



Deliverable 3.2

Design and Prototyping of SliceNet Virtualised 5G RAN-Core Infrastructure

Editor:	EURECOM
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	30/04/2018
Actual delivery date:	30/05/2018
Suggested readers:	Infrastructure providers; Communication service providers, Digital service providers; Network operators; Vertical industries
Version:	1.0
Total number of pages:	76
Keywords:	5G, RAN, Core, Infrastructure, OpenAirInterface, Mosaic5G, Network Slicing, Virtualisation

Abstract

This document reports all the activities related to the design and prototyping of a virtualized 5G Radio Access Network (RAN)-Core infrastructure, as part of the SliceNet end-to-end slicing-friendly infrastructure. A slice-friendly RAN-Core infrastructure leverages two open source ecosystems, namely, the OpenAirInterface (OAI) and Mosaic5G. OAI is an open-source, software-based, standard-compliant LTE ecosystem for prototyping 5G Mobile Networks. Building on top of OAI, Mosaic5G serves as an open-source lightweight 5G service delivery platform. In order to provide a virtualized infrastructure deployment that covers the SliceNet use cases, different methods for deploying a virtualised 5G infrastructure are highlighted via several examples for the deployment of OAI-based 5G services.

Disclaimer

This document contains material, which is the copyright of certain SliceNet consortium parties, and may not be reproduced or copied without permission.

In case of Public (PU):

All SliceNet consortium parties have agreed to full publication of this document.

In case of Restricted to Programme (PP):

All SliceNet consortium parties have agreed to make this document available on request to other framework programme participants.

In case of Restricted to Group (RE):

All SliceNet consortium parties have agreed to full publication of this document. However this document is written for being used by <organisation / other project / company etc.> as <a contribution to standardisation / material for consideration in product development etc.>.

In case of Consortium confidential (CO):

The information contained in this document is the proprietary confidential information of the SliceNet consortium and may not be disclosed except in accordance with the consortium agreement.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the SliceNet consortium as a whole, nor a certain part of the SliceNet consortium, warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, accepting no liability for loss or damage suffered by any person using this information.

The EC flag in this document is owned by the European Commission and the 5G PPP logo is owned by the 5G PPP initiative. The use of the flag and the 5G PPP logo reflects that SliceNet receives funding from the European Commission, integrated in its 5G PPP initiative. Apart from this, the European Commission or the 5G PPP initiative have no responsibility for the content.

The research leading to these results has received funding from the European Union Horizon 2020 Programme under grant agreement number H2020-ICT-2014-2/671672.

Impressum

[Full project title] End-to-End Cognitive Network Slicing and Slice Management Framework in Virtualised Multi-Domain, Multi-Tenant 5G Networks

[Short project title] SliceNet

[Number and title of work-package] WP3 – 5G Integrated Multi-Domain Slicing-Friendly Infrastructure

[Number and title of task] T3.2 Virtualised 5G RAN - CN Infrastructure

[Document title] Design and Prototyping of SliceNet Virtualised 5G RAN-Core Infrastructure

[Editor: Name, company] Navid Nikaein, EURECOM

[Work-package leader: Name, company] Navid Nikaein, Eurecom

[Estimation of PM spent on the Deliverable]

Copyright notice @ 2018 Participants in SliceNet project

Executive Summary

The SliceNet RAN-Core infrastructure described in this document relies on two open-source ecosystems, namely OpenAirInterface (OAI) and Mosaic5G. OAI wireless technology platform is a flexible platform to enable an open 4G-5G ecosystem. The platform currently provides a standard-compliant implementation of a subset of the 4G-5G systems spanning the full protocol stack of 3GPP standards in both E-UTRAN and EPC. Founded on top of OAI, Mosaic5G is an ecosystem of open-source platforms and use cases for 4G-5G research and development (R&D), with the purpose of building a lightweight 5G service delivery platform across reusable software components. Mosaic5G leverages on software-defined networking (SDN), network function virtualization (NFV) and multi-access edge computing (MEC) technology enablers to realize the service-oriented 5G vision. JOX, one of its main components, is an event-driven orchestrator for the virtualized network that natively supports Network Slicing. Together with a flexible and programmable platform for Software-Defined Radio Access Networks and a Core network controller for Software-Defined Mobile Networks, JOX provides the possibility to achieve seamless control and configuration of physical and virtual resources for both the Core and the RAN segments. In order to achieve a slice-friendly RAN-Core infrastructure, a wide range of research has been carried out to cover different methods for deploying a virtualized 5G infrastructure from the access to the core network.

Specifically, the current deliverable reports the following achievements:

- A 5G RAN-Core slicing-friendly infrastructure that could be extended to cover different SliceNet use cases.
- The design of a programmable Data and Control Plane and its prototype through OpenFlow and an SDN controller, as a part of OAI-CN and OAI-RAN implementation.
- Various methods for deploying a virtualized 5G infrastructure.
- Finally, different virtualized RAN-Core infrastructures, which have been prototyped and tested with experimental empirical results, to achieve slicing-friendly infrastructure.

List of Authors

Company	Author	Contribution
ECOM	Navid Nikaein; Tien Think Nguyen; Xenofon Vasilakos	OpenAirInterface and Mosaic5G platforms, Virtualized Slicing-Friendly 5G Infrastructure through Juju; Deployment of OAI-based 5G Services; Manual Deployment of 5G Virtualized Infrastructure through Linux Utility; Abstract; Executive Summary; Introduction; Conclusion
ORO	Marius Iordache; Elena-Madalina Oproiu	Deployment of OAI-based vEPC Services
OTE	Georgios Agapiou	Deployment of Athonet-based vEPC Services

Table of Contents

Executive Summary	4
List of Authors	5
Table of Contents	6
List of Figures.....	8
List of Tables.....	10
Abbreviations	11
Definitions	17
1 Introduction.....	18
1.1 Objectives	18
1.2 Approach and Methodology.....	18
1.3 Document Structure	19
2 OpenAirInterface and Mosaic5G Platforms	20
2.1 An Overview of OpenAirInterface	20
2.1.1 Software Platforms.....	20
2.1.2 Hardware Platforms	22
2.2 An Overview of Mosaic5G	22
2.2.1 JOX in a Nutshell.....	23
2.2.2 FlexRAN in a Nutshell	24
2.2.3 LL-MEC in a Nutshell.....	25
3 Virtualised Slicing-Friendly 5G Infrastructure	27
3.1 Automated Deployment of 5G Virtualised Infrastructure through Juju	28
3.1.1 Juju Charms	29
3.1.2 Clouds.....	31
3.1.3 Controllers and Models.....	31
3.1.4 Technical Use Cases.....	32
3.1.4.1 Deploy of slice-friendly LTE Service Chain with Juju.....	33
3.1.4.1.1 Structure of a Charm (OAI-MME)	33
3.1.4.1.2 Juju Charm Deployment	37
3.1.4.2 Deploy LTE Service Chain for Network Slicing with JOX	38
3.2 Automated Deployment of 5G Virtualised Infrastructure through OpenStack and Heat	45
3.2.1 Comparison of OpenStack Deployment Tools	45
3.2.2 Deployment of OAI-based 5G Services	47
3.2.2.1 Juju/OpenStack	47
3.2.2.2 Heat.....	53
3.2.3 Deployment of OAI-based vEPC Services.....	55

- 3.2.4 Deployment of Athonet-based vEPC Services..... 61
 - 3.2.4.1 QoS Settings 62
 - 3.2.4.2 System Management 62
- 3.3 Manual Deployment of 5G Virtualised Infrastructure through Linux Utility 63
 - 3.3.1 LXC 63
 - 3.3.2 LXD 66
 - 3.3.3 KVM 67
 - 3.3.4 Docker 68
- 3.4 Other Automated Deployment Tools 69
 - 3.4.1 Open Baton 69
 - 3.4.2 OSM 70
- 4 A Deployment Example for the SliceNet Slicing-Friendly Infrastructure 72
- 5 Conclusions..... 73
- References..... 74

List of Figures

Figure 1. OpenAirInterface software stack	21
Figure 2. Mosaic-5G.io ecosystem	23
Figure 3. JOX architecture	23
Figure 4. An example of a standard LTE chain with Domain Controllers.....	24
Figure 5. FlexRAN protocol.....	25
Figure 6. Different slice services in FlexRAN.	25
Figure 7. LL-MEC Platform [9]	26
Figure 8. Mapping the deployment tools to ETSI NFV architecture [6].....	27
Figure 9. Juju - Open Source Generic VNF [18].....	28
Figure 10. An example of a Bundle from the Juju Store	29
Figure 11. Structure of a Charm (EPC Charm as an example).....	30
Figure 12. C-RAN architecture and components.	32
Figure 13. Slice-Friendly virtualized 5G C-RAN and CORE Slice deployed by JuJu	33
Figure 14. JOX main components and properties.....	40
Figure 15. JOX architecture	41
Figure 16. JOX JSlice definition in JSON representation	42
Figure 17. Network slicing in JOX	43
Figure 18. Deployment time of VNF chains	44
Figure 19. OpenStack service overview [14]	45
Figure 20. C-RAN deployment on top of OpenStack.....	48
Figure 21. C-RAN testbed	49
Figure 22. Image of the C-RAN testbed.....	49
Figure 23. Machines commissioned in MAAS	50
Figure 24. C-RAN instances in OpenStack environment	52
Figure 25. UE connected to the deployed network – OpenAirInterface	53
Figure 26. UE connected to the deployed network	53
Figure 27. Evolved Packet Core Network components.....	56
Figure 28. OSI layers and protocols used on the control plane between the UE and MME of an LTE network.....	56
Figure 29. The UE attachment phases	57
Figure 30. Lightweight All-in-One OpenStack Virtual Private Cloud deployment for vEPC....	58
Figure 31. vEPC solution is based on a NFV-SDN Architecture	59

Figure 32. OAI as VNF within OPNFV [38]	60
Figure 33. Athonet vEPC architecture [52].....	61
Figure 34. List of deployed LXC containers	64
Figure 35. MME log without any connected UE	65
Figure 36. MME log with a connected UE.....	66
Figure 37. UE connected to the deployed network using LXC (OpenAirInterface)	66
Figure 38. List of active LXC containers (created by LXD)	67
Figure 39. Vendor ID and product ID of the RF card	67
Figure 40. Open Baton architecture [50]	70
Figure 41. OSM mapping to ETSI NFV MANO [28]	70
Figure 42. Deployment Example of a SliceNet RAN-Core Slicing-Friendly Infrastructure	72

List of Tables

Table 1. A summary comparison of OpenStack deployment tools.....	46
Table 2. Parameters of the C-RAN testbed networks	50
Table 3. Placement of OpenStack and other services at the machines of the testbed.....	51

Abbreviations

3G	Third Generation (mobile/cellular networks)
3GPP	3rd Generation Partnership Project
4G	Fourth Generation (mobile/cellular networks)
5G	Fifth Generation (mobile/cellular networks)
5G PPP	5G Infrastructure Public Private Partnership
AE	Autoscaling Engine
AES	Advance Encryption Standard
API	Application Program Interface
AWS	Amazon Web Services
BBU	Baseband Unit
CAPEX	Capital Expenditure
CIDR	Classless Inter-Domain Routing
CN	Core Network
COTS	Commercial Off-the-Shelf
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
CQI	Channel Quality Indicator
C-RAN	Centralized / Cloud Radio Access Network
DBaaS	Database as a Service
DHCP	Dynamic Host Configuration Protocol
DL	Downlink
DNS	Domain Name System
DPI	Deep Packet Inspection
DRS	Discovery Reference Signal
eMBMS	Evolved Multimedia Broadcast Multicast Services
EPC	Evolved Packet Core
EPS	Evolved Packet System

ETSI	European Telecommunications Standards Institute
E-RAB	E-UTRAN Radio Access Bearer
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
FDD	Frequency Division Duplex
FTP	File Transfer Protocol
FM	Fault Management System
GBR	Gradual Bit Rate
GCE	Google Compute Engine
GRE	Generic Routing Encapsulation
GTP	Generic Tunneling Protocol
GUI	Graphical User Interface
HARQ	Hybrid Automatic Repeat Request
HDD	Hard Disk Drive
HSS	Home Subscriber Server
IaaS	Infrastructure-as-a-Service
IETF	Internet Engineering Task Force
IMSI	International Mobile Subscriber Identity
IoT	Internet of Things
IP	Internet Protocol
ISG	Industry Specification Group
JCC	JOX Clouds Controller
JSC	JOX Slices Controller
JSlices	JOX Network Slices
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LL-MEC	Low-Latency Mobile/Multi-access Edge Computing
LTE	Long-Term Evolution
LXC	Linux Container

MaaS	Metal as a Service
MAC	Medium Access Control
MANO	Management and Orchestration
MBR	Maximum Bit Rate
MCC	Mobile Country Code
MCCH	Multicast Control Channel
MCH	Multicast Channel
MEC	Mobile/Multi-access Edge Computing
MIMO	Multiple-Input and Multiple-Output
MME	Mobility Management Entity
MNC	Mobile Network Code
MNO	Mobile Network Operator
MPLS	Multi-Protocol Label Switching
MTCH	Multicast Traffic Channel
NAS	Non-Access Stratum
NAT	Network Address Translation
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV orchestrator
NGCN	Next Generation Core Network
NGFI	Next Generation Fronthaul Interface
NR	New Radio
NS	Network Slice
NSE	Network Slicing Engine
NSI	Network Slice Instance
NSSI	Network Slice Subnet Instance
OAI	OpenAirInterface
OPEX	Operational Expenditure
OSM	Open Source MANO

OSI	Open System Interconnection
OVS	Open vSwitch
OTA	Over-The-Air
OTT	Over-The-Top
PaaS	Platform as a Service
PBCH	Physical Broadcast Channel
PC	Personal Computer
PCF	Policy and Charging Function
PCFICH	Physical Control Format Indicator Channel
PCRF	Policy and Charging Rules Function
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDSCH	Physical Downlink Shared Channel
PHICH	Physical Hybrid-ARQ Indicator Channel
PLMN	Public Land Mobile Network
PMCH	Physical Multicast Channel
PMI	Precoding Matrix Indicator
PNF	Physical Network Function
PGW/P-GW	Packet Data Network Gateway
PoC	Proof of Concept
PRACH	Physical Random Access Channel
PSS	Primary Synchronization Signals
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
QAM	Quadrature Amplitude Modulation
QCI	QoS Class Identifier
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random-Access Memory

RAN	Radio Access Network
RAU	Radio Aggregation Unit
RB	Resource Block
RCC	Radio Cloud Center
REST	Representational State Transfer
RF	Radio Frequency
RLC	Radio Link Control
RRC	Radio Resource Control
RRM	Radio Resource Management
RRS	Radio Remote System
RRU	Remote Radio Unit
R&D	Research and Development
S1AP	S1 Application Protocol
SDN	Software Defined Networking
SDK	Software Development Kit
SGW/S-GW	Serving Gateway
SGSN	Serving GPRS Support Node
SISO	Single-Input Single-Output
SLICENET	End-to-End Cognitive Network Slicing and Slice Management Framework in Virtualised Multi-Domain, Multi-Tenant 5G Networks
SNMP	Simple Network Management Protocol
SRS	Sounding Reference Signal
SSH	Secure Shell
SSS	Secondary Synchronization Signals
TAC	Tracking Area Code
TDD	Time Division Duplexing
UE	User Equipment
UL	Uplink
URL	Uniform Resource Locator

vEPC	Virtual Evolved Packet Core
VIM	Virtual Infrastructure Management
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNA	Virtualized Network Application
VNF	Virtual Network Function
VNFaaS	Virtual Network Function as a Service
VNFM	Virtual Network Function Manager
vNIC	Virtual Network Interface Card
VoLTE	Voice-over-LTE
VPC	Virtual Packet Core
VxLAN	Virtual Extensible LAN
X2AP	X2 Application Protocol

Definitions

5G Core segment	Core Network consists of the entities that provide support for the network features and telecommunication services. 5G core segment primarily refers to the Evolved Packet Core (EPC) - the core network for LTE, and the Next Generation Core Network (NGCN) in the 5G System architecture.
5G RAN segment	RAN segment consists of the entities that manage the resources of the access network and provides the user with a mechanism to access the network. It may comprise different types of accesses, e.g. 4G and 5G NR-radio accesses. The RAN consists of a set of eNBs (in LTE) or gNBs (in 5G system) connected to the Core.

1 Introduction

In the 5G era, Network Slicing becomes a key concept, as it allows multiple logical networks to be created on top of a common shared physical infrastructure. According to 5G PPP [1], Network Slicing is an end-to-end concept covering all network segments including RAN and Core among the others.

To realize slice-friendly virtualized 5G RAN-CORE infrastructure, we will leverage the existing platforms including OpenAirInterface and Mosaic-5G. OpenAirInterface (OAI) [2] is an open source project, which is developed for the purpose of softwarizing mobile network functions from the access network to the evolved packet core (EPC) of the mobile network. OAI is generally divided into two parts: the EPC software that is known as OAI-CN; and the access-network software that goes under the name of OAI-RAN. OAI currently provides a standard-compliant implementation of a subset of Release 14 LTE for all the major components of the core network, i.e. the Home Subscriber Server (HSS), the Mobility Management Entity (MME), the Serving Gateway (SGW or S-GW) and the Packet Data Network (PDN) Gateway (PGW or P-GW), as well as the access-network, i.e. the eNB, that can be deployed on standard Linux-based computing equipment either as a monolithic BS or a disaggregated BS with a Baseband Unit (BBU) and the Remote Radio Unit (RRU),.

Mosaic-5G is a complementary open source project with respect to OpenAirInterface for the purpose of building agile 5G service platform. Mosaic5G has three main platforms: FlexRAN enabling monitoring, control, programmability in the RAN domain, LL-MEC that acts as a controller for edge and core domains providing a subset of features as specified by ETSI MEC, and JoX which is an event-driven juju-based service orchestrator core with plugins to interact with different network domains.

1.1 Objectives

Virtualised 5G RAN-Core Infrastructure regards the establishment of slice-friendly cross-domain physical and virtual infrastructure layers, to provide an execution foundation for the upper layers in the SliceNet architecture. Within this context, SliceNet will contribute a virtualised 5G RAN-Core segment, to benefit from the advantages of the emerging 5G slicing paradigm, oriented towards the support of challenging use cases by verticals.

The following specific objectives are identified, based on the description of the work:

- Establish a 5G infrastructure including both the RAN and the Core segments;
- Describes various methods for deploying a virtualised 5G infrastructure together with several examples for the deployment of OAI-based 5G services;
- Design and prototype a virtualised 5G infrastructure supporting network slicing including both the RAN and the Core segments.

1.2 Approach and Methodology

The functionalities of a slicing/slice-friendly 5G RAN infrastructure should be offered for virtualised infrastructures, re-using existing mechanisms and tools (e.g., with respect to SDN), to achieve seamless control and configuration of physical and virtual resources. Network Slicing for the LTE network is composed mainly of two egalitarian parts: i) first, a slice of the core network resources and services, i.e. a grouping of physical and virtual resources bundled together with the EPC services; ii) and second, a slice of the eNB

resources and services, sharing techniques for sharing of the eNB resources (i.e, Resource Blocks (RBs)) in frequency, time, and space dimensions.

SliceNet has defined three representative vertical use cases, namely, Smart-Grid, eHealth and Smart City [3]. Each use case has different requirements regarding network infrastructure, e.g. the eHealth and Smart-Grid use cases require a Mobile/Multi-access Edge Computing (MEC) platform while the Smart-City use case does not. Additionally, the use cases require the deployment of a variety of different physical infrastructures including RAN (4G, 5G NR - New Radio) and Core (EPC, Next Generation Core Network - NGCN). As a result, in this document we try to highlight different methods for deploying a virtualized 5G infrastructure via several examples for the deployment of OAI-based 5G services. In order to provide an infrastructure deployment that covers the SliceNet use cases, we consider a C-RAN (Centralized, or Cloud Radio Access Network) architecture (with fronthaul network) in the context of this document.

1.3 Document Structure

The remainder of this document is organised as follows: Section 2 presents the OpenAirInterface and Mosaic5G platforms, as the foundation of the SliceNet RAN-CN infrastructure. Section 3 describes in details different methods for deploying a virtualized slicing-friendly RAN-CN infrastructure ranging from an automated deployment method to manual deployment through Linux utility. Moreover, Section 3 provides a number of examples regarding the deployment of OAI-based 5G services to support the SliceNet use cases. Section 4 describes a deployment example for SliceNet infrastructure covering the RAN, the MEC and the Core segments. Finally, Section 5 serves as a conclusion to this document.

2 OpenAirInterface and Mosaic5G Platforms

2.1 An Overview of OpenAirInterface

OpenAirInterface™ (OAI) [2][4] wireless technology platform is a flexible platform to enable an open 4G-5G ecosystem. The platform offers an open-source software-based implementation of a subset of the 4G-5G systems spanning the full protocol stack of 3GPP standard in both E-UTRAN and EPC. It can be used to build and customize a base station (e.g. OAI eNB or gNB), a user equipment (OAI UE) and a core network (OAI EPC) in a PC. In a 4G compatible scenario, the OAI eNB can be connected either to a commercial UE or OAI UE to test different configurations and network setups and monitor the network and mobile device in real-time. In addition, OAI UE can be connected to an eNB test equipment (e.g. CMW500) as well as a commercial eNB (e.g. Amarisoft, IP.Access, etc.).

OAI is based on a PC hosted software radio frontend architecture. With OAI, the transceiver functionality is realized via a software radio front end connected to a host computer for processing. OAI is written in standard C for several real-time Linux variants optimized for Intel x86 and ARM processors and released as free software under the OAI License Model. OAI provides a rich development environment with a range of built-in tools such as highly realistic emulation modes, soft monitoring and debugging tools, protocol analyzer, performance profiler, and configurable logging system for all layers and channels.

2.1.1 Software Platforms

Currently, the OAI platform includes a full software implementation of 4th generation mobile cellular systems compliant with 3GPP LTE standards in C under real-time Linux optimized for x86. At the Physical layer, it provides the following features:

- LTE release 10 compliant, with a subset of release 14;
- Frequency Division Duplex (FDD) and Time Division Duplexing (TDD) configurations in 5, 10, and 20 MHz bandwidth;
- Transmission mode: 1 (Single-Input Single-Output - SISO), and 2, 4, 5, and 6 (Multiple-Input and Multiple-Output - MIMO 2x2);
- Channel Quality Indicator (CQI)/Precoding Matrix Indicator (PMI) reporting;
- All downlink (DL) channels are supported: PSS, SSS, PBCH, PCFICH, PHICH, PDCCH, PDSCH, PMCH;
- All uplink (UL) channels are supported: PRACH, PUSCH, PUCCH, SRS, DRS;
- Hybrid Automatic Repeat Request (HARQ) support (UL and DL);
- Highly optimized base band processing (including turbo decoder). With AVX2 optimization, a full software solution would fit with an average of 1x86 core per eNB instance (64QAM in downlink, 16QAM in uplink, 20MHz, SISO).

For the E-UTRAN protocol stack, it provides:

- LTE release 10 compliant and a subset of release 14 features;
- Implements the Medium Access Control (MAC), Radio Link Control (RLC), Packet Data Convergence Protocol (PDCP), Radio Resource Control (RRC), S1 Application Protocol (S1AP), X2 Application Protocol (X2AP), Generic Tunneling Protocol (GTP) layers;
- Protocol service for all Rel10 Channels and eMBMS (MCH, MCCH, MTCH);
- Full reconfigurable Channel-aware proportional fair scheduling;

- Fully reconfigurable protocol stack;
- Integrity check and encryption using the Advance Encryption Standard (AES) and Snow3G algorithms;
- Support of RRC measurement with measurement gap;
- Standard S1AP and GTP-U interfaces to the Core Network;
- IPv4 and IPv6 support.

Evolved packet core network features:

- Mobility Management Entity (MME), Serving Gateway (SGW), PDN Gateway (PGW), and Home Subscriber Server (HSS) implementations. OAI reuses standards compliant stacks of GTPv1u and GTPv2c application protocols from the open-source software implementation of EPC called nwEPC¹ ;
- Non-Access Stratum (NAS) integrity and encryption using the AES and Snow3G algorithms;
- UE procedures handling: attach, authentication, service access, default and dedicated radio bearer establishment;
- Transparent access to the IP network (neither external SGW nor PGW are necessary). Configurable access point name, IP range, Domain Name System (DNS) and E-UTRAN Radio Access Bearer (E-RAB) quality of service (QoS);
- IPv4 and IPv6 support.

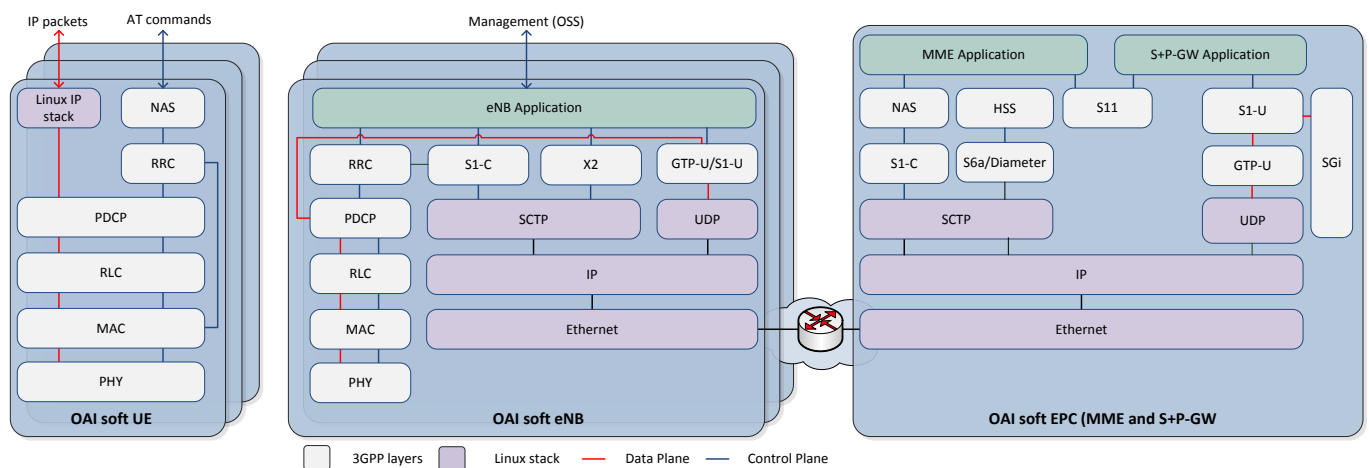


Figure 1. OpenAirInterface software stack

Figure 1 shows a schematic of the implemented LTE protocol stack in OAI. OAI platform can be used in several different configurations involving commercial components to varying degrees:

- Commercial UE ↔ Commercial eNB + OAI EPC
- Commercial UE ↔ OAI eNB + Commercial EPC
- Commercial UE ↔ OAI eNB + OAI EPC
- OAI UE ↔ OAI eNB + OAI EPC
- OAI UE ↔ OAI eNB + Commercial EPC
- OAI UE ↔ Commercial eNB + Commercial EPC

¹ nwEPC – EPC SAE Gateway, <https://sourceforge.net/projects/nwepc/>

2.1.2 Hardware Platforms

OAI is designed to be agnostic to the hardware radio frequency (RF) platforms. It can be interfaced with 3rd party Software-Defined Radio (SDR) RF platforms without significant effort. At present, OAI officially supports the following hardware platforms.

- **EURECOM EXMIMO2:** This board, developed by Eurecom, features four high-quality RF chipsets from Lime Micro Systems (LMS6002), which are LTE-grade MIMO RF front-ends for small cell eNBs. It supports stand-alone operation at low-power levels (maximum 0 dBm transmit power per channel without any power amplifier) simply by connecting an antenna to the board. RF equipment can be configured for both TDD and FDD operation with channel bandwidths up to 20 MHz covering a very large part of the available RF spectrum (250 MHz-3.8 GHz) and a subset of LTE MIMO transmission modes².
- **USRP X-series/B-Series:** This is the Ettus USRP B-series and X-series products that are supported by OAI via Ettus UHD Driver (USB3 and Ethernet)^{3,4}.
- **LIMESDR:** This is the Lime Micro Systems SDR board that is supported by OAI via the Lime USB3 driver^{5,6}.
- **BladeRF:** This is a Nuand SDR board that is also supported by OAI via the bladeRF USB3 driver⁷.

2.2 An Overview of Mosaic5G

Mosaic-5G.io [5] is an ecosystem of open-source platforms and use cases for 5G system research and development leveraging software-defined networking (SDN), network function virtualization (NFV), and multi-access edge computing (MEC) technology enablers to realize the service-oriented 5G vision. With Mosaic-5G, network services can be provisioned on demand and deployed over a virtualised infrastructure, allowing the transition from nowadays vertical dedicated networks to shared and customizable horizontal networks.

Mosaic-5G.io currently provides five main platforms⁸:

- **JOX** is an event-driven juju-based service orchestrator core with plugins architecture to interface with different network domain.
- **FlexRAN** is a flexible and programmable platform for Software-Defined Radio Access Networks.
- **LL-MEC** is an ETSI-aligned Multi-access edge computing platform that also acts as Core network controller for Software-Defined Mobile Networks.
- **OpenAirInterface RAN (OAI-RAN)** is a 3GPP compatible implementation of a subset of features of RAN release 14 with support of FlexRAN.
- **OpenAirInterface CN (OAI-CN)** is a 3GPP compatible implementation of a subset of features of CN release 12 with support of LL-MEC.

² <https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/OpenAirExpressMimo2>

³ <http://www.ettus.com/product/details/UB210-KIT>.

⁴ <https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/HowToConnectCOTSUEwithOAIeNBNew>

⁵ <https://myriadrf.org/projects/limesdr/>

⁶ <https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/how-to-setup-oai-with-lmsdr>

⁷ <https://www.nuand.com/>

⁸ <https://gitlab.eurecom.fr/mosaic5g/mosaic5g>

In addition, Mosaic-5G includes a constellation of platform packages, software development kits (SDKs), network control applications and data sets under the **Store** repository. It allows to develop and bundle plug-and-play network applications tailored to a particular use case, and compose and customize a network service delivery platform across reusable applications.

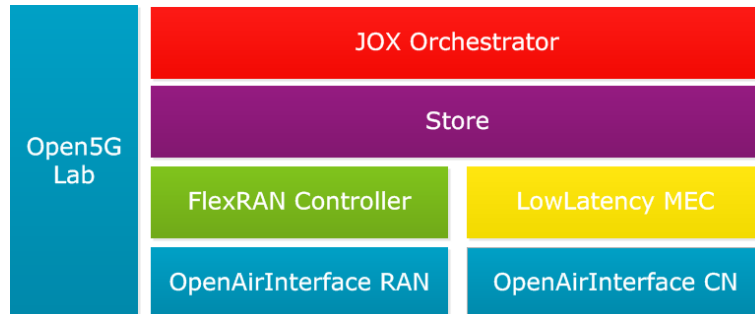


Figure 2. Mosaic-5G.io ecosystem

In the following, we briefly present the three main platforms, namely JOX, FlexRAN, and LL-MEC.

2.2.1 JOX in a Nutshell

JOX is a Juju-based orchestrator for the virtualized network that natively supports network slicing. Using JOX, each network slice can be independently optimized with specific configurations on its resources, network functions and service chains. JOX operates on top of the Juju virtual network function management (VNFM) with a plugins architecture to interface with FlexRAN, LL-MEC and virtual infrastructure management (VIM).

The JOX architecture, as shown in Figure 3, includes two main components: (a) **JOX core** that includes JSlice and Jcloud controller to control slice and cloud resources respectively, and (b) **JOX plugging framework** that enables different plugins for RAN, CN, MEC, and VIM to enable fast reactions like event handling and monitoring. Furthermore, it exposes the northbound REST API to enable several basic operations such as create, (re-)configuration, on each JSlice, connected to a JCloud.

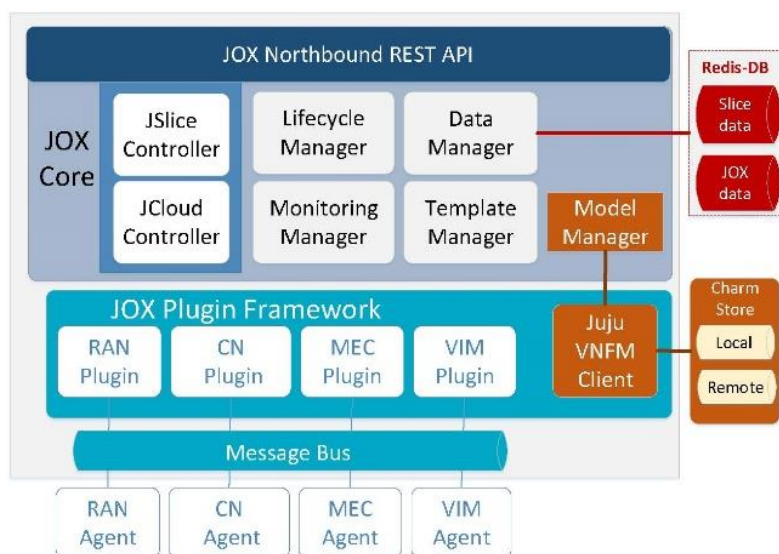


Figure 3. JOX architecture

As illustrated in Figure 4, JOX orchestrates the deployment of the standard LTE chain, i.e., eNB, MME, S/PGW, MySQL, HSS as well as FlexRAN and LL-MEC for a new JSlice, in different environments ranging from physical machine, container or virtual machine. Service dependencies may exist when deploying chains, for instance, the relationship between MySQL and HSS cannot be built until HSS is installed and configured. JOX orchestrates the service deployment and automatically handle dependencies and conflicts through Juju without any actions.

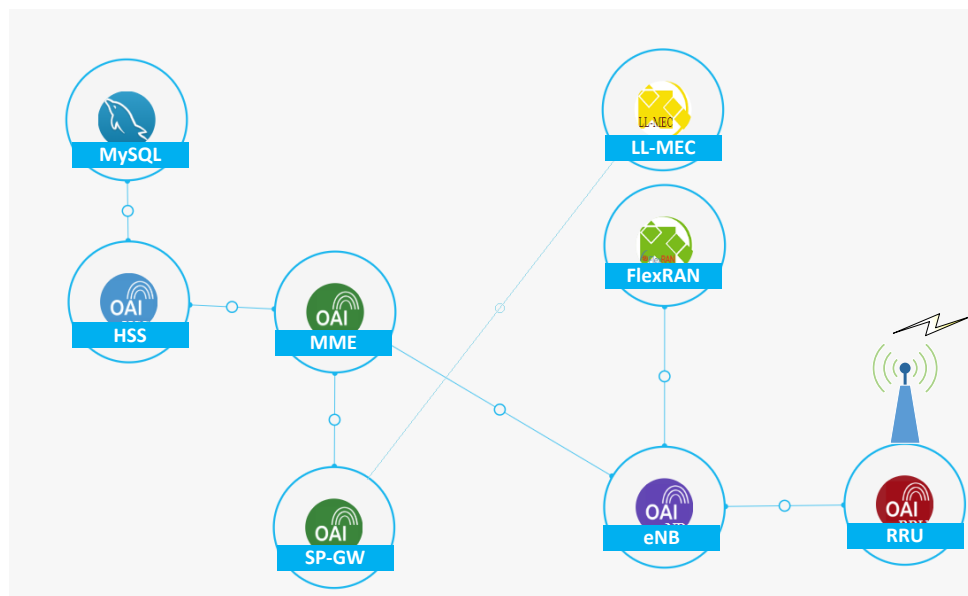


Figure 4. An example of a standard LTE chain with Domain Controllers

2.2.2 FlexRAN in a Nutshell

FlexRAN platform is the first open-source software-defined RAN platform and is designed with flexibility supporting separate control and user plane operations. Moreover, it can either centralize RAN domain control logics among multiple base stations or delegate control decisions in a distributed manner. Hence, FlexRAN provides modulated control functions, separated controller/agent control framework and well-defined APIs for “on-the-fly” control reconfiguration.

Two key elements resides in FlexRAN architecture: (a) Real-time controller (RTC) that enables coordinated control over multiple RANs, reveals network graph primitives and provision SDKs for control application, and (b) RAN runtime that acts as a local agent controlled by RTC, virtualizes underlying RAN radio resources, pipelines RAN service function chain and provides SDKs enabling distributed control application. Practically, the developed As shown in Figure 5, FlexRAN protocol between RAN runtime and real-time controller can provide several characteristics: provide statistics, enable reconfiguration, trigger event and delegate control.

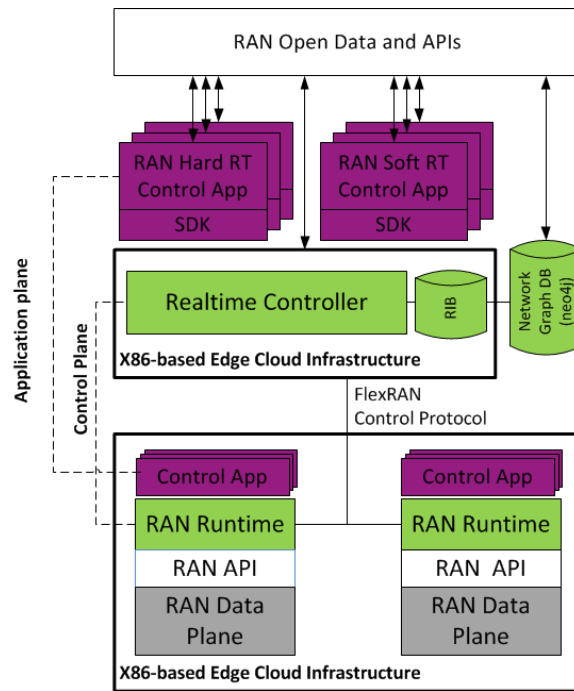


Figure 5. FlexRAN protocol

FlexRAN enables the slice-specific resource abstraction and scheduling to fulfil service requirements, such as throughput (Mbps), latency (millisecond), and reliability (packet drop). For instance, three different slice services (video, eHealth, IoT) can independently apply their customized radio resource management (RRM) control logics as shown in Figure 6.

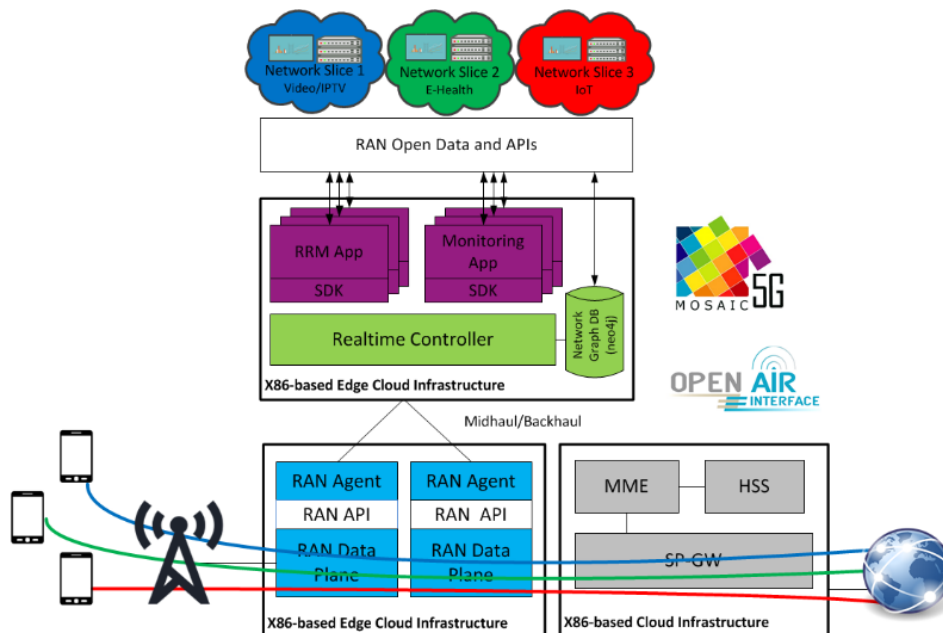


Figure 6. Different slice services in FlexRAN.

2.2.3 LL-MEC in a Nutshell

LL-MEC platform leverages SDN principle to separate user plane processing from its control logics at the edge and core networks. With OpenFlow, the user plane is abstracted for the purposes of monitoring, analysis and control. The OpenFlow protocol is applied over the

Open Virtual Switch (OVS) to enable user plane programmability. Further, SDKs are provided to enable a flexible MEC application development environment.

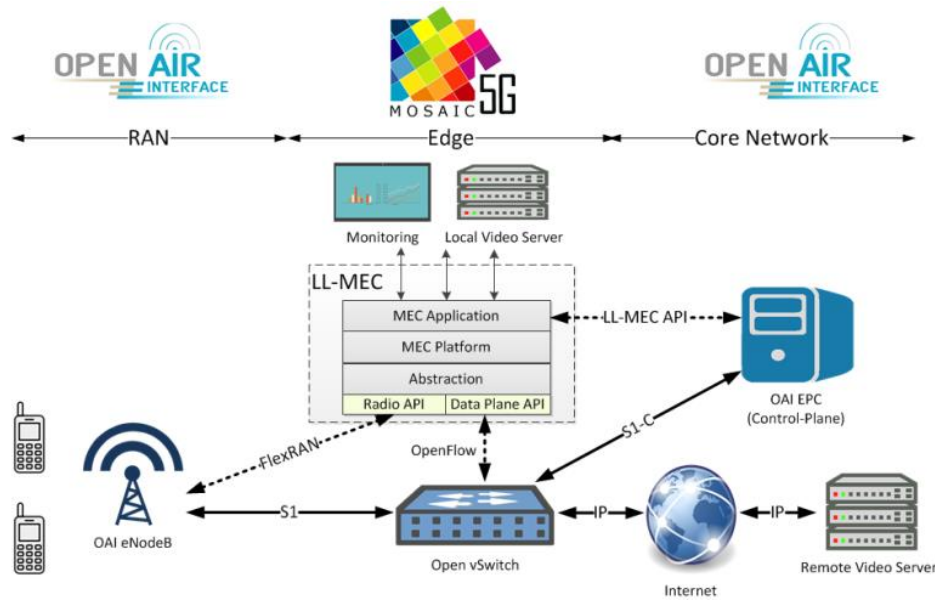


Figure 7. LL-MEC Platform [9]

LL-MEC platform is aligned with the ETSI MEC Mp1 and Mp2 reference interfaces. The Mp1 interface enables low-latency or elastic MEC applications through Core API, REST API and message bus, while Mp2 can instruct user plane how to route traffic among applications, networks, services, etc. Within LL-MEC, two services are provided: (a) Edge packet service (EPS) (equivalent to traffic rule control) that manages the static and dynamic traffic rules and handles multiple OpenFlow libraries and OVS, and (b) Radio network information service (RNIS) that exposes real-time RAN information (e.g., user and radio bearer statistics) and delegates the control decision over the user plane.

LL-MEC enables versatile applications, such as radio-aware video content optimization. It aims to adjust video quality based on the real-time per-user wireless channel quality to reduce the video stalling and utilize all available radio resources. For instance, a user with poor channel condition only receives a 240p video, while another one can enjoy 4k video streaming when it is close to the base station.

3 Virtualised Slicing-Friendly 5G Infrastructure

This section highlights different tools for deploying a virtualized 5G infrastructure. Figure 8 shows the mapping of the deployment tools to ETSI NFV reference architecture [6].

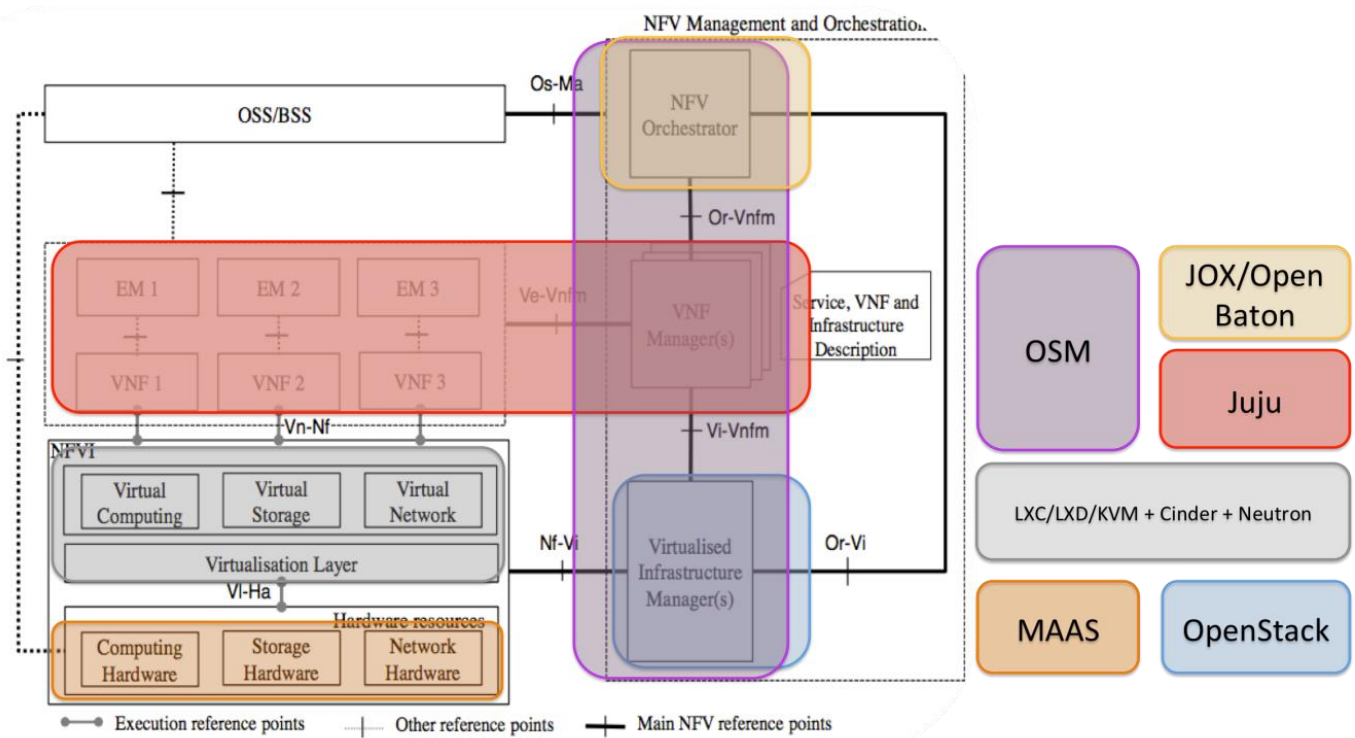


Figure 8. Mapping the deployment tools to ETSI NFV architecture [6]

A Virtual Infrastructure Management (VIM), according to the ETSI NFV management and orchestration (MANO) specification [7], is a virtual infrastructure manager that provides the Infrastructure-as-a-Service (IaaS) by assembling different NFV Infrastructures (NFVIs), each with different technologies/vendors, and abstracting them into compute, storage and network nodes/resources. Existing solutions for VIM include OpenStack [8], VMware vSphere, CloudStack, Google Kubernetes VIM, etc. As mentioned in D3.1 [9], SliceNet proposes to use OpenStack VIM as NFVI management. OpenStack is an open-source software service framework, which provides service provisioning and virtualization. OpenStack architecture is modular and pluggable, thus allows using the most appropriate modules according to the need. As an OpenStack module, Heat provides orchestration of services, however, is limited to OpenStack-based platform.

In ETSI NFV architecture, a VNFM is responsible for the lifecycle management of Virtual Network Functions (VNFs). VNFM takes care of deploying, monitoring, scaling and removing VNFs on a VIM. Juju [10], which is one of the main VNFM for ETSI Open Source MANO (OSM)[11], is mainly adopted as a VNFM in SliceNet.

An NFV Orchestrator (NFVO) is responsible for the Network Slice (NS) lifecycle management together with the VNF lifecycle (supported by the VNFM) and the NFVI resources (supported by the VIM). Based on a preliminary investigation, SliceNet proposes three solutions to explore Juju-based orchestrator (JOX) [12], Open Baton [13], and OSM [11] as an NFVO. In the context of this document, we will mainly focus on JOX while briefly introducing Open Baton and OSM.

Regarding NFVI, OpenStack, which is fundamental to the VIM, provides different key components for NFVI management including: (i) OpenStack Compute (Nova) for managing virtual or bare metal servers; (ii) OpenStack Block Storage (Cinder) for virtual storage; and (iii) OpenStack Networking (Neutron) providing virtual networking [14]. In more detail, Nova supports a wide variety of compute technologies such as Kernel-based Virtual Machine (KVM), Xen, Linux Container (LXC), Hyper-V, VMware, XenServer, OpenStack Ironic and PowerVM, which provides the flexibility in choosing a hypervisor(s). Neutron provides networking functionality between interface devices (e.g. vNICs) managed by other OpenStack services and supports advanced network services. Interestingly, Neutron has also enabled adoption of control and management technologies for software-defined networking (SDN) [14]. These SDN services may interact with other Neutron’s components through REST APIs. For instance, OpenStack can work with several SDN controllers such as OpenDaylight (ODL) [15], ONOS [16], etc.

Finally, Metal as a service (MaaS) [17], which is responsible for hardware resource management, provides an easy way to set up the hardware on which to deploy any service that needs to scale up and down dynamically.

In this section, we introduce the deployment tools together with several examples on how to use these tools for the deployment of OAI-based 5G services. Again, we remind the reader that the infrastructure itself is for C-RAN deployment.

3.1 Automated Deployment of 5G Virtualised Infrastructure through Juju

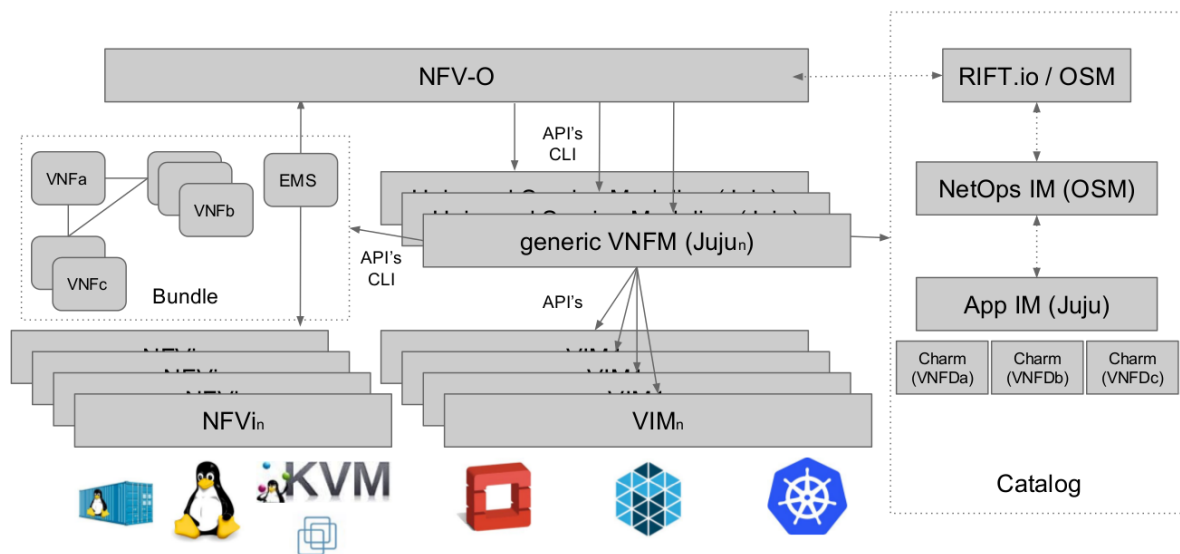


Figure 9. Juju - Open Source Generic VNFM [18]

Juju is a generic VNFM in the ETSI NFV architecture, which can be used to quickly and efficiently deploy, configure, scale, integrate and perform operational tasks in a wide selection of public clouds such as Amazon Web Services (AWS), Azure, Google Compute Engine (GCE), and Rackspace, as well as in private clouds like OpenStack and VMware, along with bare metal servers (MaaS) and containers. Juju models services, their relationships and scale regardless of the underlying infrastructure. In more detail, Juju defines a service as a group of units, which is an approach that allows to easily scale in or out services, simply by adding or removing units. In its essence, Juju takes care of installation, configuration and communication among services, yet without taking the actual decisions for a particular

service. Instead, it delegates service-specific decisions to a set of scripts called “Charms” that implement the service behavior. Charms contain all necessary instructions for deploying and configuring a service by orchestrating the entire lifecycle of the service: a Charm defines how the service should be fetched, installed and run, how configuration files should be filled up and how to react to events.

A collection of Charms that link services together is called a “Bundle”. A Bundle allows to deploy whole chunks of app infrastructure in one go. According to [19], a Charm corresponds to a service definition and a collection of Charms and Bundles corresponds to the NS catalogue according to ETSI. In addition, the process of uploading and deploying Charms into Juju corresponds to the NS onboarding and instantiation process, respectively. A global Charm catalogue containing all available Charms and Bundles that can be found in the Juju store [20].

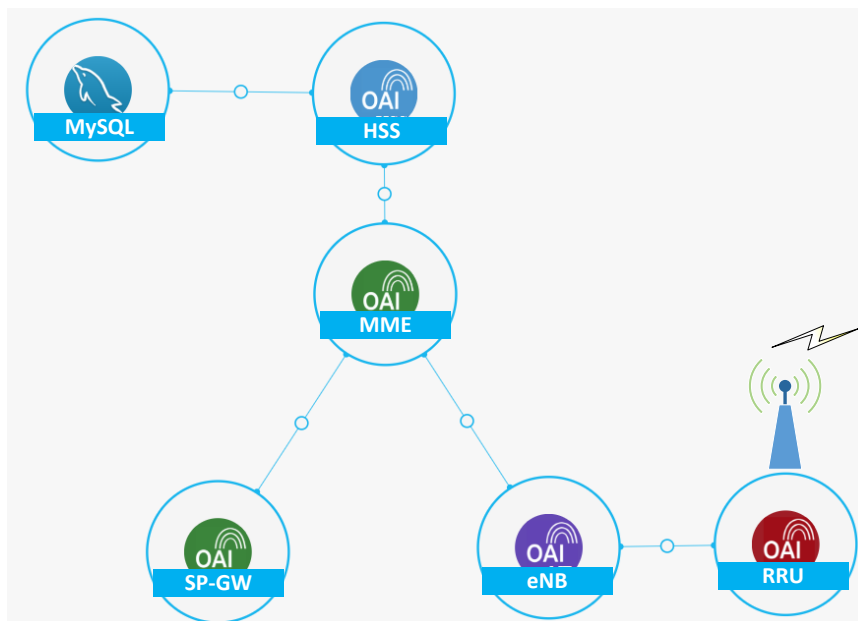


Figure 10. An example of a Bundle from the Juju Store

Using OAI C-RAN as an example, Figure 10 shows the OAI C-RAN bundle from the Juju store⁹. This Bundle defines a service template for a 5G C-RAN deployment with functional split based on OAI. It consists of the following Charms: MySQL, OAI-HSS, OAI-MME, OAI-SPGW, OAI-eNB and OAI-RRU. More details on how to create a Bundle/Charm, as well as on the structure and organization of a Charm will be provided in subsection 3.1.4.1.1.

In what follows next, we explain in more detail the notion of a Charm along with that of a Juju controller and model.

3.1.1 Juju Charms

Juju is a service modelling tool based on a concept of Charms to handle deployment and management of various cloud-based applications. Conceptually, Charms are composed of metadata, configuration data and hooks with some extra support files in order to download, configure, install, scale and maintain a service. This abstraction allows a very rapid deployment and maintenance of services, even without a detailed knowledge about the

⁹ <https://jujucharms.com/u/nauid-nikaein/oai-5g-cran/>

internals of the service itself. One important remark is that a Charm does not contain the code of a service itself, but merely the scripts, which can download the code from upstream sources.

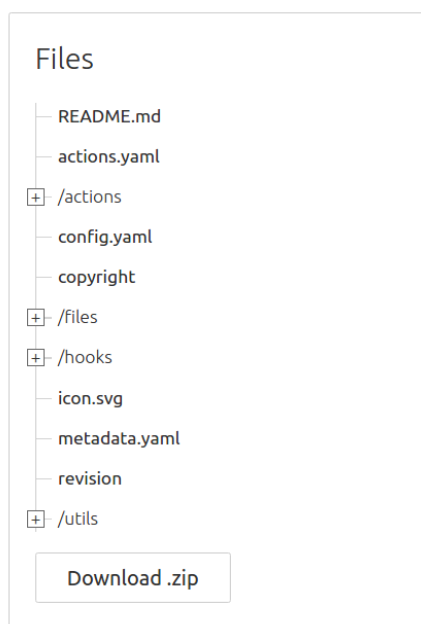


Figure 11. Structure of a Charm (EPC Charm¹⁰ as an example)

Specifically, the Charm is made up of (i) a “metadata.yaml” file describing in general the Charm with basic declarative information, defining the set of relations that the service can participate to as well as which services the Charm offers to other Charms; and (ii) a “config.yaml” with the specified options exposed to the user for service configuration and a set of hooks that are invoked by the Juju agent in order to trigger events in a Charm. Juju manages the service lifecycle with hooks (or scripts) implemented inside Charms. Currently, there are five unit hooks, namely: install, config-changed, start, upgrade-charm and stop. These hooks are invoked during the lifecycle of a service, as specified in the Charm’s configuration file. Besides this, there are four relation hooks for each interface that a Charm supports, named after the interfaces: (i) ifaceName-relation-joined, (ii) ifaceName-relation-changed, (iii) ifaceName-relation-departed and (iv) ifaceName-relation-broken to handle cases where the interface is connected to the service, or disconnected, or the configuration or settings of that interface are changed.

Generally, Charms are mostly used to model more complex deployments, potentially including many different applications and connections. As a result, a Bundle allows installing an entire working deployment easily and quickly as a Charm. The bundle consists of three main sections:

- Target machines specifications and constraints;
- Charms declaration and configuration;
- Charms relations.

Specifically, the first section of the Bundle defines a set of machines, which are required to deploy the services included in the Bundle. This information is passed to a cloud provider

¹⁰ <https://jujucharms.com/u/navid-nikaein/oai-epc/trusty/22>

(MaaS, OpenStack, AWS, GCE, etc.), which in turn provisions the machines using this specification and gives them back to Juju for further deployment of services on top of them. The user is able to define several constraints such as the number of Central Processing Unit (CPU) cores, Random-Access Memory (RAM) size, availability zones or tags to offer a hint to the cloud provider about the user's preferences about the machines that match their interest.

The second section declares all the Charms that the Bundle needs along with configuration options. These options can be set to default values, which allows the Charm to operate correctly. Then, a user can change the value of each option via Juju. Juju then requests a machine from the cloud provider and places a single Charm unit on top of it. The placement of Charm units can also be specified.

The third and last section of the Bundle specifies the relationships between Charms. Each Charm provides some capabilities while it also consumes some others. This is the modelling part of Juju. Using the GUI, a user can simply perform drag and drop actions on the Charms to connect them. One thing to remember is that Bundle defines connections between the Charms, not the Charms units.

It is important to note that a Charm unit is a single instance of an application deployed by a Charm. Most of the Charms can deploy multiple units of the application. This is the basic scalability feature of Juju. At this point, we remind the reader that service scalability depends on the number of units. For example, let us assume that an application is using the database heavily, thus we need to add a database replica to ease the load on the original one. To do so, the user can add one or multiple units of MySQL¹¹ via Juju. Juju will automatically request additional machines from the cloud provider, install MySQL on them, and configure the replication and load balancing details between the instances.

```
Juju add-unit -n5 mysql #add 5 units of MySQL12
```

3.1.2 Clouds

As mentioned earlier, Juju can use a number of public clouds (including AWS, Azure, GCE, and Rackspace) to deploy workloads, as well as private clouds (e.g. OpenStack, vSphere, MaaS) which you configure. Additionally, Juju can work directly with physical, virtual and container machines. It makes Juju independent of substrate, which is very important both for production and for the development cycle [19].

3.1.3 Controllers and Models

For management, Juju creates a special node, which is called the “Juju Controller”, during bootstrap/installation stage. This controller hosts the database, manages all the machines in the running models and responds to all events that are triggered throughout the system. It also manages the scale-out, configuration and placement of all models/applications, user account and identification, access and sharing.

In order to facilitate the management of a group of applications and resources to accommodate different workloads or use cases, Juju introduces the notion of “model”. A model is associated with a specified controller. One typical example for model is using

¹¹ <https://jujucharms.com/mysql/>

¹² *If you need to place more than one unit on a machine, use the “--to” option*

different models to deploy different regions. Model then can be replicated while keeping the same configuration of machines that Juju creates within this model, the application that get deployed on those machines and their relationships. Additionally, models can be added easily at any time.

3.1.4 Technical Use Cases

This section explains the deployment of OAI 5G C-RAN as a typical example of how to use Juju/JOX to deploy virtualized 5G infrastructure.

C-RAN Deployment - a High Level Overview

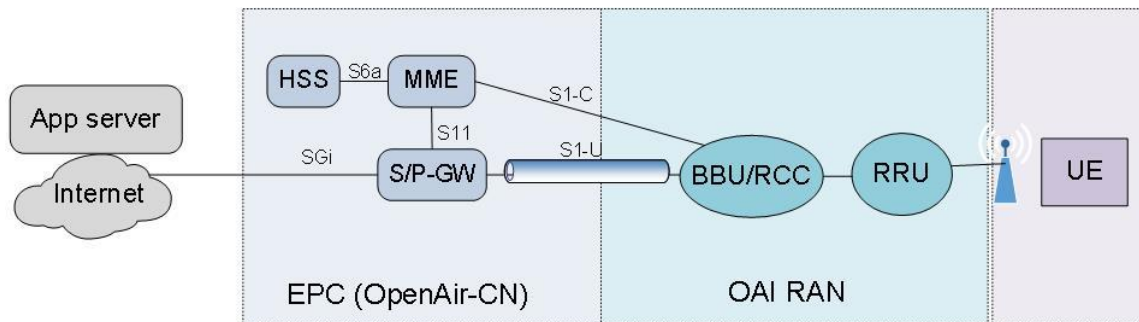


Figure 12. C-RAN architecture and components.

In recent years, C-RAN (Centralized, or Cloud Radio Access Network) with centralized processing in Baseband Units (BBUs) and Remote Radio Units (RRUs) using CPRI (Common Public Radio Interface)¹³ has been more and more deployed thanks to its significant advantages including network deployment, reduced operating costs, as well as improved network performance [21]. However, due to high bandwidth requirements, CPRI requires expensive fronthaul to carry Radio Frequency (RF) samples from BBU to RRU, resulting in rising costs. In [21], the authors argued that to meet the requirement in terms of capacity, density and other performance aspects (e.g., service delay, user bandwidth) of 5G, fronthaul interfaces need to provide low delay and high-bandwidth transmission services by means of restructuring of functions between the BBU and RRUs. As a result, the new C-RAN architecture has been proposed by splitting different parts of radio stack between different network elements (BBU and RRU) [21] [22]. Also, a new BBU and RRU interface based on packet transmission technology, namely Next Generation Fronthaul Interface (NGFI), has been defined [21] [22].

In NGFI architecture, some BBU functions are shifted to RRU. Accordingly, BBU is redefined as the Radio Cloud Center (RCC) while RRU becomes the Radio Remote System (RRS). The Radio Aggregation Unit (RAU) allows interfacing RCC with several RRUs. RCC connects with RRS via the NGFI interface. Regarding different NGFI interfaces, the authors in [23] show the potential designs of NGFI split-points for LTE network proposed by China Mobile [21]. Among them, we just highlight:

- **IF4p5 split-point:** IF4p5 corresponds to the split-point at the input (TX) and output (RX) of the OFDM symbol generator (i.e. frequency-domain signals) [22]. According to [22], IF4 is “Resource mapping and IFFT” and “FFT and Resource de-mapping”. Therefore, IF4p5 is simply compressed transmitted or received resource elements in the usable channel band.

¹³ CPRI, <http://www.cpri.info>

- **IF5 split-point** (Baseband/RF divisions): In this solution, RRU only implements RF-related functions.

The current version of OAI supports both split-points, however, we use the IF4P5 split-point as an interface between BBU/RRC and RRU in our deployment scenario.

Figure 12 shows the high-level architecture of the C-RAN deployment which consists of core network part (including HSS, MME, SGW, and PGW) and RAN part (BBU/eNB and RRU). In our deployment scenario, OAI-CN is used to deploy the functionality of an EPC while OAI-RAN for the BBU and RRU functionality.

3.1.4.1 Deploy of slice-friendly LTE Service Chain with Juju

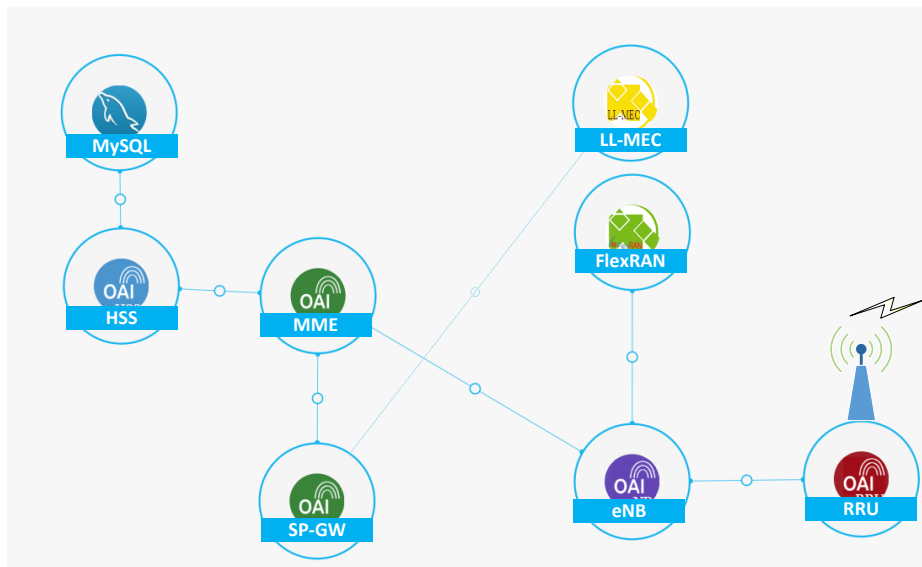


Figure 13. Slice-Friendly virtualized 5G C-RAN and CORE Slice deployed by Juju

This section explains the deployment of OAI 5G C-RAN as an example of how to use Juju to deploy a virtualized 5G infrastructure. Figure 13 shows the OAI Bundle from Juju Charms store regarding its components and their relations. This Bundle makes use of the following Charms: MySQL, OAI-HSS, OAI-MME, OAI-SPGW, OAI-eNB, OAI-RRU, FLEXRAN, and LL-MEC¹⁴ to create a slice-friendly 5G C-RAN slice.

In the following, we will describe the details structure of a Charm by taking OAI-MME as an example.

3.1.4.1.1 Structure of a Charm (OAI-MME)

As mentioned earlier, a Charm is a set of files that typically consists of instructions for deploying and configuring a service. Basically, the following files are included in a Charm:

- a **metadata.yaml** file describing the Charm with basic declarative information, defining to which relations the service can participate in and which services the Charm offers to other Charms;
- a **config.yaml** specifying the options that are exposed to the user for the service configuration;
- **Hooks** which are invoked by the Juju agent to trigger events in a Charm.

¹⁴ <https://jujucharms.com/u/navid-nikaein/>

Metadata.yaml

This file defines the features of the service Charm and the kind of relations it can participate in. The name field represents the Charm name, which is used to form the Charm Uniform Resource Locator (URL) of the online Charm store. The summary and description are to describe the Charm and its features. The tag is used to sort the Charm in the store. The “provides” and “requires” subfields list the service relations the Charm may participate in and they are complementary, so a service that provides an interface can only have that specific relation established with a service that requires the same interface, and vice versa.

```

name: oai-mme
summary: Evolved Packet Core Network (EPC) based on OpenAirInterface
maintainers:
  - Navid Nikaein <navid.nikaein@eurecom.fr>
  - Andrea Bordone Molini <bordone@eurecom.fr>
description: |
  This Charms allows you to design, deploy, provision, and dispose your 4G-5G
  OpenAirInterface EPC out of the box on any cloud infrastructure.
tags:
  - Telecom
  - 4G-5G
  - EPC
  - Core Network
  - OpenAirInterface
  - Eurecom
subordinate: false
provides:
  mme:
    interface: S1-C
requires:
  hss:
    interface: S6a-hss
  spgw:
    interface: S11

```

The interface names can be whatever and using the same interface name in another Charm allows the two to exchange information through juju infrastructure (juju state server). Specifically, OAI-MME Charm provides a service named “MME” through the S1-C interface. The name recalls the virtual interface name between the MME and the eNB/BBU in the LTE architecture, but here the purpose is simply to point out what Charm should be related to OAI-MME Charm and towards which interface. In fact, OAI-eNB (and OAI-BBU) Charm will require the OAI-MME Charm and it will use the same interface name, which represents the protocol conventionally followed by the service units to exchange information.

OAI-MME Charm will work together with OAI-HSS Charm that provides the “hss” service, therefore the OAI-MME Charm will require this service on the same interface (S6a-hss). Actually, this kind of information will be exchanged through interfaces, once the Charms are deployed, is defined by a set of files (relation hooks) in the Charms themselves and it happens a layer above the deployed software and it is necessary to provide to the services

encapsulated by the Charms the condition to properly run and coexist. With the same reason, OAI-MME Charm requires OAI-SPGW Charm.

For more information regarding Charm metadata, please refer to [24].

Config.yaml

```
options:
  branch:
    default: "develop"
    description: |
      branch indicates which installation you want to do. If you want a stable installation,
      change this option to "master".
    type: string
  revision:
    default: "head"
    description: get a specific revision from the openair-cn git repository.
    type: string
  kernel:
    default: "generic"
    description: set the default kernel, generic or low latency.
    type: string
  realm:
    default: "openair4G.eur"
    description: |
      Diameter realm of the MME. HSS and EPC have to have the same. NO empty value.
    type: string
  eth:
    default: "eth0"
    description: |
      This is the S1-C interface name.
    type: string
  maxenb:
    default: "10"
    description: Maximum number of eNB that can connect to MME. Max Value is 10.
    type: string
  maxue:
    default: "10"
    description: |
      For debug purpose, used to restrict the number of served UEs the MME can handle. Max
      Value is 50.
    type: string
  relative_capacity:
    default: "10"
    description: |
      Even though this parameter is not used by the MME for controlling the MME load
      balancing within a pool (at least for now), the parameter has to be forwarded to the eNB
      during association procedure. Values going from 0 to 255.
```

```

type: string
mme_statistic_timer:
  default: "10"
  description: |
    Displayed statistic (stdout) period. You can access the stdout: cat /srv/mme.out on the
    machine where this Charm is deployed.
  type: string
emergency_attach_supported:
  default: "no"
  description: This will attach the unauthenticated UEs (not supported).
  type: string
authenticated_imsi_supported:
  default: "no"
  description:
  type: string
verbosity:
  default: "none"
  description: sets the asn1 log level verbosity. Valid values are "none", "info", or "annoying"
  type: string
gummei_tai_mcc:
  default: "208"
  description: TAI=MCC.MNC:TAC. MCC is the Mobile Country Code. Must be three digits.
  type: string
gummei_tai_mnc:
  default: "95"
  description: TAI=MCC.MNC:TAC. MNC is the Mobile Network Code. Must be two or three
  digits.
  type: string

```

The optional config.yaml file defines how the software can be configured by the user. The objective is to expose to the user the options that he/she would be willing to tweak when deploying the service or when it is running.

The Charm should operate correctly with no explicit configuration settings. In fact, there is a default value associated to each option. Moreover, Juju allows providing a config.yaml file with the desired values for the options at the deployment time. Another option is that the user can reconfigure the software by using Juju tool when the service is running. In our case, this file allows the user to have partial control on the way the MME is built and the way the MME will behave once it is running. In a typical deployment scenario when OAI-MME is installed as a standard-alone application in a physical or a virtual machine, these options can be put in a configuration file and then to be parsed to the OAI-MME.

In the following, we highlight several options that allow the user to choose how to configure the OAI-MME:

- **branch:** specify the “git branch” from where the source code must be fetched;
- **realm:** use “realm” to assign to the MME element used to talk over the diameter protocol;
- **maxenb:** define the maximum number of eNBs that the MME can support;

- **maxue**: define the maximum number of UEs that the MME can handle;
- **gummei_tai_mcc, gummei_tai_mnc**: define the Public Land Mobile Network (PLMN), Mobile Network Code (MNC) and Mobile Country Code (MCC) to assign to the deployed mobile network.

Charm's Hooks

Hooks are a series of files that are called during the lifecycle of the service encapsulated within the Charm. A service unit's direct action is entirely defined by its Charm's hooks that will be invoked by Juju at particular times. Based on triggered events, Juju will fire up a specific hook to apply the change on the machine. Hooks may be written in any language. They run non-concurrently to inform the Charm that something happened, and they give a chance for the Charm to react to events in arbitrary ways.

The set of common unit hooks with predefined names are:

- install
- start
- stop
- config-changed
- upgrade-charm
- update-status

In the context of OAI-MME, the defined relation hooks are strictly related to which services the Charm needs or exposes. When services are related, Juju decides which hooks to call within each Charm based on the local relation name. Specifically, OAI-MME requires relations called "hss", "spgw" and provides a relation called "mme" so the following relation hooks have been defined to manage the relations' lifecycles:

- hss-relation-broken
- hss-relation-changed
- hss-relation-departed
- hss-relation-joined
- mme-relation-broken
- mme-relation-changed
- mme-relation-departed
- mme-relation-joined
- spgw-relation-changed
- spgw-relation-departed

It is not mandatory to use all units or relation hooks, so at the time Juju would call them, it may simply skip the execution of some of them.

3.1.4.1.2 Juju Charm Deployment

After preparing the machines by either leveraging on a cloud infrastructure or manually installing/creating the machines, we can use a simple Juju command to deploy the Bundle as following:

```
$juju deploy oai-5g-cran-slice.yaml
```

In our case, we deployed OAI C-RAN bundle on top of a private MaaS-cloud.

Scaling

As mentioned earlier, Juju not only makes it simple to deploy services, but also crucially makes it easy to manage them too. Its dynamic configuration ability, which allows the operator to re-configure services on the fly, add, remove, or change relationships between services, and scale in or out and up or down with ease [25]. Such scaling capabilities are crucial to achieve scalable network slices.

Spinning up another unit of a certain service allows having the chance of distributing the load over the different service instances. In general, a load balancer will be needed in front of the service units in order to actually distribute the incoming requests to the different instances. However, in our scenario the OAI software components themselves will operate the load balancing. In fact, the eNB is in charge of choosing the MME where to forward the signalling messages coming from a UE. In LTE, the scalability can be operated either for the MME or for the SPGW in different ways. For example, the following commands will add a new OAI-MME instance and link MME with other services.

```
$juju deploy oai-mme oai-mme_1
$juju add-relation oai-mme_1 hss
$juju add-relation oai-spgw oai-mme_1
$juju add-relation oai-enb oai-mme_1
```

Additionally, a network operator might want to scale up a particular service deployed inside the environment that means it might increase manually the computational power of the virtual machine where that OAI-MME service is deployed. Alternatively, we can also use another more powerful machine from the group of machines added to the manual environment, by re-deploying that service through Juju. The following commands will replace an MME instance by a new one with a different computational power.

```
$juju remove-service oai-mme_1
$juju deploy --constraints "cpu-cores=4 ram=4G" oai-mme oai-mme_1
$juju add-relation oai-mme_1 hss
$juju add-relation oai-spgw oai-mme_1
$juju add-relation oai-enb oai-mme_1
```

3.1.4.2 Deploy LTE Service Chain for Network Slicing with JOX

As mentioned in the previous section, Juju can be used to deploy a virtualized 5G infrastructure in an automated way. However, in this approach, Juju, as a VNFM, is mainly responsible for VNF lifecycle management (e.g., instantiation, update, query, scaling, and termination). As a result, it lacks the functionalities at the network service-level regarding lifecycle management, global resource management, and policy management. It is where an NFVO comes into play, especially to achieve slicing-friendly infrastructure. Generally, an NFVO, with a complete overview of the system, is responsible for [7]:

- maintaining a global view of system;
- on-boarding of application packages;
- NS lifecycle management (including instantiation, scale-out/in, performance measurements, event correlation, termination);
- global resource management, validation and authorization of NFVI resource requests;

- policy management for NS instances.

SliceNet proposes JOX - a Juju-based orchestrator, as an NFVO not only for deploying virtualized 5G infrastructure but also for MEC platform and its application (as specified in D3.1 [9]). One of the main reasons is that JOX is created as a 5G orchestration targeting network slicing. As a result, JOX inherently supports lifecycle management of network slices and orchestration for the mobile network. Specifically, it supports basic operations defined by 3GPP in TR 28.801 [26] to manage the lifecycle (preparation, instantiation, configuration, activation, runtime and decommissioning phase) of a Network Slice Instance (NSI), where all phase related API methods are exposed via the Northbound API. Besides, JOX also supports orchestration for the Mobile Network where it exploits RAN and CN specific plugins to efficiently orchestrate the network resources and services. Furthermore, JOX also supports the optimisation of the operational environment, for example, running a slice-specific logic or global optimisation on all slices applications on top of the Northbound API.

Using JOX, each network slice can be independently optimized with specific configurations on its resources, network functions and service chains. Inside the JOX core, a set of services is used to operate and control each network slice, while at the same time support the necessary interplay between resource and service orchestration, VNFM and VIMs as these are defined in the ETSI MANO architecture [7]. From the implementation perspective, JOX is tightly integrated with the Juju VNFM framework provided by Canonical [27]. The Juju system is also one of the main VNFM for ETSI OSM [11] [28].

The core JOX characteristics are summarized as follows:

- slice-specific lifecycle management and a powerful northbound API;
- core services facilitate the optimization of the orchestration procedures;
- JOX Plugin Framework where each plugin element interacts with the corresponding agent via a message bus, for example, RAN specific plugins in order to control the physical or virtualized LTE eNB;
- slice descriptors are coupled with the service configuration;
- network slice logic can be easily introduced as an application for slice optimization.

In the following paragraphs, we will highlight the architecture, components and the implementation of JOX as well as an example of how to use JOX for orchestrating LTE eNB resources.

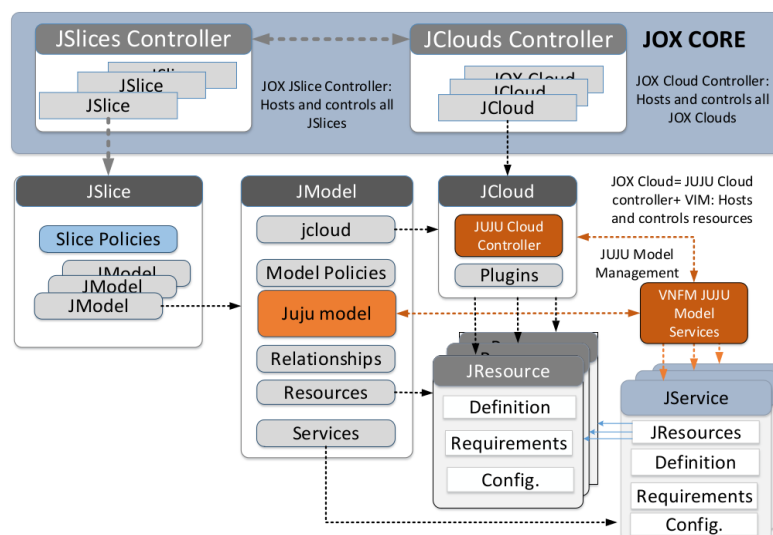


Figure 14. JOX main components and properties

Figure 14 presents the main components of JOX including JOX Network Slices (JSlices) and JOX Clouds (JClouds). In JOX, a slice is represented by a JSlice object that is defined as a set of models (called JModels) together with a policy specification. This policy may be global for all the slice models, while every model is deployed over a specific cloud infrastructure that is controlled by a single VIM (e.g. RAN-VIM). Every JModel is a Bundle of:

- **resources:** include all physical (e.g. servers, spectrum) and virtual resources (e.g., virtual machines (VMs));
- **services:** include physical or virtual network functions (PNFs and VNFs) such as eNB and vMME and virtualized network applications (VNAs) (e.g. monitoring);
- **service chains:** describe the relationship between PNFs/VNFs/VNAs (e.g. between eNB and vMME);
- **policy:** a JModel-specific policy.

Every JCloud object hosts all the underlying cloud resources and interacts with the physical infrastructure and the cloud control mechanisms through two channels: (1) the VNFM for a set of basic functionalities, and (2) directly with the VIM for fine-grain monitoring and control. Although VNFM is able to interact with the VIM, it is the direct communication between the orchestrator and the VIM that can offer the maximum level of control of the underlying physical and virtual infrastructure.

JOX is a single VNFM - multi VIM orchestrator. The VNFM is Canonical's Juju, which interacts with Charms that act as structured NFV element managers driven by Juju. A Charm encapsulates a VNF as a service and contains all the necessary hooks (i.e., scripts and primitives) to manage the life cycle of the VNF and its relationships within service chains. It contains all the logic required to deploy, configure, integrate, scale, and expose the service to the outside world, that are available to JOX through a rich Juju API. We highlight that a rich set of OAI-based 4G and a subset of 5G VNFs (for MEC, RAN and CN) are already available as Juju Charms in the Juju store (an online VNF catalogue [27]).

Juju supports a number of VIMs and a variety of the clouds including both public clouds and private ones such as AWS, Azure, GCE, Rackspace, MaaS and LXC. With JOX, the underlying cloud resources are extended with physical or virtual RAN and CN elements and a set of

resources that also include, for example, radio spectrum and resource blocks. Note that while JOX exploits all the services exposed by Juju regarding the resource management of the infrastructure, the API between Juju and VIMs is restricted to a basic set of functionalities (e.g., deploy VMs with specific requirements). In order to retrieve/analyze the real-time performance and trigger custom events, JOX exploits direct communication with the VIMs through a plugin framework.

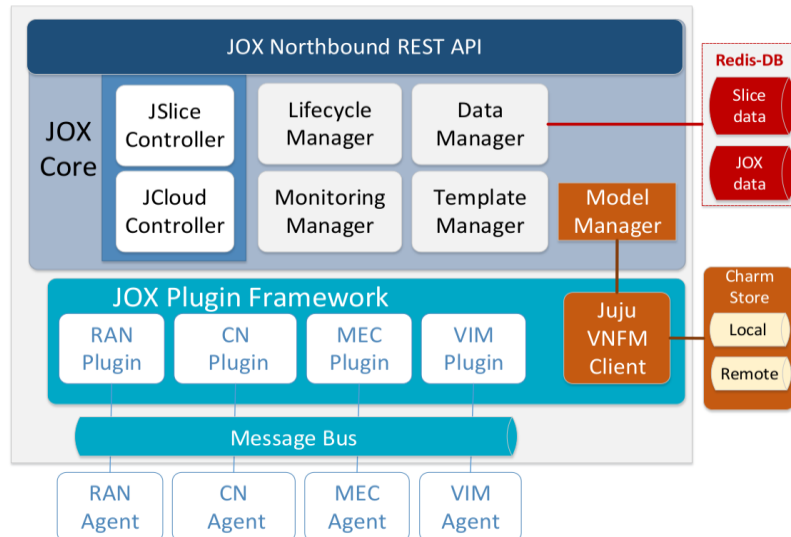


Figure 15. JOX architecture

The architecture of JOX is depicted in Figure 15. JOX exposes a northbound REST API. Using the exposed methods one can create a JSlice, connect to a JCloud and adjust all the models, resources, services and service relationships. In more detail, JOX is aligned with the recommendations of the basic operations that are defined by 3GPP in TR 28.801. According to this, the NSI lifecycle phases are preparation, instantiation, configuration, activation, runtime control and decommissioning. Through the API methods related to these phases are exposed. A set of core services is used in support of slice-specific lifecycle management, data handling, monitoring and template management. Specifically, JOX Slices Controller (JSC) is responsible to host and control all the instantiated JSlices. This is the place where global optimizations can be performed. JOX Clouds Controller (JCC) is responsible to host and control all the instantiated JClouds. JCC offers services to the JSC. JOX enables network slice lifecycle management and allows to orchestrate each network slice independently. JOX exploits RAN and CN specific plugins to efficiently orchestrate the edge network resources and services e.g. orchestrating a new slice across multiple eNBs, partitioning the radio resources and deploying a dedicated CN for this newly generated slice. It is important to note that the core of JOX framework is technology-agnostic, and is able to support 4G/5G technologies through the plugin architecture. With the same principle, it can be easily integrated with different VIMs via the plugin framework.

Network Slice Definition, Control and Management

In JOX, a network slice is represented by a JSlice. To define a JSlice, the NSI owner defines a set of resources, requirements and services, service relationships and the corresponding configurations. Currently, JOX provides a simple JSlice definition since standard templates definitions is currently an open research issue.

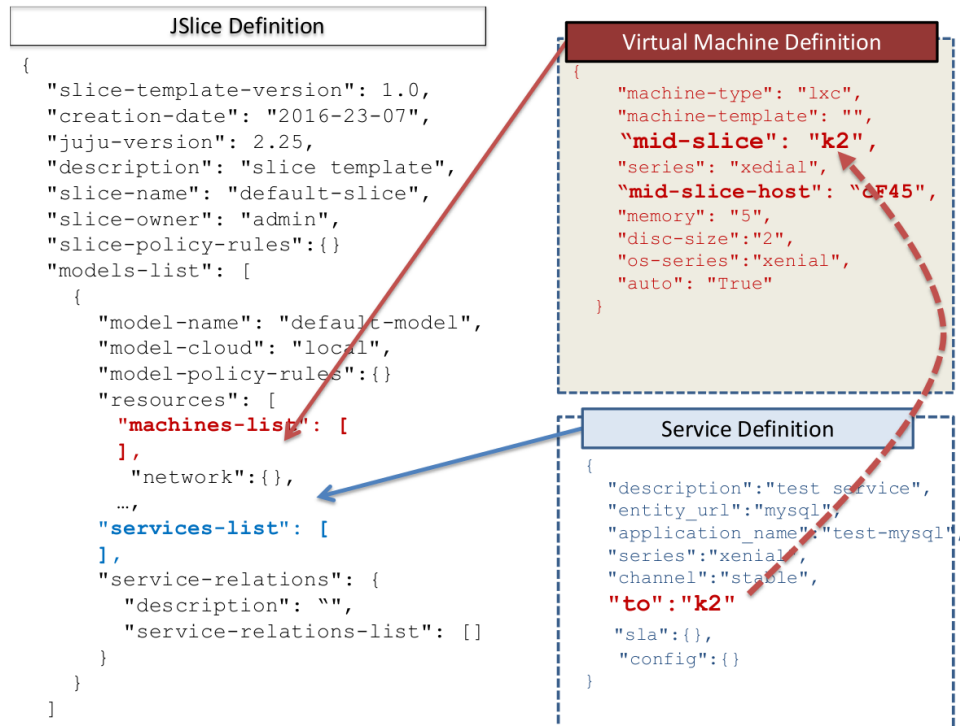


Figure 16. JOX JSlice definition in JSON representation

Figure 16 depicts an empty network slice (without any resources or services), which can be created as a POST message to JOX. In the future, template descriptions will be aligned with the work delivered in OASIS TOSCA [29] and the modelling work in the Internet Engineering Task Force (IETF) [30]. For every model, we utilize different namespaces for resources, services and the bindings to services. This way a resource (e.g. a VM) is described using a JSlice-JModel specific name. For example, the container identifier for the specific model is “k2” while the container is hosted in machine “cF45” that can be a physical machine or a container or a KVM virtual machine. Besides the information related to “where to deploy” the service, the configuration of the service is passed together with its definition; otherwise default configuration is loaded. This flexibility is enabled by the Juju framework, which triggers the corresponding Juju Charm hooks (e.g. config-changed and relation-changed). In this phase, a negotiation routine with the VIM can also be executed. Such negotiation procedures were described in [31]. While a logical definition of services and networks are initially requested by the slice, the VIM supports the requested capabilities, SLA and QoS levels based on its current state and the book-keeping information maintained by the top-level orchestrator.

JOX also supports the modification of the runtime state of a network slice in two modes: auto and manual. In the former, the network slice controller detects a performance degradation that leads to a SLA violation. In such case, specific actions need to be performed through interactions with the plugin framework, for example, increase in memory and CPU power to support the current workload, or increase the radio resources of a particular slice to increase its data rate. In the manual mode, the network slice owner is able to adjust the parameters and configurations of the network slice and the corresponding sub-elements through the northbound APIs. In both cases, a set of monitoring services is exploited at the

level of network slice (e.g., slice is healthy), the VNF, and the cloud infrastructure. These are used to either trigger the necessary action sets or facilitate the decision maker procedures.

The ability to monitor and adjust the network slice behavior and characteristics in runtime is an extremely powerful feature of JOX since it allows network slice logic to be easily introduced. In the current version of JOX, this is achieved through direct interactions with KVM, LXD hypervisors and the plugin framework for the RAN. Automatic events creation will be supported in future release.

Building LTE VNF Chains for Network Slicing with JOX

JOX exploits a rich set of functions to enable network slicing through Juju VNFs and Charms. Each Charm encapsulates every underlying LTE network module as a VNF, leveraging the OAI platform [32]¹⁵. Figure 17 shows an example of using JOX to deploy two network slices with different end-to-end logical networks. In slice A, the OAI eNB and EPC are deployed in a single VM. In slice B, a cloud-RAN chain using disaggregated RAN is deployed over different VMs. For every slice, the OAI solution can be chained using Juju relationship hooks and appropriate VNFs. For example, different functional splits can be exploited between the RRU and the BBU VNFs¹⁶. In this case, JOX orchestrates the deployment of standard LTE chain (eNB, MME, SPGW, MySQL, and HSS) for a new JSlice¹⁷. With the support of Juju VNFs, JOX has the ability to fully automate the deployment of a LTE network service chain in different execution environments ranging from a physical machine to a container or a VM.

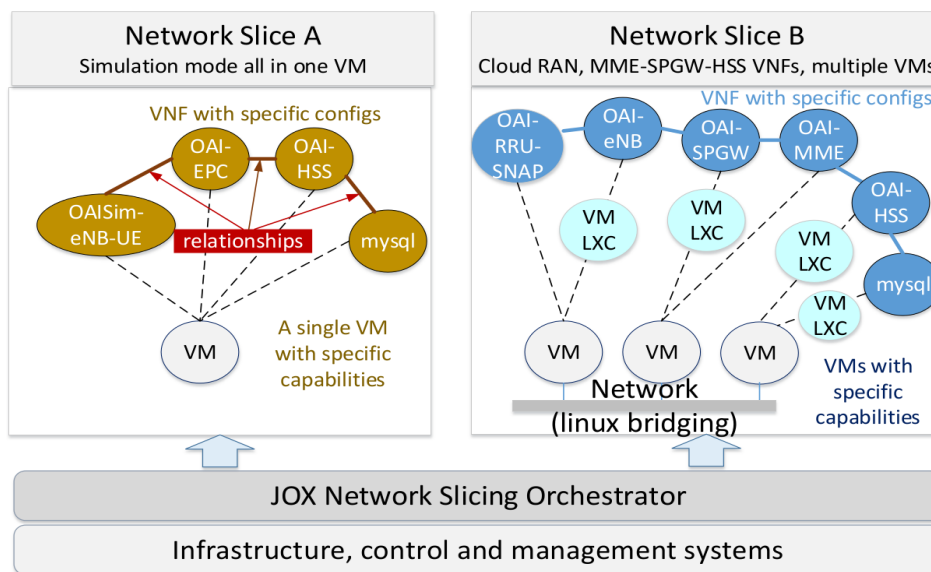


Figure 17. Network slicing in JOX

Throughout this use case, OAI service chains were deployed as a real-time LTE platform. Two virtualization environments (LXC and KVM) were considered as the targets to deploy the LTE services. For the sake of simplicity, the testbed is deployed in a physical machine. The physical server infrastructure was based on commodity Linux-based machines, equipped with 6-cores i7-3930K CPU at 3.2GHz and 16GB of RAM. Based on JOX, during the deployment for all the VNFs, all the kernel dependencies were automatically satisfied

¹⁵ OAI charms can be found at jujucharms.com/q/oai

¹⁶ Example can be found at jujucharms.com/u/navid-nikaein/oai-5g-cran/

¹⁷ Example of this service chain can be found at jujucharms.com/u/navid-nikaein/oai-nfv-4g/

(because of the scripting inside the Charms installation hooks). In Juju, a common lifecycle for a service has the following order (1) initialization: where the target environment is instantiated, such as LXC, KVM, or physical machine; (2) installation: where the service is installed on the environment; (3) configuration: where the service is reconfigured; (4) start and stop: where the service is started or stopped depending on whether the service relationships are met, and (5) relationship: where the service chain is built and dependencies are met. In our setup during installation, the Juju Charms were available from Juju remote repositories. Moreover, when chains are deployed, service dependencies may exist. For instance, the relationship between MySQL and HSS cannot be built until the HSS is installed and configured. This is the same between MME and SPGW/HSS, and between eNB and MME. This imposes time delays that cannot be avoided, due to the way the relation hooks operate in Juju. On the other hand, JOX orchestrates the service deployment and automatically handles dependencies and conflicts through Juju, without requiring any other action to be taken.

As a preliminary result, Figure 18 shows the deployment time of the LTE service chain. We observe that the installation delay dominates in most of the service lifecycle, because the services chains are deployed and built from source. The installation time can be reduced drastically when deploying the service from a local package or an image.

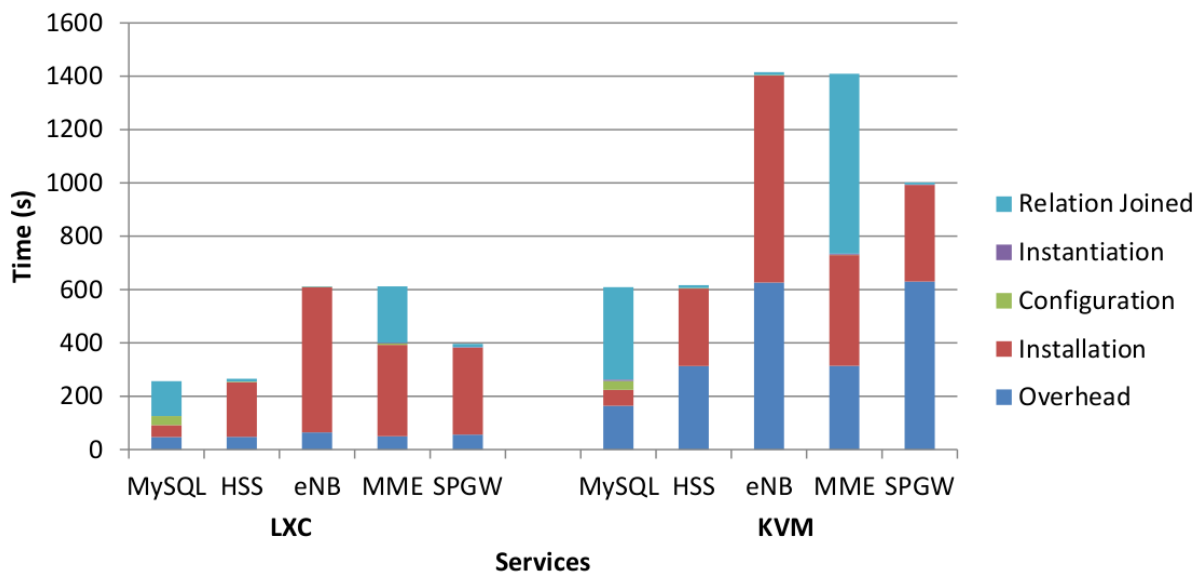


Figure 18. Deployment time of VNF chains

3.2 Automated Deployment of 5G Virtualised Infrastructure through OpenStack and Heat

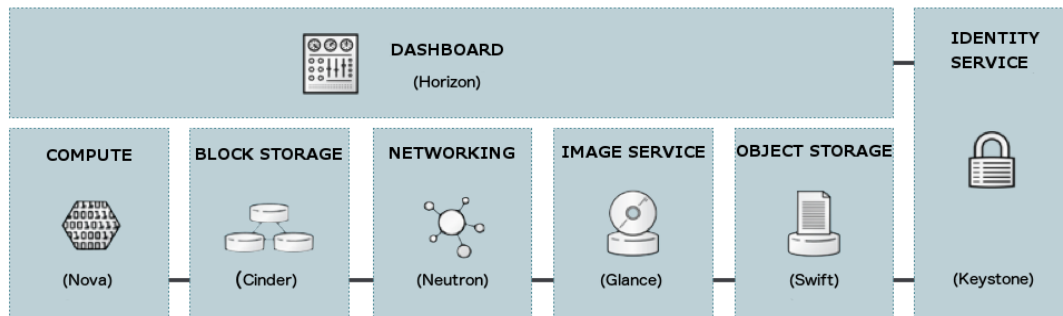


Figure 19. OpenStack service overview [14]

Figure 19 shows the logical architecture of OpenStack which mainly consists of [14]:

- The **OpenStack Compute (Nova)**: Nova is the heart of OpenStack which provides power massively scalable, on demand, self-service access to compute resources by provisioning and managing large networks of virtual machines. The Compute service facilitates this management through an abstraction layer that interfaces with supported hypervisors including KVM, LXC, Hyper-V, Docker, bare metal, etc.;
- The **OpenStack Block Storage service (Cinder)** provides block storage resources for compute instances;
- The **OpenStack Networking service (Neutron, previously called Quantum)**: Neutron provides “networking as a service” between interface devices managed by other OpenStack services including DNS, Dynamic Host Configuration Protocol (DHCP), load balancing, firewall, etc.;
- The **OpenStack Image service (Glance)** provides discovering, registering and retrieving service for disk and server images;
- The **OpenStack Identity service (Keystone)**: Keystone is a shared service that provides API client authentication, service discovery, and distributed multi-tenant authorization throughout the entire cloud infrastructure;
- The **OpenStack Dashboard (Horizon)** provides a web-based interface to OpenStack services;
- **RabbitMQ** is a message queue service, which coordinates operations and status information among OpenStack services.

3.2.1 Comparison of OpenStack Deployment Tools

Regarding OpenStack deployment, in order to avoid potential errors and to reduce the effort needed for re-deployment and management, an automated deployment tool should be used. Based on a preliminary investigation, there are several possible solutions for automated OpenStack installation relying on different tools such as Mirantis Fuel [33], Ubuntu Autopilot [34], OpenStack-Ansible [35] as well as the combination of Canonical’s MAAS [36] and Juju. A brief comparison between these tools is shown in Table 1. The first approach, i.e. a combination of Juju and MAAS, supports heterogeneous hypervisors along with containers. It is proved to be very flexible regarding the architecture layout, while remaining simple to use without extensive knowledge about OpenStack internals. The only problem with Juju is that many tasks must be done manually such as service placement,

monitoring and cluster maintenance. These issues are resolved in Autopilot, but this is a commercial product with an expansion beyond the number of ten servers being possibly quite costly. On the other hand, Mirantis Fuel is very stable, easy to use. However, it lacks support for a lightweight visualization using containers and heterogeneous hypervisors. Lastly, OpenStack-Ansible seems to be a very flexible solution, but with a much steeper learning curve than the other tools. It is a perfect tool for a big deployment where a team of system administrators can work on fine tuning of every OpenStack component. For small or proof of concept (PoC) deployments, OpenStack-Ansible requires a lot more configuration and tuning compared to other tools. As a result, in the context of SliceNet, we opted Juju and MAAS as the OpenStack deployment tool.

For more information regarding the comparison of OpenStack deployment tools, please refer to [37].

Table 1. A summary comparison of OpenStack deployment tools

	Juju and MAAS	Autopilot	Mirantis Fuel	OpenStack Ansible
Bare metal provisioning	Yes	Yes	Yes	No
Ease of service scaling	Easy	Easy	Easy	Moderate
Ease of cluster size scaling	Easy	Easy	Easy	Moderate
Ease of deployment customization	Easy	Hard	Hard	Moderate
OS Maintenance tools	No	Yes	Yes	No
OS Monitoring tools	No	Yes	Yes	No
GUI	Yes	Yes	Yes	No
LXD hypervisor support	Yes	Yes	No	No
VMware hypervisor support	Yes	Yes	Yes	Yes
KVM hypervisor support	Yes	Yes	Yes	Yes
Heterogeneous hypervisors	Yes	Yes	No	Yes
Supported operating systems	Ubuntu/ CentOS	Ubuntu	Ubuntu/CentOS	Ubuntu/ CentOS/ openSUSE
Network auto configuration	No	Yes	Yes	No
Cost per server per year	Free	Not free*	Free	Free
Commercial support available	No	Yes	Yes	No

*) First 10 servers are free

3.2.2 Deployment of OAI-based 5G Services

3.2.2.1 Juju/OpenStack

This section explains C-RAN deployment as an example of OAI-based 5G services deployment on top of OpenStack which is in turn deployed by using the combination of Juju and MAAS. The idea is that MAAS will be used to manage the hardware resources and provision the servers that, later, will be used to deploy OpenStack services automatically by Juju. Finally, the OAI services are implemented on top of OpenStack, which will be served as a cloud provider.

The high-level architecture of the C-RAN deployment, which consists of EPC part (including HSS, MME and SGW/PGW) and RAN part (BBU and RRU), is described in Figure 20. All the network entities (MME, HSS, SGW/PGW, BBU, and RRU) will be deployed in a virtualized environment on top of OpenStack. The radio card will be connected to the RRU via the interface USB-3.0 which then allows a real UE to be attached to the deployed mobile network via this wireless interface. This testbed uses OAI-CN to deploy the EPC functionality while OAI-RAN for the BBU and RRU functionality.

From a practical point of view, different network entities have different requirements in terms of latency, power and kernel support. RRU is a network element that interfaces directly with RF equipment (for example, commodity lab RF SDR platforms such as USRP B200/B210) via USB3.0 for over-the-air (OTA) experiments, its performance should be close to bare metal speeds. In addition, USB-Passthrough is needed to pass USB devices to the OpenStack instance that will be used to deploy the functionality of a RRU. On the other hand, BBU typically requires much more power in comparison to RRU. Thus, the OpenStack instances for the EPC entities and BBU can be deployed using such a high-power hardware (HW)/CPU platform while a Personal Computer (PC) or a low-power platform can be used to host RRUs. Additionally, BBU needs a low latency kernel while SGW/PGW needs a special kernel module to support GTP as required to deploy OAI software stack [4]. As a result, different types of hypervisor may be needed for different types of VM or container. For instance, both KVM and LXC can be used to deploy EPC and BBU while LXC should be used for the deployment of RRU as described in Figure 20. Therefore, two possibilities are considered as follows:

- Using LXC to deploy all network entities: In this case, the host machine needs to be installed with the kernel supporting GTP for SGW/PGW;
- Defining two different zones: one zone for KVM and another for LXC (e.g., using the notion of availability zone in MAAS). EPC entities as well as BBU could be deployed on either KVM or LXC zone while RRU on LXC zone. In this case, MAAS has to deal with tag to deploy the instances in the corresponding zone.

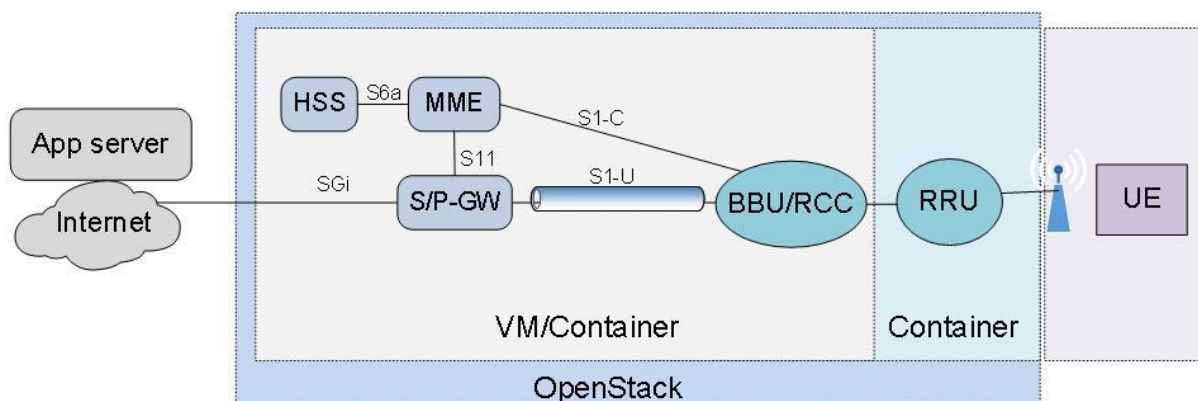


Figure 20. C-RAN deployment on top of OpenStack

For instance, we adopted the first alternative. This means that LXC is used to deploy all network entities including EPC (HSS, MME, SGW/PGW), BBU as well as RRU. Accordingly, the testbed consists of two workstations (namely Xenial1 and Xenial2), two PCs (namely Neptune and Venus), one Laptop (namely Sud) and one switch as shown in Figure 21. Mapping to the C-RAN high-level architecture, the two workstations will host the OpenStack instances to deploy EPC and BBU functionality (as well as OpenStack services) while the two PCs will be used to deploy RRUs. The laptop will be responsible for MAAS/Juju controller deployment. All devices are interconnected using a central Ethernet switch using 1GbE interfaces¹⁸. The laptop is also connected to the external network and acts as a gateway for the internal one. Here are the specifications for the hardware used to set up the testbed:

- **Two Workstations** (Xenial1 and Xenial2):
 - 10 CPU cores, 64GB of RAM, 296GB of Hard disk drive (HDD)
 - Interfaces: two Gigabit Ethernet (GbE), two small form-factor pluggable (SFP+), and one iDRAC;
- **Two PCs** (Neptune and Venus):
 - CPU cores, 32GB of RAM, 500GB of HDD
 - Interface: 1GbE
- **One Laptop** (Sud):
 - CPU cores, 8GB of RAM, 500GB of HDD
 - Interface: two GbE
- **One Cisco 2960X Ethernet switch:**
 - Interfaces: 24 x 1GbE, two SFP+

¹⁸ *Xenial1 and Xenial2 servers will be interconnected using 10GbE interfaces in the future.*

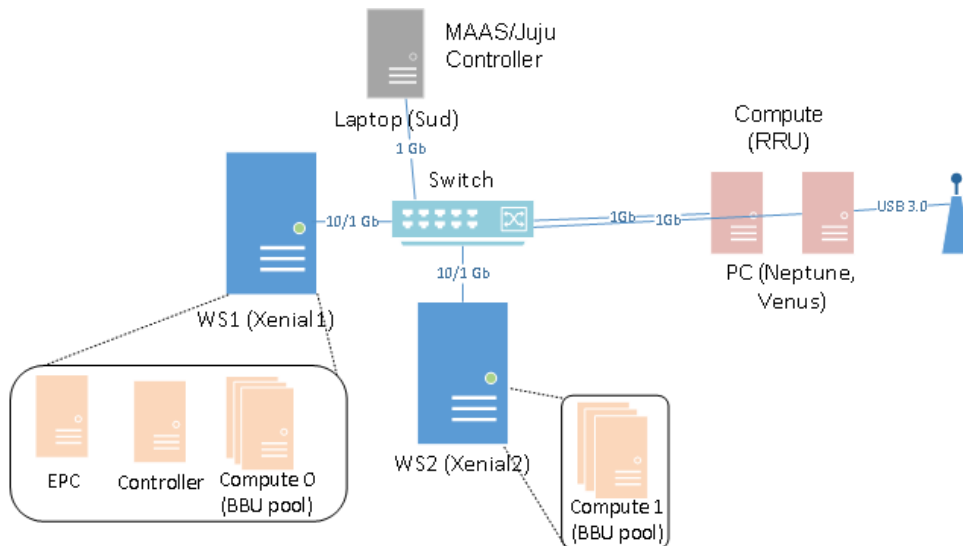


Figure 21. C-RAN testbed

Figure 22 shows the real image of the testbed deployed at Eurecom.

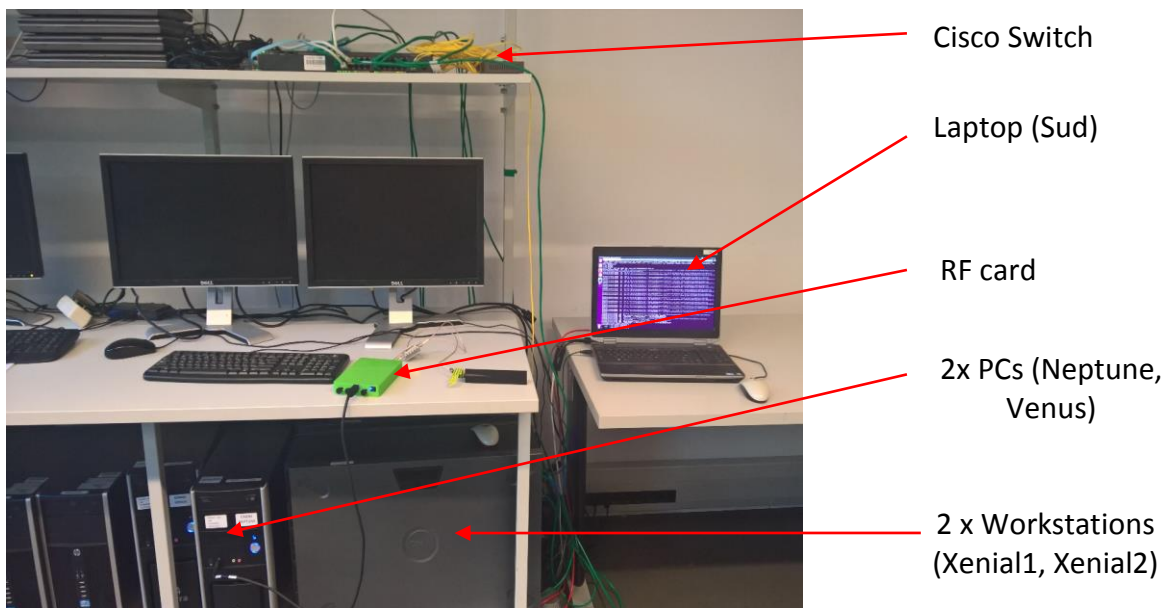


Figure 22. Image of the C-RAN testbed

Network Planning

In the context of SliceNet, network isolation is a useful feature to support network slicing. OpenStack introduces the notion of tenant networks for connectivity within projects by relying on different types of network isolation and overlay technologies such as Virtual Local Area Network (VLAN), Virtual Extensible LAN (VxLAN) and Generic Routing Encapsulation (GRE). For our testbed, VLAN is used. Again, for the sake of simplicity, only one internal network is defined for the moment. As a result, the following IP networks can be distinguished (see also Table 2):

- **External Network** - Eurecom managed network, used for the Internet access.

- **Internal Network** - Network managed by MAAS, used for management, server provisioning and OpenStack traffic (including Neutron). DNS and DHCP provided by MAAS.
- **Public Network** - Network used for publicly routable floating IPs. For the moment, this network is not necessary.

Table 2. Parameters of the C-RAN testbed networks

Name	CIDR	Gateway	DNS	DHCP
External network	192.168.12.0/24	192.168.12.100	192.168.12.100	
Internal network	10.123.0.0/24	10.123.0.1	10.123.0.1	10.123.0.2 - 10.123.0.20
Public network	-	-	-	-

It is noted that the addresses reserved for DHCP are used by MAAS to bootstrap the physical servers.

C-RAN Deployment

To deploy C-RAN testbed, several steps need to be executed. First, the physical machines need to be interconnected as described in Figure 21. The laptop, acting as a gateway for Internet connection, needs to be configured to allow the OpenStack instances connect to Internet via this computer and setup interfaces for MAAS internal network. After MAAS installation, a virtual machine, which will be served as a Juju controller, is created and commissioned by MAAS. The next step is to deploy Juju controller to the previously created virtual machine. After powering the physical servers up, they will be detected, and then will be commissioned by MAAS. At the end of this step, they are ready for the deployment of the OpenStack services. The machines commissioned by MAAS can be seen from MAAS' GUI as shown in Figure 23.

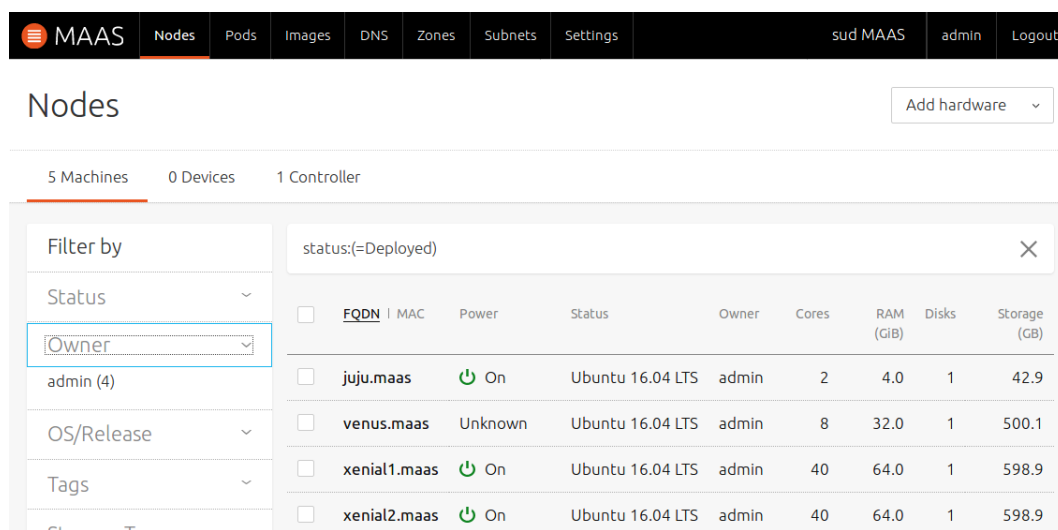


Figure 23. Machines commissioned in MAAS

The deployment of OpenStack services is then orchestrated by Juju using machines provisioned by MAAS. In this case, Juju first creates a new model for OpenStack-related

services and then deploys them using a corresponding bundle file (bundle.yaml). The bundle file defines all the required OpenStack services, their configuration, placement and interaction between them. For the moment, several OpenStack services are needed including the OpenStack Dashboard (Horizon), the OpenStack Identity service (Keystone), the OpenStack Image service (Glance), the OpenStack Compute (Nova), the OpenStack Networking service (Neutron), RabbitMQ, MySQL, and OVS. Specifically, Table 3 shows the placement of OpenStack-related services. From a practical standpoint, the following steps are executed to deploy OpenStack services.

Step 1: Switch off all of the physical machines where we wish to deploy OpenStack services (including Xenial 1, Xenial2, Venus and Neptune).

Step 2: Create a new Juju model.

```
$juju add-model os
```

Step 3: Verify that Juju controller is working.

```
$juju status
```

Step 4: To deploy OpenStack, launch the following command:

```
$juju deploy bundle.yaml
```

It takes approximately 30 minutes to deploy OpenStack services. We can observe the progress of the deployment using the following command:

```
$watch -c juju status --color "${@:1}"
```

Step 5: When Juju reports that all services are deployed and ready, execute the following command to find out the IP address of Horizon interface:

```
$juju status | grep openstack-dashboard
```

Step 6: Go to http://<openstack_dashboard_IP>/horizon and login to Horizon to manage OpenStack services.

Table 3. Placement of OpenStack and other services at the machines of the testbed

Xenial 1	Xenial 2	Neptune	Venus
Horizon (LXD)	Nova-compute-KVM/LXD	Nova-Compute-LXD	Nova-Compute-LXD
MySQL (LXD)	OVS	OVS	OVS
RabbitMQ			
Keystone (LXD)			
Glance (LXD)			
Nova-cloud-controller			
Neutron-API			
Neutron-Gateway			

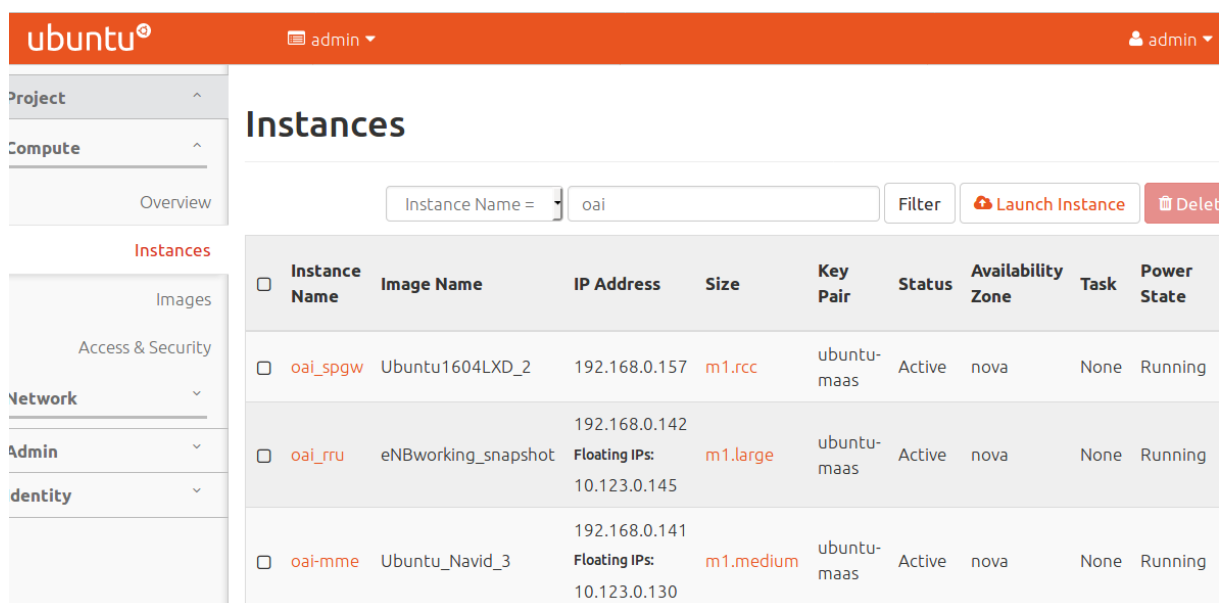
After the setup of OpenStack services, several configuration steps are required in order to obtain a minimal working cloud environment e.g., configure internal/external network and subnet, configure availability zones, import images into OpenStack Glance and setup Secure Shell (SSH) connections for the OpenStack instances. For more information regarding OpenStack configuration, please refer to [51].

Passthrough Radio Card to a Virtual Instance

As mentioned earlier, the RF card will be connected to RRU via USB 3.0, which allows user to connect to the deployed mobile network via a wireless interface. However, in a virtualized environment managed by OpenStack, it is not straightforward. In other words, USB devices are not automatically attached to an OpenStack instance. USB-passthrough is therefore needed to pass USB devices into an instance. Based on a preliminary investment, there is three possible solutions for USB-passthrough including (i) implementation of USB-passthrough feature into OpenStack; (ii) out-of-band USB-passthrough using Juju; and (iii) USB controller passthrough using PCI-passthrough [51].

As an example, we use the third solution for USB-passthrough since this solution only depends on the possibility to support passthrough under KVM/LXD (independence from Juju/OpenStack). In this solution, the script to attach USB cards to an instance can be executed as a standalone script. It can also be executed automatically as a Juju action. Basically, this script will detect the instance where the RF will be connected to and attach the card to this instance accordingly.

By using OpenStack, we created several instances for setting up the testbed as shown in Figure 24. For the first deployment, we could have to install the functionalities of all the network entities by following the instruction from OAI-RAN [2] and OAI-CN [4]. However, we then can use these instances to create the corresponding images which allow to deploy a new instance for RRC, RRU, EPC components or even a new C-RAN testbed easily and quickly in OpenStack environment.



Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State
<input type="checkbox"/> oai_spgw	Ubuntu1604LXD_2	192.168.0.157	m1.rcc	ubuntu-maas	Active	nova	None	Running
<input type="checkbox"/> oai_rru	eNBworking_snapshot	192.168.0.142 Floating IPs: 10.123.0.145	m1.large	ubuntu-maas	Active	nova	None	Running
<input type="checkbox"/> oai-mme	Ubuntu_Navid_3	192.168.0.141 Floating IPs: 10.123.0.130	m1.medium	ubuntu-maas	Active	nova	None	Running

Figure 24. C-RAN instances in OpenStack environment

Finally, we have a fully functional mobile network. We then use a commercial off-the-shelf (COTS) UE to verify the functionality of the deployed mobile network. As expected, the UE

can successfully attach to the deployed mobile network and establish a PDN connection to the Internet (as can be seen in Figure 25). Figure 26 shows the MME log with the information related to the connected eNB and the connected UE.

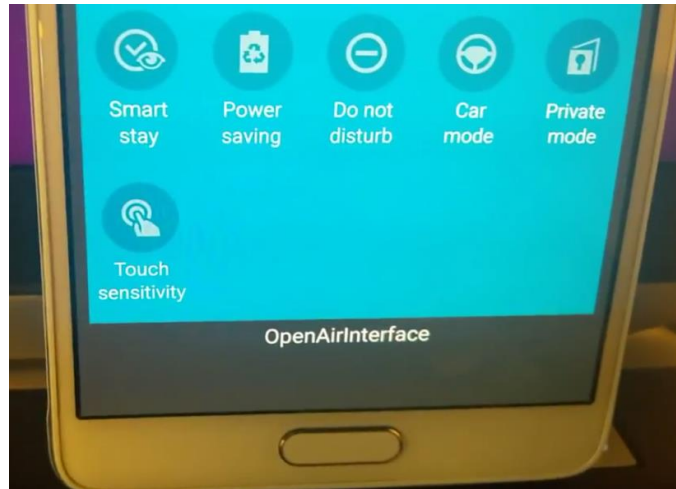


Figure 25. UE connected to the deployed network – OpenAirInterface

```

===== STATISTICS =====

```

	Current Status	Added since last display	Removed since last
Connected eNBs	1	0	0
Attached UEs	1	0	0
Connected UEs	1	0	0
Default Bearers	1	0	0
S1-U Bearers	1	0	0

Figure 26. UE connected to the deployed network

3.2.2.2 Heat

Heat [52] is OpenStack's main orchestration component, which is capable of launching deployments of complex cloud applications described in text files (called templates). The following is a simple Heat template (Heat Orchestration Template, or HOT) describing a typical OAI-based LTE deployment with four main components – HSS, MME, S/P GW, and eNB.

```

heat_template_version: 2015-05-23
description: LTEaaS
parameters:
  key_name:
    type: string
    description: Name of a KeyPair to enable SSH access to the instance
    default : cloudkey

resources:

```

HSS:

```

type: OS::Nova::Server
properties:
  image: hss-1
  flavor: HSS.med
  key_name: cloudkey
  networks: [{ network: PUBLIC_NETWORK }]
  user_data: |
    #!/bin/bash
    MY_IP=`ip addr show dev eth0 | awk -F'[ /]*' '/inet /{print $3}`
    sed -i 's/MY_IP/'$MY_IP'/g' /etc/hosts
    hostname hss-1
    run_hss

```

SPGW:

```

type: OS::Nova::Server
properties:
  image: epc-3
  flavor: EPC.med
  key_name: cloudkey
  networks: [ {network: PUBLIC_NETWORK } ]
  user_data:
    str_replace:
      template: |
        #!/bin/bash
        MY_IP=`ip addr show dev eth0 | awk -F'[ /]*' '/inet /{print $3}`
        sed -i 's#MY_IP_S1#'$MY_IP'/24#g' spgw.conf
        sed -i 's#MY_IP#' $MY_IP'/24#g' spgw.conf
        run_spgw

```

MME:

```

type: OS::Nova::Server
properties:
  image: epc-3
  flavor: EPC.med
  key_name: cloudkey
  networks: [ {network: PUBLIC_NETWORK } ]
  user_data:
    str_replace:
      template: |
        #!/bin/bash
        MY_IP=`ip addr show dev eth0 | awk -F'[ /]*' '/inet /{print $3}`
        sed -i 's/MY_IP/'$MY_IP'/g' /etc/hosts
        sed -i 's/HSS_IP/'$HSS_IP'/g' /etc/hosts
        sed -i 's#MY_IP_S11#' $SPGW_IP'/24#g' mme.conf

```

```

sed -i 's#MY_IP_S1#$MY_IP'/24#g' mme.conf
sed -i 's#MY_IP#$MY_IP'/24#g' mme.conf
hostname mme-1
run_mme
params:
  $HSS_IP: { get_attr: [HSS, first_address] }
  $SPGW_IP: { get_attr: [SPGW, first_address] }

ENB:
type: OS::Nova::Server
properties:
  flavor: eNB.med
  image: enb-usrp
  key_name: cloudkey
  networks: [{ network: PUBLIC_NETWORK }]
  user_data:
    str_replace:
      template: |
        #!/bin/bash -v
        MY_IP=`ip addr show dev eth0 | awk -F'[ /]*' '/inet /{print $3}'`/24
        sed -i 's#MY_IP_ADDRESS_REPLACE#$MY_IP#g' enb.conf
        sed -i 's#MME_IP_ADDRESS_REPLACE#$MME_IP#g' enb.conf
        ./lte-softmodem -O enb.conf

params:
  $MME_IP: { get_attr: [MME, first_address] }

```

A Heat template typically contains several sections:

- **heat_template_version** is used to specify the version of the template syntax that is used.
- **description** is used to provide a description of what the template does.
- **parameters** provides input parameters which allow users to customize a template during deployment.
- **resources** is the most important section in a template. It defines compute instances together with the information related to which flavour, image, public key and network to use for these instances. In this example, three instances are defined: HSS, EPC, and eNB.

Please refer to [53] for more information regarding HOT syntax.

3.2.3 Deployment of OAI-based vEPC Services

In order to satisfy all the needs of the operators to come up with an NFV architecture that will meet all the requirements for virtualizing the mobile packet core, it is imperative that we will first present first the requirements of the mobile services and applications that will be running on the top of the mobile packet core.



Figure 27. Evolved Packet Core Network components

The user equipment connects to the network over the uU interface to the eNB. The connection terminates on the PDN Gateway. The MME, HSS and SGW are other important components in establishing the connection and providing subscriber service and make up the evolved packet core.

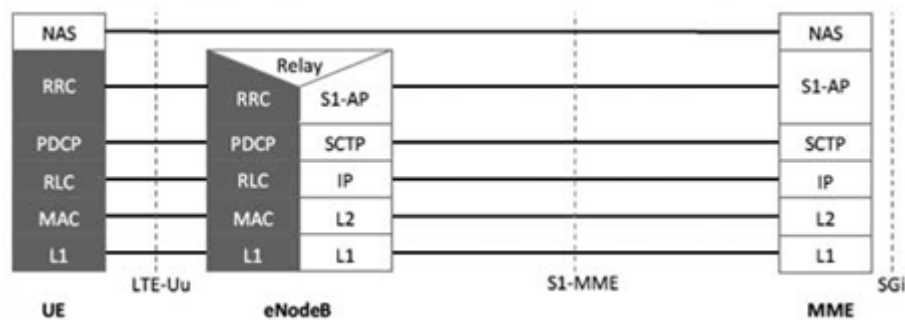


Figure 28. OSI layers and protocols used on the control plane between the UE and MME of an LTE network

The process of attaching an UE to an LTE network has to take place in five phases involving six of the functional nodes from the network. The five phases are the following:

- **UE Identity acquisition**, in this phase the UE identifies itself to the network by communicating its International Mobile Subscriber Identity (IMSI) identifier, another identifier used, based on the network implementation, is the Globally Unique Temporary Identifier (GUTI). IMSI is a unique ID that globally identifies a mobile subscriber. It is composed of two parts, namely PLMN ID and Mobile Subscription Identification Number (MSIN). A PLMN ID is an ID that globally identifies a mobile operator (is a combination of MCC and MNC). MSIN is a unique ID that identifies a mobile subscriber within a mobile operator. In contrary, GUTI comes as a security improve of the IMSI identification in the radio link connection of the mobile subscriber. Unlike IMSI, a GUTI is not permanent, but changed into a new value whenever generated. When the process of initial attach for an UE to an LTE network takes place, it sends its IMSI to the network for authentication to be identified. Once connection is established, the MME delivers a GUTI value through Attachment Accept message to the UE. This value is then can be used as its ID instead of IMSI when it reattaches to the network.
- **Authentication**, in this phase the mutual authentication is realized by the EPS-AKA method. This method supposes that all the keys that are needed for various security mechanism are derived from an intermediate key which is viewed as the local master key for the subscriber in contrast to the permanent master key. Inside the network side, the local master key is stored in the MME and the permanent master key is stored in the AuC. This approach provides the following advantages:
 - It enables cryptographic key separation.

- The system is improved by providing key freshness and it is possible to renew the keys used in security mechanism.
- **Non-Access Stratum Security Setup**
- **Localization Update**, in this phase the MME informs the HSS that it manages the UE and recovers the services to which the UE has subscription.
- **EPS Session Establishment**, in this phase the bearer is realized based on the QCI.

All the above detailed phases can be identified in the Figure 29.

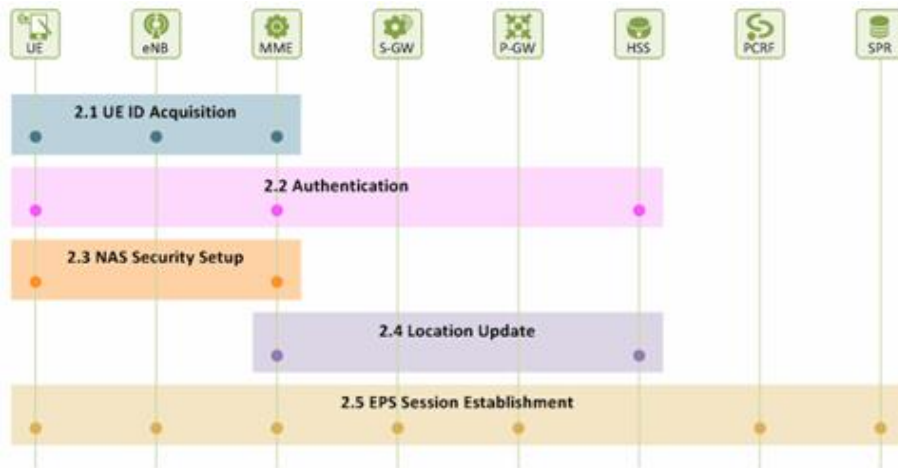


Figure 29. The UE attachment phases

In an NFV architecture, two models need to be taken into account:

- SGW, PGW, MME and HSS collocated in the same data center;
- MME + SGSN (WCDMA) located in the regional data centers and PGW or SGW + HSS + IP services located in the core or the national data center.

In the second case where one or more components may be distributed, the underlying infrastructure design including OpenStack will be impacted. This is referred to as Distributed NFV. If only some 3GPP functions are placed in the regions/ edge of the network, we may choose to deploy on compute nodes in that region as long as the latency is below the latency threshold of OpenStack and application control plane. Those compute nodes may be integrated with storage and deployed as Hyper Converged Infrastructure. In some cases, a smaller deployment of OpenStack may be used in these regions. This raises a lot of design questions regarding the shared identity infrastructure (referring to the OpenStack identity Service (Keystone)) and image service (referring to the OpenStack Image Service (Glance)).

Virtual Private Clouds may be deployed in many different ways:

- Dedicated Private Cloud for VPC;
- Collocated with other VNFs in operators private cloud;
- Hosted in public clouds (IaaS, Platform as a Service (PaaS) or Virtual Network Function as a Service (VNFaaS));
- Hosted in vendors private cloud and offered as a service.

VNF vendors tend to bundle combinations of the above services based on functional requirements of operators. These combinations could lead to varying deployment models all the way from the number of VMs to High Availability (HA), scale, traffic mix, and throughput requirements.

Depending on the type of services being deployed, the VNFs may be deployed in one of the following ways:

- Lightweight deployment using an All-in-One OpenStack deployment that typically runs on a single server:
 - If multiple functions are required, multiple servers are deployed, each running the VNF. This deployment scenario can be observed in Figure 30.

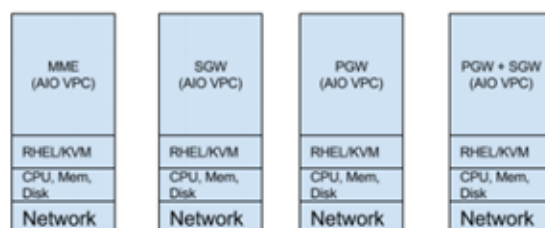


Figure 30. Lightweight All-in-One OpenStack Virtual Private Cloud deployment for vEPC

- Runs on blade or rack-mount servers;
- Suitable for cloud deployment on private, public or hybrid-clouds;
- Running on pre-bundled, fixed configuration blade-servers over KVM hypervisor;
- Full HA OpenStack Platform deployment comprising of multiple servers.

The Internet facing interface of the EPC network is referred as SGi in 3GPP specifications. As a packet leaves the gateway (PGW), it has no longer a context about the subscriber. It is a pure IP packet. IP services such as DPI, Parental Control, Video Optimization, Web Optimization, URL filtering/enrichment, Firewall and NAT are applied to the IP packets as they leave the mobile network towards the Internet. Since these services reside on the SGi interface, they are commonly referred to as **GI-LAN** services. Such services are typically deployed in some combinations and form a logical chain.

In most of the mobile networks, there is some variation of GI-LAN service, but restricted to APN granularity, rather than subscriber level. This is because traditional GI-LAN service requires physical connections to form the chain. In today's solutions, SDN is used to create logical chains between GI-LAN elements. In this case, the actual application runs in a virtual environment typically on top of an OpenStack environment, rather than purpose-specific appliances. This approach offers a huge advantage because applications can scale based on demand and conversely shrink. A typical example of this would be when people return from work and turn on their TV or Over-The-Top (OTT) video, this creates a surge in traffic and creates a demand on GI-LAN elements.

If we analyze the situation from the infrastructure point of view, GI-LAN will closely resemble the vEPC VNF, it will have the same requirements of orchestrations, VNF lifecycle management, performance and security. Alongside those common requirements as mentioned earlier, GI-LAN will require metering capability to determine the resource usage, to be able to grow and shrink upon demand. This requirement is aiming at the future capability of growing and shrinking automatically. Another requirement for the GI-LAN is regarding the compatibility of the OpenStack with the Service Function Chaining that will have to support. Service Function Chaining encapsulation can leverage Network Service Header or Multi-Protocol Label Switching (MPLS).

vEPC components such as SGW and PGW may be placed either centrally at the core data center or regionally to serve the local cell sites or exit points. In LTE, the SGW selection is typically based on the network topology as well as the location of the UE (Tracking Area Code, TAC). The subscriber connection terminates on the PGW and the decision of which PGW to assign to the subscriber depends on the subscription context including the information related to the Access Point Name (APN), among the others. vEPC enables mobile network operators and enablers to use a virtual infrastructure to host voice and data services, rather than using an infrastructure built with physical functions only.

Network slicing or network multi-tenancy, being a capability also enabled by vEPC, pose a prerequisite for providing multiple services simultaneously. By using the vEPC approach, the mobile network operators (MNOs) can reduce OPEX and CAPEX, while speeding up delivery and enabling on-demand scalability.

In a vEPC, most of the above-mentioned functions may be virtualized, including PGW, SGW, MME, PCRF, FW, Router, DPI, Switches and LB. This case is typically for the operators who are building a new mobile packet core or upgrading and adopt virtualization approach.

The approach to deploy vEPC is based on NFV technology, where vEPC is seen as a NFV use case. However, within a general Virtual Packet Core (VPC) case, the MNOs may have more specific use cases depending on what services they provide.

The vEPC solution is based on a NFV-SDN Architecture, all entities (MME, HSS, PGW, SGW and PCF) being implemented in this case with support of VMs (or containers), instantiated through a Controller node (OpenStack deployment). The VMs are then instantiated and managed from Cloud, based on the ETSI MANO approach with resources for the VNF given by the NFVI. This approach is straightforward in order to implement the functions of a classic EPC, so each NE can be implemented as a single or multiple VMs, as a multi-tenancy implementation.

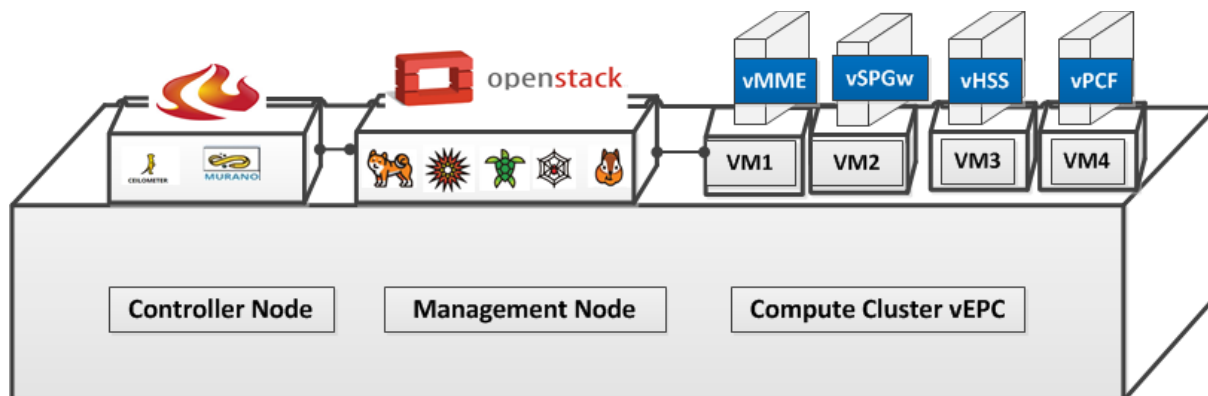


Figure 31. vEPC solution is based on a NFV-SDN Architecture

OpenAirInterface (OAI) as is presented in [2] is an open source Rel-10/Rel-14 3GPP compliant reference implementation of EPC and E-UTRAN that runs on general purpose computing platform. The software is capable of interface with commodity lab RF SDR platforms for OTA experiments with commercial devices. The potential of OAI can be exploited within the R&D SliceNet scope for open vEPC core network, integrated into a scenario combining the virtualized RAN and Core Infrastructure, which will be then applied to the SliceNet use-cases and integrated within the SliceNet architecture.

In the context of OPNFV project, OAI offers the potential to test OPNFV infrastructure within the framework of Functest project [39], offering several open source 3GPP 4G/5G VNFs, for example, EPC (HSS, MME, S/PGW), BBU, OAISIM (OAI Simulator for 3GPP RAN), RRU, and UE. The communication between the different VNFs within OAI can happen over standard IP Communication interface thus avoiding the need of special purpose servers/RF equipment for testing OAI.

The current plan for OAI is to integrate OAI EPC as a VNF within OPNFV Functest as a part of Danube, the fourth OPNFV release, as described in [40].

OAI community is working in disaggregating OAI EPC into HSS, MME, SGW, and PGW. All the different EPC components will run in their own virtual environments and be chained together with service orchestrator to provide EPC functionality.

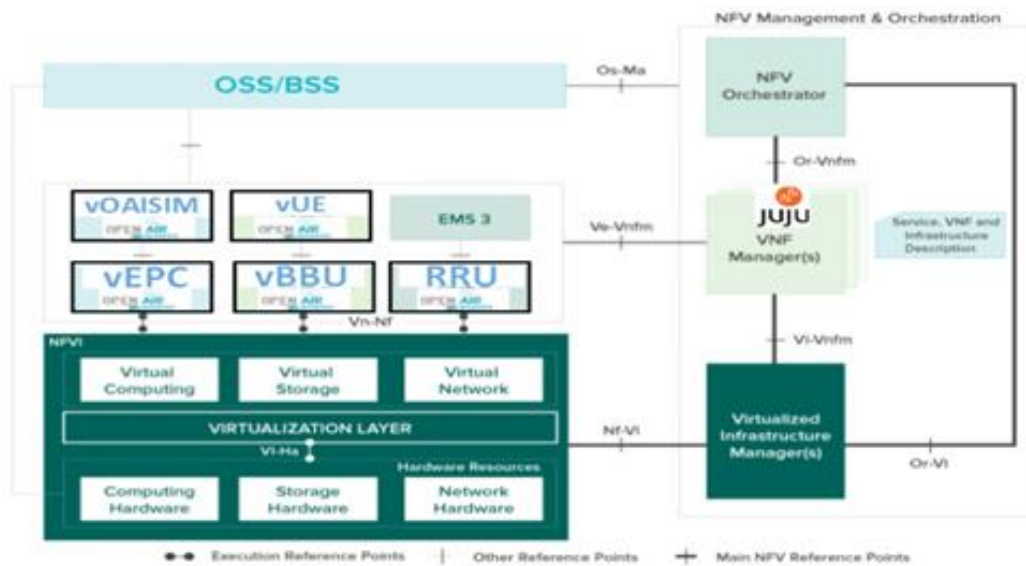


Figure 32. OAI as VNF within OPNFV [38]

OAI can provide several interesting use cases around 4G/5G cellular deployment within OPNFV. The OAI community is also working towards creating SDN interfaces which can be leveraged for more complex test cases involving SDN controllers such as ODL and ONOS M-CORD¹⁹.

OAI integration in the KVM4NFV project [41] shows that the RAN virtualization may make higher demand on computing capability and the hypervisors are not designed or targeted for the specific Telco NFVI requirements, mainly due to the performance features. The Pharos project [42] is developing an OPNFV lab infrastructure integrating OAI as VNF, managed by Juju which opens up interesting possibilities for further integration and testing within the Pharos test labs.

The OAI Community also aims to work closely with ETSI NFV/ETSI MEC ISG, for example in terms of providing PoCs demonstrating key concepts of these work groups.

In the future, OAI continues to work in close collaboration with OPNFV communities for joint demonstration and for working towards an end-to-end ETSI NFV platform based on open source tools. OAI is also working to develop its core software for future 3GPP releases towards 5G. Furthermore, OAI testing within OPNFV Pharos Labs using OPNFV Functest

¹⁹ Mobile CORD, <https://wiki.opencord.org/display/CORD/Mobile+CORD>

provides valuable feedback to the OAI community. OAI integration with OPNFV in conjunction with other open source projects has the potential to create an end-to-end reference platform built on open source software that can be potentially used by 3GPP/ETSI/NGMN for PoC and demonstration.

3.2.4 Deployment of Athonet-based vEPC Services

Athonet [54] provides a complete software-based mobile packet core solution (EPC) which also includes a HSS, Voice-over-LTE (IMS for VoLTE), and LTE Broadcast (eMBMS). The industry's most efficient mobile core solution that can be deployed in fully virtualised environments (NFV), enterprise data centres or on standard off-the-shelf servers. It can be used in highly distributed deployments in Tier 1 Mobile Operators and OTE has deployed it in its lab Athonet vEPC architecture.

Athonet's LTE mobile core complies with the default 3GPP interfaces as shown in the figure below.

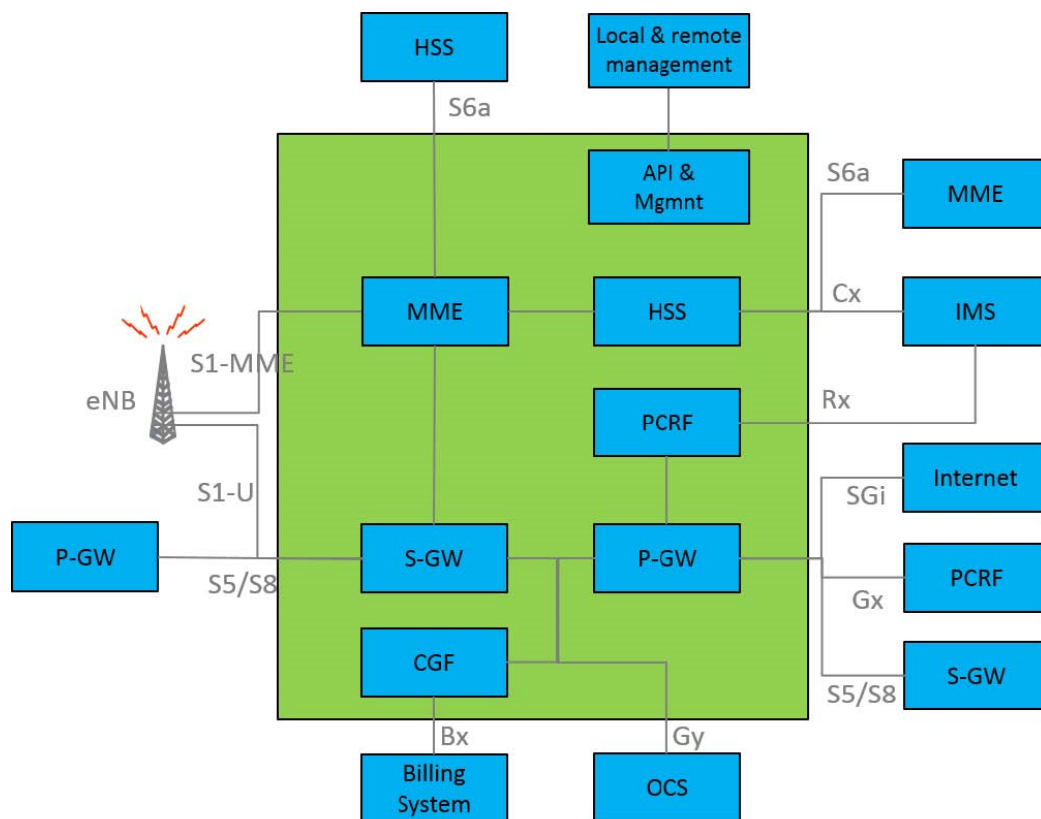


Figure 33. Athonet vEPC architecture [52]

The EPC, which contains the core network nodes MME, SGW (S-GW), PGW (P-GW), HSS, PCRF, connects externally via the following 3GPP compliant interfaces:

- S1: connects EPC to access nodes
- S5/S8: connects SGW and PGW
- S6a: connects MME and HSS for the authentication of user access and profiling
- Gx: connects PCRF and PGW and enables the PCRF to prioritize certain type of data
- Rx: connects the PCRF to external AF (application function such as IMS)
- Gx: connects PCRF and PGW and enables the PCRF to prioritize certain type of data traffic

- Gy: for online charging
- Bx: FTP(S) based interface which allows billing systems
- Cx: Diameter interface
- SGi: connects the PGW to Intranet and Internet

MME

The MME supports the following features:

- S1-MME: it is the interface to connect the MME to eNBs
- S6a: Diameter interface for authorization of the users

SGW

The SGW follows 3GPP specifications Rel-12 and supports the following:

- S1-U: GTPv1-U interface for connecting the SGW to the eNBs
- S5/S8: GTPv2 interface to connect SGW and PGW
- X2 handover
- S1 Release procedure

PGW

The PGW follows 3GPP specifications Rel-12 and supports the following features:

- SGi: connects the PGW to Intranet and Internet
- S5/S8: GTPv2v interface to connect SGW and PGW
- Gx: connects PCRF and PGW and enables PCRF to prioritize traffic
- VRF support

PCRF

The PCRF also follows Rel-12 and supports the features:

- Rx: to connect PCRF to external application function
- Create/delete bearers
- Subscription repository
- Policy based services

HSS

The HSS follows also 3GPP Rel-12 specifications and supports the following features:

- S6a: Diameter interface for transfer transcription
- USIM credentials
- EPC user profile management

3.2.4.1 QoS Settings

The QoS settings can be done according to QCI, Gradual Bit Rate (GBR), Maximum Bit Rate (MBR) and Priority values.

3.2.4.2 System Management

Athonet has implemented Element Management System (EMS) for the core management system. It can manage system configuration and 3GPP nodes, user management and QoS profile management, detailed user activity and secure access.

Integration points are available for connecting the vEPC to third parties:

- SNMP for KPI and performance monitoring
- SNMP traps for alarm monitoring
- RESTful API for user provisioning, profile assignment, activating and de-activating users

Supported protocols, interfaces and standards

- Architecture enhancements for non-3GPP access, according to 3GPP 23.402
- Intra-domain connection of RAN nodes to multiple CN nodes according to 3GPP 23.236
- Network sharing according to 3GPP 23.251
- Stream Control Transmission Protocol (SCTP) according to RFC 4960
- User Datagram Protocol according to RFC 768
- Internet Protocol according to RFC 791
- Transmission Control Protocol according to RFC 793
- Internet Protocol version 6 (IPv6) specification according to RFC 2460
- GTP-U based interfaces according to 3GPP 29.060-29.281
- QoS architecture according to 3GPP 23.107
- Diameter interfaces according to 3GPP 29.230
- S1-AP according to 3GPP 36.413
- S1 data transport according to 3GPP 36.414
- NAS-EPS according to 3GPP 24.301
- Gy interface according to 3GPP 32.299 and RFC 4006
- Bx interface according to 3GPP 32.251, 3GPP 32.297, 3GPP 32.298
- Rx interface according to 3GPP 23.203
- Gx interface according to according to 3GPP 29.212
- Cx interface according to 3GPP 29.228-9

3.3 Manual Deployment of 5G Virtualised Infrastructure through Linux Utility

In this subsection, we deploy a simplified version of C-RAN (as shown in Figure 20) by relying on different Linux utilities such as LXC, LXN, KVM and Docker. In this case, instead of deploying HSS, MME, and SGW/PGW in separate entities, we install all the EPC functionalities on one container/virtual machine.

3.3.1 LXC

Linux Containers (LXC) [43] is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host. Linux containers can offer an environment as close as possible to a standard Linux installation without the need for a separate kernel and all the hardware simulation.

The following paragraph will briefly go through the main steps to install LXC and use LXC to deploy a C-RAN testbed based on OAI-RAN and OAI-CN.

Step 1: LXC installation

To install LXC and the required packages, execute the following command:

```
$sudo apt-get install lxc lxc-template bridge-utils
```

Before creating the containers, we need to configure the network for the containers which then allow to set up different private networks (please refer to [44] for more information).

Step 2: Creating a container

To create a new container, we use “lxc-create” command. For example, the following command is to create a new container namely `bbu_lxc` based on Ubuntu 16.04 (64-bit).

```
$sudo lxc-create -n bbu_lxc -t ubuntu -- -r xenial -a amd64
```

To deploy the C-RAN testbed, three containers are needed for deploying EPC, BBU and RRU.

```
$sudo lxc-create -n oai-epc -t ubuntu -- -r xenial -a amd64
```

```
$sudo lxc-create -n oai-bbu -t ubuntu -- -r xenial -a amd64
```

```
$sudo lxc-create -n oai-rru -t ubuntu -- -r xenial -a amd64
```

Step 3: OAI installation

After creating the containers, we start these containers and then use these containers to deploy OAI software.

We first start the containers by using the following commands:

```
$sudo lxc-start -n oai-epc
```

```
$sudo lxc-start -n oai-bbu
```

```
$sudo lxc-start -n oai-rru
```

After that, we can see the list of deployed containers by using “lxc-ls” command:

```
voiture@oai:~$ sudo lxc-ls -f
NAME      STATE    AUTOSTART  GROUPS  IPV4      IPV6
oai-bbu   RUNNING  0          -       10.0.1.183 -
oai-cn    RUNNING  0          -       10.0.1.159 -
oai-rru   RUNNING  0          -       10.0.1.152 -
```

Figure 34. List of deployed LXC containers

In order to deploy the BBU/RRU functionality, we login to `oai-bbu/oai-rru` container and follow the installation guide from [2].

```
$sudo lxc-console -n oai-bbu
```

For more details, we first get the OAI source code from OAI Git repository²⁰.

```
$git clone https://gitlab.eurecom.fr/oai/openairinterface5g.git
```

```
$cd openairinterface5g
```

```
$git checkout develop
```

We then build OAI as a BBU by using the following commands:

```
$cd openairinterface5g
```

```
$source oaienv
```

```
$cd cmake_targets
```

```
$/build_oai -l
```

```
# install SW packages from internet
```

```
$/cmake_targets/build_oai -c -x -t ETHERNET -w USRP --eNB
```

²⁰ <https://gitlab.eurecom.fr/oai/openairinterface5g/>

Similarly, we install OAI-EPC on top of oai-epc container by following the installation guide from [4]. The BBU then needs to be configured to talk with the EPC and RRU. The same step should also be done for EPC and RRU. Please refer to [2] [4] for more information on OAI installation.

Step 4: USB-passthrough

Similar to the C-RAN deployment on top of OpenStack, USB-passthrough needs to be done to allow the radio card to be attached to the oai-rru container. One possible solution is to modify the LXC configuration file of the corresponding container, for example, as following

```
lxc.mount.entry = /dev/bus/usb/XXX dev/bus/usb/XXX none bind,optional,create=dir
# USB Dongle for weather station
lxc.cgroup.devices.allow = c YYY:* rwm
```

Where XXX is the bus ID where the USB device is attached and YYY is the character device ID.

Step 5: Launch the testbed

We first access to oai-cn container and launch the HSS, MME and SGW/PGW accordingly.

```
$cd openair-cn/scripts
$./run_hss
$./run_mme
$./run_spgw
```

We then access to oai-rru and oai-bbu to launch RRU and BBU functionality.

For example, the following commands are to launch BBU:

```
$cd openairinterface5g/cmake_targets/lte_build_oai/build/
$sudo ./lte-softmodem -O
/home/ubuntu/openair5g/openairinterface5g/targets/PROJECTS/GENERIC-LTE-
EPC/CONF/rrc.band7.tm1.usrpb210.conf
```

After this step, a fully functional mobile network is deployed. We can see from MME that the deployed BBU (eNB) has been connected to the core network, however, without any connected UE.

```
===== STATISTICS =====
```

	Current Status	Added since last display	Removed since last display
Connected eNBs	1	0	0
Attached UEs	0	0	0
Connected UEs	0	0	0
Default Bearers	0	0	0
S1-U Bearers	0	0	0

Figure 35. MME log without any connected UE

The deployed network then hosts a COTS UE (see Figure 36 and Figure 37). Again, we can clearly see that the UE is attached to OpenAirInterface network.

	Current Status	Added since last display	Removed since last
Connected eNBs	1	0	0
Attached UEs	1	0	0
Connected UEs	1	0	0
Default Bearers	1	0	0
S1-U Bearers	1	0	0

Figure 36. MME log with a connected UE

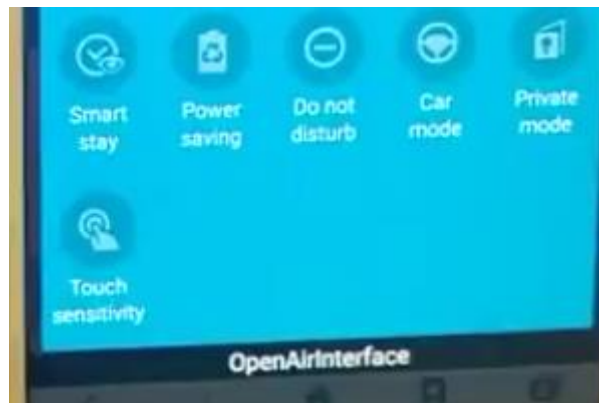


Figure 37. UE connected to the deployed network using LXC (OpenAirInterface)

3.3.2 LXD

LXD [45] is a container hypervisor providing a REST API to manage LXC containers. In fact, LXD is building on top of LXC to provide a new, better user experience.

Using LXD to deploy a C-RAN testbed, as LXC, similar steps are executed to install LXD, configure LXD environment, create LXC containers for EPC/BBU/RRU and install OAI software on these containers. From a technical standpoint, the following steps are executed.

Step 1: Install LXD

```
$sudo apt update
$sudo apt install lxd
```

Step 2: Setup LXD and configure LXD environment

We first execute “sudo lxd init” command and then follow the instruction to provide additional information related to the network configuration.

Step 3: Launch a container

The following commands create three LXC containers for oai-epc, oai-bbu and oai-rru.

```
$lxc launch ubuntu:16.04 oai-epc
$lxc launch ubuntu:16.04 oai-bbu
$lxc launch ubuntu:16.04 oai-rru
```

We can check the active containers by using “lxc list” command.

```
voiture@oai:~$ sudo lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
oai-bbu	RUNNING	10.191.88.172 (eth0)	fde5:4875:ddfb:3406:216:3eff:fe33:9fa5 (eth0)	PERSISTENT	0
oai-epc	RUNNING	10.191.88.212 (eth0)	fde5:4875:ddfb:3406:216:3eff:fea1:ced2 (eth0)	PERSISTENT	0
oai-rru	RUNNING	10.191.88.184 (eth0)	fde5:4875:ddfb:3406:216:3eff:fe76:7c03 (eth0)	PERSISTENT	0

Figure 38. List of active LXC containers (created by LXD)

Again, OAI functionality is deployed on these LXC containers similar to that in the subsection 3.3.1.

Regarding USB-passthrough, LXD version 2.5 or higher is required. To achieve that, we install LXD version 2.5 from the source code²¹ and deploy this version to the host where we will attach the RF card via USB3.0 [51]. We then execute the following command from the selected host to attach the USB device to the oai-rru container.

```
$sudo lxc config device add oai-rru 5G_card usb vendorid=2500 productid=0020
```

Where “5G_card ” is the name of the RF card. The information regarding the vendorid and productid can be found by using “lsusb” command as follows:

```
voiture@oai:~$ lsusb
Bus 002 Device 003: ID 0a5c:5800 Broadcom Corp. BCM5880 Secure Applications Processor
Bus 002 Device 012: ID 2500:0020
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 004: ID 1bcf:2980 Sunplus Innovation Technology Inc.
Bus 001 Device 003: ID 413c:8187 Dell Computer Corp. DW375 Bluetooth Module
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Figure 39. Vendor ID and product ID of the RF card

Finally, we can launch all the entities in a similar way as mentioned in the previous section and attach a COTS UE to the deployed mobile network.

3.3.3 KVM

Unlike LXC, KVM (Kernel-based Virtual Machine) [46] virtualization requires a separate kernel instance and dedicated resources to run. KVM, categorized as full-virtualized type, abstracts hardware and operating system by emulation or pass-through hardware. KVM is a favourite hypervisor of the OpenStack project and is used in most OpenStack distributions. However, with the release of LXC 2.0 and LXD, LXC/LXD increasingly gains the support from OpenStack. In comparison with KVM, LXC allows to reduce the overhead, which in turn provide the capability to support a large number of containers and allow delivering bare metal performance. In our case, as mentioned earlier, LXC may be suitable for deploying RRU and eNB (in a typical LTE scenario) which need to perform as close as possible to bare-metal speeds. On the other hand, KVM may be more suitable when some kernel requirements are taken into account. It is worthy to note that for the moment how to attach the RF to a KVM instance via USB-passthrough is still an open issue. Both KVM and LXC/LXD are supported by OpenStack.

Here are some quick steps to deploy OAI-based 5G services under KVM.

Step 1: Install KVM

²¹ <https://github.com/lxc/lxd>

```
$sudo apt-get install qemu-kvm libvirt-bin bridge-utils virtinst
```

Step 2: After configuring networking (by relying on the bridged mechanism) and adding user to the libvirtd group, create a KVM instance by using the following commands:

```
$sudo virt-install -n web_devel -r 512 --disk  
path=/var/lib/libvirt/images/web_devel.img,bus=virtio,size=4 -c  
ubuntu-16.04-i386.iso --network network=default,model=virtio  
--graphics vnc,listen=0.0.0.0 --noautoconsole -v
```

Please refer to [47] for more information regarding KVM installation under Ubuntu.

3.3.4 Docker

Docker [48] similar to LXC is relied on the Linux kernel features such as cgroups, namespaces and apparmor to create a virtualized isolated environment. As a result, the performance of both LXC and Docker is very similar. Docker typically acts as an application container which packages the application and all its dependencies in a virtual container that can run on any Linux server that supports the container runtime environment. In other words, Docker containers typically run only a single process per container.

Here are some basic steps to install Docker (Community Edition - CE) e.g., using the Docker repository [49].

Step 1: Setup the Docker repository

```
$sudo apt-get update  
$sudo apt-get install apt-transport-https ca-certificates curl software-properties-common  
$curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -  
$sudo add-apt-repository \  
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \  
$(lsb_release -cs) \  
Stable"
```

Step 2: install Docker CE

To install the latest version of Docker, launch the following commands:

```
$sudo apt-get update  
$sudo apt-get install docker-ce
```

To install a specific version of Docker CE, list the available versions in the repo, then select and install:

```
$apt-cache madison docker-ce  
$sudo apt-get install docker-ce=<VERSION>
```

Step 3: Verify that Docker CE is installed correctly by running the hello-world image.

```
$sudo docker run hello-world
```

Step 4: Deploy OAI functionalities (EPC, eNB/BBU, RRU) as follows:

We first create a network from which static IP addresses are used to assign to the containers.

```
$sudo docker network create --driver=bridge --subnet=172.19.0.0/24 --gateway=172.19.0.1  
oainet
```

```
$sudo docker run --ip=172.19.0.11 --net=oainet -t -i --privileged --rm --name="oai5g_enb" -v /dev/bus/usb:/dev/bus/usb ubuntu:14.04
```

The above command will open a shell inside Docker container, which then allows you to install OAI eNB/BBU as mentioned in the previous section.

Similarly, a container for RRU is created with the following command:

```
$sudo docker run --ip=172.19.0.12 --net=oainet -t -i --privileged --rm --name="oai5g_rru" -v /dev/bus/usb:/dev/bus/usb ubuntu:14.04
```

It is noted that, for EPC, we need to load GTP module in the Docker container, the following command is used:

```
$sudo docker run --ip=172.19.0.9 --net=oainet -t -i --rm -P --privileged --cap-add=ALL -v /dev:/dev -v /lib/modules:/lib/modules -h "yang" --name="oai_epc" ubuntu:14.04 /bin/bash
```

3.4 Other Automated Deployment Tools

A part from the above-mentioned automated deployment tools, Open Baton and Open Source MANO (OSM) are also potential candidate for the virtualized 5G infrastructure deployment.

3.4.1 Open Baton

Open Baton [13] is an open source platform, which provides a standard aligned implementation of the ETSI NFV MANO specification. The architecture of Open Baton is presented in Figure 40. Open Baton is easily extensible. It integrates with OpenStack as main VIM implementation. Additionally, it provides a plugin mechanism for supporting additional VIM types. For more details, beside the NFVO which is fully compliant with ETSI MANO, Open Baton supports [50]: (i) A generic VNFM, which can be easily extended for supporting different type of VNFs; (ii) An Autoscaling Engine (AE system) which can be used for automatic runtime scaling operation of VNFs; (iii) A Fault Management System (FM) for automatic management of faults occurring at the NFVI/VNF level; (iv) a Network Slicing Engine (NSE) to ensure a specific QoS for a Network Slice Instance (NSI) or Network Slice Subnet Instance (NSSI); (vi) A set of libraries for integrating new network services e.g., for building your own VNFMs; and (vii) A dashboard for easily managing all the VNFs.

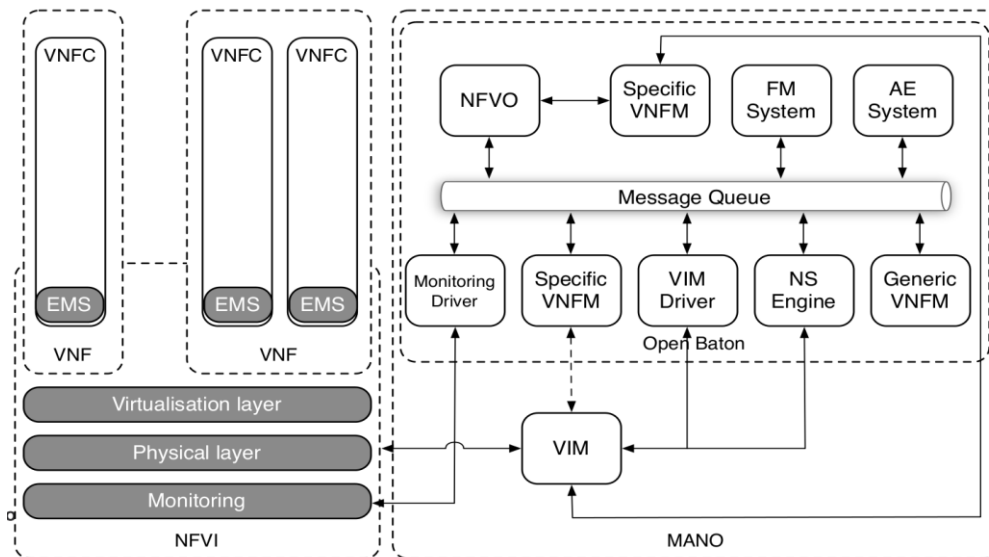


Figure 40. Open Baton architecture [50]

3.4.2 OSM

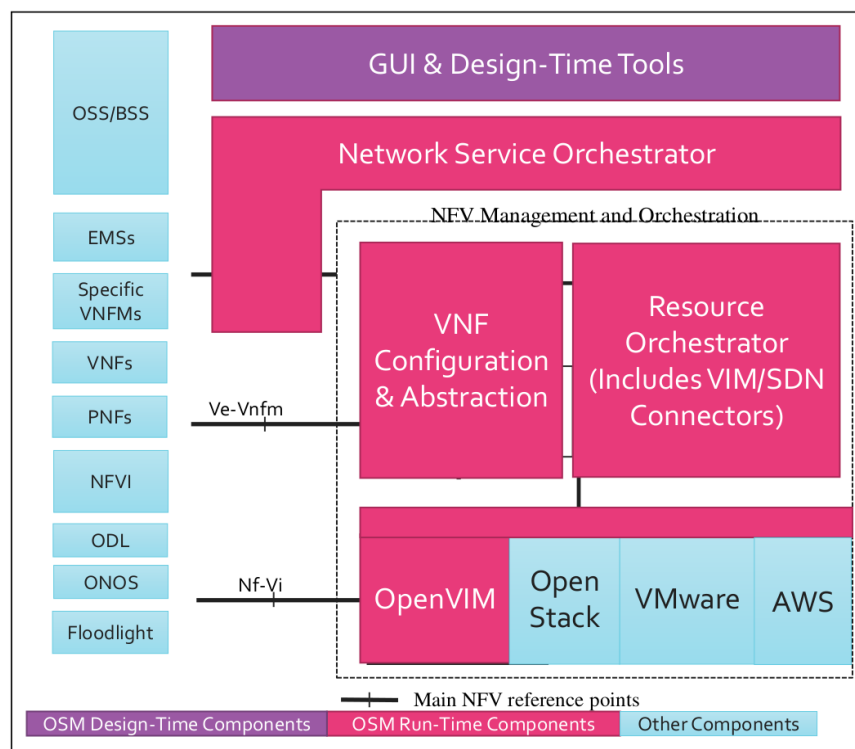


Figure 41. OSM mapping to ETSI NFV MANO [28]

Open Source MANO (OSM) [11][28] is an ETSI-hosted project to develop an Open Source NFV management and orchestration (MANO) software stack aligned with ETSI NFV which is to enhance interoperability with other components (VNFs, VIMs and SDN controllers), and create a plug-in framework to make platform easy to extend and maintain. OSM is published under Apache v2 license, integrates existing open source modules from Telefonica’s OpenMANO project, Canonical’s Juju Charms and Rift.io orchestrator.

Figure 41 shows the approximate mapping of scope between the OSM components and the ETSI NFV MANO logical view. OSM scope covers both design-time and run-time aspects to deliver a production-quality MANO stack. For more information regarding OSM, please refer to [11] [28].

4 A Deployment Example for the SliceNet Slicing-Friendly Infrastructure

Figure 42 shows a deployment example for the SliceNet Infrastructure which is based on the OAI, Mosaic-5G FlexRAN and LL-MEC platforms. This infrastructure offers the following features:

- A RAN runtime slicing system, which enables the dynamic creation of slices with QoS support, while providing functional and resource isolation among different slices (e.g., verticals);
- LL-MEC platform leverages the SDN principle to separate user plane processing from its control logics at the edge and core networks to enable user plane programmability as per slice requirements;
- Dedicated core networks on per slice basis enabling isolation among different slice.

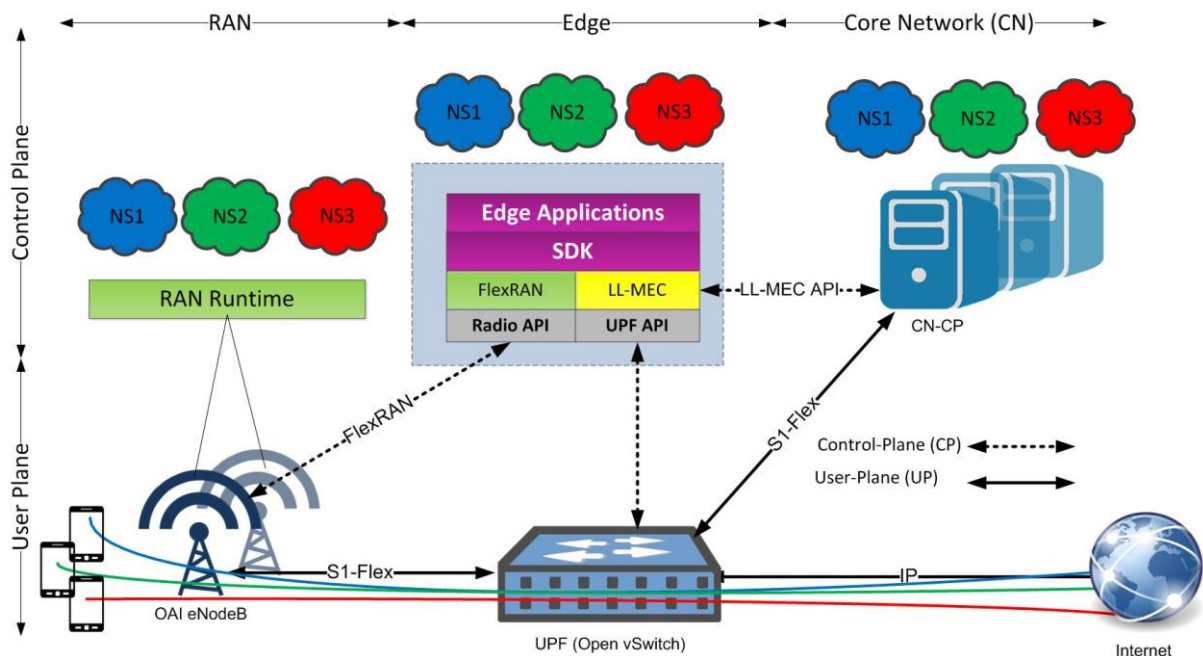


Figure 42. Deployment Example of a SliceNet RAN-Core Slicing-Friendly Infrastructure

5 Conclusions

The main objective of this document is to present the activities related to the design and prototyping of a virtualized 5G RAN-Core infrastructure to achieve an end-to-end slicing-friendly infrastructure. The proposed SliceNet RAN-Core infrastructure leverages OAI and Mosaic5G open-source platforms to provide a flexible platform for the dynamic control and allocation of radio and core network resources (including radio spectrum and resource blocks) and services in response to the needs of the deployed services. The design, implementation and validation of a prototype for integrated network programmability for the RAN and core network is presented with different flavours.

In addition, the current document serves as an in-depth analysis of the different tools for deploying a virtualized 5G infrastructure from the access to the core network in a holistic manner. Lastly, several technical use cases have been prototyped with empirical results in order to validate the essential technical approaches proposed to enable a slicing-friendly RAN-Core infrastructure.

References

- [1] 5GPPP Architecture Working Group, “5G Architecture White Paper: View on 5G Architecture”, Dec. 2017, available online at: <https://5g-ppp.eu/wp-content/uploads/2018/01/5G-PPP-5G-Architecture-White-Paper-Jan-2018-v2.0.pdf>
- [2] OpenAirInterface, Apr. 2018, [Online]. Available: <http://www.openairinterface.org/>
- [3] SliceNet, Deliverable 2.1 - Vertical Sector Requirements Analysis and Use Case Definition, Oct. 2017.
- [4] OpenAirInterface - Core Network (OAI-CN), Apr. 2018, [Online]. Available: <https://github.com/OPENAIRINTERFACE/>
- [5] Mosaic5G.io, Apr. 2018, [Online]. Available: <http://mosaic-5g.io/>
- [6] ETSI GS NFV 002, “Network Functions Virtualisation (NFV); Architectural Framework”, Oct. 2013, available online at http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf
- [7] ETSI GS NFV-MAN 001, “Network Functions Virtualisation (NFV); Management and Orchestration”, Dec. 2014, available online at http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf
- [8] OpenStack, Apr. 2018, [Online]. Available: <https://www.openstack.org/>
- [9] SliceNet, Deliverable 3.1 - Design and Prototyping of SliceNet Virtualised Mobile Edge Computing Infrastructure, Apr. 2018.
- [10] Juju, Apr. 2018, [Online]. Available: <https://jujucharms.com/>
- [11] Open Source MANO, Apr. 2018, [Online]. Available: <https://osm.etsi.org/>
- [12] K. Katsalis, N. Nikaein, and A. Huang, “JOX: an event-driven orchestrator for 5G network slicing”, in Proc. IEEE/IFIP Network Operations and Management Symposium, 2018.
- [13] Open Baton, Apr. 2018, [Online]. Available: <https://openbaton.github.io/>
- [14] OpenStack Foundation Report, “Accelerating NFV Delivery with OpenStack Global Telecoms Align Around Open Source Networking Future”, 2016, available online at <https://www.openstack.org/assets/telecoms-and-nfv/OpenStack-Foundation-NFV-Report.pdf>
- [15] OpenDaylight, Apr. 2018, [Online]. Available: <https://www.opendaylight.org/>
- [16] ONOS, Apr. 2018, [Online]. Available: <https://onosproject.org/>
- [17] Metal as a service (MaaS), Apr. 2018, [Online]. Available: <https://maas.io/>
- [18] Canonical, “Implementing vCPE with OpenStack and Software Defined Networks”, Apr. 2018, [Online]. Available: <https://www.slideshare.net/PLUMgrid/implementing-vcpe-with-openstack-and-software-defined-networks>
- [19] Juju for Telcos and Service Providers Pt. 2, Apr. 2018, [Online]. Available: <https://insights.ubuntu.com/2015/07/23/juju-for-telcos-and-service-providers-pt-2>

- [20] Juju Charm Store, Apr. 2018, [Online]. Available: <https://jujucharms.com/store>
- [21] China Mobile Research Institute, "White Paper of Next Generation Fronthaul Interface", Jun. 2015
- [22] OpenAirInterface (OAI), NGFI Whitepaper, Apr. 2018, [Online]. Available http://www.openairinterface.org/?page_id=1695
- [23] N. Nikaein, E. Schiller, R. Favraud, R. Knopp, I. Alyafawi, and T. Braun, "Towards a Cloud-Native Radio Access Network", Book Chapter of "Advances in Mobile Cloud Computing and Big Data in the 5G Era", Springer, 2016, ISBN: 978-3-319-45143-5
- [24] Juju Charm Metadata, Apr. 2018, [Online]. Available: <https://jujucharms.com/docs/1.25/authors-charm-metadata>
- [25] Canonical Ltd. Juju Documentation, Apr. 2018, [Online]. Available: <https://jujucharms.com/docs/stable/getting-started>
- [26] 3GPP TR 28.801, "3rd Generation Partnership Project, Technical Specification Group Services and System Aspects; Telecommunication management; Study on management and orchestration of network slicing for next generation network", Jan 2018
- [27] Juju VNFM Framework, Apr. 2018. [Online]. Available: <http://jujucharms.com/>
- [28] OSM Release Three, Apr. 2018. [Online]. Available: <https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseTHREE-FINAL.pdf>
- [29] ETSI GS NFV-SOL 001, "Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; NFV Descriptors based on TOSCA; TOSCA-based NFV descriptors", v 0.0.2, July 2016
- [30] IETF, "Technology Independent Information Model for Network Slicing." Available: IETF: draft-qiang-coms-net-slicing-information-model-00
- [31] K. Katsalis, N. Nikaein, E. Schiller, A. Ksentini, and T. Braun, "Network slices toward 5G communications: Slicing the LTE network," IEEE Communications Magazine, vol. 55, no. 8, pp. 146–154, 2017.
- [32] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, "OpenAirInterface: A flexible platform for 5G research," ACM SIGCOMM Computer Communication Review, vol. 44, no. 5, pp. 33–38, 2014.
- [33] Mirantis Fuel, Apr. 2018. [Online]. Available: <https://www.mirantis.com/software/openstack/fuel/>
- [34] Autopilot, Apr. 2018. [Online]. Available: <https://pages.ubuntu.com/Autopilot-welcome.html>
- [35] OpenStack-Ansible, Apr. 2018. [Online]. Available: <https://www.ansible.com/integrations/cloud/openstack>
- [36] Ubuntu MAAS, Apr. 2018. [Online]. Available: <https://www.ubuntu.com/server/maas>
- [37] E. Cebrat, K. Maslowski, "Comparison of OpenStack deployment tools", Semester project Report, Mar. 2017

- [38] OPNFV Upstream Projects – OpenAirInterface, Apr. 2018, [Online]. Available: <https://www.opnfv.org/community/upstream-projects/openairinterface>
- [39] OPNFV - Base system functionality testing (Functest), Apr. 2018, [Online]. Available: <https://wiki.opnfv.org/display/functest/Opnfv+Functional+Testing>
- [40] OpenAirInterface - VNFs for 3GPP Cellular Stack, Apr. 2018, [Online]. Available: <https://wiki.opnfv.org/display/functest/OpenAirInterface+-+VNFs+for+3GPP+Cellular+Stack>
- [41] KVM4NFV Project, Apr. 2018, [Online]. Available: <https://wiki.opnfv.org/display/kvm/Nfv-kvm>
- [42] Pharos Project, Apr. 2018, [Online]. Available: <https://www.opnfv.org/community/projects/pharos>
- [43] Linux Container - LXC, Apr. 2018, [Online]. Available: <https://linuxcontainers.org/>
- [44] LXC – Server guide, Apr. 2018, [Online]. Available: <https://help.ubuntu.com/lts/serverguide/lxc.html>
- [45] LXD, Apr. 2018, [Online]. Available: <https://linuxcontainers.org/lxd/introduction/>
- [46] KVM, Apr. 2018, [Online]. Available: <https://help.ubuntu.com/community/KVM>
- [47] Ubuntu Libvirt, Apr. 2018. [Online]. Available: <https://help.ubuntu.com/lts/serverguide/libvirt.html>
- [48] Docker, Apr. 2018, [Online]. Available: <https://www.docker.com/>
- [49] Get Docker CE for Ubuntu, Apr. 2018. [Online]. Available: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- [50] G. Carella, “Open Baton - The Open Source Network Function Virtualization Orchestrator (NFVO)”, Apr. 2018, [Online]. Available: <https://wiki.opnfv.org/download/attachments/6819410/OpenBaton-OPNFV-16-9-v0.1.pdf>
- [51] T. Nguyen, C. Bonnet, N. Nikaein, P. Matzakos, and K. Bountouris, “Research Report: Testbed Deployment - OAI as a Service”, Dec. 2017.
- [52] Heat – OpenStack Orchestration, Apr. 2018. [Online]. Available: <https://docs.openstack.org/heat/latest/>
- [53] Heat Template Guide, Apr. 2018. [Online]. Available: https://docs.openstack.org/heat/latest/template_guide/index.html
- [54] Athonet, Apr. 2018. [Online]. Available: <https://www.athonet.com>

[End of document]