. Maximum vCPU is fixed to prevent a pod from consuming vCPU resources allocated to other pods. By fixing these values, the number of the maximum and minimum number of pods hosting the service is fixed.

$$\begin{aligned} & P_{max} = R_{vCPU} / min_{vCPU} \\ & P_{min} = R_{vCPU} / max_{vCPU} \\ & P_{min} \leqslant K \leqslant P_{max} \end{aligned} \tag{4.12}$$

4.5 Heuristic Approach

The proposed problem formulation is similar to the well-known bin packing problem. In a classical bin packing problem [40], the aim is to minimize the number of bins used to fit items of variable volume. These bins have a fixed volume. Whereas in the mentioned problem nodes with variable vCPU resemble bins, and pods with variable sizes resemble items. The classical bin packing problem is NP-hard [40] and there are heuristic algorithms to solve it. In the mentioned problem bin capacity is variable, the number of items is variable with variable capacity and there are additional constraints like service availability. Which results in an NP-hard problem as well. This section presents Dynamic Resource Allocation and Placement (DRAP) a heuristic algorithm, that solves the mentioned problem in polynomial time.

The DRAP algorithm aims to minimize the number of nodes required to place the pods by adjusting the vCPU allocated to each pod. Adjustable vCPU allocation allows the algorithm to increase or decrease the number of pods based on the required service availability.

Input: $r > 0, C(j), N, R_{vCPU}, max_{vCPU}, min_{vCPU}$
Output: K, W, X

$i = 1$
Sort C(j) in decreasing order
while $t \geq min_{vCPU}/max_{vCPU}$ **do**
  for j in [1,N] **do**
    while $C(j) \geq 0$ **do**
      $w_{ij} = min(C(j), t * max_{vCPU})$
      if $w_{ij} \leq min_{vCPU}$ **then**
        break
      end if
      $C(j) = C(j) - w_{ij}$
      $x_{ij} = 1$
      if $sum(W) \geq R_{vCPU}$ **then**
        break
      end if
      if $i \leqslant P_{max}$ **then**
        $i = i + 1$
      else
        break
      end if
    end while
    if $A(X) \geq RA$ **then**
      $K = i$
      return W, X, K
    else
      break
    end if
  end for

28: $t = t - r$
29: reinitialize,K, W, X
30: **end while**

DRAP starts with sorting the nodes in decreasing order of available vCPU capacity. The tuning factor t is used to adjust the size of the pods. PACA dictionary W stores i pod number (pod name), vCPU allocation w_{ij}, and node number j. The tuning rate r iterates by reducing the vCPU allocation from the maximum to the minimum possible. The algorithm iterates over several possible combinations of K, W, and X until it finds the pod placement matrix X that satisfies the availability constraint, or it assigns the minimum possible vCPU min_{vCPU} to the maximum number of pods P_{max} that a service can have. If there is not enough capacity available in the cluster, the algorithm will return no solution.

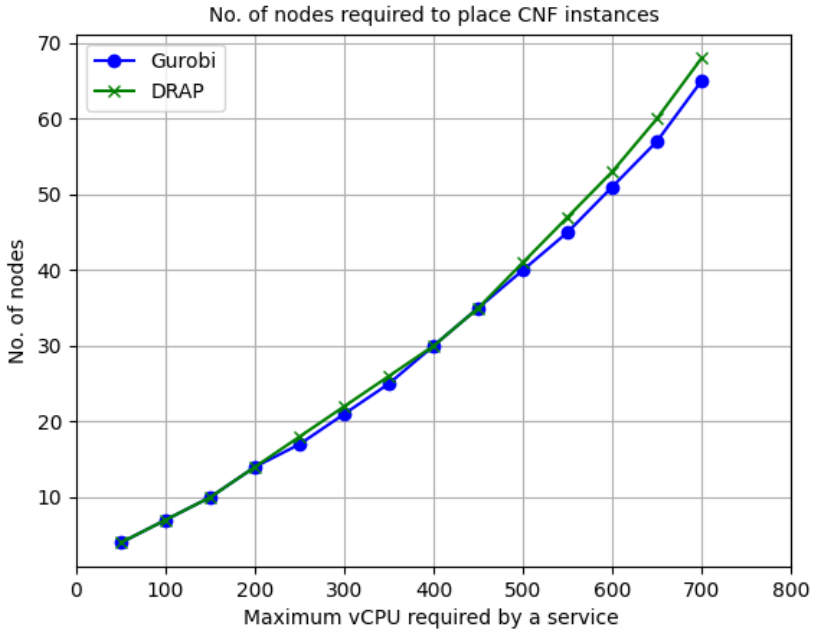
4.6 Performance Evaluation

All the simulations were performed on Intel Core i5-9400F with 6 CPU@2.90GHz and 32GiB of RAM. The academic license of the Gurobi Optimizer was used to solve the ILP stated in Equation 4.11. The node and pod failure probabilities were constant throughout the evaluation process, $h_j = 0.001$, $g_i = 0.001$ $\forall i \in [1,K], j \in [1,N]$. The minimum and maximum vCPU which can be allocated to pods are, 2000m and 4000m vCPU units, respectively.

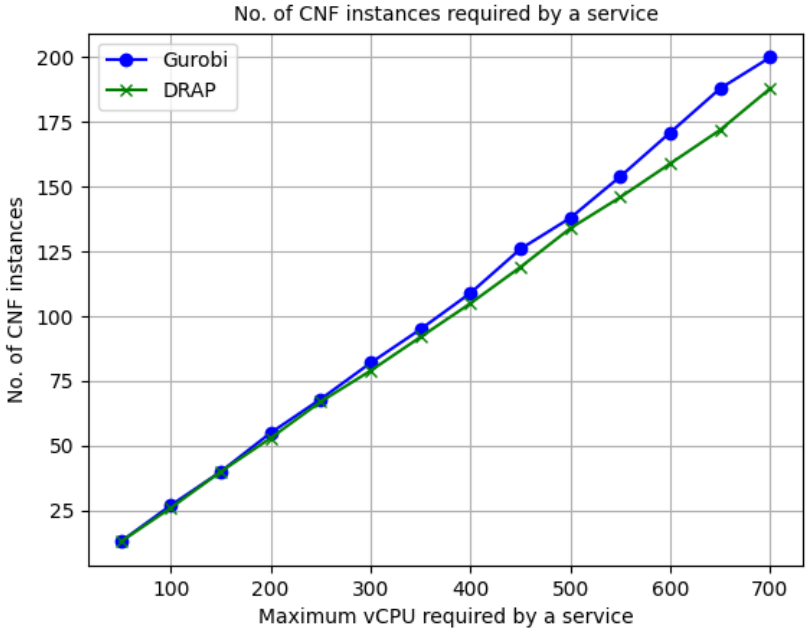
Figure 4.3 depicts the number of nodes and pods required to host CNF instances of different cloud native network services. Each service requires different vCPUs, 50, 100 up to 700vCPUs respectively, and 99.999% service availability. The services were placed on a cluster of 100 nodes, and each node has a capacity between 2 and 16vCPUs selected uniformly at random. The placement of each service was independent of the other services. The cluster nodes had the same vCPU distribution for all the services at the time of placement. The performance of DRAP is close to Gurobi in terms of reducing the number of nodes. Both of them provide similar availability for each service.

Figure 4.4 considers a network service requiring 600vCPUs and 99.999% availability. The minimum and maximum vCPU allocation for pods is fixed to 1000m and 3000m vCPU units, respectively. The service was placed on clusters having 200, 400, to 2000 nodes. Every node of a cluster has 8vCPUs. The sharp increase in Gurobi's execution time justifies that the problem is NP-hard. The heuristic algorithm DRAP provides the solution in a very short amount of time. The time taken by DRAP to provide the PACA dictionary varies between 13ms and 108ms.

Figure 4.5 compares the strict and relaxed availability models defined previously. Five services were placed on a cluster of 50 nodes where each service requires 20, 40 up to 100vCPUs. Each cluster node has a capacity between 2 and 16vCPUs selected uniformly at random. The aim is to achieve maximum service availability, which can be promised by both models. For example, for a service requesting 60 vCPU with DRAP 97.823% of the time there will be no QoS degradation for any user and 99.999% time, service will be available, but some users might be affected. Whereas for Gurobi, these values are 98.019% under strict and 99.999% under the relaxed model. Gurobi and DRAP have nearly similar performances.



(a) Number of nodes required



(b) Number of pods required

Figure 4.3: Number of nodes and pods required by a cloud-native network service

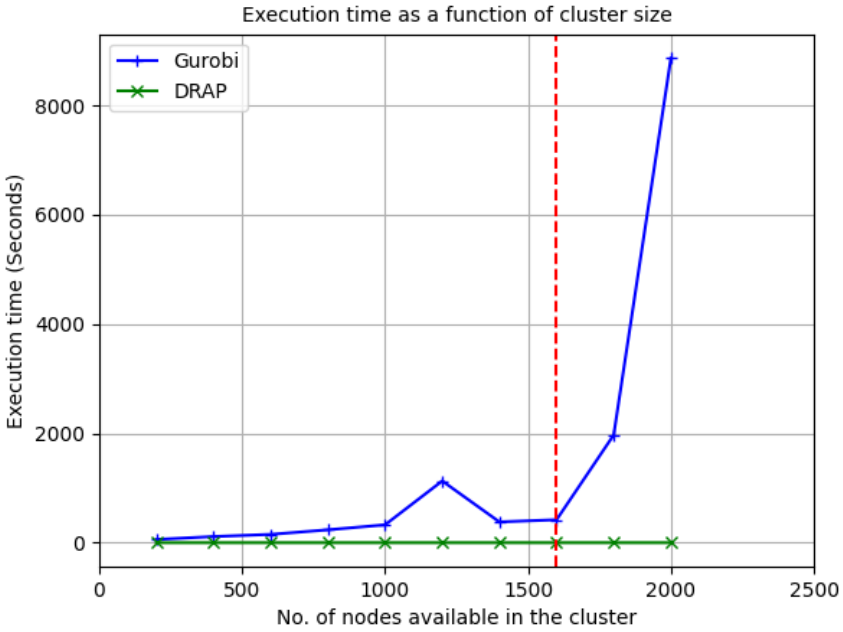
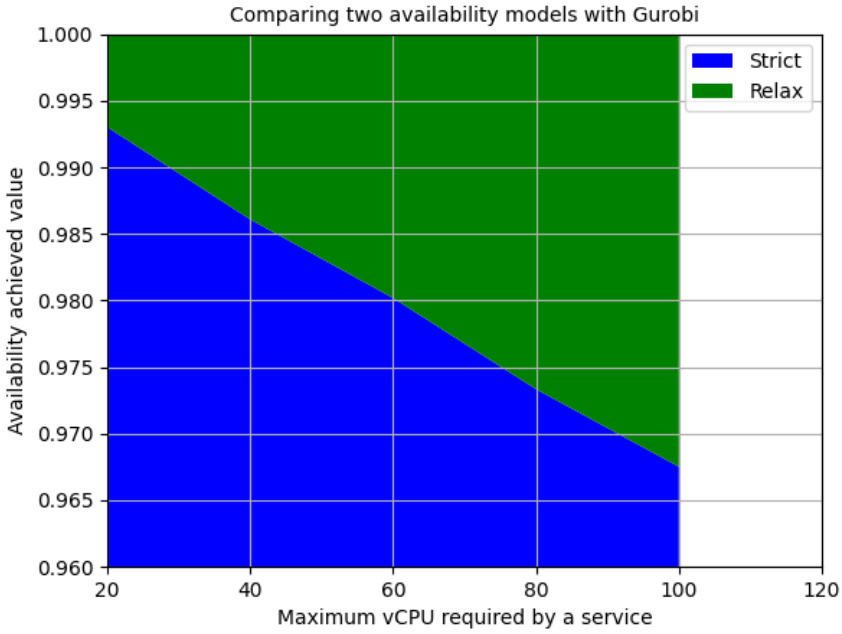
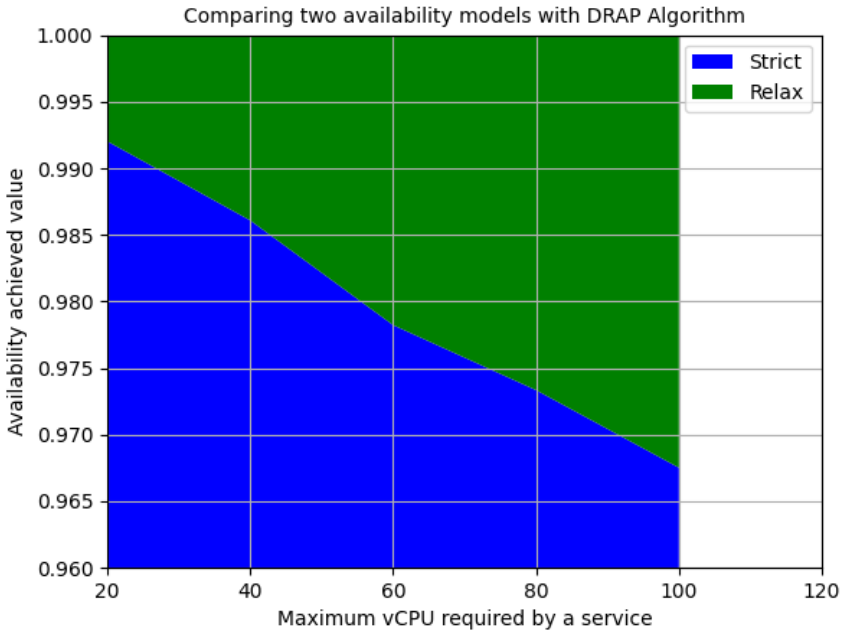


Figure 4.4: Execution time as a function of cluster size



(a) Gurobi



(b) DRAP

Figure 4.5: Comparing two availability models

4.7 Summary

The chapter proposed the use of a cloud-native approach of having multiple replicas without over-provisioning resources to provide resilience. The proposed algorithm benefits from allocating vCPU dynamically and at a finer granularity. It can be considered that by allocating vCPU at a finer granularity node capacity can be efficiently utilized. If a node has a low computational capacity, then the pod requirements can be adjusted. The proposed approach can be beneficial for service providers in reducing their infrastructural costs. The considered service is simple because it has replicas of the same type of CNF. If there is a service that has different types of CNFs and each CNF has replicas, then there can be complications related to affinity, which are not considered in the proposed algorithm.

The next chapter proposes a Lightweight edge Slice Orchestration Framework. This framework is based on the knowledge gathered in this chapter. The placement algorithm presented in this chapter can be used in the framework to place MEC applications on the edge of the network.

Chapter 5

Lightweight edge Slice Orchestration Framework

5.1 Introduction

Chapter 2 discusses how applications hosted at the MEC use traffic redirection or DNS-based redirection rules to steer traffic to MEC Applications instead of the Internet. As the traffic redirection needs to be done dynamically at the instantiation of the MEC application, ETSI MEC defined the traffic redirection as a rule in the AppD describing the MEC application. Besides, the traffic redirection is enforced by the MEC Platform element that acts as an interface between the MEC and 5G domains.

According to 3GPP, when a network slice is deployed, it is known as a Network Slice Instance (NSI). It is composed of one or more Network Slice Subnet Instances (NSSI), which may be dedicated to a NSI or shared among other NSI's. NSSI's contain either VNFs or PNFs or radio resources or transport resources. NSSIs are deployed on top of different technological domains (i.e., radio, edge, transport network, cloud) and are stitched together to build the end-to-end network slice corresponding to a NSI. Usually, a Slice Orchestrator (SO) manages the lifecycle of a NSI, which is composed of four steps: preparation, instantiation and configuration, activation, and decommissioning. Several LCM steps are delegated to the sub-slice orchestrators that manage the technological domain where a NSSI is deployed. In this context, a MEC Application can be considered to be a part of an edge sub-slice (i.e., NSSI), and the MEC orchestrator (MEO) discussed in chapter 2 can be regarded as the edge sub-slice orchestrator.

The well-known existing orchestration frameworks, such as ONAP and OSM, were designed to manage the lifecycle of VNFs or network services, a connected graph of multiple VNFs and PNFs. Currently, none of them perform the lifecycle management of MEC applications. Though they can instantiate a MEC application at the edge, they do not communicate to MEP or provide MEP capabilities to interface with the 5G network and steer traffic to the freshly deployed MEC application instance. Moreover, these frameworks have a complex design and high resource consumption, making their positioning unsuitable for the edge cloud.

This chapter proposes a novel framework for orchestrating and managing the life-cycle of Edge Sub Slices (ESS) that are crucial for low latency demanding services. The framework's architecture is designed following microservices and cloud-native principles discussed in chapter 1. It allows for orchestrating and managing multiple slices at the same time. To summarize, the chapter will present:

1. A cloud-native Lightweight edge Slice Orchestration (LeSO) framework that is specifically designed to orchestrate cloud-native MEC Applications and deploy them as fully isolated edge-sub-slices

2. An Edge Sub-Slice Template (ESST) describing an ESS to manage cloud-native container-based applications following microservices design. The template contains a modified version of ETSI MEC AppD.

The chapter also showcases the placement of the framework in the global network slicing orchestration architecture. Finally, the chapter provides performance evaluation results of the LeSO framework to prove its low computing resource consumption as it is assumed to be deployed at the edge. The performance evaluation results were obtained via experiments as LeSO was deployed in EURECOM 5G trial facility [41] deployed in EURECOM, Sophia Antipolis.

The chapter is organized into 3 major sections: framework and implementation, performance evaluation, and conclusion.

5.2 Lightweight edge Slice Orchestration Framework

This section introduces the LeSO framework and highlights its placement in the global end-to-end network slicing architecture and its design and implementation. It also describes the skeleton of the proposed ESST.

5.2.1 Global Architecture

Network Slice Orchestrator or simply a Slice Orchestrator (SO) communicates with several sub-slice or domain-specific orchestrators, which manage the sub-slice of their own domain. In terms of basic functionality, SO and other sub-slice orchestrators can be considered equivalent to 3GPP Network Slice Management Function (NSMF) and Network Sub Slice Management Function (NSSMF), respectively. Later chapters will discuss in detail the placement of these orchestrators. At the moment it can be considered that these orchestrators are placed in their domain or near to their domain and close to their managed resources.

Each of these domains is responsible for managing different types of network functions (VNFs or PNF) or resources. For example, the edge domain for latency-sensitive network functions or vertical applications, the core domain for core network functions, the transport domain for physical or virtualized transport resources, and the RAN domain for physical or virtualized RAN resources.

Figure 5.1 proposes the placement of the LeSO framework in the global NS LCM architecture. This figure only shows edge and cloud sub-slice orchestrators, but for end-to-end life cycle management, there are also transport and radio sub-slice orchestrators.

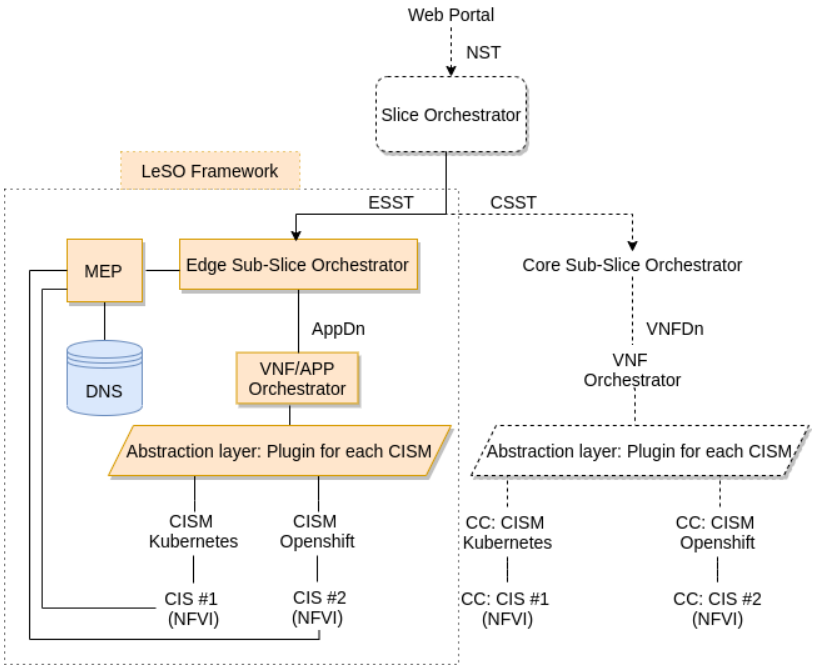


Figure 5.1: Placement of the LeSO Framework in Global NS LCM

5.2.2 Edge Sub-Slice

The framework considers an ESS to contain multiple MEC Applications or CNF or both, not connected or connected to each other. MEC Applications and CNFs consume the services of one another via their own service discovery mechanism if needed. Figure 5.2 shows an example of an edge slice that contains, *MEC App1* which follows microservice architecture divided in *MEC App1:1* and *MEC App1:2*. MEC App2 and CNF are independent.

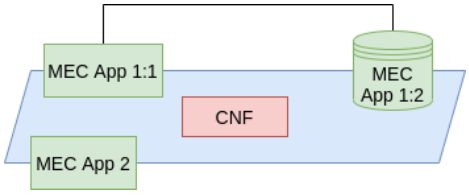


Figure 5.2: Edge Sub Slice Skeleton

The slice of figure 5.2 can be described using the proposed Edge Sub Slice Template (ESST) that contains multiple modified AppDs. Chapter 2, defines the limits of AppD and why it can not be used to describe a microservices-based container application. Figure 5.3 defines a skeleton of ESST.

Multiple AppDs allow defining multiple MEC applications or CNFs connected or not connected to each other. Below are the major modifications proposed in the original AppD to adapt it for the cloud-native world. The definition of the AppD fields remains the same as described in the original AppD,

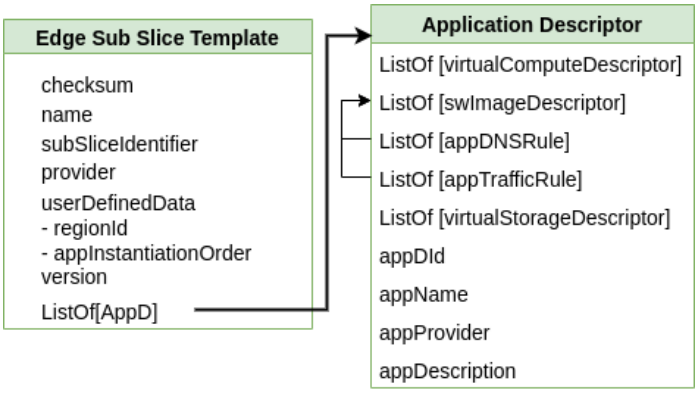


Figure 5.3: Skeleton of Edge Sub-Slice Template

- Array of *swImageDescriptor* instead of single cardinality and use it to define multiple container images and application configuration.
- Included a parameter *configuration* in *swImageDescriptor*, to configure application container specific configuration.
- Included a parameter *port* in *swImageDescriptor*, to define networking ports exposed by the container. This parameter is used to provide the orientation of the exposed ports. It can be either towards:
  - Mobile Network (MN) if the application running inside the container needs to be exposed to MN. This is for edge use cases where user equipment will use the application deployed at the edge via traffic redirection rather than going to the internet.
  - Container Network (CN) for internal communication between AppDs of the same ESST
  - The Internet (IN), for container application to expose its GUI towards the Internet.
- Array of *virtualComputeDescriptor* instead of single cardinality to define resource consumption of each container of a pod.

MEC applications can connect to each other in a cloud-native way by communicating with the other application using its CN-exposed port. The service attached to the exposed port can be accessed using the Pod Fully Qualified Domain Name (FQDN) and the port number. Other important fields of ESST are; (i) **appDInstantiationOrder** which is a list of appDId that describes in which order the AppDs will be treated. This parameter is important for connected AppDs, which have dependencies on other AppDs present in the same ESST. (ii) **regionId** is a list of edge sites where ESST will be created.

The LeSO framework only requires an ESST to handle the lifecycle management of the MEC applications. It does not require any other package or descriptor which are used by other VNF or service orchestrators. ESST is designed to be described by the owners of

the MEC applications, who should be unaware of the underlying platform or infrastructure. ESST is agnostic to the type of platform or infrastructure the MEC Application will run.

5.2.3 LeSO Design

The edge clouds are designed to support latency-sensitive applications, which do not consume high computational resources. The LeSO framework is specifically designed for edge clouds, where computational resources are scarce.

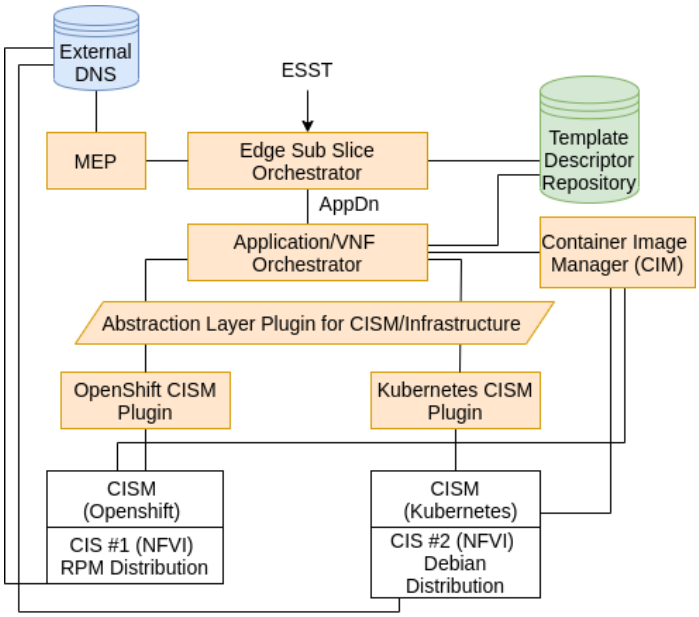


Figure 5.4: Lightweight edge Slice Orchestration Framework

All the components expose a restful northbound Application Programming Interface (API) for communication. The design and deployment follow cloud-native principles. The current version of the framework can deploy MEC Applications or simple CNFs on top of container-based clouds, such as Openshift[1] and Kubernetes. Below is the component description,

- **Edge Sub-Slice Orchestrator (ESSO)**: Manages the lifecycle of ESST and communicates with MEP and SO.
- **Container Infrastructure Service Management (CISM)** and **Container Infrastructure Service (CIS)**: Corresponds to the container orchestration platform and CIS to the infrastructure, respectively. Their functioning is defined in chapter 2.
- **Application/VNF Orchestrator**: Manages the lifecycle of MEC applications or CNFs. It is agnostic to the type of CISM, and it communicates to CISM via the CISM plugin.

[1] https://www.redhat.com/en/technologies/cloud-computing/openshift

- **Template and Descriptor Repository (TDR)**: Database to store ESSTs and application descriptors.
- **MEC Platform**: Performs traffic redirection or DNS redirection by communicating with the 5G core and DNS, respectively.
- **External DNS**: All the CISM have a DNS running in their cluster. The cluster DNS relies on this external DNS to resolve FQDN that is not present in the cluster DNS. This DNS is managed by the MEC platform to perform DNS redirection.
- **CISM Plugin**: This is an abstraction layer that takes the information provided by the application orchestrator and converts it into CISM-specific Yet Another Markup Language (YAML) templates. That is needed by CISM to create objects and workloads.
- **Container Image Manager (CIM)** manages container images. It can pull container images from all the public repositories and make them from the source code present in a git repository. Lastly, it provides the possibility to download container images in tar format from a secure link.

ESSO can handle multiple edge sites, where each site has a specific **regionId**. ESST uses the **regionId** field of ESST to define its placement. ETSI MEC in NFV framework or ETSI NFV-MANO framework proposes to have a dedicated VNF manager (VNFM) for each VNF or MEC application. This results in an extra entity that consumes more computational resources. Instead, the framework proposes an App/VNF Orchestrator that manages the lifecycle of each MEC Application or CNF.

The abstraction layer provides the flexibility to include different CISMs. Northbound of all the CISM plugins is uniform, and their southbound adapts to the CISM API.

5.2.4 Isolation between slices

LeSO framework is slice-aware by creating fully isolated edge slices. All the applications and CNFs of a slice run in an isolated environment, known as a namespace (corresponding to Kubernetes namespace). It provides basic isolation in terms of CISM workloads and object segregation. The computational resource isolation is provided by fixing the amount of vCPU, RAM, and Storage that an application or CNF can use. Network isolation is provided using Openshift and Kubernetes networking policies and their Container Network Interface (CNI). Figure 5.5 depicts isolation between two edge sub-slices.

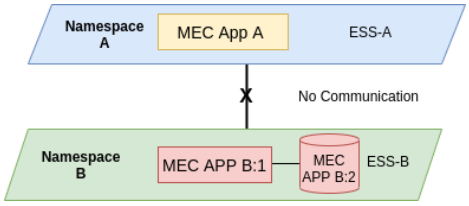


Figure 5.5: Isolation between slices

5.2.5 Working

Northbound API exposed by ESSO can be used by SO or any other entity to perform LCM (Life Cycle Management) on ESST. An ESST goes through three different lifecycle phases, creation phase, modification phase, or deletion phase. These phases have four different stages,

- **On-boarding**: Gathering and reserving resources. In this stage, the application orchestrator gathers all the container images via CIM and reserves the computational and network resources required by each container image of each AppD. The orchestrator defines required CISM objects needed for instantiation and stores them in the repository.
- **Instantiation**: Create the CISM objects defined earlier and instantiate the application described by each AppD in the order described by **appDInstantiationOrder**. Once the ESST is instantiated and if required, it requests the MEP for traffic redirection or DNS redirection.
- **Termination**: Gracefully deleting each application of the AppD in order as described by **appDInstantiationOrder**.
- **Off-boarding**: Removing the stored container images and un-reserve the computational and network resources for each application.

Figure 5.6 shows the phases and stages through which ESST goes during its LCM.

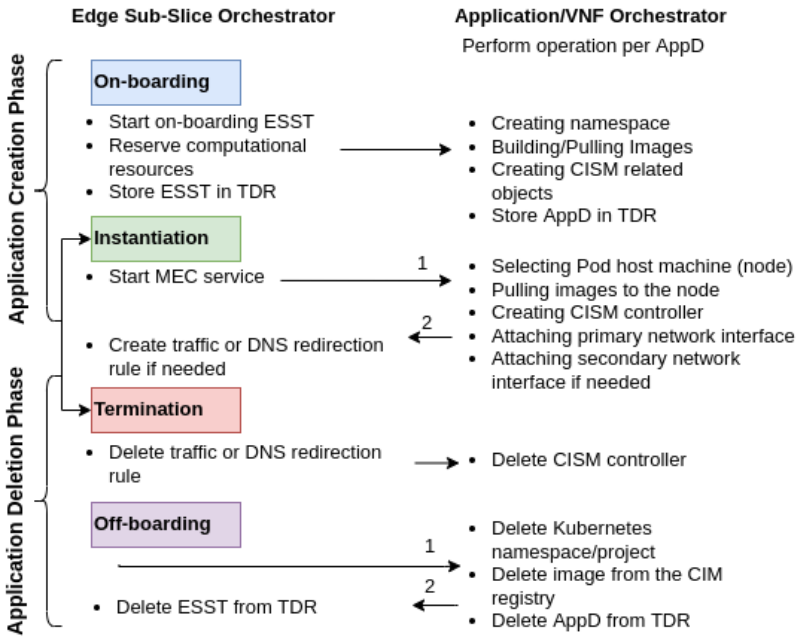


Figure 5.6: LCM of ESST

ESSO via the Application Orchestrator provides the possibility to modify the computational resources required by an application or CNF. However, it results in terminating the application and re-instantiating the new application. This behavior is because it is not possible to change the computational resources allocated to a container at run time. It is considered that the application is able to preserve its state in a persistent volume or store it in a database before a graceful termination. The modification phase includes the termination and instantiation stages.

5.3 Performance Evaluation

At the time of doing performance evaluation, the 5G trial facility deployed in EURECOM had a production-grade Openshift cluster with 7 worker nodes and 3 supervisor nodes to run container-based VNFs. In total, there were 320 CPU Cores and 624 GiB of RAM present in the cluster.

To analyze the resource consumption of the LeSO framework, a real trial scenario was replicated by creating four different edge slices using four ESSTs. Each ESST contains a different number of container images and AppDs. The container images of ESST ES1, ES2, and ES3 were already present in the cluster image repository. Hence their on-boarding time was shorter. One of the images of ESST ES4 was present in the public image repository, and the remaining were present locally in the cluster. Table 5.1 describes the time taken to create and delete each ESST. Table 5.2 describes the time spent on each stage of the LCM of the slice, the runtime of each edge slice was 30 mins.

ESST	Creation Time (s)	Deletion Time (s)	AppD	SwImage
ES1	20.42	10.32	1	6
ES2	15.26	10.32	1	1
ES3	30.52	15.29	2	6,1
ES4	50.43	20.4	3	1,1,1

Table 5.1: Time spent (in seconds) in LCM of 4 slices

ESST	On-boarding Time (s)	Instantiation Time (s)	Termination Time (s)	Off-boarding Time (s)
ES1	15.26	5.16	5.17	5.15
ES2	5.16	10.1	5.16	5.16
ES3	20.17	10.35	10.1	5.19
ES4	25.31	25.12	15.24	5.16

Table 5.2: Time spent (in seconds) in each stage of ESST LCM

Figure 5.7 shows the computational resources consumed by the LeSO framework in a time span of 1 hour when the trial was conducted. CPU core consumption is in milli (m) CPU, 1000m CPU is one CPU core. Black triangles indicate the time at which the slice

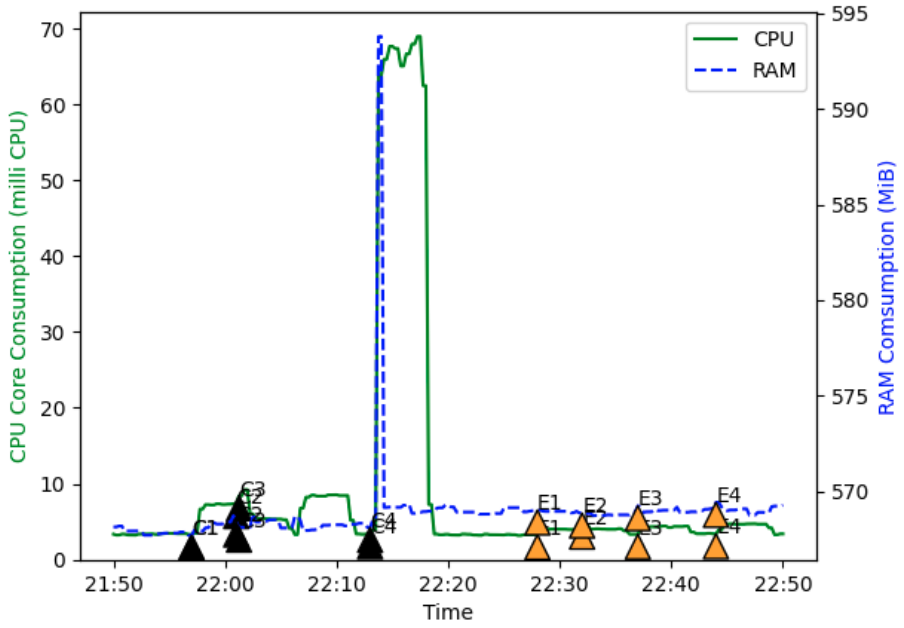


Figure 5.7: Computational Resource Consumption for 4 Edge Slices

creation request was received, and the orange triangle indicates the ending of the slice. The time span between CX and EX indicates the lifespan of a slice, $\forall X \in [1,4]$. The peak in CPU and RAM consumption is due to the use of CIM for pulling an image for ES4 for other slices the image was locally present. CIM is using Podman[2] for image management; hence its resource consumption behavior is not controlled by this framework. From the tables and the figure below analysis can be drawn,

- The composition of ESST affects the time spent in each LCM stage.
- Slice creation is a computationally expensive task than slice deletion.
- Resource consumption before C1 and after 22:20 is the same. Hence, the framework does not consume resources until a new slice creation request or modification request is received.

Multiple slices were created and deleted at the same time to analyze the multi-tenancy of the proposed framework. This allowed understand how the framework handles parallel requests. The slice creation started with 10 slices, and after 30 seconds of runtime, they were deleted. This pattern was continued for handling 10, 20 to 50 edge sub-slices. All the slices used the same ESST A, and the container image was already present in the cluster image repository. The experiments were performed 100 times for each data point, using Monte Carlo simulation. These data points were collected over a period of 4 days.

Figure 5.8 only shows the creation time of these slices. The figure proves that the framework can handle parallel requests and it immediately starts processing the request

[2]https://podman.io/

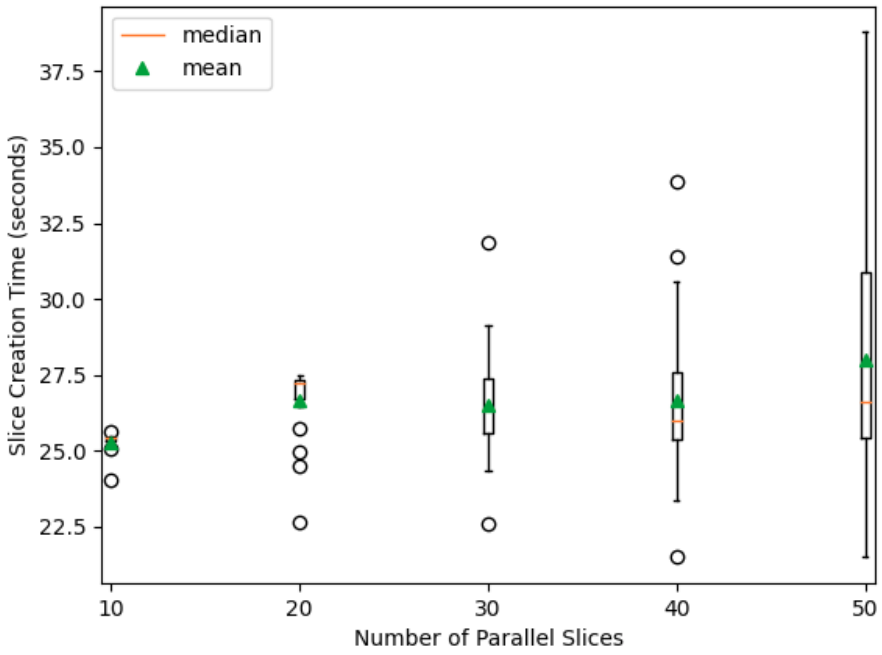


Figure 5.8: Parallel Slice Creation Time

once received. Slice creation is the most time-consuming process where the CISM has to pull or build software images, schedule the CNF, attach a network interface, and assign an IP address. The mean and median time needed to create a slice varies between 25 and 28 seconds on the EURECOM Openshift cluster. The absolute value of creation time depends on the cluster design, its hardware configuration, and ESST design. To create 50 slices, the framework consumed 226m CPU Core and 1.19 GiB memory.

Orchestrator	vCPU	Memory (GiB)
OSM-11	2	6
ONAP (Honolulu)	112	224
LeSO	1	2

Table 5.3: Resource Requirement Comparison

To summarize, the performance analysis LeSO framework requires 1 vCPU and 2 GiB of RAM for deployment and management of Edge Sub-Slices. Table 5.3 compares the resource requirement of LeSO, OSM, and ONAP, referred from [8], [42]. It should be noted that other frameworks require much higher computational resources, and they do not provide the capability to instantiate MEC applications. Hence, the low resource consumption of the LeSO framework makes it suitable to be deployed at the edge cloud and handle the LCM of MEC applications.

5.4 Summary

The chapter introduced LeSO a cloud-native orchestrator that handles LCM of edge slices. LeSO encloses a MEP element that allows dynamic deployment of edge slices. LeSO framework has a nominal resource consumption and requires 1 CPU and 2 GiB of RAM for installation. The multi-tenancy feature allows performing LCM on multiple slices at the same time. The ESST abstracts the platform and infrastructure-related information from the MEC Application providers and allows describing a cloud-native MEC Application which was not possible using the original AppD.

The next chapter will show how the design of an edge application can affect its latency and availability. It will use the AppD proposed in this chapter and will provide a comparative analysis between different deployment models of MEC applications.

Chapter 6

Availability and Latency Aware Deployment of Cloud Native edge Slices

6.1 Introduction

Edge computing is a critical enabler for uRLLC services [43]. The strict latency and availability demands of the application and network functions can be achieved by placing them at the edge of the network. Besides low latency capability, edge computing reduces the amount of traffic to be transported all the way to the central cloud. Indeed, with edge computing, traffic can be treated locally, keeping data privacy in the case of machine learning model training using data collected from sensors and actuators or other elements.

Meanwhile, with the advent of containerization, which led to the emergence of cloud-native principles, applications are no longer deployed as a monolithic block but rather as loosely coupled microservices. In the cloud-native world, each microservice is deployed as a container and managed using container orchestration engines and platforms like Kubernetes. However, edge deployment has still not embraced this trend. Indeed, the key standard of edge computing is ETSI MEC [44]. The ETSI MEC group has issued several specifications covering: application packaging, orchestration, and traffic redirection; but still considering monolithic blocks when deploying applications at the edge. ETSI MEC considers that applications are deployed in VM or containers but one container or VM per application. All the process of orchestration and management relies on this assumption, which is no longer a reality with the emergence of cloud-native orchestration platforms, highly favoring the usage of microservices.

To overcome this situation chapter 5 proposed a new edge slice orchestration framework that allows describing multiple applications using multiple microservices interacting with each other. Chapter 5 also devised a novel template, namely ESST, aiming at defining an edge slice containing multiple applications designed using multiple microservices. This template eases the orchestration of containers and microservices on industry defacto cloud-native orchestration platform Kubernetes.

Basically, to deploy containers Kubernetes uses Pod. The latter is the smallest schedulable entity. It provides an ecosystem for multiple containers to interact. All the Kubernetes-based platforms run containerized microservices in pods. The latency between microservices and their availability depends on whether they are running inside the same pod or different pods and pods are deployed on the same machine or different. The most preferred way to deploy microservices is one pod per microservice i.e., a 1 to 1 mapping between microservice and pod. But, it is also possible to deploy multiple microservices

inside one pod, i.e., 1 pod and N microservice inside it. This can be considered as the default solution adopted in chapter 5. In this context, it is important to understand what is the appropriate solution when considering edge application constraints (i.e., latency, availability, etc.), but also the cost induced by the orchestration and management. The choice of the two possible mappings could be critical if it is not well investigated.

This chapter fills this gap by studying the performance of both solutions in terms of availability, latency, and management cost. The chapter combines analytical models and experimentation to quantify the latency and availability metrics. The edge slice is described using the previously proposed ESST. To summarize the chapter will present:

1. A detailed deployment model for mapping microservices of an edge slice inside one pod and re-modeling the one-to-one deployment model by classifying microservices as critical and non-critical,

2. Markov Chain based availability model for each deployment,

3. Latency analysis between microservices for each deployment model.

The chapter is organized into 4 major sections: deployment models for edge slices, availability modeling, performance evaluation, and summary.

6.2 Deployment Models for edge Slice

In the formation of an ESST, the number of MEC Applications and microservices depends on the edge slice template provider or MEC Application provider. The enablers of edge slice functionality are microservices; together, they deliver MEC Application's desired features. Hence, it can be considered an ESST composed of microservices mapped to containers, and these containers can be placed inside the same pod or different pods. These pods can run on the same machine or different machines.

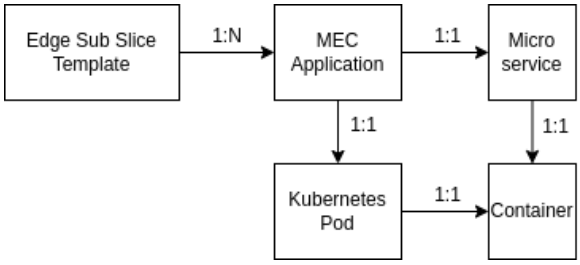


Figure 6.1: 1 to 1 Mapping of MEC Applications and Microservices

Based on this, two different deployment models for an ESST with N microservices can be considered. Figure 6.1 shows a 1 to 1 mapping between MEC Application and microservices, one container per pod. The edge slice will have N MEC applications or pods and N microservices or containers. Figure 6.2 shows 1 to N mapping between MEC Application and microservices, i.e., N containers in one pod. The slice will have 1 MEC application or pod and N microservices or containers.

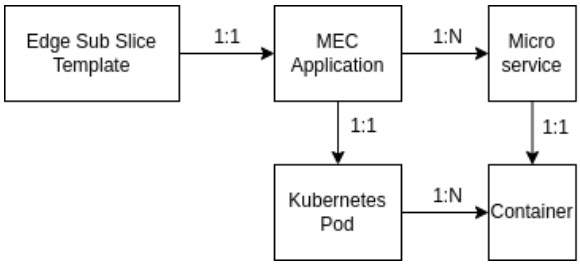


Figure 6.2: 1 to *N* Mapping of MEC Application and Micro-services

The following subsections will analyze and discuss each deployment model, its benefits, and drawbacks considering critical performance criteria: (i) Orchestration and management, (ii) Availability, and (iii) Low-Latency.

6.2.1 Orchestration and Management

The process of orchestration and management is one of the key functions in the context of containerization. It concerns the Life Cycle Management (LCM) of a network slice and hence all the applications that constitute the network slice. The most time-consuming task in the LCM of a slice is slice creation. It involves fetching container images of the microservices, scheduling, computational and storage resource allocation, IP address allocation, instantiation, etc.

The number of container images remains the same for both deployment models. Containers mostly consume the computational and storage resources, and in both deployment models, the number of containers remains the same, so the computational and storage resource consumption will be the same. However, if there is a significant pod overhead[1] then the 1 to N deployment model will use fewer resources than 1 to 1. Each pod is allocated an IP address; hence 1 to 1 deployment model will have N IP addresses, whereas 1 to N will have only one IP address.

Pod scheduling and instantiation are the two most critical stages. They result in variable creation times for both deployment models. In pod scheduling, the scheduler has to look for host machines that have enough computational resources for these pods. In the 1 to N model, all the containers stay in the same pod. Hence, if there is no such machine with the requested computational resources, it will result in scheduling failure. In contrast, the 1 to 1 model can schedule pods on different nodes; hence the computational resources can be utilized efficiently. Once the pods are scheduled, their Cgroups, network interfaces, etc., are created. This step's time depends on whether the pods are instantiated in parallel or serial (if pods require a particular order of instantiation). The containers inside a pod only start parallel. Thus, if the pods are started serially, the 1 to N deployment takes less time, and if pods are started in parallel, the 1 to 1 deployment takes less time.

[1] https://kubernetes.io/docs/concepts/scheduling-eviction/pod-overhead/

6.2.2 Availability

Availability is a critical component for any type of telecom and vertical service, especially when the service belongs to the uRLLC category. It measures the length of time a system or network is functioning. In carrier-grade systems, the availability should ensure the 5 nine uptime, i.e., 99.999% available. To ensure availability in cloud-native systems, it is important to reduce downtime and guarantee that all microservices, and hence pods, are active. The different reasons that lead to reduced availability are

- The container running the microservice died due to an internal error (software bug, high resource consumption, etc.) in the microservice.
- The pod died, and hence all the containers inside the pod also died.
- The node hosting the pod died due to hardware or software errors.

The reasons for the above failure can be voluntary or involuntary. In case of a node failure, pods with their containers are immediately scheduled on another node. Therefore, a slice will be disrupted for the time the pods are getting instantiated.

If we consider the first deployment model, i.e., 1 to 1, we expect better availability. Indeed, if the container or the pod dies, only one microservice dies, and the time to reboot 1 pod with 1 container is very short. However, the pod needs to be re-scheduled on another node if the node dies. If another node is available, then the time to reboot the pod is short. Otherwise, the downtime can be higher. In the second deployment model (1 to N), when a pod dies, all the microservices running inside the pod die with it. This means that the time needed to reboot the pod with all its containers depends on the number of containers to reboot. In the 1 to 1 deployment model, the pods can be scheduled across different nodes; even if some pods are not available, the edge slice will be partially available. Whereas, in the other model, if there is any disruption, the edge slice will not be available completely; all the containers are running in the same pod.

6.2.3 Latency

As stated earlier, latency is the critical KPI that motivates the deployment of cloud-native microservices at the edge. In cloud-native and virtualized systems, it is important to distinguish the communication latency among the microservices and the service latency (i.e., the collective functionality of all the microservices). The service latency is composed of inter-microservice latency and microservices processing time. Service latency may increase due to downtime if a container, a pod, or a node dies. Regarding the communication latency, it can be assumed that the 1 to N model will achieve the best performance as all the microservices are inside the pod and use the pod's loopback interface or IPC via Unix sockets that ensure merely instantaneous communications among microservices.

In contrast, the 1 to 1 deployment model may have a communication overhead, particularly if pods are deployed on different nodes, which requires packets to traverse through tunnels, which negatively impacts communication latency. One solution that can improve the performance of the 1 to 1 model is to use a placement algorithm that ensures

that all pods are grouped in the same node, which avoids using tunnels and hence reduces communication latency. If the node hosting all these pods dies, the whole slice will not be functional, impacting the availability strongly. Another solution would be to use special Container Network Interfaces (CNIs) that use Data Plane Development Kit (DPDK)[2] in combination with Single-root input/output virtualization (SR-IOV) to improve latency and throughput metrics. Most edge providers may choose this option to enable low-latency demanding services.

Concerning the service latency, and as discussed in the section 6.2.2, the 1 to 1 deployment model should achieve better results in comparison to 1 to N as it minimizes the downtime.

6.3 Modeling Availability

To evaluate the availability of both deployment models, the proposal is to model them using Markov Chains. Let us assume that:

- An ESST is composed of N microservices
- Each microservice in a slice is categorized into critical and non-critical. The non-critical ones provide extra functionality/features to the slice. The critical ones are responsible for the primary/principal functionality of the slice.
- If a non-critical microservice dies, the edge slice will be partially disrupted. Indeed, it can still work with limited functionality. But, if a critical microservice dies, the edge slice will be completely disrupted.
- The microservices have no software-related bugs, and the computational resources needed by the microservices are properly allocated to their container. This avoids the failure of the microservice container due to internal errors. Only pod failure enclosing the microservice container will be considered.
- It is assumed that the failure rate of pods and the recreation rate of pods follow an exponential distribution with rates of f and r, respectively.

Let us denote M and K by the number of non-critical and critical microservices, respectively. Here, $N = M + K$.

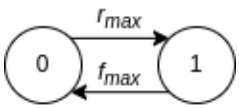


Figure 6.3: Markov chain for 1 to N deployment model

Figure 6.3 represents the slice deployed using the 1 to N deployment model, where N microservices or containers are inside one pod. Here, due to the compact nature of the

[2]https://www.dpdk.org/

deployment, if the pod fails, all the containers will fail. Besides, there is no differentiation between critical and non-critical microservices. It is assumed that the recreation period of a pod is composed of the time needed to recreate all the containers, which can be modeled using an exponential distribution with a rate r_{max}. Similarly, the failure rate can be modeled using an exponential distribution f_{max}. Therefore, the system is modeled with a two-state Markov chain $X = X(t), t \geqslant 0$ on the two states 0 and 1; where 0 indicates that the system is in failure, while 1 means that the system is fully available. A transition between state 0 to 1 with the rate r_{max} indicates that the pod is recreated, and a transition between state 1 to 0 with the rate f_{max} indicates that the system has failed. Figure 6.3 illustrates the transition graph.

In this scenario, the time a slice with N microservices will be available corresponds to the probability to be in state $S = 1$ denoted by,

$$A = r_{max}/(r_{max} + f_{max}) \tag{6.1}$$

Regarding the other deployment model 1 to 1, the Markov chain $X = X(t), t \geqslant 0$ is defined on the state space S defined by $S = \{(m,k) | m = 0, ..., M \text{ and } k = 0, \ldots, K\}$, for every $K \geqslant 1$. In this model, $X(t) = (m,k)$ indicates that at time t, there are m active non-critical microservices and k active critical microservices. While $s = (0,0)$ indicates that all the pods are down, $s = (M,K)$ indicates that all pods are healthy and work properly. Figure 6.4 illustrates the transitions graph of the envisioned system.

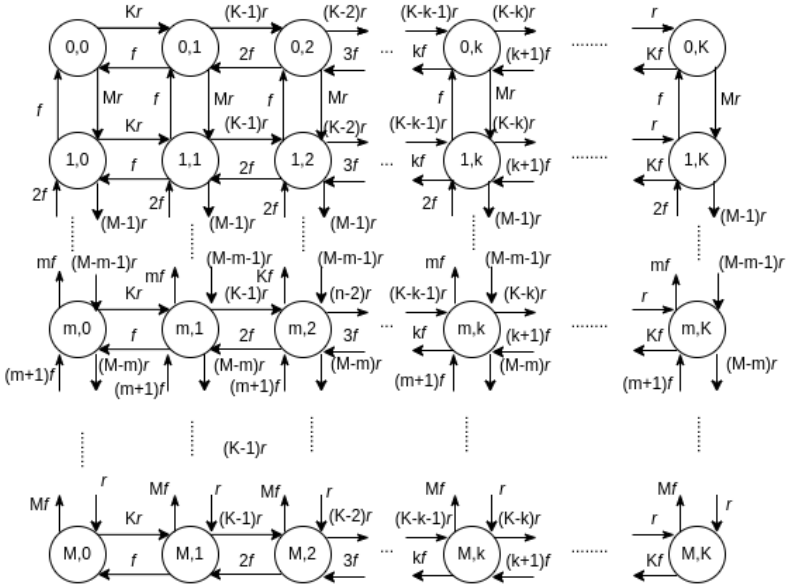


Figure 6.4: Markov Chain for 1 to 1 deployment model

- If a non-critical pod gets recreated while already m $(0 \leqslant m \leqslant M-1)$ are active and k critical containers are active then there is a transition from state (m,k) to state $(m+1,k)$ with rate $(M-m)r$.

- If a critical pod gets recreated while already k $(0 \leqslant k \leqslant K-1)$ are active and m non-critical containers are active then there is a transition from state (m,k) to state $(m,k+1)$ with rate $(K-k)r$.
- If a non-critical pod fails when m $(0 \leqslant m \leqslant M-1)$ are active and k critical containers are active then there is a transition from state (m,k) to $(m-1,k)$ with rate $(m)f$.
- If a critical pod fails when k $(0 \leqslant k \leqslant K-1)$ are active and m non-critical containers are active then there is a transition from state (m,k) to $(m,k-1)$ with rate $(k)f$.

Let Q be the infinitesimal generator matrix for the chain. Each entry q_{mk} such as $s_m, s_k \in S$ and $m \neq k$ of the matrix corresponds to the instantaneous transition rate from state m to state k. Diagonal entries are chosen to ensure null rows of Q, i.e.:

$$q_{mm} = -\sum q_{mk}, s_k \in S, m \neq k \tag{6.2}$$

The objective is to analyze the system in the long run; i.e., inter-events time is neglected over the running time of the system. It matches the steady-state behavior of the analyzed system. $\forall s_m \in S$, we note $\pi_m = \pi_{mk} = \lim_{t\to\infty} P\{s(m,k)\}, m \in (0,M), k \in (0,K), M+K=N$, the stationary probability distribution of the chain. The Markov chain of 1 to 1 is a homogeneous, finite, and irreducible process. In the steady state of the system, we assume that the total probability flux out of a state is equal to the total probability flux into the state. For a state $s_m \in S$:

$$\pi_m * \sum_{s_k \in S, m \neq k} q_{mk} = \sum_{s_k \in S, m \neq k} \pi_k * q_{mk} \tag{6.3}$$

Let π be the vector containing all model states. By combining 6.2 and 6.3, the below equation can be formulated:

$$\pi * Q = 0 \tag{6.4}$$

The normalization condition of the chain is:

$$\sum_{s_k \in S} \pi_m = 1 \tag{6.5}$$

Solving global balance and normalization condition equations 6.4 and 6.5 leads to determining vector π. Getting the steady-state probabilities will allow determining the probability of having m non-critical pods and k critical pods active. This, in turn, will help to derive the availability of the 1 to 1 deployment model.

The slice applications are fully available when it is in the state $s = (M,K)$ i.e., all pods are running, it is denoted by A_{full}

$$A_{full} = P(s(M,K)) \tag{6.6}$$

The slice applications are available with limited capabilities/functionalities when it is in the state $s = (m,K), \forall m \in (0,M)$

$$A_{limited} = \sum_{m=0}^{M} P(s(m,K)) \tag{6.7}$$

In special scenarios where it is not possible to distinguish between critical and non-critical, i.e., all microservices are critical, we can use the Markov chain corresponding to the 1 to N model. The slice will be available when all the pods are running.

6.4 Performance Evaluation

Latency and availability KPIs are highly dependent on the design of the infrastructure, computational resources of the node hosting pods, and connectivity between the nodes. To evaluate the performance of the deployment model irrespective of the infrastructure, two managed Kubernetes cloud environments were chosen to analyze and evaluate the proposed deployment models. The testbed1 is a 5G trial facility [41] deployed in EURECOM. The testbed is using Openshift SDN-based CNI. The testbed2 is a managed Kubernetes service rented from public cloud OVH. It has 5 VM-based worker nodes with 2 vCPU and 7 GiB of RAM connected via Canal CNI[3].

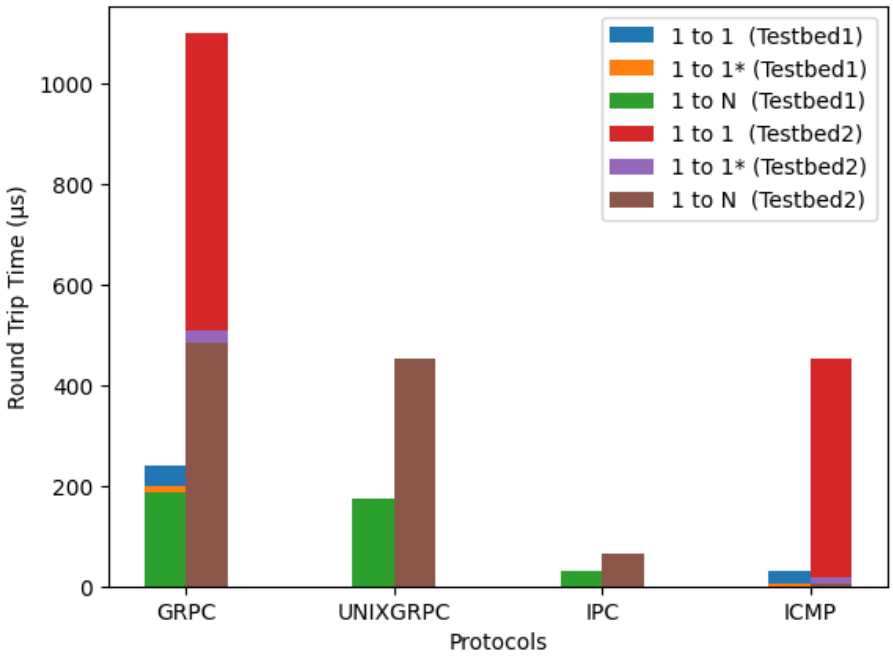


Figure 6.5: Inter Microservice RTT (μS) for different models and testbeds

Figure 6.5 shows the Round Trip Time (RTT) to understand the latency between microservices when they are deployed with two different models. Several protocols were used, Google Remote Procedure Call (GRPC), one of the most used protocols in microservices architecture, IPC over Unix socket, and GRPC over Unix socket. The unix socket-based communication can only happen in a 1 to N type of deployment model as the microservices

[3]https://github.com/projectcalico/canal

share the same network namespace. The * in the 1 to 1 deployment model depicts communication between microservices deployed on the same machine and without * on different machines. ICMP ping results are only added for the readers who specifically use ping as a latency metric. The below conclusion can be drawn from Figure 6.5:

- Different RTT values in two testbeds are due to different CNIs, computational, and network resources associated with the cluster.
- In both the testbeds 1 to N deployment model has the lowest RTT as it uses a loopback interface or UNIX sockets. Accordingly, it is better to use IPC based for latency-sensitive applications.

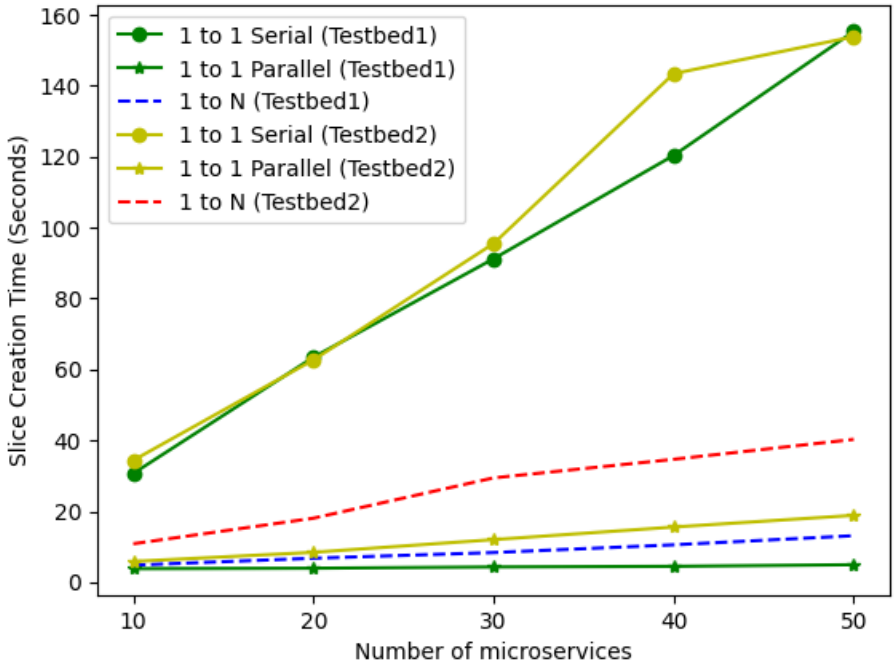


Figure 6.6: Slice Creation Time (S) for different models and Testbeds

Slice with 10, 20 to 50 microservices were created on both testbeds to evaluate the slice creation time. The same container image was used for all the microservices and the resource requirement was 10 milli CPU and 100MB RAM. The slice with 1 to 1 deployment model can be created with or without dependency between the microservices, i.e., serial or parallel pod creation. Figure 6.6 shows the slice creation time. It can be concluded that:

- 1 to 1 model takes less time when the microservices do not have any dependency among them, i.e., pods are created parallelly.
- 1 to N deployment model in testbed1 takes less time than 1 to 1 in testbed2. This behavior is due to the different computational capacities of the two clusters.

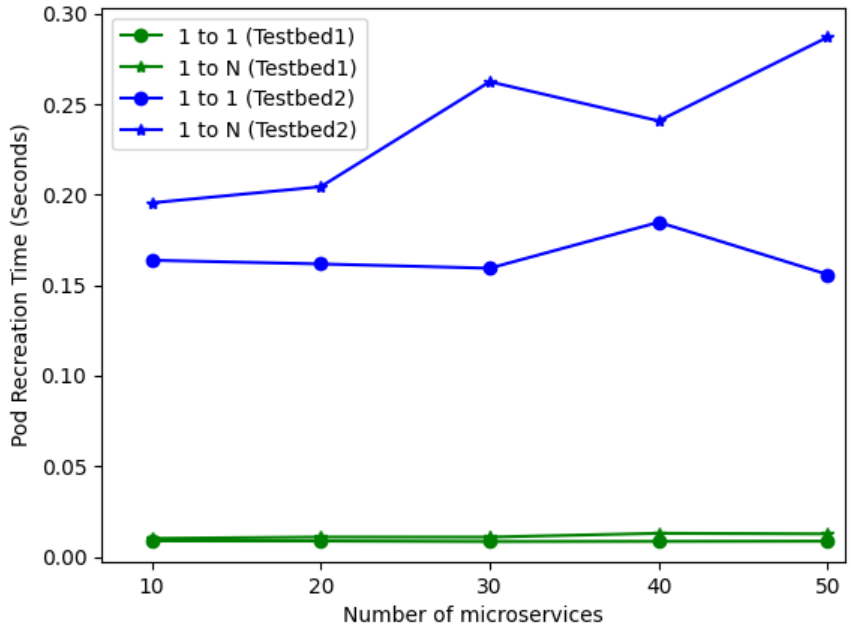


Figure 6.7: Pod Recreation Time for different models and Testbeds

The pods were deployed using the Kubernetes deployment controller, which watches for pod availability. If the pod is not running, it immediately creates a new pod. The pods were forcibly deleted to simulate the pod recreation in the 1 to 1 and 1 to N deployment models. Figure 6.7 shows the average pod recreation time with the increasing number of microservices. It is assumed that microservices are again available once the pods are recreated.

These values of pod recreation time are used to estimate the availability for a slice with 10 and 50 microservices deployed with 1 to 1 and 1 to N deployment models. It is assumed that the pod failure rate is 100 times in a year, and it can happen at any point in time. Table 6.1, shows the full availability of 1 to 1 and 1 to N deployment models using the formula derived via solving the Markov Chains, equation 6.6 and 6.1 respectively. To correlate the availability with a combination of critical and non-critical microservices, scenario II and III depicts two different combinations for a slice with 50 microservices.

Table 6.1: Availability of a Slice with 10 and 50 Microservices

Scenario	Testbed	Critical	Non-Critical	1 to 1 (Full)	1 to N
I	Testbed1	5	5	99.9998%	99.9999%
I	Testbed2	5	5	99.997%	99.9992%
II	Testbed1	25	25	99.999%	99.9999 %
II	Testbed2	25	25	99.9856%	99.9989%
III	Testbed1	40	10	99.9986%	99.9999%
III	Testbed2	40	10	99.9765%	99.9989%

Following conclusions can be drawn from Figure 6.7 and Table 6.1,

- Computational resources of the testbed affect the pod recreation time, which indeed affects the slice availability
- Scenarios II and III, depict the effect of critical microservices on the slice availability. If the majority of the slice microservices are critical, then there are more chances of a critical microservice failing. Hence, the availability will be reduced.

The 1 to N deployment model has higher availability than the 1 to 1 model, but this is highly subjective to the availability of a node with required computational requirements. 1 to 1 deployment has an advantage over 1 to N, which is the limited availability. If a non-critical microservice fails, the slice will still be available with limited functionality. In contrast, in 1 to N deployment, the slice will be completely non-functional until a suitable node is found.

6.5 Summary

This chapter presented a novel methodology to model a cloud-native network slice defined with microservices. The two deployment models and the presented approach of classifying microservices into critical and non-critical are suitable for different types of applications. Both deployment models are useful in serving different purposes. The 1 to N deployment model promises low latency communication and high availability with the condition that the computational resources required by the microservices are available all the time in one of the cluster nodes. It might not be the case all the time. In the 1 to 1 deployment model, availability is subjected to the application's design in terms of the number of critical and non-critical microservices. This model efficiently utilizes the cluster's computational resources by allowing microservices to spread across different cluster nodes. It is suitable to use this model for applications prioritizing availability over low latency and can be categorized into critical and non-critical microservices.

The next chapter will showcase a multi-domain network slice orchestrator. The edge sub-slice designed in this chapter can be instantiated using the multi-domain network slice orchestrator.

Chapter 7

Cloud Native Lightweight Slice Orchestration Framework

7.1 Introduction

The previous chapters have iterated that NFV is the key enabler of Network slicing in 5G and beyond networks. It enables hosting multiple communication services on top of the same physical infrastructure without compromising their QoS. Chapter 1 section 1.3 explains multi-domain slice orchestration. Handling a network slice that spreads across different technological domains is a difficult task. Indeed, a network slice is divided into sub-slices, which are managed by domain-specific service orchestrators. These orchestrators translate domain-specific Service Level Objectives (SLOs) to resource-level objectives and forward them to resource orchestrators. The resource orchestrators manage the infrastructure for network functions. The job of a network slice orchestrator is to manage interactions with sub-slice/service orchestrators and deliver a network slice.

Sub-slice or service orchestrators are responsible for handling the network function's lifecycle. The transition from VM-based VNFs to CNFs has created new challenges for service orchestrators. One such challenge is following cloud-native application design principles, as discussed in chapter 1, containerization, microservice, and on-demand scaling. Containerization is the process of packaging software in containers. It simplifies the software packaging, orchestration mechanism, and reduces software deployment time with lower computational cost as compared to VMs. All these benefits of using containers amplify when fused with cloud-native principles. Hence, the transition from VNF to CNF requires re-designing VNFs and service orchestrators. Whereas, as discussed in chapter 2 the recent release of ETSI NFV-MANO [45] adds another layer of components to orchestrate containers alongside VMs. Rather than re-designing the service orchestration framework to fully support containers. This approach complicates the VNF descriptors and hides the simplicity of using containers. However, ETSI MEC [46] specifications for orchestrating applications at the edge of the network, still do not clearly mention container-based MEC applications.

The well-known service orchestration frameworks, ONAP and OSM [8] followed the same path as ETSI-NFV MANO and added the support for CNF orchestration to their legacy orchestration mechanism. Leading to a complicated workflow and multiple redundant components in the service orchestration framework to support both containers and VMs. In addition, these orchestrators can not orchestrate MEC applications or provide MEP services. These frameworks do not provide a clear mechanism for orchestrating isolated network slices and ONAP demands high computational and networking resources. Hence, this inability to orchestrate MEC applications, high resource consumption, complicated service

orchestration workflow, and missing isolation between network slices is the inspiration behind proposing an end-to-end network slice orchestration framework based on cloud-native design principles.

Cloud-native Lightweight Slice Orchestration (CLiSO) framework is designed to handle RAN, Core Network, and MEC domains. The framework can be deployed in resource constraint environments due to its lean design and low resource consumption. The framework allows hosting CNFs on public, private, or hybrid clouds. This proposed framework is an extension of the work presented in chapter 5, the Lightweight edge Slice Orchestration (LeSO) framework. The previous framework only focused on the MEC domain, whereas the new framework adds additional support for RAN and Core Network domains.

On the other hand, Zero-touch Service Management (ZSM) [47] plays an important role in enabling self-managed services. ZSM allows each service orchestrator to perform a closed-loop automation and heal their services in case of errors. Existing orchestrators or ETSI ZSM specification provides a notion of ZSM via state-of-the-art machine learning and artificial intelligence algorithms. Whereas, such a solution would entirely rely on the intelligence of the service orchestrator and could result in a single point of failure.

To address this challenge the chapter proposes Domain Specific Handlers (DSHs), which is a management network function designed to manage the lifecycle of one or many network functions via communicating with service orchestrator's APIs. Service orchestrators deploy DSH as a CNF and are responsible for managing DSH's health. However, DSH communicates with service orchestrators using a dedicated interface to manage the health and resource consumption of its network functions. This will allow offloading sub-slice operational management from service orchestrators to DSH. DSH may contain vendor-specific logic to handle its network functions. This provides the freedom to customize network function life cycle management rather than relying on the service orchestrator's generic mechanism to handle all the network functions.

Finally, management of network slices enabling mission-critical services requires dynamic management of infrastructure resources. For example, a fleet of drones acting as edge servers and hosting a mission-critical service. The proposed slicing framework provides an interface to dynamically manage infrastructure resources. It allows a container-based VIM to offer its resources to the hardware resource pool managed by the resource orchestrator. Facilitating on-demand life cycle handling of mission-critical services. To summarize, the chapter will present:

1. An end-to-end Cloud-native Lightweight Network Slice Orchestration (CLiSO) framework. Capable of orchestrating CNFs and MEC applications on public, private, and hybrid cloud and PNF. It allows dynamic management of infrastructure.

2. Concept of Domain Specific Handlers (DSHs) to allow Zero-touch Service Management of sub-slices.

3. NST based on a modified version of ETSI PNFD, VNFD and ETSI MEC AppD.

The CLiSO framework is an extension of the LeSO framework introduced in chapter 5. The key differences between these two frameworks are:

- LeSO framework is designed to orchestrate only edge or MEC sub-slices. It does not allow orchestrating CNFs that require multiple interfaces such as 5G UPF or PNFs.
- The ESST was not aware of slices of different domains. Whereas, CLiSO modifies the LeSOs ESST to be multi-domain slice-aware and allows the possibility to share an edge slice with multiple core or ran slices.
- CLiSO allows sharing the sub-slices with other slices. Which was not the case with LeSO.
- CLiSO provides the functionality to dynamically register and de-register CISM and MEPs. The registered CISM can have any number of physical or virtual interfaces. This ability was missing from LeSO.
- The CISM plugin of CLiSO provides basic monitoring information which was not present in the LeSO CISM plugin.

7.2 Cloud-native Lightweight Slice Orchestration (CLiSO) Framework

This section will introduce the architecture of the CLiSO framework. The roles and functioning of its different components. The concept of Domain Slice Handler(s) and how they can communicate with sub-slice orchestrators to manage their sub-slice(s). The section also presents the skeleton of the proposed cloud native NST and its different fields and their parameters.

7.2.1 CLiSO Framework Architecture

The proposed framework has a hierarchical architecture starting from the top, Network Slice Orchestrator (NSO), Network Sub-Slice Orchestrator (NSSO) (ran, edge, and core domain), CISM, and Container Image Registry (CIR). To compare the framework with 3GPP's proposed network slice management framework, NSO is analogous to NSMF and NSSO is analogous to NSSMF. Apart from these components, there is a template registry and a global Domain Name Server (DNS); these components are not shown in Figure 7.1. The transport domain is out of the scope of the proposed slice orchestration framework. The different layers are highlighted to distinguish the role of the components. The Service Orchestration Layer (SOL) is responsible for translating SLOs to Resource Level Objectives (RLOs) and coordinating with resource controllers. Whereas the Resource Orchestration Layer (ROL) manages the resources via communicating with the underlying resource pool. The purpose of each component of the framework is described below,

- Network Slice Orchestrator (NSO): It is responsible for creating Network Sub Slice Templates (NSST) for different domains and coordinating the life cycle management of sub-slices. It receives the monitoring data from different sub-slice orchestrators to extract slice-level monitoring information.
- Network Sub-Slice Orchestrators (NSSO) or Service Orchestrators: They are responsible for handling sub-slices of their respective domains and collecting monitoring data to share with NSO. They expose APIs for DSH to consume slice-specific KPIs,

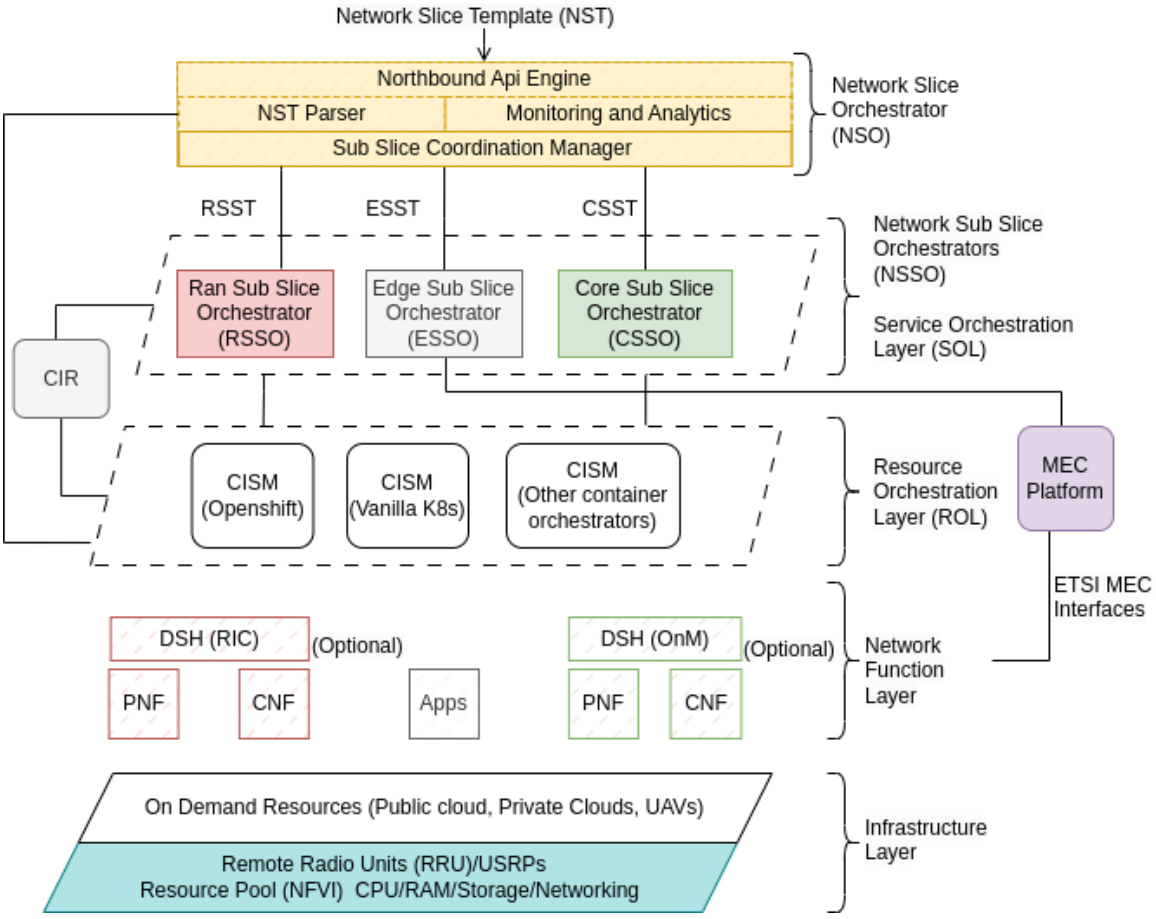


Figure 7.1: Proposed Cloud-native Lightweight Slice Orchestration Framework

- RAN Sub-Slice Orchestrator (RSSO): Handles the lifecycle of ran sub-slices composed of PNFs or CNFs. It communicates with CISM to manage the CNFs and PNFs. To manage each PNF, RSSO creates ephemeral containers with short lifespans when needed. CISMs managed by RSSO are capable of handling radio access network functions, Radio Unit (RU), Distributed Unit (DU), Central Unit (CU) or Central Unit-Control Plane (CU-CP), Central Unit-User Plane (CU-UP).
- Edge Sub-Slice Orchestrator (ESSO): Handles the lifecycle of edge sub-slices composed of MEC Applications. It coordinates with the MEC Platform to provide the necessary services like traffic redirection, DNS-based redirection, or Radio Network Information Service (RNIS) to the MEC Apps. ESSO only handles container-based MEC Apps. MEC Platform discussed in chapter 3 is also part of the proposed framework. The detailed functioning of ESSO is described in 5.
- Core Sub-Slice Orchestrator (CSSO): CSSO handles the lifecycle of core network sub-slices composed of PNFs or CNFs. It communicates with CISM to manage PNFs and CNFs.

- CISM: As described in section 2.1.3 of chapter 2, it is responsible for orchestrating containers. It creates the necessary communication links between network functions to deliver the required slice behavior. NSSOs communicate with CISM via a CISM agent hosted on the CISM platform. The framework is capable of orchestrating containers on different distributions of Kubernetes, Openshift, Vanilla Kubernetes (also known as K8s), and K3s[1]. A new distribution can be supported by creating a plugin. The Vanilla Kubernetes plugin can be used for orchestrating network functions on the public cloud Kubernetes distribution. It has most of the required functionalities.

- CIR: Manages and stores Open Container Initiative (OCI) format container images. It is capable of pulling images from public or private repositories and building images from source code.

- Template Registry (TR): It has a similar role as LeSOs TDR. It stores all the NSTs and NSSTs. RSST, ESST, and CSST are RAN, Edge, and Core sub-slice templates respectively.

- Global Domain Name Server (DNS): Similar to LeSOs external DNS, it allows the network functions hosted on different CISM infrastructures to resolve the fully qualified domain name of other network functions belonging to the same slice.

The proposed framework follows cloud-native design principles, service-based and microservices architecture, and supports on-demand scaling of the components. Each component exposes a REST API. It is possible to downsize the framework if required and deploy only selected components. For example, to manage only core sub-slices, it is required to deploy only CSSO. RSSO and ESSO are optional.

[1] https://k3s.io/

The framework proposes dynamic resource management based on the CISM registration mechanism. This mechanism was not present in chapter 5 LeSO. In LeSO CISM and MEP were registered at the startup time. Unlike ETSI NFV, CLiSO does not follow the concept of fixed infrastructure. CISM is added to the individual NSSO CISM repository via NSO. A CISM agent should be running on the CISM platform. It is registered using the CISM registration template shown in Figure 7.2. The fields in underline are required fields and others are optional. *listOfSubnet* allows CISM to describe its subnets and this information is used to allocate IP addresses to network functions. *dpdk* boolean value is to specify if the interface supports data plane acceleration technology, Data Plane Development Kit (DPDK). *annotations* field allows mentioning resources apart from CPU and RAM that CISM exposes, for example, hugepages.

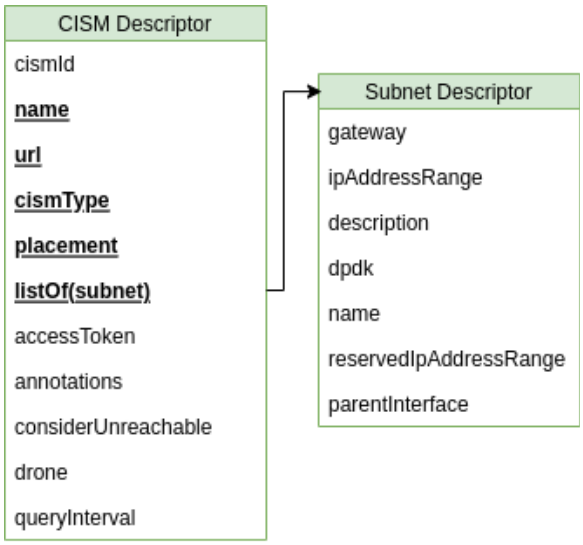


Figure 7.2: Proposed CISM Descriptor

Registered CISM shares regular heartbeats with their respective handlers in NSSOs. This allows NSSOs to be infrastructure aware and make decisions when there is a problem at the infrastructure level. The field *drone* allows registering UAV-based CISM. This field instructs the CISM agent not to remove the network slice when a heartbeat is not being shared with NSSO. CISM can manage PNFs by using ephemeral containers, short-lived containers that can communicate with PNFs and perform any required action based on the instructions provided in NSST.

MEC platforms are registered dynamically only to ESSO via NSO. ESSO directly communicates with MEC platforms and gets the list of services hosted at the MEP. Figure 7.3 shows the MEP descriptor. The fields in underline are required fields and others are optional. ESSO has the capability to query the MEP regularly for heartbeats and the list of hosted services.

7.2.2 Domain Specific Handler (DSH)

Domain Specific Handlers are management network functions responsible for handling the lifecycle of one or many network functions belonging to the same slice. They

Figure 7.3: Proposed MEP Descriptor

are analogous to an Element Management System (EMS) but designed with an ability to communicate with the CLiSO framework to negotiate the SLOs of the slice. A slice may have multiple DSHs depending on the NST provider. Their administrative domain is restrictive to their slice. Few key points about DSH:

- They get lifecycle notifications of all the network functions from NSSO. They have a direct communication link with all the network functions of their administrative domain. They fetch network function level KPIs and push them to the CLiSO monitoring engine.
- They can subscribe to infrastructure-level KPIs, computation, and networking resources consumed by the network functions. Based on this information, they can request the NSSO to increase or decrease the resources consumed by the CNF or MEC application.
- They can request the NSSO to upgrade the container software image of a CNF instance.

DSH can provide ZSM if they collect KPIs from network functions and the CLiSO framework to handle the lifecycle of their managed network slice. DSH may use Artificial Intelligence and Machine Learning algorithms to perform ZSM. CLiSO Framework does not have an inbuilt DSH as they are external to the proposed framework. There are no standards around designing an EMS similarly nor for DSH. But to communicate with the NSSO they should have a dedicated interface. DSH provides the freedom to network function providers or vendors to manage their network functions rather than using NSO's generic management algorithm.

Figure 7.1 shows some examples of network functions that can be used as DSH if appropriate APIs are implemented to communicate with CLiSO. 1) RAN Intelligent Controller (RIC) that manages RAN network functions via analyzing their and infrastructure level KPIs from CLiSO. 2) Operations And Maintenance (OAM) that manages the lifecycle of core network functions via subscribing to notifications from Access and Mobility Management Function (AMF), Session Management Function (SMF), and Network Data Analytics Function (NWDAF) and consuming infrastructure level KPIs from CLiSO framework. However, DSHs are not mandatory components of a sub-slice. If network functions have a self-management mechanism, they can communicate with NSSOs to manage their own resource consumption.

7.2.3 Network Slice Template

The Network Slice Template proposed in Figure 7.4 is deployment oriented and contains network function deployment-specific information. The framework considers that slice or sub-slice level SLOs can be translated to the network function's configuration. The proposed NST contains dedicated sections to define Core, Edge, and RAN sub-slices. NSO accepts a NST package in tarball format. This allows providing dedicated YAML files for each sub-slice template and any additional files like scripts or configuration files required by the PNF, CNF, or MEC applications. Apart from the network slice template the proposed framework does not require any additional deployment-related packages like helm-charts or juju-charms[2] as required by OSM and ONAP. To stay aligned with ETSI's proposed VNFD instead of using the term CNF Descriptor (CNFD) the framework retains VNF Descriptor (VNFD) in the proposed NST. Below is the description for some of the fields of NST:

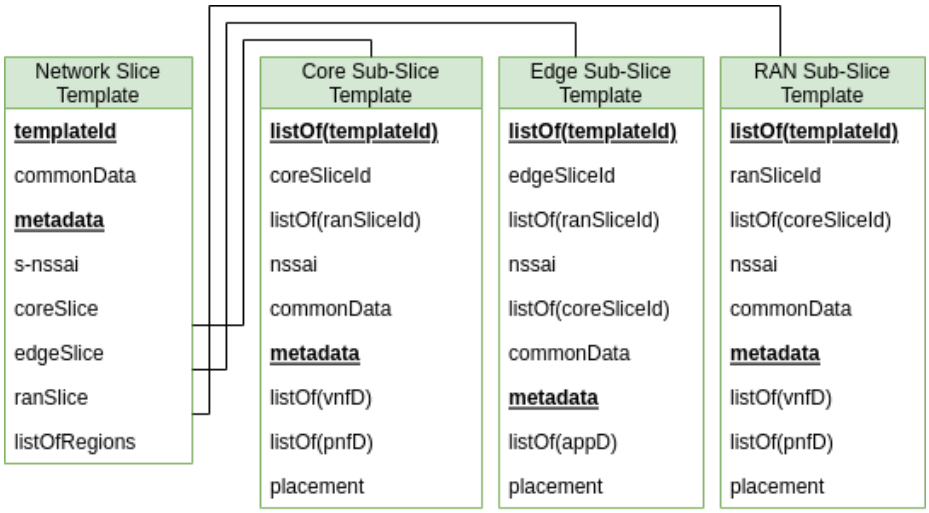


Figure 7.4: Proposed Network Slice Template

- *templateId*: A unique identifier for an onboarded slice. If the field is present for a slice then instead of creating a new slice the orchestrator will update the onboarded slice.
- *metadata*: This field contains only slice name as a mandatory parameter and other parameters are optional, annotations and order in which sub-slices should be instantiated.
- *commonData*: This field contains data that is common to all the sub-slices and it will be provided to all the sub-slices of a slice. It can contain any user-defined information and it will be forwarded to network functions of the sub-slice. For example, this field can contain SLO information and DSH can use this information to tune the configuration of other network functions to achieve the required SLO.

[2]https://juju.is/

- *s-nssai*: single network slice selection assistance information is a unique network slice identifier defined by 3GPP. It contains Slice Service Type (SST), a mandatory parameter to identify slice characteristics, and an optional parameter Service Differentiator (SD).

- *listOfRegions*: This field allows defining the regions where the network slice will be created. The regions are defined in the CISM descriptor. A region can be the name of a city, state, or custom value defined when a new CISM is added to the repository.

- When providing a new NST to the orchestrator it is mandatory to define one of the sub-slices. If the sub-slice unique identifier, *coreSliceId*, *ranSliceId* or *edgeSliceId* is provided in the respective section then the orchestrator will automatically fetch the sub-slice template from the template registry. It is possible to define the subSlice template in a separate YAML file and provide its relative location to the package.

  All the sub-slice templates have a similar design,

- They all contain a list of NST *templateId*, as a sub-slice can be shared among multiple slices.

- The unique sub-slice identifiers *coreSliceId*, *ranSliceId* or *edgeSliceId* are only present when the sub-slice is successfully on-boarded. If the field is present in the sub-slice template the sub-slice orchestrator will update the on-boarded sub-slice instead of on-boarding a new template.

- The *placement* field allows specifying the placement of the sub-slice network functions in a region. This can be the unique identifier of a CISM or the position tag, for example, FAR EDGE, ASSOCIATED EDGE, or CENTRAL CLOUD. These tags are customized and can be defined in the CISM descriptor. This placement will be common for all the CNFs of a sub-slice. Unless the CNF or MEC App descriptors also have the *placement* field and specifics the *cismId*. The placement field can be manually added in the NST, if not then NSO will automatically instruct the domain orchestrators about the placement.

- coreSlice contains a list of *ranSliceIds* using the coreSlice. Similarly, ranSlice contains a list of *coreSliceIds* it is using. edgeSlice contains a list of *coreSliceId* and *ranSliceId* through which the edgeSlice is reachable. This mapping is used to provide security isolation among sub-slices. For example, only allowed *ranSliceIds* can use the coreSlice.

- Edge sub-slice template is restricted to defining only MEC applications.

- *nssai* is a list of *s-nssai*. It allows enabling the sharing of sub-slices with sub-slices of different network slices. This parameter is also used to provide isolation among different slices.

The vnfD, pnfD, and appD used by coreSlice, ranSlice, and edgeSlice templates are modified versions of ETSI NFV VNFD, PNF, and ETSI MEC AppD. However, the recent version of VNFD allows describing containers using *osContainerDesc* field but there is no simple possibility to define the container's exposed ports, subnets it should be connected to, initial startup configuration, and placement-related information of VNF. To overcome such shortcomings the framework proposes a modified VNFD and PNFD in Figure 7.5. It should be noted that the figure contains only the fields added or modified for the proposal.

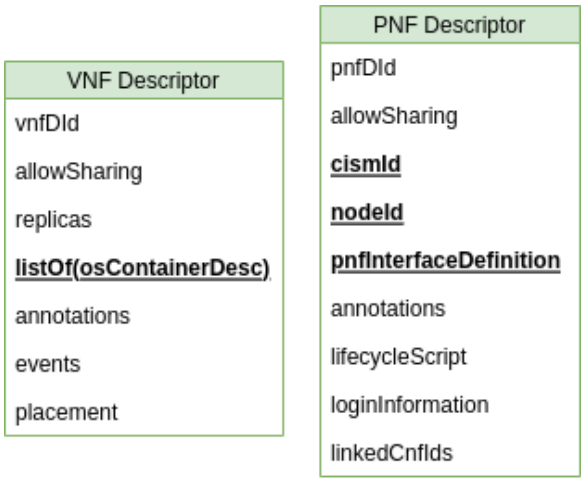


Figure 7.5: Proposed VNF and PNF Descriptor

- *vnfDId* and *pnfDId*: Are unique identifiers allocated by CISM when the network functions are already on-boarded. If already present, CISM will update the on-boarded descriptor rather than on-boarding a new descriptor.
- *osContainerDesc*: Describes the container images required to deploy the CNF. It contains several sub-fields to allow describing computational resources required and their limit, container image location, storage-related information, and monitoring parameters to be tracked for this osContainer. The model was extended to add information related to ports exposed by the container, the possibility to pass configuration to containers, startup command in case the container wants some initial command, liveliness to understand network function is always running, and readiness probe to know when the network function is ready to use, etc.
- *replicas*: Number of replicas of the CNF. The CNF should support load balancing between different replicas.
- *events*: Each CNF can subscribe to events published by CSSO or RSSO. This field is mostly relevant for DSH. The events give information about all network functions of the slice.
- *placement*: If mentioned it will supersede the placement defined in the sub-slice template.

- *cismId*: Mandatory field for PNF to define its location, and in case PNF is only accessible via one of the nodes of the CISM then it is mandatory to define *nodeId*.
- *pnfInterfaceDefinition*: It allows describing the interfaces that are connected to the PNF and their IP addresses.
- *linkedCnfIds*: This field defines the list of CNFs that will use the PNF. This will allow deciding the placement of CNFs to enable communication with their PNF. For example, if 5G New Radio DU will be placed on the node through which it can connect to the Radio Unit (RU).
- *lifecycleScript*: A relative location or URL of a script to execute in the PNF at the time of on-boarding. It is executed via a short-lived container instantiated in the same network or host from where PNF is reachable.
- *annotations*: This field is to consume the resources exposed by CISM.

The application descriptor used here is slightly updated from the one described in chapter 5. The field *swImageDescriptor* is replaced with *osContainerDesc* as it is more relevant according to the ETSI NFV standards. ETSI MEC appD standards are still not adapted for container-based VNFs or CNFs.

7.2.4 Isolation between slices

Isolation between network slices is important for security concerns while providing the required QoS to the network slice. The isolation is divided into two categories, first resource isolation, and second communication isolation.

1. Resource Isolation: Isolation at the level of infrastructure resources used by a CNF. In the *osContainerDesc* field, it is mandatory to define the required CPU and RAM needed by the container and their limits. The CISM will always guarantee that these resources are allocated to the container.
2. Communication Isolation: For each container of CNF in *osContainerDesc* there is a subfield *ports* to mention the exposed ports of the container and the network in which they are exposed. Outside this network, the ports will not be reachable. *nssai* field controls the communication between sub-slices of different slices. A sub-slice will only be able to communicate with slices defined in *nssai* else all the outgoing traffic to other slices will be rejected. Each instantiated CNF has an infrastructure-level container. This container guards the ingress and egress traffic of the CNF. An example of the tools that can manage resource isolation is Linux iptable rules.

7.2.5 Monitoring and Logging

To keep the framework lightweight, it only exposes a limited set of KPIs using the inbuilt capability of the CISM. The framework provides CPU and RAM consumed by each network function of a slice. The PNF or their linked management CNFs can push

their resource consumption to the monitoring engine. The framework provides the standard output logs of each CNF of the slice. The PNF or their linked VNF have to push their logs to the logging engine. The proposed framework can be plugged into the monitoring framework proposed by the authors in [48] to get elaborated KPIs related to each network function.

7.2.6 Working of the Framework

A NST template defines a network slice instance. Each network slice instance has four crucial phases,

1. Preparation phase: It includes calculating the requirement of the network slice instance, onboarding the required container image, reserving the required resources at the CISM level, and creating CISM deployment-specific definitions.

2. Commissioning phase: After the successful onboarding of the slice network functions, the network functions are instantiated and in the case of MEC applications if required, ESSO communicates with MEP to provide requested services. At this phase, the slice is ready and operational. The KPI collection can be started.

3. Operational phase: The slice instance can serve its consumers and KPI can be monitored. At this phase, it is possible to use the output of the monitoring engine to update the slice resource consumption.

4. De-commissioning phase: The slice instance will be terminated, all the reserved resources will be released and the container images will be off-boarded.

NSO exposes a northbound REST API to allow CSMF to request the creation of a network slice instance described using NST. NSO using the NST parser disintegrates the NST in different NSSTs based on the domains defined in NST. If the *listOfRegions* is defined in the NST and there is no *placement* field in NSST, it will calculate the placement of the sub-slice and add the *placement* field in the NSST. The sub-slice coordination manager forwards these NSSTs to their respective domain orchestrators. The domain orchestrators can simultaneously communicate with multiple CISM agents spread across different regions or in the same region to create the sub-slice. If the VNFD defines the placement of the CNF then it will supersede the placement value defined in NSST. Domain orchestrators add the infrastructure level container based on the *nssai* field of the NSST to enable communication isolation and translate the PNFD, VNFD, and appD to CISM level definition.

A CNF or MEC application corresponds to a Kubernetes Pod. All the containers are mapped to one Pod. CISM level objects define the resource requirements of the pod, virtual interface definition, and software configuration, etc. Domain orchestrator communicates with CISM via CISM agent to instantiate Pods. CISM communicates with PNFs via short-lived containers. The containers use the ICMP ping mechanism or Linux NETCAT command to check the connectivity with the PNF. They execute *lifeCycleScript* if provided in PNFD. CISM agent responds back to domain orchestrators with network information of the CNFs, PNF, and MEC application. If MEC applications request MEP services, DNS redirection, or traffic redirection as described in appD. ESSO communicates with the desired or default MEP and provides the requested services.

After the successful creation of all the network functions of a sub-slice, the slice instances are ready to serve their customers. The termination of a slice instance follows a similar mechanism and results in terminating all the network functions, disabling services used by the MEC application, and off-boarding all the container images.

7.3 Performance Evaluation

CLiSO framework was evaluated on public and private clouds. Public cloud as Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), and OVH provide Kubernetes as Platform as a Service (PaaS). The Kubernetes lifecycle and underlying infrastructure are managed by the cloud provider. Private cloud was created using Minikube[3] and Red Hat OpenShift Local[4], two command line tools to install Vanilla Kubernetes and Openshift, respectively. The proposed framework was evaluated using three different experiments with different motives:

1. Compatibility Testing: The motive of this test was to evaluate the compatibility of the framework with different Kubernetes distributions.

2. Configuration Testing: There were two motives for this test, First to evaluate the capability of the CLiSO framework to manage multi-domain slicing. Second, if the framework can be disintegrated and deployed on different platforms.

3. Scalability Testing: The motive was to evaluate multi-tenancy and resource consumption when multiple slice creation requests arrive.

All the experiments have used OpenAirInterface (OAI)[12] 5G Core and RAN Network Functions. Every experiment has a different slice configuration, different number of CNFs, or different deployment complexity to understand how many CNFs in a slice the framework can support.

7.3.1 Compatibility Testing

To evaluate the compatibility of the CLiSO framework with various public clouds, the lifecycle of a 5G Core Network Slice was managed on different production-level Kubernetes platforms. In the experiment, the core network slice went through all four phases on each public cloud. The core network slice contained AMF, SMF, NRF, UPF, AUSF, UDM, UDR, and two replicas of MYSQL. Each of the network functions and MYSQL instances was deployed as CNFs, slice contained 9 CNFs or Kubernetes Pods or 9 containers.

All the CISMs were single-node clusters. Their hardware details are mentioned in Table 7.1. To manage the lifecycle of the core network slice, only necessary components of CLiSO were used, image registry, database, CSSO, and NSO. The CLiSO framework was deployed on an OVH cloud instance with 4vCPU, 15 GB RAM, and Kubernetes version 1.25.4-1.

[3]https://minikube.sigs.k8s.io

[4]https://developers.redhat.com/products/openshift-local

Table 7.1: CISM Details for Compatibility Testing

Platform	Location	Kubernetes Version	Resources (vCPU, RAM)
Azure (AKS)	Paris	1.24.9	4, 16GB
Google (GKE)	Paris	1.24.8-gke.2000	4, 16GB
OVH(I)	Strasbourg	1.25.4-1	4, 15GB
Amazon (EKS)	Ireland	1.24	4, 7.5GB
OVH(II)	Strasbourg	1.23.14	4, 15GB
Kubeadm	Local	1.23.17	4, 16GB

To compare the results of the CLiSO framework with ETSI OSM. ETSI OSM was deployed in an Ubuntu 20.04 VM with 4vCPU and 16GB RAM. The deployment instructions were used from the website [49]. Inside the VM OSM deployer script deployed OSM on a Kubeadm-based Vanilla Kubernetes cluster, a tool to create a Kubernetes cluster. OVH (II) and Kubeadm were used as CISM, their hardware details are mentioned in Table 7.1. The reason for using different OVH configure was because of OSM, as it only supports an older version of Kubernetes. In OSM terminology CISM is referred as a dummy VIM. The helm charts used by the 5G core network slice descriptor were taken from the OAI git repository [50].

Table 7.2: Core Network Slice Life cycle On Various Cloud Platforms via CLiSO and OSM

Orchestrator	Platform	CISM Registration(s)	Creation (s)	Deletion (s)
CLiSO	Azure	0.502	30.048	1.436
	Google	0.396	43.926	2.317
	OVH(I)	0.513	44.1	2.394
	Amazon	0.718	65.46	1.538
OSM	Kubeadm	1.846	97.68	29.97
	OVH(II)	2.930	138.98	34.78

Table 7.2 highlights the time taken (in seconds) to register different CISMs, create a core network slice and delete a core network slice using CLiSO and OSM. The values are averaged over 10 iterations. From the table below conclusions can be drawn:

1. CLiSO takes less time to deploy core network slice than OSM.
2. CISM agent is compatible with different public cloud platforms and can deploy a core network slice on various public cloud platforms.
3. CLiSO framework can be deployed partially, if only the core network slice has to be handled then only CSSO is required.

Apart from this, the CLiSO framework can deploy replicas of a CNF. For example, MYSQL in this scenario. This is done by mentioning the replicas in VNFD. In OSM the replicas were only possible via manipulating the helm-charts manually.

7.3.2 Configuration Testing

The motive of this test is to understand the capability of the CLiSO framework to manage a multi-domain slice hosted on different CISM instances. Figure 7.6 shows the slice and required placement of the CNFs. This configuration of 5G network slice is meant for applications requesting low latency, as the user plane is hosted at the edge. The CLiSO framework was disintegrated, image registry and its database were deployed in public cloud OVH with 2vCPU and 4GB RAM. The rest of the components, CSSO, RSSO, NSO, and their database were deployed in a local Kubernetes instance with 4vCPU and 4GB RAM. The image registry has to be located in the public cloud where all the CISM can communicate. Hence, the image registry requires a public IP address in this scenario. Hardware information related to CISMs is below:

1. Public Cloud (OVH): 4vCPU and 16GB RAM, location Strasbourg
2. Local Openshift: 4vCPU and 16GB RAM, Openshift Local version 4.12
3. Local Minikube Kubernetes: Baremetal Kubernetes with 4CPU (No Hyper-threading) and 16GB RAM, connected with USRP B210.

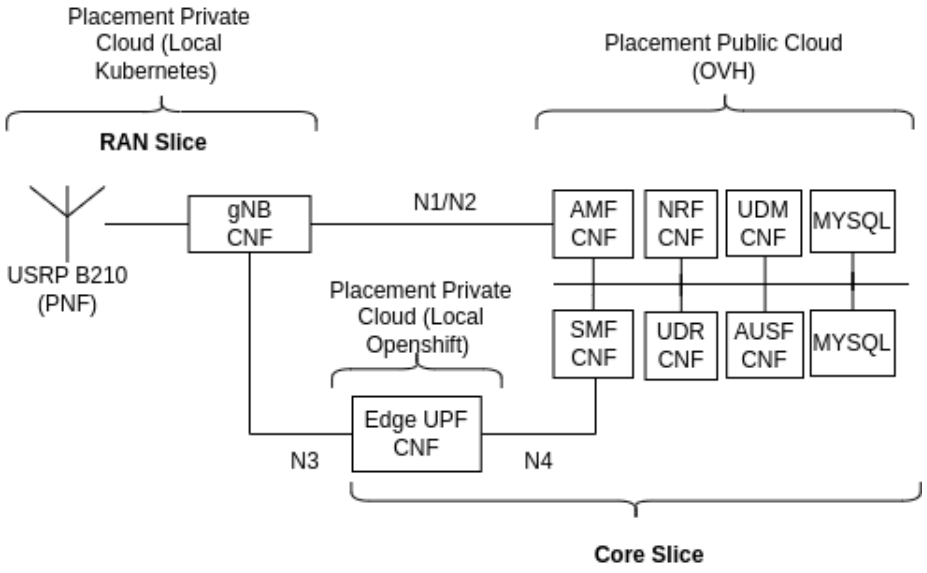


Figure 7.6: Ran and Core Slice Deployment

The public cloud slice had AMF, SMF, NRF, AUSF, UDM, UDR, and two replicas of MYSQL. At the edge, there was one UPF managed by Openshift CISM, and on the other edge instance, gNB CNF managed by Minikube CISM. In total, there were 10 CNFs or 10 Kubernetes Pods and USRP as PNF managed by gNB CNF. Table 7.3 shows the time taken (in seconds) to on-board, instantiate, terminate, and off-board the network slice. The values are averaged over 10 iterations. Though the above values are subjected to the hardware capabilities, the table concludes that the framework can manage a multi-domain slice

Table 7.3: Ran and Core Network Slice Life cycle Time (seconds)

Creation		Deletion	
43.042		3.545	
Onboard	Instantiate	Terminate	Offboard
3.259	39.783	2.431	1.114

hosted on multiple CISM instances. Like other orchestrators, OSM and ONAP connectivity between public and private clouds is a prerequisite and the CLiSO framework assumes that CISM instances can communicate with each other. ETSI OSM at the time of evaluating the framework did not support RedHat Openshift.

7.3.3 Scalability Testing

To inspect the multi-tenancy capability of the complete CLiSO framework, it was required to select a slice configuration spread across all three technological domains and hosted on multiple CISM platforms. In this experiment the RAN slice used 3GPP proposed three split RAN architecture, Distributed Unit (DU), Central Unit-Control Plane (CU-CP), and Central Unit-User Plane (CU-UP) deployed on three different CISM respectively. Rather than using physical radio units, the experiment used OAI DU in RF-simulated mode. The control plane of the core network slice was deployed on one CISM and User Plane Function on another CISM. The MEC Application connected to UPF was a content caching server. In total 12 CNFs or 12 Kubernetes Pod were included in the end-to-end network slice.

The experiment used four instances of OVH Kubernetes, two instances with 8 vCPU and 30GB RAM to imitate central and associated edge cloud. The other two instances with 4 vCPU and 16GB RAM for far edge cloud and hosting CLiSO components. For this experiment all the components of the CLiSO framework were necessary. Due to computational resource constraints, the CNFs and MEC applications were deployed in sleep mode rather than functional mode. The slice network functions went through the four lifecycle stages on-boarding, instantiation, termination, and off-boarding.

Figure 7.8 shows the slice creation time and CPU core consumption by the CLiSO framework while handling multiple slices, 2, 4 up to 14 at the same time. Due to Kubernetes default constrained of 110 Pods per host machine, it was not possible to scale the experiment beyond 14 slices on the selected infrastructure. In the OVH-managed Kubernetes service, this value is not configurable. Though the slice creation time depends on the hardware and the CNF, the graph reflects that the framework is capable of managing the lifecycle of multiple slices at the same time. To create two slices the complete framework consumed 0.275 CPU and 340MB RAM. Whereas, for 14 slices the framework consumed 1.33 CPU and 365MB RAM. In addition, the framework consumes most of the resources at the time of slice creation. Once the slice is created the slice orchestrator and sub-slice orchestrators just share heartbeats among themselves and with CNFs. After the slice is created user can interact with the slice CNFs via CNFs management API or DSHs API/graphical user interface.

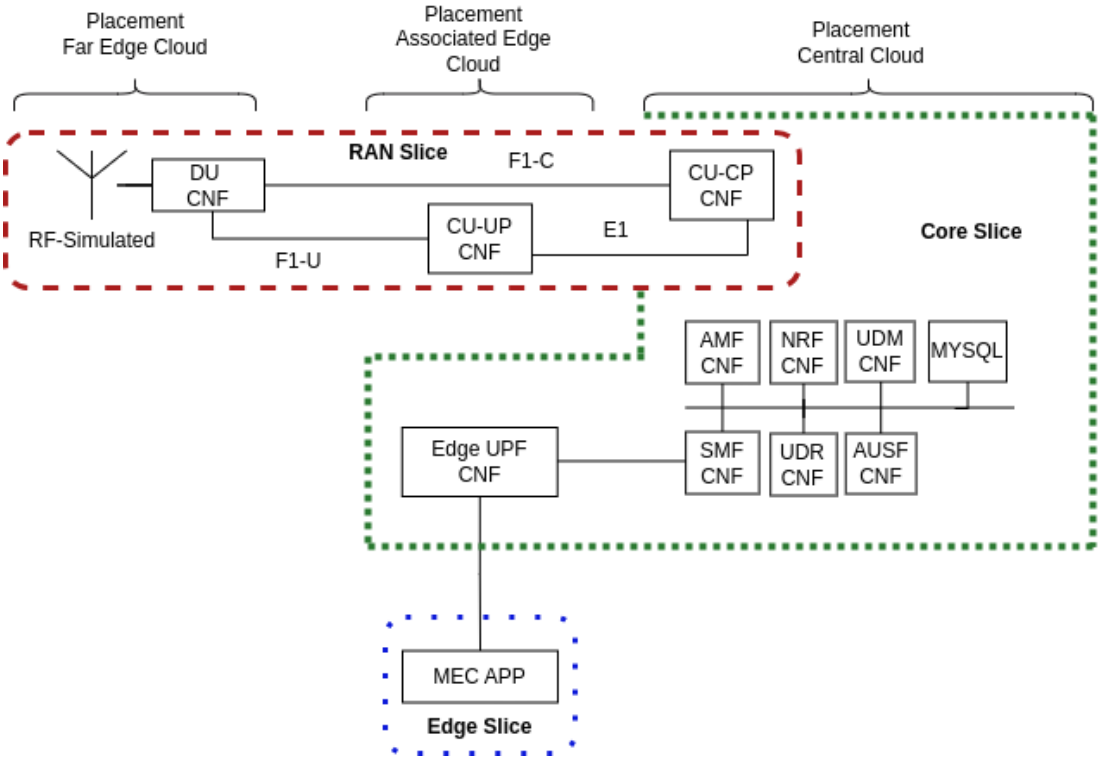


Figure 7.7: Slice in Three Technological Domains

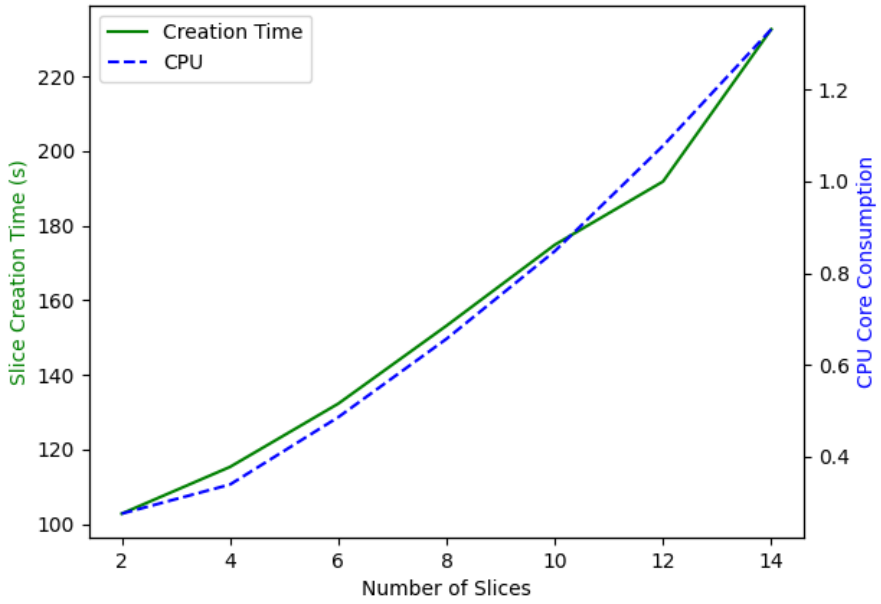


Figure 7.8: Slice Creation time vs CPU Consumed

Table 7.4: Resource Requirement OSM and ONAP

Orchestrator	vCPU	Memory (GB)
ONAP (Jakarta)	112	224
OSM-13	2	8

Table 7.4 shows the resource requirement of OSM and ONAP taken from their official website. CLiSO resource consumption is based on the scaling test presented above but there are no similar results for other orchestrators. It should be noted CLiSO framework supports the orchestration of MEC applications and interactions with MEP. Whereas other orchestrators do not have this ability. All the components of the CLiSO framework can be deployed on any publicly managed Kubernetes distribution or private Kubernetes instance.

7.4 Summary

The chapter presents a novel end-to-end orchestration framework CLiSO. The framework is lightweight when compared with other orchestrators. It takes less time to deploy slices when compared with OSM. It allows orchestrating container-based VNFs or CNFs on multiple Kubernetes distribution, Amazon Elastic Kubernetes Service, Azure Kubernetes Service, Google Kubernetes Engine, OVH, RedHat Openshift, Vanilla Kubernetes and K3S a lightweight Kubernetes distribution supporting AArch64 (ARM64) architecture. It is possible to deploy the components of the CLiSO framework on all these Kubernetes distributions. Hence, making the framework cloud-native, and easy to port to multiple cloud platforms. The framework is designed with a plugin approach and network sub-slice orchestrators abstract the type of CISM. This allows supporting any other container orchestration framework. CLiSO framework is highly customizable and can be deployed partially, as shown in the evaluation section. The deployment-centric and CISM agnostic network slice template allows describing sub-slices and their required network functions or MEC applications by abstracting infrastructure-related information. The proposed concept of Domain Slice Handlers (DSH) distributes the responsibility of managing a sub-slice among themselves and network sub-slice orchestrators. DSH allows zero-touch service management of the sub-slice network functions. DSHs are network function provider-centric, allowing the providers to personalize the life cycle management of their network functions.

Chapter 8

Concluding Remarks and Perspectives

The rise of containerization technology has a big role to play in the adoption of NFV. The main benefits of containerization are faster upgrades, easy scalability, easy monitoring, and reduced operational costs. These benefits can not be achieved without applying cloud-native fundamentals. The cloud-native fundamentals are relatively new for telecommunication and the standards are still evolving. This thesis examines the gaps in the current standards and the orchestration frameworks to understand the state-of-the-art orchestration of cloud-native network functions and MEC applications. It takes a setup further and provides a path towards joining NFV, MEC, and Network Slicing with cloud-native fundamentals.

The journey of this thesis starts by creating a MEC platform and RNIS, then creating an algorithm to place cloud-native MEC applications cost-effectively with high availability. The knowledge acquired while investigating ETSI MEC and Network Slicing was later used to propose a Lightweight edge Slice Orchestration (LeSO) Framework. A novel framework allowing orchestration of edge Slices. The edge slice deployment template proposed by the framework provided different possibilities to model an edge application. These different modeling possibilities were further studied to understand the correlation between availability and latency.

Finally, the last chapter of the thesis uses all the work presented in the previous chapters to develop a novel multi-domain network slice orchestration framework, Cloud-native Lightweight Slice Orchestration (CLiSO) framework. The service-based architecture, low resource footprint, rich feature set, rigorous performance evaluation of the CLiSO framework, and its comparison with the well-known service orchestrators provide it an edge over the existing orchestrators. CLiSO framework is designed to fulfill the orchestration requirements of 5G and beyond networks.

The existing containerization tools were not designed for packaging network functions. There is a lot of work ongoing in improving their functionality, especially for their usage in Telecommunications. It is possible that containers may not be the methodology for designing every network function or MEC application. Containers were fundamentally designed to share the Kernel with the host machine. Network functions or applications that require a dedicated Kernel or use tools such as extended Berkley Packet Filter (e-BPF) may not be able to benefit from containers. Such network functions or applications can benefit from Unikernels [51]. Today developing Unikernel applications is complicated but maybe in the future, it will be improved. The CLiSO framework modular design and infrastructure abstraction layer allows extending its functionality beyond orchestrating container-based network functions or MEC applications.

8.1 Perspectives

The increasing adoption of NFV will certainly create new challenges, and successful NFV adoption will result in higher dependency on NFV in 6G. One such challenge in front of NFV is to reduce carbon emissions. To tackle these new challenges and to keep up with new advancements in the field of NFV standards, the work presented in each chapter can be potentially extended. Below are some perspectives for future work,

1. **Complex placement problems**: The placement algorithm presented in chapter 4 is only applicable to simple cloud-native network services. Whereas, in the future, a similar algorithm is required to place a multi-domain network slice. That comprises network service from each domain. In this scenario, again, cost and availability will be two important parameters that have to be optimized. To reduce the carbon footprint, the operational cost will include carbon emissions. In such a case, the CNF or PNF should be placed, in a location, where electrical energy is produced from sustainable resources.

2. **CLiSO framework extensions**: There are several areas where the CLiSO framework can be extended:

    - **Exposing energy metrics**: There is a need to evaluate the energy consumption of 5G and beyond network functions to dynamically downscale or upscale their instances or resource consumption to optimize their energy consumption. In the future, the CLiSO framework can have the in-built capability to expose energy consumption at the network function level. This metric can be exposed as a service or MEC application to be consumed by Domain Specific Handlers (DSH) or other MEC applications to reduce the energy consumption of their slice or their own application.
    - **Zero-touch Service Management**: The framework requires the slice providers to use the proposed concept of DSH, a slice management function to perform life cycle management for their slice. To enable general slice management in the absence of DSH, the CLiSO framework can use Artificial Intelligence (AI) and Machine Learning (ML) models. These models will observe the behavior of each network function of a slice based on the triggers provided by the slice provider or pre-configured triggers to perform life-cycle management. The most challenging task here would be to identify when the network functions have abnormal behavior and predict their failure well in advance to avoid service disruption.
    - **Multi-domain network slice placement**: CLiSO provides the ability to deploy a network slice in multiple regions at the same time. Each of these regions can have multiple CISM. Currently, if the slice provider does not specify the CISM in NSST then in a region with multiple CISM, CLiSO will randomly select a suitable CISM. This approach may have issues when there is a large number of CISM in a region as the computational and networking resources are not effectively utilized. Hence, there is a need to implement algorithms for

placing a multi-domain network slice that is spread across different regions and on multiple CISM.

- **Transport Sub-Slice Orchestrator**: The framework does not manage the transport network between different NFVI sites or PNF. Currently, CLiSO assumes that different CISM can communicate and they manage their own transport network. In the future, the CLiSO framework can use such an orchestrator to dynamically manage the connectivity between different NFVI sites and logical networks between different CISM.
- **Ability to allocate isolated computing resources at CNF level**: Certain network functions that require near real-time performance such as CU-UP, UPF, or containerized DU need isolated/dedicated CPUs and a real-time Linux kernel. The underlying CISM such as Kubernetes does not have the ability to provide dedicated CPUs to a CNF. Further, Vanilla Kubernetes does not allow deploying its management plane components on a specific CPU, to avoid using the same CPUs for management components and CNFs. This task challenges the architecture of Kubernetes and requires an investigation to overcome this challenge. After solving the challenge, the CLiSO NST can have a special field for CNFs if they require dedicated CPUs.

3. **Intent-oriented slicing templates**: The proposed NST or NSSTs are deployment oriented. They require the slice owners to provide the network function's computational resource consumption, required networking interfaces, and placement preferences in case the network function prefers a specific Operating system or Linux kernel version. The intent-oriented slicing template should only require the SLOs for the slice, and based on this, the slice orchestrator should create the NSSTs by choosing the appropriate network functions from the repository.

4. **Investigating Alternatives of Containerization**: Containerization is not suitable for every application, as described earlier. In the coming future, there will be a need to investigate or develop alternatives to containerization technology. WebAssembly[1] (WASM) can be one such alternative. Initially, WASM was designed to package non-java-script applications to run in web browsers. However, it is capable of packaging any application and deploying it on multiple hardware platforms as containers using WebAssembly System Interface (WASI). WASM, unlike containers, is a binary instruction format for stack-based virtual machines such as Java Virtual Machine. Another alternative is Unikernel which has existed for a long time. Exploring such alternatives and integrating them into the global orchestration framework will require adapting the proposed framework.

[1] https://webassembly.org/

Chapter 9

Résumé

9.1 The 5G Journey

La 2ème génération du réseau mobile est le catalyseur qui a fait des téléphones mobiles une nécessité dans notre vie quotidienne. C'est la première génération de réseaux mobiles qui était présente dans le monde entier. Elle permettait d'envoyer des messages textuels et multimédias, ce qui était inimaginable à l'époque. La 3 ème génération a permis d'utiliser les téléphones mobiles pour consulter des sites web et regarder des vidéos. La 4 ème génération a véritablement révolutionné notre société en permettant une connexion de données permanente à des prix plus abordables, ce qui n'était pas possible avec les autres générations. C'est la raison pour laquelle le nombre d'utilisateurs de téléphones portables a considérablement augmenté au cours des dix dernières années. Chaque génération s'efforce d'utiliser efficacement le spectre limité pour répondre aux demandes sans cesse croissantes des utilisateurs de communications mobiles.

La 4G a créé une opportunité pour de nombreuses nouvelles entreprises qui s'appuyaient sur la connectivité internet par téléphone mobile pour atteindre leurs clients. L'utilisation intensive du streaming et de la diffusion de médias, des jeux en ligne et des appareils connectés intelligents a remis en question la conception unique de la 4G. En effet, la 4G n'a pas été conçue pour fournir à chaque application une qualité de service adaptée (QoS), en particulier pour les applications qui nécessitent une faible latence ultra fiable. Cette lacune a été bien comblée par la 5ème génération de réseaux mobiles. Elle est conçue pour fournir une qualité de service adaptée aux exigences des applications grâce au concept de découpage du réseau en tranches (Network Slicing) [1].

9.2 Network Slicing, MEC and NFV

Le projet de partenariat de troisième génération (3GPP) a introduit le découpage en tranches pour partager (logiquement) le réseau physique (partagé) afin de gérer différents modèles de trafic. Toutes les tranches logiques sont hébergées sur la même infrastructure physique, ce qui accroît l'efficacité de l'infrastructure sous-jacente. La version 17 du 3GPP identifie cinq types différents de tranches en fonction des modèles de trafic des applications,

- enhanced Mobile Broadband (eMBB) : Cette tranche est adaptée à la gestion d'applications exigeantes en termes de débit descendant élevé. Par exemple, la diffusion de médias en ultra-haute définition (UHD).

- Ultra Reliable Low Latency Demanding Applications (URLLC) : Cette tranche est adaptée aux applications qui nécessitent une communication ultra-fiable et à faible latence. Par exemple, le contrôle de drones, les opérations médicales à distance et la manipulation d'instruments de musique lors d'un concert à distance.

- Massive Internet of Things (MIoT) : Cette tranche est adaptée au traitement des communications massives de type machine à machine pour l'industrie 4.0 [3].

- Vehicular to Everything (V2X) : Cette tranche est adaptée à la gestion de la conduite de voitures autonomes et des scénarios de pelotons de camions.

- High-Performance Machine Type Communication (HMTC) : Cette tranche est adaptée à la gestion des dispositifs IoT qui nécessitent une communication à large bande passante.

Le 3GPP offre la possibilité de définir des tranches réseau personnalisées, vu que les tranches susmentionnées ne peuvent pas supporter tous les types de trafic. Les applications émergentes de la 5G, telles que les services critiques, la réalité augmentée (AR) et la réalité virtuelle, nécessitent les capacités de l'eMBB et de l'URLLC. Ces services peuvent être rendus possibles par l'intégration de l'informatique de périphérie multi-accès (MEC) dans l'infrastructure 5G. Edge cloud ou MEC [4] [5] fournit un foyer aux applications exigeantes à faible latence. La proximité du MEC avec les utilisateurs finaux le rend apte à déployer les services émergents de la 5G.

L'ETSI MEC Industry Standard Group (ISG) a été formé pour créer des normes d'orchestration des applications à la périphérie du réseau. Les applications hébergées à la MEC nécessitent une redirection du trafic ou des règles de redirection basées sur le DNS pour rediriger le trafic vers les applications de la MEC plutôt que vers l'Internet. Sachant que la redirection du trafic devant être effectuée de manière dynamique lors de l'instanciation de l'application MEC, l'ETSI MEC a défini la redirection du trafic comme une règle dans le descripteur d'application (AppD) décrivant l'application MEC. En outre, la réorientation du trafic est appliquée par l'élément de la plateforme MEC (MEP) qui sert d'interface entre les domaines MEC et 5G. MEC offre un service d'information sur le réseau radio (RNIS) qui peut être utilisé par les applications pour obtenir des informations radio pour leurs abonnés. Le RNIS permet d'adapter la qualité de l'expérience (QoE) au niveau de l'abonné. Ce qui fait de MEC une entité importante dans les réseaux 5G et au-delà.

La complexité architecturale des réseaux mobiles augmente d'une génération à l'autre. Chaque génération nécessite des méthodologies de déploiement et de surveillance différentes. La virtualisation des fonctions de réseau (NFV), introduite en 2012, a proposé une autre façon de déployer les fonctions de réseau plutôt que les fonctions de réseau physique (PNF). La NFV propose de découpler la pile logiciel du matériel des fonctions de réseau physique et de déployer le logiciel dans un environnement virtualisé dans le nuage. L'objectif de la NFV est d'apporter un changement de paradigme en passant de l'utilisation d'un matériel spécialement conçu à celle d'un matériel commercial prêt à l'emploi (COTS) pour le déploiement des fonctions de réseau. Cela permet des mises à niveau plus rapides du réseau. L'ETSI NFV ISG, formé en 2012, est responsable de la publication de normes pour la gestion du cycle de vie des VNF.

Les machines virtuelles (VM) ont fait leurs preuves dans le domaine de la virtualisation des logiciels. La première architecture NFV, les descripteurs d'instanciation des VNF et la méthodologie d'orchestration reposaient en grande partie sur des VNF basés sur des machines virtuelles. Bien que les VM aient toujours été difficiles à gérer, les problèmes d'interopérabilité entre les hyperviseurs, les longs délais d'instanciation et les exigences élevées en matière de ressources sont quelques-uns des principaux problèmes liés à l'utilisation des VM. En revanche, les conteneurs ont un temps d'instanciation plus rapide, une empreinte de ressources plus faible et permettent une architecture microservices pour la conception de logiciels. L'architecture microservices permet de décomposer une application logiciel monolithique en une pile logiciel facilement gérable afin de faciliter la gestion et la mise à jour du logiciel.

En effet, la conteneurisation existe depuis longtemps, mais les industries informatiques ont commencé à utiliser les conteneurs lorsque docker [6] a été introduit en 2013. Le groupe NFV a commencé à considérer les conteneurs comme une alternative aux VM après la publication du "Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS" de l'ETSI NFV [7] en 2019. Le terme cloud-native, tel inventé par la Cloud Native Computing Foundation[1] est une approche logiciel pour le développement et la gestion d'applications natives pour tous les types de clouds, publics, privés et hybrides. Les orchestrateurs de services réseau bien connus tels que ETSI Open-Source MANO (OSM), Open Network Automation Platform (ONAP) [8] et les orchestrateurs présentés par les auteurs dans [9], [10] sont capables d'orchestrer des VNF basés sur des conteneurs ou des fonctions réseau cloud-natives (CNF). Un service réseau est une fonctionnalité fournie par le VNF, le PNF ou une combinaison connectée des deux. Cloud-native et NFV ont joué un grand rôle en influençant l'architecture basée sur les services (SBA) et le découpage du réseau du cœur de la 5G. L'ensemble du réseau central 5G peut être considéré comme un service de réseau; Faisant du NFV cloud-native l'épine dorsale des réseaux 5G et au-delà.

La conteneurisation est l'une des exigences de base pour une application cloud-native. D'autres exigences ont été définies par la CNCF [2] :

- Conception à couplage lâche et basée sur les API : Les applications doivent suivre l'architecture basée sur les microservices et communiquer à l'aide d'appels d'interface de programmation d'application (API) à distance. Le réseau central 5G SBA répond à ce critère.

- Gérable : La configuration de l'application doit se faire en dehors de l'application. Cela permet de modifier facilement les fonctionnalités de celle-ci.

- Observabilité : L'application doit présenter des mesures permettant d'observer son état et son fonctionnement.

- Scalabilité : La CNCF ne mentionne pas explicitement l'évolutivité, mais dans les environnements cloud, les applications sont conçues pour être dimensionnées et redimensionnées de manière dynamique afin d'équilibrer le rapport coût/disponibilité.

[1] https://www.cncf.io/

[2] https://github.com/cncf/toc/blob/main/DEFINITION.md

Les applications doivent prendre en charge l'évolutivité verticale, horizontale ou les deux.

9.3 Multi-domain Network Slicing

Une tranche de réseau peut s'étendre sur plusieurs domaines technologiques, à savoir le réseau d'accès radio (RAN), le réseau de transport, le réseau de coeur 5G (CN) et l'informatique en périphérie (Edge). Le domaine de l'informatique en périphérie suit le standard MEC, comme proposé par les auteurs dans [11]. La figure 9.1 représente une tranche de réseau composée de tous les domaines technologiques.

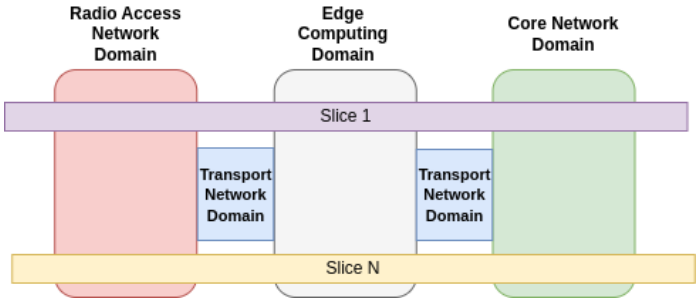


Figure 9.1: A Multi-domain Network Slice

Chaque domaine contient des fonctions de réseau spécifiques à un domaine ou le domaine de l'informatique périphérique contient des applications MEC. Dans la 5G, les fonctions de réseau sont comprises dans une instance de sous-réseau de tranche de réseau (NSSAI) suivant leurs caractéristiques. Cela permet d'effectuer un découpage du réseau en tranche à la granularité des fonctions de réseau, c'est-à-dire d'utiliser une seule fonction de réseau partagé par plusieurs tranches. Par ailleurs, une tranche de réseau peut également avoir des fonctions de réseau dédiées qui ne prennent en charge qu'une seule tranche. Il est également possible de combiner les deux scénarios.

9.4 Challenges in Orchestrating a Multi-domain Network Slice

L'orchestration de tous ces différents scénarios de découpage du réseau peut s'avérer difficile, en particulier lorsque le découpage est réparti entre plusieurs domaines. L'état de l'art actuel ne se concentre pas sur la réalisation d'une orchestration de tranches de réseau multi-domaines cloud-native. Voici quelques-uns des défis posés par les orchestrateurs existants :

- Pas de prise en charge native de l'orchestration des tranches de réseau : ils ont été conçus pour orchestrer uniquement des services réseau, ce qui limite leur capacité à prendre en charge l'orchestration de tranches de réseau de bout en bout.
- Conception héritée basée sur les VM : Leur conception architecturale est fortement basée sur des VNF utilisant des VM. Ce n'est que récemment que les VNF utilisant des conteneurs a été pris en compte.

- Pas de support pour l'orchestration des applications MEC : Les orchestrateurs peuvent gérer les fonctions réseau à la périphérie, mais aucun d'entre eux ne fournit les fonctionnalités MEP, telles que la redirection du trafic, qui sont nécessaires aux applications MEC.

- Infrastructure et plateforme dépendantes : La plupart des orchestrateurs utilisent Kubernetes[3], un moteur d'orchestration de conteneurs pour orchestrer des VNF basés sur des conteneurs. Leur architecture et leurs descripteurs de tranches de réseau ou de services de réseau dépendent fortement de la distribution communautaire de Kubernetes. Or, il existe plusieurs distributions de Kubernetes et, à l'avenir, il pourrait y avoir d'autres moteurs d'orchestration. Cela ajoute une obligation d'être conscient de l'infrastructure.

- Architecture complexe et empreintes de ressources élevées : La prise en charge des conteneurs au-dessus de l'infrastructure traditionnelle basée sur les machines virtuelles a augmenté la complexité de l'architecture. Les orchestrateurs tels que l'ONAP nécessitent de grandes quantités de ressources de calcul et de stockage, ce qui les rend inadaptés pour fonctionner dans des environnements à ressources limitées.

9.4.1 Challenges related to Availability

Outre ces défis, la littérature existante sur l'orchestration des fonctions réseau cloud-natives n'aborde pas l'aspect de leur disponibilité. Les fonctions réseau cloud-natives doivent respecter la disponibilité de 99,999

- Défi I : La façon cloud-native d'atteindre cette disponibilité est d'avoir plusieurs répliques de la fonction réseau. Cela peut conduire à un surapprovisionnement en ressources infrastructurelles, ce qui augmente les coûts de déploiement et de gestion. Un problème de décision se pose donc : combien de répliques de chaque fonction de réseau faut-il déployer ? Sans surapprovisionner les ressources informatiques afin d'éviter les coûts élevés et de fournir la disponibilité du service comme promis dans l'accord de niveau de service (SLA) ?

- Défi II : Les fonctions de réseau déployées à la périphérie sont sensibles à la latence. L'architecture microservices des applications cloud-natives peut avoir une incidence sur leur latence. Un déploiement étroitement couplé de ces applications aura une meilleure latence mais une disponibilité plus faible. En revanche, un déploiement faiblement couplé offrira une meilleure disponibilité mais une latence plus élevée. Quel est le compromis à faire ?

9.5 Motivation de la thèse

Suite à ces défis, cette thèse est motivée par l'absence d'un :

[3]https://kubernetes.io/

1. Cadre d'orchestration de découpage multi-domaine léger et facile à utiliser. Un cadre qui peut orchestrer simultanément des tranches de domaine courant, périphérique et central.

2. Un cadre qui suit les principes cloud-native pour orchestrer et gérer les CNF sur les clouds publics, privés et hybrides.

3. un cadre qui extrait les informations relatives à la plate-forme et à l'infrastructure du propriétaire de la tranche de réseau (cadre de travail qui fait abstraction des informations relatives à la plateforme et à l'infrastructure du propriétaire de la tranche de réseau).

4. un algorithme de placement qui fournit un déploiement conscient des coûts et de la disponibilité des fonctions réseau cloud-natives.

5. Une méthodologie autour du déploiement de la fonction réseau cloud-native à la périphérie du réseau qui nécessite une faible latence et une disponibilité de niveau telco de 99,999

L'objectif final de cette thèse est de réaliser un cadre d'orchestration de tranches multi-domaines pour combler les lacunes entre les normes et les orchestrateurs existants.

9.6 Résumé des chapitres

La thèse commence par un historique des terminologies pertinentes. Le chapitre 2 donne un bref aperçu de NFV, MEC, Network Slicing, Netowrk Service/Slice Orchestrators et 5G Core Network Service Based Architecture. La figure 9.2 représente le plan visuel de la thèse.

- **Chapitre 3**: Le service d'information sur le réseau radio (RNIS) est l'un des principaux services fournis par une plateforme MEC (MEP). Il est chargé d'interagir avec le réseau d'accès radio (RAN), de collecter des informations au niveau du RAN sur l'équipement de l'utilisateur (UE) et de les exposer aux applications mobiles de périphérie. Ces dernières peuvent à leur tour les utiliser pour ajuster dynamiquement leur comportement afin de s'adapter de manière optimale aux conditions du réseau d'accès radio. Ce chapitre présente une implémentation RNIS conforme aux normes, basée sur OpenAirInterface, et étudie les aspects de performance critiques pour sa fourniture en tant que VNF. Étant donné que la conception et le fonctionnement du RNIS suivent le modèle de publication et d'abonnement, le chapitre étudie d'autres mises en œuvre utilisant différentes technologies Publish/Subscribe (RabbitMQ et Apache Kafka) et compare leur utilisation et leurs performances afin d'évaluer leur aptitude à fournir le RNIS en tant que service.

- **Chapitre 4**: Ce chapitre propose une approche cloud-native pour fournir de la résilience à des services de réseau cloud-native simples. Un service de réseau cloud-native simple comprend un seul CNF. Étant donné qu'un CNF est capable de fournir des services à d'autres CNF, le CNF est considéré comme un service. Le chapitre

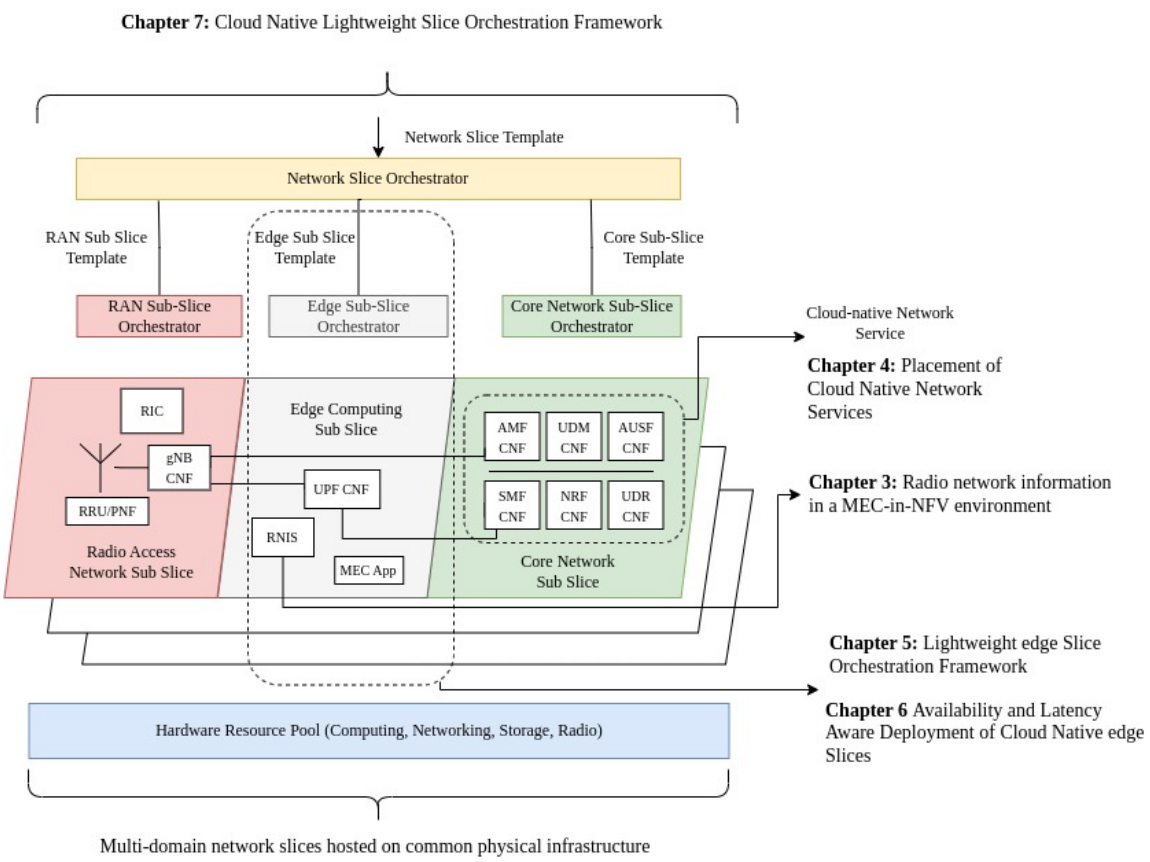


Figure 9.2: Représentation visuelle du plan de la thèse

présente un algorithme d'allocation de ressources et de placement dynamique pour modéliser et placer un service de réseau cloud-native simple. L'algorithme vise à minimiser l'utilisation des ressources infrastructurelles sous la contrainte de respecter la disponibilité du service mentionnée dans l'accord de niveau de service.

- **Chapitre 5**: Ce chapitre propose un cadre d'orchestration Lightweight edge Slice Orchestration (LeSO). Cet orchestrateur dit cloud-native orchestre et gère le déploiement de microservices en tant que sous-tranches à la périphérie. Alors que les cadres d'orchestration existants sont avides de consommation de ressources informatiques et ne parviennent pas à s'intégrer dans le domaine du Multi-access Edge Computing (MEC), LeSO est, de par sa conception, très léger et intègre un composant similaire à la plateforme MEC pour garantir le pilotage du trafic afin d'automatiser le déploiement des tranches de bordure. Les résultats de l'expérience montrent que LeSO nécessite une petite quantité de CPU et de mémoire, même lorsqu'un grand nombre de tranches de bordure sont déployées.

- **Chapitre 6**: Dans le paradigme cloud-native, fortement adopté par les fournisseurs de cloud, les fonctions réseau et les applications sont décomposées en microservices qui s'exécutent dans un conteneur. Le moteur d'orchestration de conteneurs par défaut, à savoir Kubernetes, déploie plusieurs conteneurs à l'intérieur d'un pod. La correspondance entre les microservices et les conteneurs affecte fortement la disponibilité et la latence des microservices déployés et, par conséquent, l'application en cours d'exécution. Ce chapitre propose de nouveaux modèles de déploiement tenant compte de la disponibilité et de la latence pour un service périphérique composé de plusieurs applications conçues comme de multiples microservices. Les deux déploiements envisagés sont analysés et évalués à l'aide de l'expérimentation et d'un modèle analytique, en tenant compte des critères de performance critiques pour les services orientés vers la périphérie, tels que les exigences en matière de disponibilité et de latence.

- **Chapitre 7**: Pour surmonter les défis de l'orchestration d'une tranche de réseau multi-domaine, ce chapitre propose un nouveau cadre d'orchestration de tranche légère cloud-native (CLiSO) étendant le cadre d'orchestration de tranche légère de bord (LeSO) proposé dans le chapitre 5. En outre, ce chapitre présente un modèle de tranche de réseau agnostique sur le plan technologique et orienté vers le déploiement. Pour permettre une gestion autonome des tranches de réseau, le cadre fournit un concept de gestionnaire spécifique au domaine. Le cadre a fait l'objet d'une évaluation approfondie via l'orchestration de fonctions réseau de conteneurs OpenAirInterface[12] sur des plates-formes de nuage publiques et privées.

Le dernier chapitre 8 de cette thèse conclut le travail présenté dans cette thèse et discute des travaux futurs. Le tableau 9.1 contient une liste des publications réalisées au cours de la thèse.

Type	Reference	Ch. 4	Ch. 5	Ch. 6	Ch. 7	Ch. 8
C	S. Arora, P. A. Frangoudis and A. Ksentini, "Exposing radio network information in a MEC-in-NFV environment: the RNISaaS concept," *2019 IEEE Conference on Network Softwarization (NetSoft)*, Paris, France, 2019, pp. 306-310	**X**				
C	S. Arora and A. Ksentini, "Dynamic Resource Allocation and Placement of Cloud Native Network Services," *ICC 2021-IEEE International Conference on Communications*, Montreal, QC, Canada, 2021, pp. 1-6		**X**			
C	S. Arora, A. Ksentini and C. Bonnet, "Lightweight edge Slice Orchestration Framework," *ICC 2022-IEEE International Conference on Communications*, Seoul, Korea, Republic of, 2022, pp. 865-870			**X**		
M	S. Arora, K. Boutiba, M. Mekki and A. Ksentini, "A 5G Facility for Trialing and Testing Vertical Services and Applications," in *IEEE Internet of Things Magazine*, December 2022 vol. 5, no. 4, pp. 150-155			**X**		
C	S. Arora, A. Ksentini and C. Bonnet, "Availability and Latency Aware Deployment of Cloud Native Edge Slices," *GLOBECOM 2022-2022 IEEE Global Communications Conference, Rio de Janeiro, Brazil*, 2022, pp. 3441-3446				**X**	
J	S. Arora, A. Ksentini and C. Bonnet, "Cloud Native Lightweight Slice Orchestration (CLiSO) Framework," *Computer Communications*					**X**

C = Conference, J = Journal, M = Magazine

Table 9.1: Publication and their contributions to the chapters

Bibliography

[1] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[2] *System architecture for the 5G System (5GS)*, 3GPP Technical Specification 123 501 V17.5.0, Jul. 2022.

[3] K. Zhou, T. Liu, and L. Zhou, "Industry 4.0: Towards future industrial opportunities and challenges," in *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2015, pp. 2147–2152.

[4] *Mobile Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, Dec. 2020.

[5] S. Dutta, T. Taleb, P. A. Frangoudis, and A. Ksentini, "On-the-fly qoe-aware transcoding in the mobile edge," in *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016, pp. 1–6.

[6] S. Singh and N. Singh, "Containers & docker: Emerging roles & future of cloud technology," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 804–807.

[7] *Network Functions Virtualisation (NFV) Release 3; Architecture; Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS*, ETSI Group Specification NFV-IFA 029 V3.3.1, Nov. 2019.

[8] G. M. Yilma, Z. F. Yousaf, V. Sciancalepore, and X. Costa-Perez, "Benchmarking open source nfv mano systems: Osm and onap," *Computer Communications*, vol. 161, pp. 86–98, 2020.

[9] J. Mangues-Bafalluy, J. Baranda, I. Pascual, R. Martínez, L. Vettori, G. Landi, A. Zurita, D. Salama, K. Antevski, J. Martín-Pérez, D. Andrushko, K. Tomakh, B. Martini, X. Li, and J. X. Salvat, "5g-transformer service orchestrator: design, implementation, and evaluation," in *2019 European Conference on Networks and Communications (EuCNC)*, 2019, pp. 31–36.

[10] 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)*, 2017, pp. 973–978.

[11] S. Kekki, "MEC in 5G networks," ETSI, Tech. Rep., Jun. 2018. [Online]. Available: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp28_mec_in_5G_FINAL.pdf

[12] OpenAirInterface. [Online]. Available: http://www.openairinterface.org/

[13] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[14] C. Tranoris, "Openslice: An opensource oss for delivering network slice as a service," 2021.

[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, p. 70–93, jan 2016. [Online]. Available: https://doi.org/10.1145/2898442.2898444

[16] *Generic Network Slice Template Version*, GSMA Group Specification 7.017, Jun. 2022.

[17] A. Ksentini and P. A. Frangoudis, "Toward slicing-enabled multi-access edge computing in 5g," *IEEE Network*, vol. 34, no. 2, pp. 99–105, 2020.

[18] *Mobile Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, Mar. 2016.

[19] *Mobile Edge Computing (MEC); Radio Network Information API*, ETSI Group Specification MEC 012, Jul. 2017.

[20] *Mobile Edge Computing (MEC); Deployment of Mobile Edge Computing in an NVF environment*, ETSI Group Report MEC 017, Mar. 2018.

[21] S. Bolettieri, D. T. Bui, and R. Bruno, "Towards end-to-end application slicing in multi-access edge computing systems: Architecture discussion and proof-of-concept," *Future Generation Computer Systems*, vol. 136, pp. 110–127, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22001984

[22] J. Pérez-Romero, V. Riccobene, F. Schmidt, O. Sallent, E. Jimeno, J. Fernandez, A. Flizikowski, I. Giannoulakis, and E. Kafetzakis, "Monitoring and analytics for the optimisation of cloud enabled small cells," in *Proc. IEEE CAMAD*, 2018.

[23] Y. Li, P. A. Frangoudis, Y. Hadjadj-Aoul, and P. Bertin, "A Mobile Edge Computing-assisted video delivery architecture for wireless heterogeneous networks," in *Proc. IEEE ISCC*, 2017.

[24] Y. Tan, C. Han, M. Luo, X. Zhou, and X. Zhang, "Radio network-aware edge caching for video delivery in MEC-enabled cellular networks," in *Proc. IEEE WCNC Workshops*, 2018.

[25] *Mobile Edge Computing (MEC); Radio Network Information API*, ETSI Group Specification MEC 012, Feb. 2022.

[26] *Mobile Edge Computing (MEC); Location API*, ETSI Group Specification MEC 013, Jul. 2017.

[27] P. Schmitt, B. Landais, and F. Y. Yang, "Control and User Plane Separation of EPC nodes (CUPS)," 3GPP, Tech. Rep., Jul. 2018. [Online]. Available: http://www.3gpp.org/cups

[28] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. P. Kontovasilis, "Flexran: A flexible and programmable platform for software-defined radio access networks," in *Proc. ACM CoNEXT*, 2016.

[29] R. Schmidt, M. Irazabal, and N. Nikaein, "Flexric: An sdk for next-generation sd-rans," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 411–425. [Online]. Available: https://doi.org/10.1145/3485983.3494870

[30] Running Kafka in Production. [Online]. Available: https://docs.confluent.io/current/kafka/deployment.html

[31] *Network Functions Virtualisation (NFV); Resiliency Requirements*, ETSI Group Specification NFV-REL 001 V1.1.1, Jan. 2015.

[32] L. Yala, P. A. Frangoudis, G. Lucarelli, and A. Ksentini, "Cost and availability aware resource allocation and virtual function placement for cdnaas provision," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1334–1348, 2018.

[33] S. Agarwal, F. Malandrino, C. F. Chiasserini, and S. De, "Vnf placement and resource allocation for the support of vertical services in 5g networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, pp. 433–446, 2019.

[34] S. Imadali and A. Bousselmi, "Cloud native 5g virtual network functions: Design principles and use cases," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, 2018, pp. 91–96.

[35] B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou, "Cloud-native service function chaining for 5g based on network service mesh," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

[36] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz, "Service function chaining in next generation networks: State of the art and research challenges," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 216–223, 2017.

[37] *Network Functions Virtualisation (NFV) Release 3; Virtualised Network Function; Specification of the Classification of Cloud Native VNF implementations*, ETSI Group Specification NFV-EVE 011 V3.1.1, Oct. 2018.

[38] K. Community. (2020-06-01) Kubernetes CPU. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

[39] ——. (2020-06-01) Podoverhead. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/pod-overhead

[40] E. G. Coffman, G. Galambos, S. Martello, and D. Vigo, *Bin Packing Approximation Algorithms: Combinatorial Analysis*. Boston, MA: Springer US, 1999, pp. 151–207. [Online]. Available: https://doi.org/10.1007/978-1-4757-3023-4_3

[41] S. Arora, K. Boutiba, M. Mekki, and A. Ksentini, "A 5g facility for trialing and testing vertical services and applications," *IEEE Internet of Things Magazine*, vol. 5, no. 4, pp. 150–155, 2022.

[42] ETSI OSM Community. (2021-09-20) OSM Resource Requirements. [Online]. Available: https://osm.etsi.org/docs/user-guide/03-installing-osm.html#pre-requirements

[43] A. Ksentini, P. A. Frangoudis, A. PC, and N. Nikaein, "Providing low latency guarantees for slicing-ready 5g systems via two-level mac scheduling," *IEEE Network*, vol. 32, no. 6, pp. 116–123, 2018.

[44] B. Brik, P. A. Frangoudis, and A. Ksentini, "Service-oriented mec applications placement in a federated edge cloud architecture," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.

[45] *Network Functions Virtualisation (NFV) Release 4, Management and Orchestration, Architectural Framework Specification*, ETSI Group Specification NFV 006 V4.4.1, Dec. 2022.

[46] *Mobile Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, Mar. 2022.

[47] C. Benzaid and T. Taleb, "Ai-driven zero touch network and service management in 5g and beyond: Challenges and research directions," *IEEE Network*, vol. 34, no. 2, pp. 186–194, 2020.

[48] M. Mekki, S. Arora, and A. Ksentini, "A scalable monitoring framework for network slicing in 5g and beyond mobile networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 413–423, 2022.

[49] ETSI OSM Community. (2023-04) OSM Deployment Guide. [Online]. Available: https://osm.etsi.org/docs/user-guide/latest/20-tutorial.html

[50] OpenAirInterface Software Alliance. (2023-04) OAI Core Network Helm Charts. [Online]. Available: https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed/-/tree/v1.5.0/charts

[51] T. Kurek, "Unikernel network functions: A journey beyond the containers," *IEEE Communications Magazine*, vol. 57, no. 12, pp. 15–19, 2019.