

6G BRAINS Deliverable D5.3

Final integration for AI-based E2E Network Slicing control and MANO

Editors:	Capgemini Engineering
Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU), embargoed until March 2024 <i>Consortium Confidential (CO) until March 2024</i>
Contractual delivery date:	30 September 2023
Actual delivery date:	11 December 2023
Suggested readers:	5G and beyond (6G) stakeholders
Version:	1.0
Total number of pages:	151
Keywords:	Network slicing, Management and Orchestration, End-to-End, Artificial Intelligence, Machine Learning, Deep Reinforcement Learning, Radio Access Networks

Abstract

This deliverable presents the final enabling technologies and integrated system achieving smart end-to-end (E2E) Network Slicing. It presents the final version of the prototyped subsystems covering RAN and backhaul/backbone Network Slice control, E2E Network Slice management and orchestration, multi-agent reinforcement learning platform and algorithms. It also presents the system integration among the various subsystems, with empirical testing, validation and evaluation results.

Disclaimer

This document contains material, which is the copyright of certain 6G BRAINS consortium parties, and may not be reproduced or copied without permission.

In case of Public (PU):

All 6G BRAINS consortium parties have agreed to full publication of this document.

In case of Restricted to Programme (PP):

All 6G BRAINS consortium parties have agreed to make this document available on request to other framework programme participants.

In case of Restricted to Group (RE):

All 6G BRAINS 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 6G BRAINS 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 6G BRAINS consortium as a whole, nor a certain part of the 6G BRAINS 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 EC flag and the 5G PPP logo reflects that 6G BRAINS receives funding from the European Commission, integrated in its 5G PPP initiative. Apart from this, the European Commission and the 5G PPP initiative have no responsibility for the content of this document.

The research leading to these results has received funding from the European Union Horizon 2020 Programme under grant agreement number 101017226 – 6G BRAINS – H2020-ICT-2020-2. The content of this document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

Impressum

[Full project title]	Bring Reinforcement-learning Into Radio Light Network for Massive Connections
[Short project title]	6G BRAINS
[Number and title of work-package]	WP5 AI-enabled Directional Slicing RAN Slicing and E2E Network Slicing
[Number and title of task]	<p>T5.1 Novel AI-based Directional Radio Access and Backhaul Network Slicing</p> <p>T5.2 Novel Hybrid Network Slicing for the Backbone Network</p> <p>T5.3 Autonomous and Intent-based Management and Network Orchestration for Integrated E2E Network slicing</p> <p>T5.4 Multi-agent Deep Reinforcement Learning Scheme</p>
[Document title]	D5.3 Final integration for AI-based E2E Network Slicing control and MANO
[Editor]	Capgemini Engineering
[Work-package leader]	Qi Wang and Jose M. Alcaraz Calero, UWS
[Estimation of PM spent on the Deliverable]	65 PMs

Executive summary

Effective end-to-end (E2E) Network Slicing plays a crucial role in realizing resource-efficient network traffic management and meeting the diverse quality of service demands across a range of use cases within the context of advanced 5G and 6G networks. This document concludes the integration of AI-driven E2E Network Slicing control Management and Orchestration (MANO) prototypes and components within the framework of the EU 5G-PPP 6G BRAINS project. These prototypes and components encompass various network domains, including the radio access network (RAN), edge infrastructure, transport infrastructure, and core network. In particular, the following accomplishments stand out:

- Development of slicing enablers for both RAN and Core, leveraging the established open-source 5G and beyond OAI platform including the FlexRIC, a versatile and efficient Software Development Kit (SDK) designed to facilitate the construction of service-oriented RAN controllers. Furthermore, notable advancements were made in the realm of Light Fidelity (LiFi) Network slicing over Optical Wireless Communication (OWC) networks.
- Finalised the FlexSlice RAN Slicing Framework, ensuring its compliance with O-RAN standards through integration with the FlexApp xApp framework.
- The development of Twin5G, a cutting-edge optimized Multi-Agent Reinforcement Learning (MARL) framework, was completed where it was specifically designed for operational use within real RAN environments.
- Successfully integrated FlexSlice with NS-3 channel simulations, enabling the creation of prototypes for directional RAN slicing. Integration efforts have linked TSG's theRLib with an abstracted version of OAI for training purposes and with Twin5G for comprehensive testing applications.
- Finalised versions of hardware-based Network Slice control enablers based on XDP smart network interface cards in addition to NetFPGA to benefit from hardware acceleration for data plane programmability, complimentary to the cost-efficient software-based enablers (Linux Traffic Control and Open Virtual Switch) reported before.
- Prototyped an additional slicing enabler Process slicing in the service process domain to guarantee QoS for industrial service processes with computing resource constraints, complimentary to the Network Slicing enablers concerned with networking resource constraints.
- Integration of End-to-End (E2E) Network Slice Control Enablers to facilitate Multi-Level Support (edge, transport and core networks). Different technical methodologies have been explored, all aimed at harnessing the potential of data plane programmability. These efforts have been directed towards the realization of the network flows using the developed enablers to configure and guarantee the QoS.
- Integration of topology discovery and monitoring capabilities, based on an extension to IEEE 802.1ab, with the ONAP-based MANO system to create a topology-aware enterprise-grade Network Slice management solution.

- Finalised version of the multi-agent reinforcement learning platform TheRLib, with improved packaging and ease-of-use, inter-operability with MATLAB-based training simulations, off-line training algorithm with Imitation Learning, and capabilities for the deployment of trained AI agents.
- Finalised version of the multi-agent reinforcement learning algorithms for radio link control, has taken the form of the Matlab Network Simulation of the Reinforcement Learning Radio Link Control, The Markov Decision Process (MDP) Radio Link Control model using Matlab Reinforcement Learning toolbox, and TheRLib which is required to cope with scalability and to cope with large number of User Equipment (UE) (in our case 12) deployed in the network using light weight TheRLib Python Code instances of the Reinforcement Learning agents for each UE as opposed to the much larger footprint Matlab instances of the RL agents. Each of these three individual parts and how they relate to each other have been successfully completed and reported In this deliverable.
- Finally, the fully integrated AI-enabled E2E Network Slicing with complete control encompass a complex solution that seamlessly combines AI, MANO, the concept of Virtual Industry Assistant and the Network Slicing controllers. The designed solution allows to manage and optimize network resources in advanced communication network, particularly in the context of beyond 5G and 6G technologies.

List of authors

Company	Author	Contribution
TSG	Alexandre Kazmierowski Lewis Sear	Leading Section 3.4 (MA-DRL). Section 5.2.9.
UWS	Antonio Matencio Escolar Pablo Salva-Garcia Ignacio Sanchez Navarro Mohamed Khadmaoui-Bichouna Angel Gama-Garcia Mohammad Alselek Jimena Andrade-Hoz Jose M. Alcaraz Calero Qi Wang	Overall structure of the deliverable. Coordination of inputs to the deliverable. Leading Sections 3.1.4 (LiFi-based slicing), 3.2 (hybrid wired segment slicing), 4.1 (overall approach for integration), 4.2 (integration testbed), 4.3 (E2E Network Slicing integration), 5.3.2 (software-based slicing evaluation), 5.3.3 (hardware-based slicing evaluation), 5.3.4 (E2E slicing evaluation), 5.3.5 (process slicing evaluation), and Section 6 (conclusions).
ECOM	Chieh-Chun Chen Ta Dang Khoa Le	Sections 3.1.1 and 5.3.1. Sections 3.1.2, 3.1.3, 4.5, and 5.3.2.
CAP	Paulo Duarte João Fonseca Bruno Mendes Marco Araújo	Contribution for the common sections: executive summary, introduction, and overall achievements. Leading Section 3.3 (E2E Network Slicing MANO). Sections 4.4.1 (integration of ONAP and Network Slice manager), 4.4.2 (integration of ONAP and virtual assistant), 5.2 (testing scenarios), 5.3.5 (E2E Network Slicing and evaluation), and Section 7 (References). Deliverable Editor.

UBRU	John Cosmas Prince Kwaku Boakye Kareem Ali John Miguel Maysam Abbod (review)	Section 3.4.2.3 RAN Radio Link Control DRL algorithms Section 4.6 Integration of TheRLib MA-DRL platform and the MatLab-based RAN link control Section 5.2.5 AI-based RAN radio link control (UBRU) Section 5.3.7 AI-based RAN radio control testing and evaluation
EURES	Anastasius Gavras (review)	

Table of Contents

Executive summary	4
List of authors.....	6
Table of Contents	8
List of figures and tables	10
Abbreviations	14
1 Introduction.....	18
1.1 Objective of this document	18
1.2 Overall methodology.....	19
1.3 Structure of this document	19
2 Overall achievements.....	21
2.1 Enabling technologies achieved	21
2.2 System integration achieved	24
3 Final version of sub-systems	25
3.1 RAN slicing.....	25
3.1.1 FlexSlice: flexible and real-time programmable RAN slicing.....	25
3.1.2 Directional RAN slicing	29
3.1.3 AI-based RAN slicing.....	30
3.1.4 Li-Fi Network Slicing	32
3.2 Hybrid wired segment slicing	36
3.2.1 XDP Hardware-based SmartNIC slicing enablers	36
3.2.2 Topology monitoring based on extension to IEEE 802.1ab	38
3.2.3 Process slicing.....	40
3.3 E2E Network Slice MANO	42
3.3.1 ONAP-based Network Slice MANO	42
3.3.2 Autonomous and Intent-based Network Slice MANO	43
3.4 MA-DRL.....	58
3.4.1 Final version of the MA-DRL platform.....	58
4 Final system prototyping.....	76
4.1 Overall approach	76
4.2 Testbed for integration	76
4.3 Integration for E2E Network Slice control enablers.....	81
4.3.1 Integration of SCA and FlexRIC.....	83

4.4	Integration for E2E Network Slice management and orchestration.....	86
4.4.1	Integration of ONAP and Network Slice Manager	87
4.4.2	Integration of ONAP and Virtual Industrial Assistant	95
4.5	Integration of TheRLib MA-DRL platform to support RAN slice control.....	98
4.6	Integration of TheRLib MA-DRL platform and the MatLab-based RAN link control.	99
4.6.1	Overview	99
4.6.2	Modulation and coding scheme (MCS) and channel quality indicator (CQI)...	100
4.6.3	Modification and Adaptation of Reinforcement Learning Generic Markov Decision Process (MDP) Example Model for Radio Link Control in Matlab.....	103
4.6.4	Linux Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Matlab Development Environment	106
4.6.5	Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab	106
5	Final system testing and performance evaluation.....	118
5.1	KPIs and testing scenarios	118
5.1.1	Summary of KPIs.....	118
5.1.2	Summary of testing scenarios	118
5.2	Validation and performance evaluation results.....	120
5.2.1	FlexSlice RAN slicing testing and evaluation	120
5.2.2	Advanced RAN slicing testing and evaluation	123
5.2.3	OVS software-based Network Slicing enabler testing and evaluation	125
5.2.4	XDP hardware-based Network Slicing enabler testing and evaluation	129
5.2.5	E2E Network Slicing testing and evaluation.....	131
5.2.6	Process slicing testing and evaluation.....	134
5.2.7	AI-based RAN radio link control testing and evaluation	135
5.2.8	Interfacing Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab to the 5G Network Model	144
5.2.9	Interfacing Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab to TheRLib.....	145
6	Conclusions.....	148
7	References.....	149

List of figures and tables

List of figures:

<i>Figure 3.1.1.1: 6G BRAINS functional architecture for smart Network Slicing</i>	21
<i>Figure 3.1.1.1: Hierarchical control for RAN user plans (left) and three potential topologies for control logic (right)</i>	25
<i>Figure 3.1.1.2: High-level architecture of FlexSlice framework with three abstraction control logic examples.</i>	26
<i>Figure 3.1.1.3: Control logic abstraction and Redesigned radio resource scheduler for recursive operation.</i>	28
<i>Figure 3.1.1.4: Algorithm of recursive radio resource scheduling</i>	29
<i>Figure 3.1.2.1: Directional Slicing is executed by grouping UEs on distance-related metrics.</i>	30
<i>Figure 3.1.3.1: The four techniques employed to train an RL agent that optimizes the QoS of a target UE on real RAN software</i>	31
<i>Figure 3.1.3.2: Latency-optimized Model Serving and Replacement</i>	32
<i>Figure 3.1.4.1: Overview of a 5G-Li-Fi hybrid architecture</i>	33
<i>Figure 3.1.4.2: Li-Fi based Network Slice architecture</i>	34
<i>Figure 3.1.4.3: OSI model of the Li-Fi device and Li-Fi modem</i>	35
<i>Figure 3.2.1.1: Data-path of the XDP-based slice enabler with kernel bypass</i>	37
<i>Figure 3.2.2.1: Topology example for LLDP neighbour discovery from TIA</i>	39
<i>Figure 3.2.3.1: Process slicing architecture</i>	41
<i>Figure 3.3.1.1: ONAP High Level Architecture Integration</i>	42
<i>Figure 3.3.2.1: Network Slicing Phases</i>	44
<i>Figure 3.3.2.1.1: Service Design and Control for Network Slice Templates Modelling</i>	44
<i>Figure 3.3.2.1.2: Service Characteristics Template</i>	45
<i>Figure 3.3.2.2.1: Network Service Topology Model Design</i>	46
<i>Figure 3.3.2.2.2: Part of Topology Discovery Workflow</i>	47
<i>Figure 3.3.2.2.3: AAI view of Discovered Topology Result</i>	48
<i>Figure 3.3.2.3.1: AAI model structure for slice creation</i>	49
<i>Figure 3.3.2.3.2: Network Slice Creation Workflow</i>	49
<i>Figure 3.3.2.3.3: Create Custom Communication Service and Send to NSMF</i>	50
<i>Figure 3.3.2.3.4: Network Slice Creation Logic</i>	51
<i>Figure 3.3.2.3.5: Result After creating a network slice via ONAP</i>	52
<i>Figure 3.3.2.4.1: ONAP BPMN Workflow for Slice Attachment</i>	53
<i>Figure 3.3.2.5.1: Delete Slice BPMN Workflow</i>	53
<i>Figure 3.3.2.6.1: Control Loop for RAN Optimization Using ONAP as Non-RT-RIC</i>	54
<i>Figure 3.3.2.6.2: Flow for RAN Measurements and Policy Enforcement</i>	55
<i>Figure 3.3.2.6.3: VES Collector Data Collection</i>	56
<i>Figure 3.3.2.6.4: Logs on E2 Agent showing the creation of Slices</i>	57
<i>Figure 3.4.1.1.1: TheRLib MA-DRL Platform architecture</i>	58
<i>Figure 3.4.1.1.2: Semi-automated CI-CD pipeline</i>	59
<i>Figure 3.4.1.2.1: Overview of the new training architecture</i>	60
<i>Figure 3.4.1.2.2: Overview of a training script</i>	61
<i>Figure 3.4.1.2.3: Overview of the MetricCollection parameters and instantiation</i>	62
<i>Figure 3.4.1.3.1: MATLAB Interface Python code illustration</i>	63
<i>Figure 3.4.1.3.2: MATLAB Interface MATLAB code illustration</i>	64
<i>Figure 3.4.1.3.3: Illustration of Overview of the MATLAB Interface</i>	65
<i>Figure 3.4.1.3.4: Episode Reward and Time Taken for a CartPole experiment within a MATLAB Environment (red) and a Gym Environment (blue)</i>	66
<i>Figure 3.4.1.4.1: Main principles of the TheRLib-deploy component</i>	67
<i>Figure 3.4.1.4.2: Deployment pipeline: connect to Therdash and query experiment options</i>	68
<i>Figure 3.4.1.4.3: Deployment pipeline: query which neural network weights are available</i>	69
<i>Figure 3.4.1.4.4: Deployment pipeline: query the metric data associated to the neural network weights</i>	69

<i>Figure 3.4.1.4.5: Deployment pipeline: download and decode the adequate neural network weights</i>	70
<i>Figure 3.4.1.5.1.1: Behavioural Cloning experiment process</i>	71
<i>Figure 3.4.1.5.1.2: Behavioural Cloning experiment dashboard view</i>	71
<i>Figure 3.4.1.5.3.1: Relationship between Network Simulation, Matlab's Markov Decision Process Model</i>	73
<i>Figure 3.4.1.5.3.2: Relationship between Network Simulation, Matlab's Markov Decision Process Model and TheRLib MA-DRL platform</i>	73
<i>Figure 3.4.1.5.3.3: Detail of Matlab MDP model</i>	74
<i>Figure 3.4.1.5.3.4: Applying TheRLib MA-DRL platform for both training and deployment of MDP</i>	74
<i>Figure 3.4.1.5.3.5: Applying TheRLib MA-DRL platform for Neural Network training and deployment</i>	75
<i>Figure 3.4.1.5.3.6: Applying TheRLib MA-DRL platform for Neural Network training and deployment</i>	75
<i>Figure 4.2.1: High level view of the 6G-BRAINS testbed for WP5 integration</i>	77
<i>Figure 4.2.2: CORE compute node for 6G-BRAINS integration</i>	79
<i>Figure 4.2.3: RAN compute node deployed in Edge segment for 6G-BRAINS integration</i>	80
<i>Figure 4.2.4: Compute deployed in Edge segment hosting THALES RLib 6G-BRAINS component</i>	81
<i>Figure 4.3.1: Slice Control Agent (SCA) - Logical Architecture</i>	82
<i>Figure 4.3.1.1: RMC architecture</i>	84
<i>Figure 4.3.1.2: RMC endpoints</i>	84
<i>Figure 4.4.1.1.9.1: High Level Architecture Integration ONAP - Topology Controller and Slice Manager</i>	94
<i>Figure 4.4.1.1.9.2: Topology Slicing View</i>	95
<i>Figure 4.5.1: The inner mechanism of our abstracted FlexRIC-OAI simulation</i>	98
<i>Figure 4.5.2: Differences between Training Expectation and Actual Tested results</i>	99
<i>Figure 4.6.2.1: Conventional Choosing the best Modulation and Coding Scheme (MCS) from Channel Quality Indicator (CQI)</i>	100
<i>Figure 4.6.2.3.1: MCS, CQI & BLER</i>	103
<i>Figure 4.6.3.1.1: Original MDP Model</i>	104
<i>Figure 4.6.3.1.2: Reference Simulation for BLER</i>	104
<i>Figure 4.6.3.1.3: move_function embedded deep inside successive function calls from the top level program</i>	105
<i>Figure 4.6.3.1.4: Reinforcement Learning Episode Manager</i>	105
<i>Figure 4.6.5.1.1: Defining the State Space</i>	107
<i>Figure 4.6.5.1.2: Specifying the Transition Matrix</i>	108
<i>Figure 4.6.5.1.3: Reference Simulation for Reward Matrix</i>	108
<i>Figure 4.6.5.1.4: MDP against BLER for single state transitioning [12]</i>	109
<i>Figure 4.6.5.1.5: MDP against BLER for up to two state transitioning[13]</i>	110
<i>Figure 4.6.5.1.6: Reference Simulation for Block Error Rate</i>	110
<i>Figure 4.6.5.2.1: Simulation of creating Q-learning agent.</i>	111
<i>Figure 4.6.5.4.1: Simulation of training Q-learning agent</i>	113
<i>Figure 4.6.5.4.2: Simulation of Validate Q-Learning Results</i>	113
<i>Figure 4.6.5.5.1: MCS, CQI & BLER</i>	114
<i>Figure 4.6.5.6.1: Reference simulation of the function signature</i>	114
<i>Figure 4.6.5.6.2: Reference Simulation for initialization</i>	116
<i>Figure 4.6.5.6.3: Reference Simulation of State Transition logic</i>	116
<i>Figure 4.6.5.6.4: Reference Simulation of Reward Calculation</i>	116
<i>Figure 4.6.5.6.5: Reference Simulation of BLER</i>	117
<i>Figure 5.2.1.1: Reference Simulation of Reward Calculation</i>	120
<i>Figure 5.2.1.2: Memory consumption comparison.</i>	120
<i>Figure 5.2.1.3: Description of six scenarios (Here s1 and s2 represent slice 1 and slice 2, respectively).</i>	122
<i>Figure 5.2.1.4: Network and User performance in a fully loaded scenario with a centralized topology.</i>	122
<i>Figure 5.2.2.1: Learning and testing curves of the 3 training regimes; the regime with 10MS control-loop is excluded from testing due to bad training performance.</i>	124
<i>Figure 5.2.2.2: Comparing DQN variations (RAN inter-slicing) and difference RAN intra-slicing</i>	124
<i>Figure 5.2.3.1: Empirical analysis of the scalability of the time required by the SCA and UWS-OVS data plane agent to create a Network Slice.</i>	125

<i>Figure 5.2.3.2: Average delay per slice and use case (in μs) when processing heterogeneous network traffic from 280,920 IoT devices and 432 slices by the UWS-OVS data plane agent.</i>	128
<i>Figure 5.2.3.3: Average bandwidth per slice for each use case in a scenario with heterogeneous 5G IoT traffic.</i>	129
<i>Figure 5.2.4.1: Packet Per Second, 1 Socket</i>	130
<i>Figure 5.2.4.2: Packet Per Second, 16 Socket</i>	130
<i>Figure 5.2.4.3: Packet Loss, 1 Socket</i>	130
<i>Figure 5.2.4.4: Packet Loss, 16 Sockets</i>	130
<i>Figure 5.2.5.1.1: Slice manager E2E Network Slice creation time</i>	132
<i>Figure 5.2.5.2.1: ONAP Topology Synchronization Process Times</i>	133
<i>Figure 5.2.5.2.2: ONAP Slice Creation Processing Times</i>	133
<i>Figure 5.2.5.2.3: ONAP Slice Deletion Processing Times</i>	134
<i>Figure 5.2.6.1: Metrics data for computing resources.</i>	135
<i>Figure 5.2.7.3.1: Single State Transitioning Episode 2</i>	136
<i>Figure 5.2.7.3.2: Single State Transitioning Episode 11</i>	137
<i>Figure 5.2.7.3.3: Single State Transitioning Episode 20</i>	137
<i>Figure 5.2.7.4.1: Single State Transitioning Episode 24</i>	138
<i>Figure 5.2.7.4.2: Single State Transitioning Episode 50</i>	138
<i>Figure 5.2.7.4.3: Single State Transitioning Episode 86</i>	139
<i>Figure 5.2.7.5.1: Single State Transitioning Episode 130</i>	140
<i>Figure 5.2.7.5.2: Single State Transitioning Episode 210</i>	140
<i>Figure 5.2.7.5.3: Episode 240</i>	141
<i>Figure 5.2.7.5.4: Single State Transitioning Episode 250</i>	142
<i>Figure 5.2.7.6.1: Up to two State Transitioning Episode 2000</i>	143
<i>Figure 5.2.7.7.2: Episode reward for rIMDPEnv with riQAgent</i>	144
<i>Figure 5.2.8.1: Matlab's Network Topology Visualisation of Intercell Interference Model</i>	144
<i>Figure 5.2.8.2: DRL Link Control Markov Decision Process</i>	145
<i>Figure 5.2.9.1: Training curve with the Soft Actor Critic (SAC) algorithm</i>	146
<i>Figure 5.2.9.2: Training curve with the Proximal Policy Optimisation (PPO) algorithm</i>	147

List of tables:

<i>Table 3.3.2.6.1: A1 Policy Example for Slicing Enforcement</i>	57
<i>Table 4.3.1: Integration Status of Slice Control Enablers with the SCA</i>	82
<i>Table 3.4.2.1: Add a new slice related to a base station node</i>	85
<i>Table 3.4.2.2: Attach new UE to a slice</i>	86
<i>Table 3.4.2.3: Deleting an existing slice related to a base station node</i>	86
<i>Table 3.4.2.4: Deleting all slices related to a base station node</i>	86
<i>Table 3.4.2.1: Get Topology</i>	89
<i>Table 3.4.2.2: Create Slice</i>	90
<i>Table 3.4.2.3: Get Slice</i>	90
<i>Table 3.4.2.4: Get All Slices</i>	91
<i>Table 3.4.2.5: Attach Service to Slice</i>	92
<i>Table 3.4.2.6: Get All Attached Services</i>	93
<i>Table 3.4.2.7: Detach Service</i>	93
<i>Table 3.4.2.8: Detach Service</i>	94
<i>Table 4.4.2.1: Service Order Request Add Slice</i>	98
<i>Table 4.6.2.1.1: MCS Index Table 5.1.3.1-3 in [20]</i>	101
<i>Table 4.6.2.2.1: 4-bit CQI Table 5.2.2.1-2 in [12]</i>	102
<i>Table 5.2.2.1: Uplink specification of the AGV video streaming slice</i>	123
<i>Table 5.2.3.1: Testbed set up for empirical evaluation of UWS-OVS with heterogeneous traffic</i>	126
<i>Table 5.2.3.2: Example of proposed priorities mapping different network traffic profiles</i>	127

Abbreviations

5G	Fifth Generation (mobile/cellular networks)
5G PPP	5G Infrastructure Public Private Partnership
6G	Sixth Generation
6G BRAINS	Internet of Radio Light (project)
A&AI	Active and Available Inventory
AI	Artificial Intelligence
AMF	Access and Mobility Functions
BSR	Buffer Status Reports
CFN	Cell Free Network
CNF	Cloud-native Network Function
CQI	Channel Quality Indicator
CSMF	Communication Service Management Function
CTDE	Centralised Training with Decentralised Execution
DAS	Dynamic Autoscaling
DPI	Deep Packet Inspection
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
DPAS	Data Plane Abstraction Service
E2E	End-to-end
eBPF	Extended Berkeley Packet Filter
eMBB	Enhanced Mobile Broadband
FlexCN	Flexible Core controller
FlexRIC	Flexible RAN intelligent controller

GUI	Graphical User Interface
HARQ	Hybrid Automatic Repeat Request
HAS	Hyper Autoscaling
iApps	Controller-internal Applications
IVA	Industrial Virtual Assistant
MAB	Maximum Allowed Bandwidth
MA-DRL	Multi-Agent Deep Reinforcement Learning
MANO	Management and Orchestration
MARL	Multi-Agenmt Reinforcement Learning
MCS	Modulation and Coding Scheme
MDP	Markov Decison Process
MEC	Mobile Edge Computing
MGB	Minimum Guaranteed Bandwidth
ML	Machine Learning
NBI	North Bound Interface
NFP	Netronome Flow Processor
NFV	Network Function Virtualisation
Non-RT-RIC	Non Real Time Radio Intelligent Controller
NPU	Network Processing Unit
NSMF	Network Slice Management Function
NST	Network Slice Template
OAI	Open Air Interface
ONAP	Open Network Automation Platform
ONNX	Open Neural Network Exchange

OOF	ONAP Optimisation Framework
OVS	Open vSwitch
PBT	Population Based Training
PCI	Peripheral Component Interconnect
PDU	Packet Data Unit
PMI	Precoding Matrix Indicator
PNF	Physical Network Function
PPO	Proximal Policy Optimisation
PPS	Packets per Second
PRB	Physical Resource Blocks
QoS	Quality of Service
R&D	Research and Development
RAN	Radio Access Network
RAN UP	RAN User Plans
rAPPs	RAN application located at NON-RT-RIC
RAT	Radio Access Technology
RBG	Resource Block Group
REST	Representational State Transfer
RIC	Radio Intelligent Controller
RSS	Receive Side Scaling
SAC	Soft Actor Critic
SCA	Slice Control Agent
SC SN	Slicing Control Service Model
SCTP	Stream Control Transmission Protocol

SDK	Software Development Kit
SDN	Software Defined Networks
SLA	Service-level Agreement
SMAC	StarCraft Multi-Agent Challenge
SMF	Session Management Function
SO	Service Orchestrator
TIA	Topology Inventory Agent
UPF	User Plane Function
URLLC	Ultra-reliable Low Latency
VL	Virtual Link
VM	Virtual Machine
VNF	Virtual Network Function
WAN	Wide Area Network
xApps	Controller-external Applications
XDP	Express Data Path

1 Introduction

In the context of evolving network technologies, Network Slicing remains a pivotal element in both 5G and future 6G networks. As outlined in 6G Brains D2.1, it serves as a fundamental solution catering to diverse requirements across various vertical industries, particularly in Industry 4.0. This report empathizes the importance of advanced End-to-End(E2E) Network Slicing, specifically bringing Artificial Intelligence (AI) for Radio Access Network(RAN) slicing. It highlights the need of high-speed and flexible backbone for Network Slicing, encompassing Multi-Access/Mobile Edge Computing (MEC), Transport Network and the Core Network. Also, it shows the importance of Management layers such as MANO and Slice Controllers allowing to explore advanced data plane programmability to ensure robust Service-Level Agreements (SLAs).

This document serves as the final progress report, focusing on the advancements achieved in intelligent E2E Network Slicing promoting an open-source framework. Specifically, it highlights the outcomes of T5.3 by designing and prototyping the Management and orchestration (MANO) plane for E2E Network Slicing, including Slice controllers and Slice Inventory Agents, the advancements in the RAN, Backhaul/backbone slicing, and the results from T5.4 in prototyping the Multi-Agent Deep Reinforcement Learning (MA-DRL) platform for training intelligent resource allocation policies within E2E Network Slicing tasks, and updates from T5.1 and T5.2 in prototyping technologies for E2E Network Slicing. Furthermore, it elaborates on the integration efforts within WP5, emphasizing the integrations between T5.1 and T5.2 regarding different enablers, the integration between T5.2 and T5.3 concerning MANO and Integration of the MA-DRL with AI applications in E2E Network Slicing between T5.4 and T5.1, and the integration with some components of WP4 including the Virtual Voice Assistant in the final E2E Network Slicing framework.

Finally, the report elucidates the outcomes derived from the testing and evaluation phase pertaining to E2E Network Slicing.

1.1 Objective of this document

The objectives of this deliverable are listed below:

- Final architecture details, where a unified framework is presented enabling Network Slicing within the RAN, meeting D5.2 requirements.
- Describe the technical advancements achieved during the project, regarding design, implementation, and experimental results, including AI implementation for Network Slicing.
- Describe the technical details and experiments results of the design integration and prototypes related to E2E Network Slice control.
- Promote open source based on technical approach for advanced Network Slicing.
- Describe the final prototype including the integration of components across RAN segment, slicing management and network controllers, harmonizing MANO framework and AI-based controllers.

- Finally, the report details the findings from the testing and evaluation phase concerning E2E Network Slicing.

1.2 Overall methodology

The overall methodology adopted for this project was structured around several key developmental phases and integration efforts:

- **Network Slicing Enablers:** On the RAN side, we implemented FlexSlice as a RAN function within OpenAirInterface (an open-sourced 5G stack) to enable RAN slicing for different control topologies. We then extended FlexRIC (an open-sourced SD-RAN controller) with FlexApp to enable the development O-RAN-compliant RAN slicing xApps. Additionally, advancements in Core slicing and, interestingly, Light Fidelity (Li-Fi)-based slicing over Optical Wireless Communication were also achieved.
- **Twin5G MARL Framework:** To enable accurate emulations of Industrial 5G scenarios, we departed from traditional Gym-based RL to train directly on the real RAN. In the end, we achieved both scalable and near-discrete-time emulations, removing the need for RAN simulators in MARL.
- **Integration Efforts:** Successful integration was achieved between FlexSlice, NS-3 channel simulations, FlexApp, Twin5G, and TheRLib in a factory-inspired testing scenario. During training, TheRLib was interfaced with an abstracted simulator of OAI, leading to both useful (convergence) and interesting insights (learning stability).
- **Hardware-Based Enablers:** Development was completed on hardware-based Network Slice control enablers, utilizing eXpress Data Path (XDP) smart network interface cards and NetFPGA for hardware acceleration, complementing previously reported software-based enablers.
- **Process Slicing and E2E Network Slice Control:** Prototyping process slicing for Quality of Service (QoS) assurance in industrial service processes, in addition to integrating E2E Network Slice Control Enablers across multiple network levels (edge, transport, core) through various technical methodologies focusing on data plane programmability and QoS configuration.
- **Topology Discovery and Monitoring Integration:** Integration of topology discovery and monitoring capabilities, based on IEEE 802.1ab extension, with the ONAP-based MANO system. This facilitated the creation of a Topology-aware Network Slice Management solution suitable for enterprise-grade environments.
- **The TheRLib MA-DRL Platform:** the finalization of TheRLib, a multi-agent reinforcement learning platform, makes it easier to install and use. This phase added inter-operability with MATLAB-based learning environments, capabilities for off-line learning with Imitation Learning (IL) techniques, and proof-of-concept for deployment of trained agents. This contributed significantly to the project's overall framework.

1.3 Structure of this document

The rest of the document is organized as follows:

- Section 2 presents the overall achievements, mainly the achieved assets and the various integrations.
- Section 3 describes the final version of the sub-systems that compose the final AI-based E2E networking slicing control and MANO system.
- Section 4 describes the details of the final integration of the sub-systems, components and enablers delivered as outcomes in WP5, that enables the End-to-End Network Slicing capabilities in 5G and beyond infrastructures.
- Section 5 presents the final system testing and performance evaluation.
- Section 6 concludes the document.

2 Overall achievements

Figure 2.1 illustrates the 6G BRAINS functional architecture for smart Network Slice control, management, and orchestration, as introduced in D2.2 and further developed in D5.1 and D5.2.

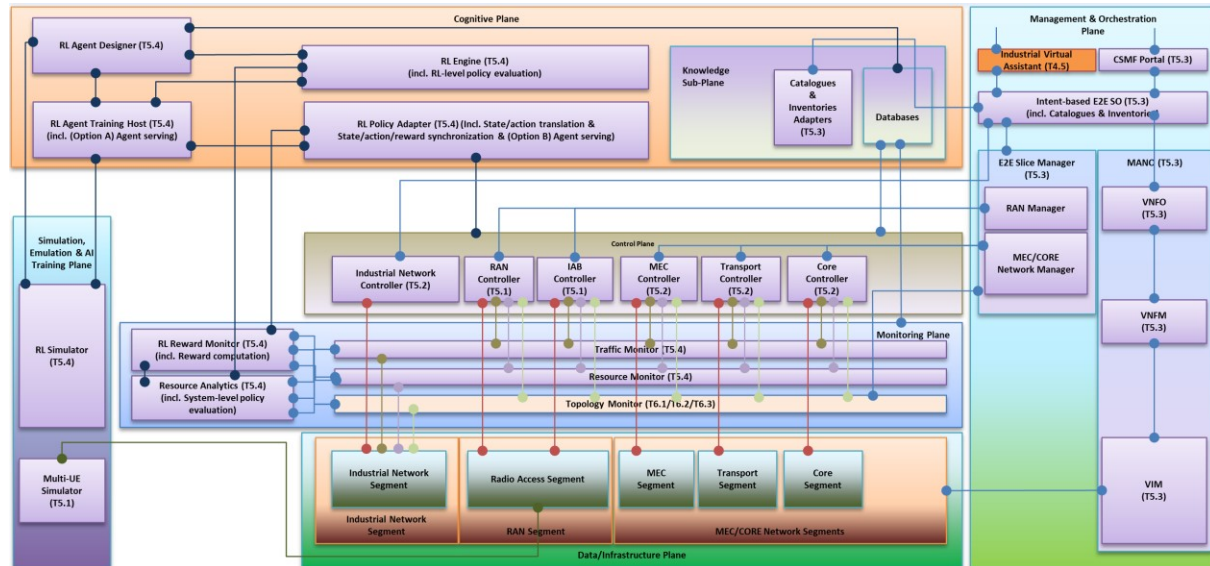


Figure 3.1.1.1: 6G BRAINS functional architecture for smart Network Slicing

From the comprehensive functional architecture, distinct planes delineate specific functionalities. Initiating with the control plane, we can find the O-RAN compliant RAN slicing framework (T5.1) alongside the backbone network's Network Slicing (T5.2). From the Management and Orchestration (MANO) plane (T5.3), pivotal components encompass the MANO, E2E Network Slice Manager, and the Intent-Based Service Orchestration. Positioned with the cognitive plane, lies the network's intelligence and knowledge repository, housing reinforcement learning platform and algorithms dedicated to radio resource optimization. This structured delineation showcases the architecture's organization, delineating functionalities across planes for seamless Network Slicing orchestration and optimization. The subsequent chapters present concise summary of the main enablers developed throughout the project and their integration efforts. These summarizes pivotal advancements achieved within the project timeline, highlighting the development and integration of key components that contributed to the realization of the project objectives.

2.1 Enabling technologies achieved

RAN Slicing

- The FlexApp low-latency and multi-language xApp framework for the O-RAN RIC; achieved sub-millisecond control-loop time even in Python.
- The FlexSlice programmable RAN slicing framework, which executes RAN slicing for multiple topologies at various trade-offs. FlexSlice is extended with NS-3 channel simulations to prototype power-based Directional Slicing.

- Implemented and tested Hyper Autoscaling (HAS), a *Resource-constrained Off-policy Learning (ResourceOff)*-based approach to RAN slicing, on OAI and FlexRIC using *Twin5G* (a real-RAN-based MARL framework). Twin5G and BudgetOff are listed as achievements in the Reinforcement Learning section.
- Network Slicing control loop for RAN mechanism was created facilitating the RAN metric collection and analysis through rApp, aiming to create Network Slices based on predefined policies.
- Li-Fi Network Slicing over OWC networks, as a new enabler for the Factory of the future use cases, especially the Internet of Things (IoT) networks for demanding industrial applications. It provides an alternative solution in a complementary and underutilised spectrum for the last hop, beyond Network Slicing over wireless technologies that operate in the commonly used Radio Frequency (RF) spectrum. It is noted that Li-Fi Network Slicing can co-exist with RF and backhaul/backbone Network Slicing in a hybrid system.

Backhaul/backbone Network Slicing

- Hardware-based data path Network Slicing solutions based on eXpress Data Path(XDP) kernel bypassing (e.g., Netronome) and FPGA (e.g., NetFPGA) smart network interface cards (NICs), providing performance boost benefiting from hardware acceleration. In particular, the XDP-based enabler combines both hardware offloading and in-driver enabling to achieve more efficient network traffic control and rapid traffic engineering in the application space, leading to improved performance compared with NetFPGA. In addition, it facilitates on-demand, real-time network card programming.
- Software-based data path Network Slicing solutions based on the Linux Traffic Control (TC) and the Open Virtual Switch (OVS), offering cost-efficient programmable data path capabilities in the kernel and/or user space of the system. Both TC and OVS are widely available in most of the 5G/6G infrastructure, and thus are recommended as the default data path Network Slicing enablers.
- Software and hardware-based solutions can be exploited in a hybrid manner wherever appropriate along the series of control points for Network Slicing over the E2E path. For instance, hardware-based enablers should be deployed in critical control points where performance is critical. On the other hand, software-based enablers would be recommended in any control points that are not equipped with higher-performance programmable data-plane network devices such as those based on SmartNICs or kernel bypass techniques.
- Process slicing for providing QoS for industrial service processes by prioritising computing resources, complementary to the Network Slicing enablers above, which resolve the competition for networking resources. By combining both Network Slicing and process slicing, bottlenecks in both resource domains can be removed to achieve further improved performance for mission-critical industrial applications that operate over demanding computing resources.

Network management and orchestration (MANO)

- Enterprise-grade MANO based on ONAP, capable to proficiency manage the life-cycle of the Network Slices, using third party controllers, and efficiently maintain their inventory within the network infrastructure.
- Intent-based Network Slicing Management leveraging voice commands as user interface. This approach allows for execution of Network Slicing operations and management tasks through natural language voice commands, offering a user-friendly and intuitive control mechanism.
- Autonomous Network Slicing using ONAP as a Non-Real Time Radio Intelligent Controller (Non-RT RIC), integrated with FlexRic, allowing metrics collection via a VES (Virtual Event Streaming) collector. It ensures self-regulating management of Network Slicing by collecting relevant metrics and enforcing policies based on predefined criteria aiming the Network Slicing creation.
- The Slice Control Agent (SCA) framework for technology-agnostic Network Slice control over different network segments. The SCA plays a central role in dealing with and integrating the heterogeneous Network Slicing enablers across RAN (FlexRIC, Li-Fi etc.) and backhaul/backbone networks into an E2E Network Slice control framework and are compatible with both hardware- and software-based enablers.
- The Topology Inventory Agent scheme for topology discovery and monitoring in complex networking environment. This topology solution is based on a novel extension to the IEEE 802.1ab standard. It enables overriding the automatically set capabilities to improve topology discovery and is useful to discover the E2E network topology for the subsequent E2E Network Slice creation.

Reinforcement learning platform and algorithms

- The TherLib Multi-Agent Reinforcement Learning platform for AI algorithm training and deployment. This solution provides single-agent and multi-agent Deep Reinforcement Learning algorithms for online training with simulations (OpenAI Gym based, MATLAB-based, etc.) and Imitation Learning algorithms for off-line training. It also comes with management and monitoring system of the AI trainings, accessible with a web dashboard, and capabilities for the deployment of trained AI agents.
- Twin5G, a near-real-time Multi-Agent Reinforcement Learning framework that runs on the real RAN, to accurately emulate Private 5G scenarios of *any* scale.
- *Resource-constrained Off-policy Learning*, a new class of MARL algorithms that allow separately-trained RL agent to optimally run together in a resource-compete environment.
- The radio link control algorithm based on Multi-Agent Reinforcement Learning to allow optimised RAN operation consists of two parts: (1) Matlab Network Simulation of the Reinforcement Learning Radio Link Control, which is described in Deliverable D5.2 Section 6.3; (2) The Markov Decision Process (MDP) Radio Link Control model using Matlab Reinforcement Learning toolbox, which is described in this D5.3 Section 5.3.7. How they and TherLib relate with each other. which is described in D5.3 Section 3.4.2.3, is required to cope with scalability and to cope with large number of User

Equipment (in our case 12) deployed in the network using light weight TheRLib Python Code instances of the Reinforcement Learning agents for each UE, which is shown in D5.3 Section 5.2.9, as opposed to the much larger footprint Matlab instances of the RL agents. Each of these three individual parts and how they relate to each other have been successfully completed and reported In this deliverable.

2.2 System integration achieved

- Integrated ECOM's RAN slicing with UWS's backhaul/backbone Network Slicing to create end-to-end Network Slice control. The RAN slicing enabler FlexRIC and selected hybrid backhaul/backbone enablers including TC, OVS, and NetFPGA have been integrated through the southbound interface of the Slice Control Agent framework, and are able to work independently and together along the heterogeneous control points along the E2E data path. Thanks to the Slice Control Agent, which handles and hides such heterogeneity from upper layers, the network management and orchestration system is enabled to focus on intents only.
- Integrated UWS's Slice Control Agent framework and CAP's intent-enabled ONAP-based MANO to achieve technology-agnostic end-to-end Network Slice management. The MANO has been integrated to the northbound interface of the Slice Control Agent so that the intents received from the user can be interpreted properly and enforced to the right network segment, RAN or backhaul/backbone, using the most appropriate Network Slicing enabler per control point along the E2E route.
- Integrated UWS's Topology Inventory Agent scheme with CAP's Intent-enabled ONAP-based MANO to achieve Enterprise-grade Topology-aware MANO for improved E2E Network Slice Management. This integration has been achieved through the northbound interface of the UWS's Slice Manager and has significantly enlarged the capability of network topology discovery and monitoring of the ONAP platform.
- Integrated the Industrial Voice Assistant (from WP4) with CAP's MANO platform to achieve voice-controlled Network Slice Management. Using plain natural language the user can send the intents to create Network Slices using voice. Then the request is translated on the request known by ONAP Northbound Interface (NBI).
- Integrated TSG's TheRLib platform with ECOM's abstracted OAI during training, and with ECOM's Twin5G, FlexApp, and FlexSlice during testing. Both training and testing targeted Industrial 5G scenarios.
- Integrated TSG's TheRLib platform with UBRU's MATLAB-based Network Simulation of the Reinforcement Learning Radio Link Control and Markov Decision Process (MDP) Radio Link Control model using Matlab Reinforcement Learning toolbox.

3 Final version of sub-systems

3.1 RAN slicing

This section first describes T5.1's O-RAN-compliant RAN slicing framework (3.1.1), how it is extended for directional RAN slicing (3.1.2), and AI-based RAN slicing (3.1.3). Finally, we conclude the section with a novel RAN slicing technology: hybrid 5G and Li-Fi (3.1.4).

3.1.1 FlexSlice: flexible and real-time programmable RAN slicing

A. FlexSlice Introduction

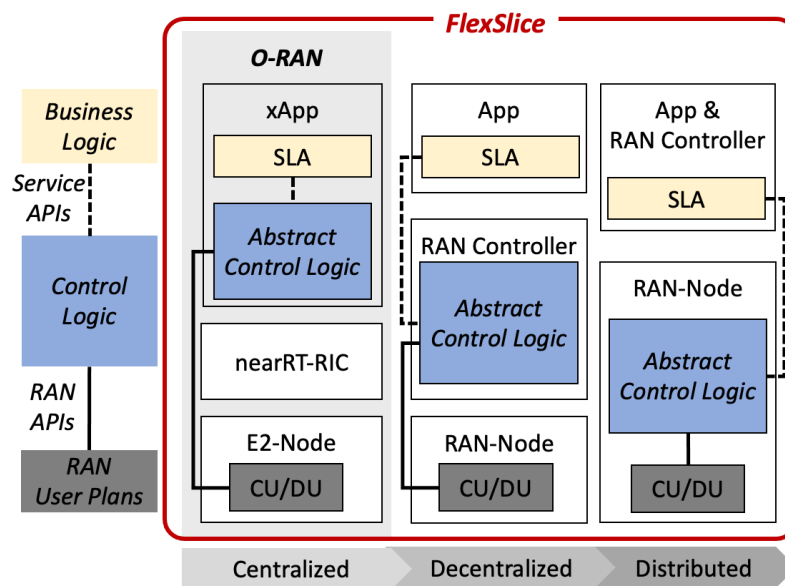


Figure 3.1.1.1: Hierarchical control for RAN user plans (left) and three potential topologies for control logic (right)

In the envisioned Open RAN model outlined by the O-RAN Alliance, the complexity of RAN control becomes even more intricate. In this scenario, an application (like an xApp in the O-RAN architecture) operating on top the RAN controller (such as the near real-time RAN intelligent controller [nearRT-RIC]) must manage multiple RAN-Nodes. These nodes include both Centralized Units (CU) and Distributed Units (DU) referred to as E2-Nodes in the O-RAN architecture. Additionally, while sending control commands to these underlying RAN-Nodes, the RAN controller must synchronize various applications and minimize potential conflicts. Another challenge with the O-RAN design is its inability to provide real-time control and monitoring of sub-10ms control loops, which are necessary for enabling RAN slicing in 5G-Advanced (Release 18) services.

To generalize the O-RAN approach and introduce our proposed FlexSlice framework, we introduce three main components in Figure 3.1.1.1: (1) business logic; (2) control logic; and (3) RAN User Plans (RAN UP). Service providers define the business logic in terms of required Key Performance Indicators (KPIs) like expected bandwidth and latency, specified in the Service Level Agreement (SLA). This business logic is then translated into control logic, encompassing parameters such as Radio Resource Control (RRC) and Radio Resource

Management (RRM) rules, using available service Application Programming Interfaces (APIs). The control logic is subsequently applied to the RAN UP, such as the radio resource scheduler at the Medium Access Control (MAC) layer, utilizing accessible RAN APIs.

As depicted in blue on the right side of Figure 3.1.1.1, the control logic can be centralized within the App, decentralized within the RAN controller, or distributed across RAN-Nodes. The key trade-offs involve control loop latency and RAN-Node complexity. Furthermore, by adopting an appropriate control logic architecture, additional use-cases, like scenarios involving mobile cells/DUs with non-ideal links between RAN-Node and RAN controller, can be realized. Consider the centralized approach (as in the current O-RAN model) where the control logic mainly resides in the App. This approach offers the lowest RAN complexity but suffers from higher control loop latency, limiting its real-time control capabilities. In contrast, both decentralized and distributed topologies are better suited for dynamically controlling RAN UP, as they require sub-millisecond control latency to adapt to shorter Transmission Time Intervals (TTIs), e.g., < 1ms. Moreover, this design simplifies the App's structure (see Figure 2), enabling the same App to work across various RAN controller platforms.

B. FlexSlice Framework

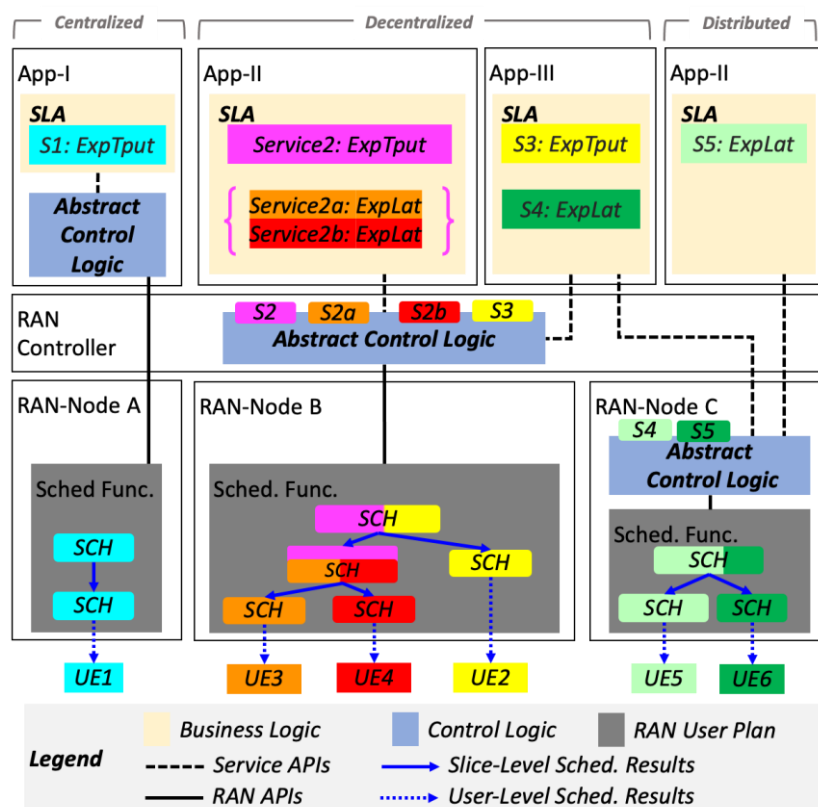


Figure 3.1.1.2: High-level architecture of FlexSlice framework with three abstraction control logic examples.

The FlexSlice framework, as proposed, evolves the existing O-RAN architecture by offering flexible deployment options for control logic: centralized, decentralized, or distributed (refer to Figure 3.1.1.2). This flexibility is aimed at achieving a reduced control loop latency. Additionally, the framework employs a recursive approach to realize the radio resource

scheduling function, enhancing its adaptability and extensibility. This recursive concept can also be extended to various other functions within the RAN, enabling RAN nodes to support multiple services and tenants simultaneously.

In Figure 3.1.1.1, an overarching architecture of FlexSlice is depicted, comprising three key components: business logic (yellow), control logic (blue), and RAN UPs (gray). The service provider outlines the business logic, defining the necessary Key Performance Indicators (KPIs) for a specific service, such as throughput and latency, as part of the Service Level Agreement (SLA). The control logic is responsible for fulfilling these SLAs by creating appropriate Network Slices in RAN UPs. Depending on the chosen topology (as seen in Figure 3.1.1.1), the control logic can be placed within the App, the RAN controller, or directly in the RAN-Node.

- **Centralized:** Apps handle business logic and control logic. As shown in Figure 3, App-I abstracts the expected throughput of Service-1 into control logic, e.g., PRB number, in a comprehensible manner to the radio resource scheduler at RAN-Node A.
- **Decentralized:** Apps define business logic and use service APIs exposed by the RAN controller. In Figure 3.1.1.1, App-II and App-III request the expected throughput for Services 2 and 3, respectively, and Service 2 asks for extra expected latency for its two sub-services (2a and 2b). The RAN controller abstracts all these requests into the control logic and uses RAN APIs toward RAN-Node B.
- **Distributed:** RAN-Nodes have RAN UPs and control logic, allowing shorter control loops in real-time. Consider App-III and App-IV in Figure 3.1.1.2: RAN-Node C embeds the abstracted control logic from the expected latency for Services 4 and 5 directly into the radio resource scheduler.

In Figure 3.1.1.1, the abstract control logic simplifies RAN UP functions and Service Models (SMs) across multiple dimensions, across various vendors and Radio Access Technologies (RAT) RAN-Nodes. This abstraction layer customizes control logic based on requested Key Performance Indicators (KPIs), a crucial aspect of FlexSlice.

Inside RAN UPs (Figure 3.1.1.2), scheduling functions in RAN-Nodes A, B, and C handle Services 1, 2 and 3, and 4 and 5, respectively. These functions are realized by employing the recursive approach (denoted as SCH in Figure 3.1.1.2) which allows for multi-level scheduling. At the slice level, radio resources are partitioned based on (sub-)slice information (e.g., SLA). These per-slice resources are then allocated to the associated UEs based on UE information, e.g., Quality of Service (QoS).

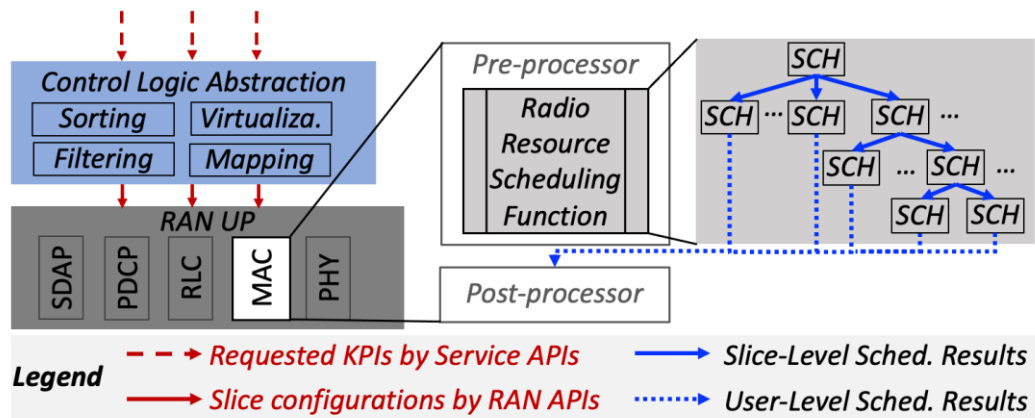


Figure 3.1.1.3: Control logic abstraction and Redesigned radio resource scheduler for recursive operation.

The aim of control logic abstraction is to allow network operators to define the characteristics of each slice in a flexible manner based on the KPIs of each service without disclosing specific deployment details (e.g., multi-vendor and multi-RAT at various sites). As depicted in Figure 3.1.1.3, the process of controlling RAN slices is done by the mix-and-match control logic abstraction between the requested KPIs for each service and the respective slice configuration (e.g., radio resource share), which includes setting up slices, and deciding the scheduling algorithm and parameters. Specifically, four tasks are included as shown in Figure 3.1.1.3:

1. Sorting: Arrange services based on their requested KPIs or non-technical criteria
2. Virtualization: Virtualize radio resources of RAN-Nodes with tailored approach (e.g., PRB to vRB)
3. Filtering: Adjust the requested KPIs in accordance with the (virtualized) resources of RAN-Nodes
4. Mapping: Create slice configurations by mapping the requested KPIs to use respective RAN functions (e.g., scheduling algorithm and parameters)

RAN function redesign is key to FlexSlice, and we focus on the radio resource scheduler. First, it is divided into two parts in Figure 3.1.1.3: (a) pre-processor and (b) post-processor. The former incorporates the radio resource scheduling function and can be controlled as a RAN function, while the latter allocates resources to physical channels based on the results from the pre-processor. Then, the radio resource scheduling function is redesigned recursively (denoted as SCH in Figure 3.1.1.3) to dynamically apply different algorithms and parameters based on performance requirements and scheduling constraints, effectively realizing both slice- and user-level scheduling. By doing so, the original scheduling problem can be decomposed, further increasing flexibility and extensibility.


```

1: procedure SCH( $l, r, \mathcal{N}$ )
2:   if  $r > 0$  then
3:     if  $l = \text{slice}$  then
4:        $i \leftarrow \text{SelectSlice}(\mathbf{C}, \mathcal{N})$ 
5:        $\mathbf{S}_i[0] \leftarrow \min(r, \text{AllocRB}(r, \mathbf{C}[i], 0))$ 
6:        $r \leftarrow r - \mathbf{S}_i[0]$ 
7:        $\mathcal{N} \leftarrow \mathcal{N} \setminus \{c_i\}$ 
8:       if  $\text{length}(\mathbf{S}_i) > 1$  then
9:          $\text{SCH}(\text{user}, \mathbf{S}_i[0], \mathcal{N})$ 
10:      end if
11:      if  $|\mathcal{N}| > 0$  then
12:         $\text{SCH}(\text{slice}, r, \mathcal{N})$ 
13:      end if
14:     else
15:        $j \leftarrow 1$ 
16:       while  $r > 0$  and  $j < \text{length}(\mathbf{S}_i)$  do
17:          $\mathbf{S}_i[j] \leftarrow \min(r, \text{AllocRB}(r, \mathbf{C}[i], j))$ 
18:          $r \leftarrow r - \mathbf{S}_i[j]$ 
19:          $j \leftarrow j + 1$ 
20:       end while
21:     end if
22:   end if
23: end procedure

```

Figure 3.1.1.4: Algorithm of recursive radio resource scheduling

Practically, this recursive approach is provided in details in Figure 5, and we first define two sets: $\mathcal{C} := \{c_1, \dots, c_{|\mathcal{C}|}\}$ contains all slices to be scheduled, and $U_i := \{u_1, \dots, u_{|U_i|}\}, \forall c_i \in \mathcal{C}$ includes all associated UEs with the i^{th} slice. Then, two global variables are formed in Figure 3.4: \mathbf{C} is a vector of size $|\mathcal{C}|$ comprising all slice configurations, and \mathbf{S}_i is a vector of size $|U_i| + 1$ including the number of scheduled resources for the i^{th} slice (i.e., $\mathbf{S}_i[0]$ as the first element of \mathbf{S}_i) and for its associated UEs (i.e., $\mathbf{S}_i[j], \forall j \in [1, |U_i|]$). Finally, $\mathbf{S}_i, \forall c_i \in \mathcal{C}$ will be passed to the post-processor as the scheduling results.

Then, three inputs are defined: l denotes the scheduling level (i.e., equal to *slice* or *user*), r is the number of available RBs, and set \mathcal{N} contains the unscheduled slices. To start the recursive operation, the following arguments are used: $l \leftarrow \text{slice}$, $\mathcal{N} \leftarrow \mathcal{C}$, and $r \leftarrow R$ where R is the maximum number of RBs can be scheduled. At the slice-level from line 4 in Figure 3.4, we first select an unscheduled slice using the $\text{SelectSlice}(\cdot)$ function, allocate RBs to this slice using the $\text{AllocRB}(\cdot)$ function, recursively call SCH to assign RBs to its associated UEs (see line 9), and move to the next unscheduled slice (see line 12). At the user-level from line 15, every associated UE is looped through to schedule all slice-level resources, i.e., $\mathbf{S}_i[0]$.

3.1.2 Directional RAN slicing

In this subsection, we discuss how this O-RAN-compliant slicing framework can be extended to consider UE positions, i.e., Directional Slicing. The key research question is that: “If we further divide a slice into sub-slices base on power-related metrics, can we achieve higher QoS at smaller resource usage?”

Description: We consider a centralized xApp that gets RSRP stats from the PHY service model and executes weight-based resource allocation for ranges of RSRPs:

- **Excellent signal:** -40 dBm to -80 dBm, each UE get 1 resource point.
- **Good signal:** -81 dBm to -100 dBm, each UE get 2 resource points.
- **Poor signal:** -101 dBm to -120 dBm, each UE get 4 resource points.
- **Unusable:** -121 dBm to -140 dBm, each UE get 1 resource point to keep the connection.

Then, using RSRP distribution of UEs, we compute the total number of points, i.e., P points, each of which is valued at R/P vRBs, where R is the current slice budget. The mapping from RSRPs to slice configuration is then completed.

We illustrate this Directional Slicing concept and its control loop in Figure 3.1.2.1.

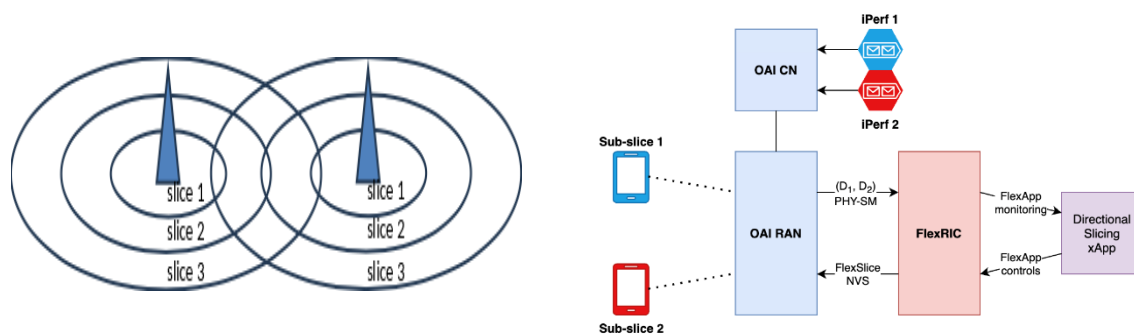


Figure 3.1.2.1: Directional Slicing is executed by grouping UEs on distance-related metrics.

Our experimental results, conducted on top of OpenAirInterface and FlexRIC, using channel simulation results of NS-3 will be discussed in *Subsection 5.3.2*.

3.1.3 AI-based RAN slicing

In this subsection, we first recall the motivation, problem formulation, and algorithm design of our Industrial RAN Slicing use case, condensing *Deliverable 5.2's subsection 3.2*. Then, we describe the key elements of our digital-twin and multi-agent Reinforcement Learning (RL) solution, towards bringing upfront (i.e., before deployment) QoS guarantees for each UE in a target factory.

Motivation: We address the needs to be highly modular and re-configurable of modern manufacturing lines in order to serve smaller and more customized manufacturing batches. The goal is to make a target factory wireless (versus wired) to enable the (re-)assembling of mobile machinery at minimal rewiring cost [1].

Problem Formulation: The key challenge is to connect as many UEs as possible while ensuring predictable QoS for each UE, despite varied requirements and stochastic workloads. We formulated this challenge as *predictable RAN resource optimization per-UE* [2][3], solved with RL to avoid explicitly modelling the workloads [4].

Algorithm Design: Since our RL agents are trained per-UE but deployed in a scarce resource environment, we extend standard RL algorithms to *protect each agent's trained optimal trajectory from in-production resource competition*. This is done by 2 techniques:

1. Per-episode resource budget for each UE, which constrains their per-transition resource usage on average and, hence, limits the scale of resource conflicts.
2. A policy search algorithm with derived trajectories robust against a chosen resource “cutting” scheme.

More details about this algorithm can be found in *Deliverable 5.2's subsection 3.2*, here we briefly describe its mathematical notations.

$$\pi_{\mathcal{A}}^* = \operatorname{argmax}_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^T \text{QoS}(s_t, f(a_t)) \right]$$

$$\text{s.t. } \sum_{t=0}^T \text{cost}(f(a_t)) \leq B.$$

Where B , $f(a)$, and \mathcal{A} are per-UE resource budget, system resource cutting scheme, and per-UE policy-search, respectively.

Scalable and Near-discrete-time Digital Twin: As Deep RL is sensitive to modelling errors [5], predictable QoS requires accurate emulation of in-production traffic, channel, and RAN-stack dynamics, i.e., real-RAN-software digital twins. However, training on the real RAN is restrictive due to scalability (e.g., multiple UEs with mobility) and non-discrete-time transitions [6]. To solve this, we propose Twin5G, a novel training methodology that can correctly executes RL on the real RAN for scenario of *any* scale. Twin5G consists of 2 key ideas:

1. Instead of training with multiple UEs connected, we separately train for each target UE (i.e., the UE we want to optimize) by emulating the interference other UEs may cause to its in-production traffic and channel dynamics.
2. To achieve discrete-time transition, one can make the total control-loop time negligible in comparison to the control interval [6]. Since the minimum control interval of the near-RT RIC is 10ms[7], we target a total control-loop time of sub-2ms.

These ideas are illustrated in Figure 3.1.3.1, which visually describes the 4 techniques used to realize them.

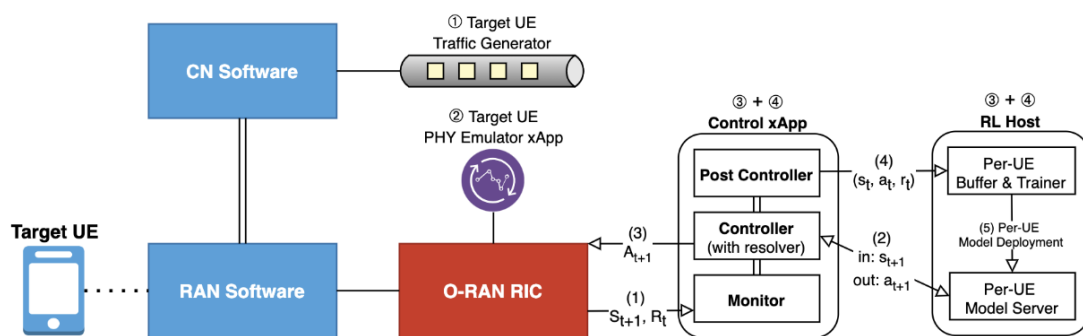


Figure 3.1.3.1: The four techniques employed to train an RL agent that optimizes the QoS of a target UE on real RAN software

Techniques ONE and TWO are used to emulate a target UE’s in-production traffic and channel dynamics. With RAN remains the only bottleneck in factory scenarios, traffic emulation can be

done independent of other UEs, i.e., no interferences. For PHY-layer interferences, the relevant multi-UE ray-traced propagation model is on-board to get and set relevant PHY variables, emulating the radio link as if it is inside the real factory.

Techniques THREE and FOUR are used to reduce the total control-loop time. While THREE bypasses OpenAI's Gym interface to decouple the background training loop from the foreground control loop, FOUR redesigns model serving to be latency-optimized for Python instead of throughput-optimized like current state-of-the-arts. The former is achieved by introducing a *Post Controller* and *multi-replica Model Servers* that allows our *Controller* to not wait for the training process. The latter is achieved through 5 design decisions, illustrated in Figure 3.1.3.2:

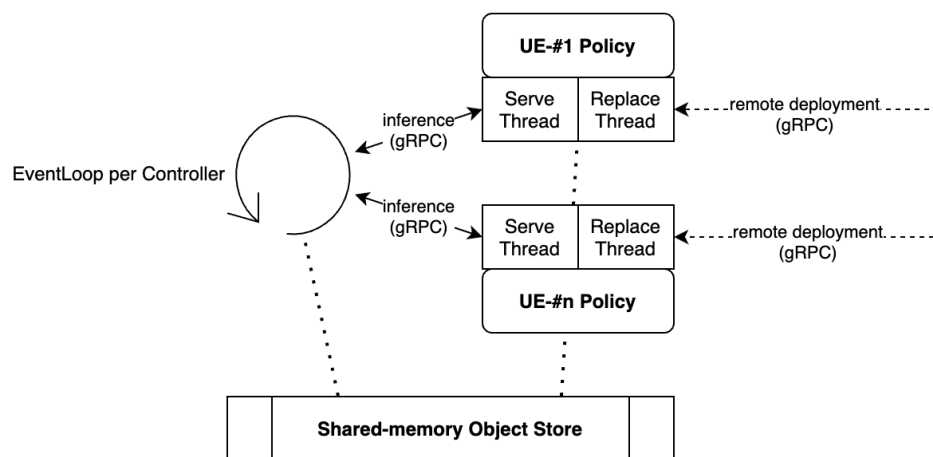


Figure 3.1.3.2: Latency-optimized Model Serving and Replacement

Firstly, we adopt the model server approach, instead of disk loading, to allow remote deployment and faster replacement time. Secondly, we make *Controllers* event loops and *Model Servers* having “replace” threads so that control inference time is stable regardless of the number of UEs. Third- and fourthly, control inference is done over gRPC on object IDs, instead of REST and serialized objects, to further reduce latency at no extra cost of Protobuf definitions. Finally, control inference has a strict deadline for return, beyond which it returns default replies to avoid lock-based queuing that may incurs latency.

In conclusion, these 4 techniques help us build real-RAN-based digital twins that can correctly execute RL for a variety of industrial 5G scenarios. Our experimental results will be discussed in *Subsection 5.3.2*.

3.1.4 Li-Fi Network Slicing

To further expand RAN slicing to different types of networks and contribute to an Integrated Access and Backhaul paradigm, 6G BRAINS has created a hybrid architecture between 5G backhaul and Li-Fi access technologies to introduce Network Slicing over this architecture. A new design has been developed to bring these two networks together and to apply Network Slicing on both: 5G and Li-Fi. The choice of Li-Fi as an extension of the 5G network is due to the following advantages that this technology brings:

- Li-Fi uses a light spectrum instead of radio frequency, and this fact makes communications unaffected by other wireless technologies such as Wi-Fi, 4G or 5G among others.
- Furthermore, the transmission of information becomes completely private, secure, and un-hackable from outside walls thanks to the use of this new range of frequencies. Using a light spectrum, the transmission can only be conducted if there is direct line of sight between the transmitter and receiver. Anyone who wants to obtain the information exchanged must be in the same room as the Li-Fi endpoints.
- In addition, Li-Fi is a technology considered to be ready for 6G, and thus 6G BRAINS has investigated to explore the potential of implementing Network Slicing over Li-Fi, which aims to ensure high-quality communication services for prioritized applications.

Figure 3.1.4.1 shows how this hybrid architecture look like in an industrial environment, where both networks are used. In this case, Li-Fi Network Slicing is designed on top of the advantages it offers as listed above. To create this hybrid network and therefore, be able to add the Network Slicing methodology to the Li-Fi technology, the Oledcomm Li-Fi hardware has been employed. In this architecture there are two segments. The first segment is where 5G network is deployed. In summary, it is composed by 5G RAN, 5G edges and the 5G core with their specific network virtual functions. The second segment consists of the Li-Fi network. To create this network, a 5G-UE is used as the Li-Fi access point. Therefore, this device acts as the gateway between Li-Fi and 5G. With this configuration, the Li-Fi network uses the 5G resources to be able to provide service to the endpoints.

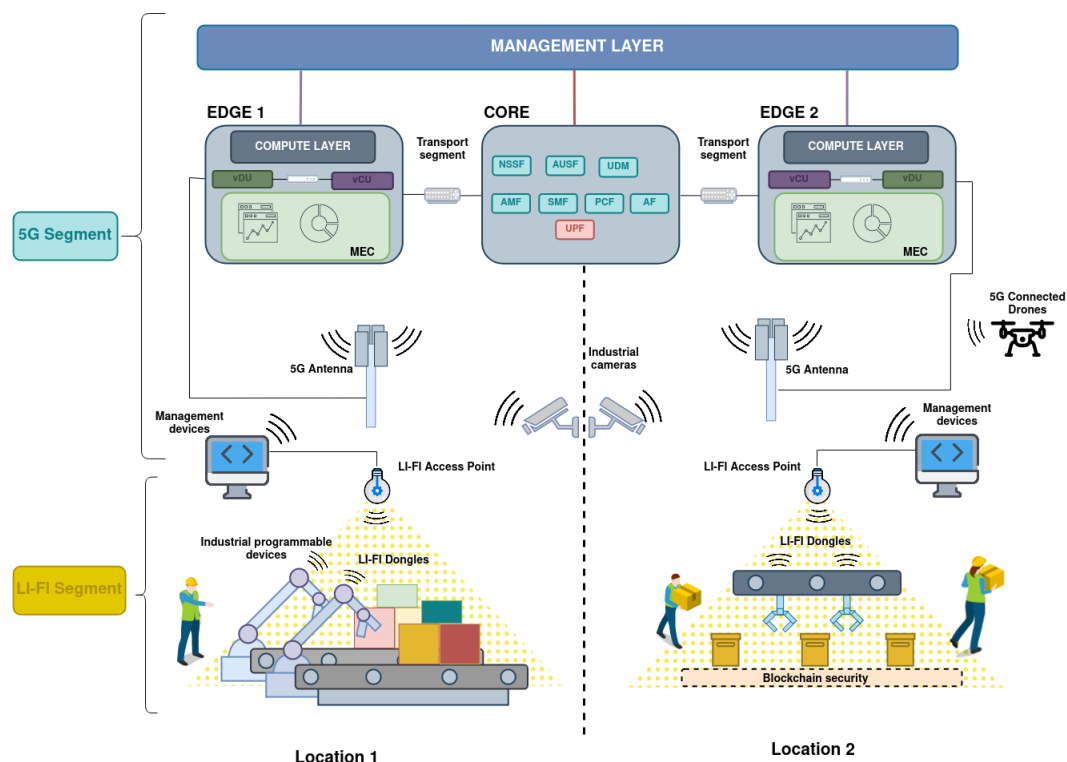


Figure 3.1.4.1: Overview of a 5G-Li-Fi hybrid architecture

Having the scenario configured, we have designed and developed an architecture that allows the enforcement of Network Slicing. To do this, as shown in Figure 3.1.4.2, the Slicing Control

Agent (SCA) component has been extended to provide slicing support over the Li-Fi technology. After that, SCA has been installed in the devices connected to the Li-Fi network. The SCA performs the control of the Data Plane Abstraction Service in which the different network technologies (agents) are controlled. In this scenario, the technology used is called Li-Fi agent. This technology is provided by the Li-Fi devices (dongles and access point).

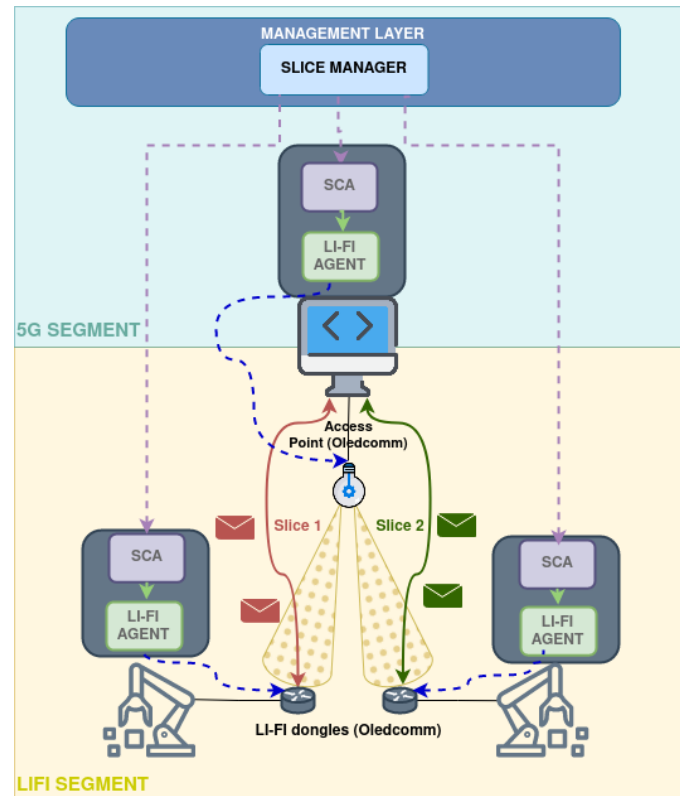


Figure 3.1.4.2: Li-Fi based Network Slice architecture

The SCA allows the control of the Li-Fi agent, enabling the assurance of the QoS over the Li-Fi transmission of data. The control can be implemented at two different levels. The first level is achieved by changing bandwidth and the second one is varying the latency of the transmission. Figure 3.1.4.3 explains this control over the Li-Fi agent.

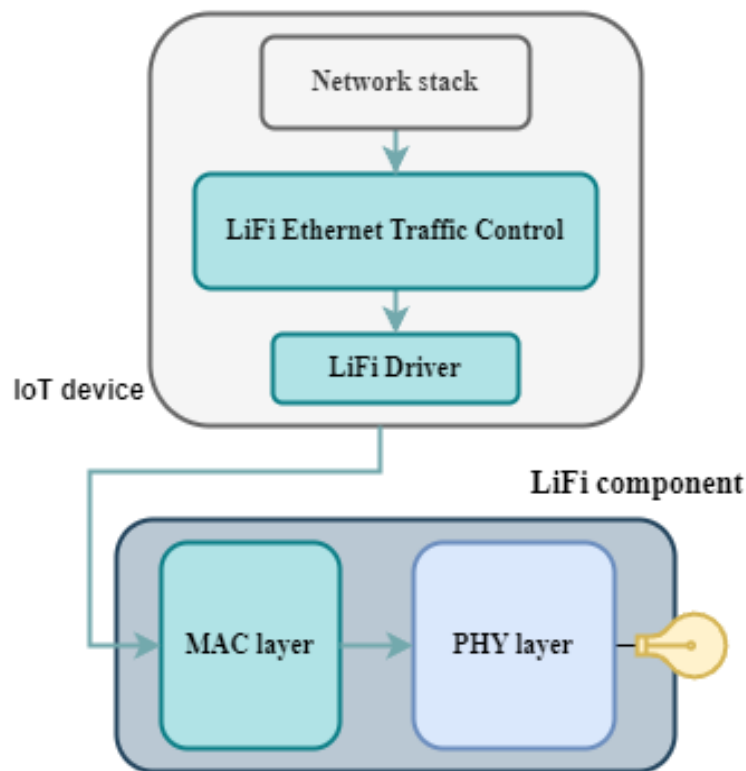


Figure 3.1.4.3: OSI model of the Li-Fi device and Li-Fi modem

The Oledcomm Li-Fi Max devices used consists of 2 OSI layers. The physical layer related to the physical connections and is responsible of the data transmission itself. The second layer is the Medium Access Control (MAC) and its responsibility is to control the hardware that interacts with the physical layer. This layer can be found in Li-Fi components and in the host device (computer, mobile phone, industrial IoT, etc).

In the Li-Fi component, the MAC layer is in charge of enforcing the Network Slicing of the downlink. Since the components that provide the Network Slicing capabilities are installed in all the Li-Fi components, the SCA can act on each Li-Fi component and enforce the QoS traffic control in both senses: uplink and downlink. With this architecture, it is possible to create and maintain a dedicated QoS for each Li-Fi user device by creating slices in the Li-Fi network. Using Li-Fi as an extension of the 5G network ensures improved telecommunications performance isolation and security without losing the Network Slicing functionality that 5G brings.

3.2 Hybrid wired segment slicing

Regarding the wired segments of the infrastructure, the 6G BRAINS project foresees a diverse range of hybrid slice enablers, including both hardware and software. These enablers facilitate the programming of specific points within the infrastructure data-plane to implement distinct Quality of Service (QoS) requirements at each point. These requirements, determined by the slice, aim to guarantee that all network traffic corresponding to the slice criteria (such a specific service), satisfies network performance requirements throughout its end-to-end slice journey. In order to enable flexible slice instantiations that cater to specific network and system requirements, it is absolutely critical to explore, evaluate, and contrast the different methodologies. Furthermore, it is equally important to discover new opportunities in the latest techniques and technologies for programming the data plane.

The subsequent subsections provide an overview of the progress, advancements, and new discoveries made in the 6G BRAINS project with regards hybrid wired segment slicing. These developments not only enhance performance but also introduce new capabilities related to the analysis and programming of the data-plane.

3.2.1 XDP Hardware-based SmartNIC slicing enablers

In Deliverable D5.2 (Section 4.1), 6G BRAINS provided an innovative, hardware-based slice enabler, utilizing Extended Berkeley Packet Filter (eBPF) and XDP. This new enabler was aimed to address certain limitations associated with the previous hardware-based enabler discussed in Deliverable D5.1, which was reliant on the NetFPGA platform. The architecture, design, and functional aspects of this XDP-based technology are comprehensively detailed in Deliverable D5.2, alongside a comparative analysis of the XDP-based and NetFPGA-based slice enablers. This comparison highlighted the superior performance of the XDP-based enabler, demonstrating enhanced delay performances, jitter, and packet loss metrics, in comparison to the NetFPGA-based counterpart. Moreover, the XDP technology delivers additional advantages, including heightened flexibility - facilitating on-demand, real-time network card programming. Its broad compatibility is also noteworthy, with a greater number of NIC vendors supporting the architecture.

While the advantages of offloading customised eBPF programs directly into the network card, and employing the in-hardware XDP architecture for Network Slicing are noteworthy, the UWS research team uncovered several limitations that necessitate further investigation. In scenarios where network traffic routes through a point in the data-plane that is designed to forward the traffic (acting as a switch or router) to another network device element, without requiring the traffic to reach the user space, the hardware offloading approach performed admirably, delivering exceptional network performance. However, the picture changed when network traffic associated with a slice needed to reach a service or an application. In these scenarios, the traffic had to traverse the Linux network stack to reach an application in the user space, and this process significantly reduced network performance. The Linux network subsystem still has the responsibility of forwarding the packet to its ultimate destination. When incoming packets enter from the NIC and arrive to the kernel network stack, it

undergoes various processing stages, including protocol processing and socket buffering. A potential bottleneck can arise during the socket buffering phase. This is where packets are temporarily stored in buffers before the application processes them. If buffer sizes are incorrectly configured, or if the application struggles to process packets quickly enough, the result can be packet loss and diminished network performance. This issue is evident in one of the 6G Brains research works [25], where to overcome that, network traffic had to be rerouted to other servers for processing, as the system's network stack was incapable of handling high-rate packet processing.

This challenge highlights the necessity for sophisticated solutions that can sustain high performance even when user-space interactions are required. To address this issue, the UWS has developed and implemented an in-driver solution that bypasses the kernel, enabling more efficient network traffic management. In this system, network traffic first reaches the NIC, where it is classified and processed at the hardware level, and after that it is sent to the driver level. If the traffic matches the slice definition at this stage, it is directly dispatched to the user-space application or service. This innovative approach combines both XDP techniques - hardware offloading and in-driver enabling - leading to a much more efficient management of network traffic and avoids the time-consuming networking steps of the Operating System, thus preventing potential network performance degradation caused by bottlenecks.

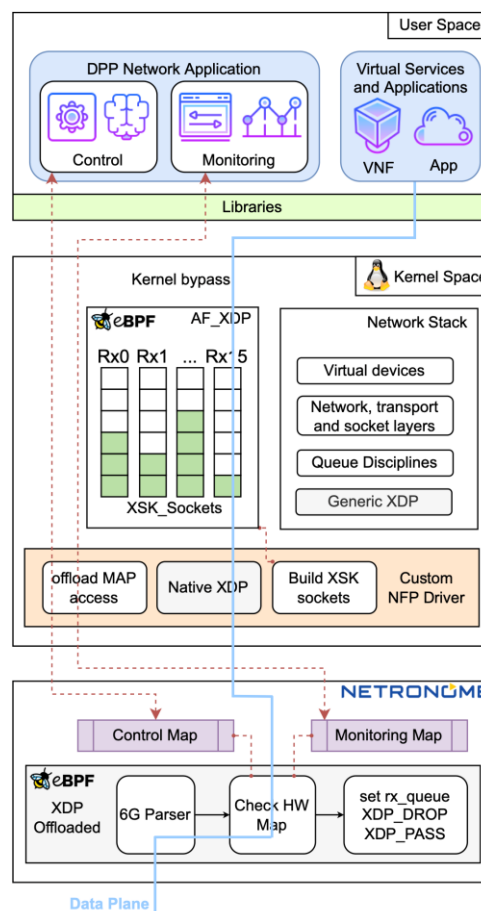


Figure 3.2.1.1: Data-path of the XDP-based slice enabler with kernel bypass

The diagram in Figure 3.2.1.1 provides a visual representation of the data-path (illustrated with a blue line) for incoming network traffic that targets a user-space service. The bottom of the figure features a SmartNIC, equipped with a custom XDP offloaded firmware and two distinct eBPF hardware maps for control and monitoring. Each rule in the control map includes a 32-bit key and value. These are split into two 16-bit segments, one representing the action and the other indicating the action's output value if needed. For instance, it may be set to a particular *rx_queue*. Each rule in the monitoring map also contains a 32-bit key, but a 64-bit value. The latter is divided into two 32-bit parts: one for packet length, and the other for the count of packets that fit a specific rule. The XDP offloaded firmware, shown in grey, comprises three components: the 6G parser, a logic module, and an output decision maker. The 6G parser processes network flows and extracts necessary meta-data, which is then used to calculate a hash identifier for the matching module. This module compares the hash identifier with those in the eBPF control map to decide the network packet's destination. The packet can either be assigned to a specific queue, discarded, or passed on to the kernel network stack via the default queue. When network packets arrive at the kernel space, they are processed by a driver containing a custom version of the NFP driver, which facilitates socket forwarding. This driver utilizes AF_XDP [26] and XSK sockets to expedite communication between the SmartNIC and user space, bypassing the network stack. This solution supports as many sockets as hardware queues the network card provides and allows access to the hardware control and monitoring maps. It also offers a versatile data path with Native XDP support.

This driver customization enabled 6G BRAINS to enhance performance and efficiency, outperforming more generic drivers. Consequently, the data plane could satisfy the rigorous demands, in terms of network performance, of 6G Network Slices.

3.2.2 Topology monitoring based on extension to IEEE 802.1ab

The Topology Inventory Agent (TIA) is an essential component of the 6G BRAINS framework described in Section 7.3.1 of Deliverable 5.2. TIA gathers information about the network topology and reports it periodically so that the E2E Slice Manager can control, manage, and orchestrate Network Slices in real time. To do so, TIA uses different Linux tools that access the information using different protocols. One of those protocols is the IEEE 802.1AB also known as Link Layer Discovery Protocol (LLDP). TIA uses this protocol to discover neighbour computers as well as other neighbour managed devices where TIA cannot be deployed (e.g., managed switches, managed routers). Also, if LLDP is enabled in virtual computers and containers, the physical host of those devices could use LLDP to discover them as they would answer to the LLDP request.

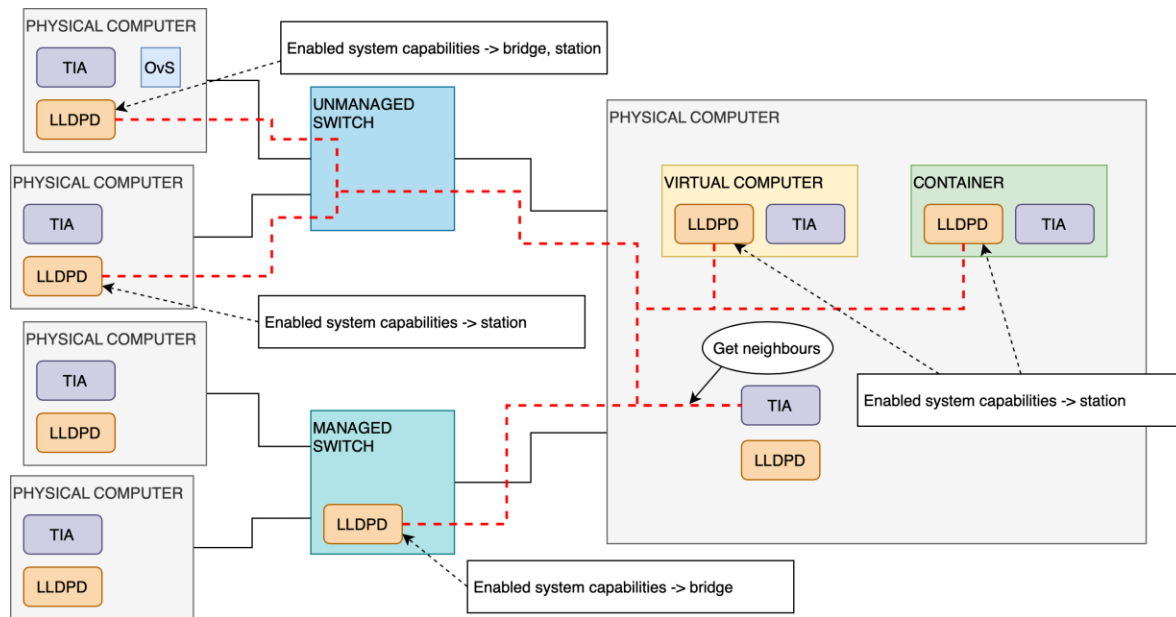


Figure 3.2.2.1: Topology example for LLDP neighbour discovery from TIA

Figure 3.2.2.1 depicts an example of a topology with different examples of connections and devices. Some of the physical computers are connected to a managed switch and some others are connected to an unmanaged switch. In this case, the difference between the managed and the unmanaged switch is that managed switch has a LLDP daemon running. TIA is deployed in the physical computers, virtual computers, and containers.

The red line in Figure 3.2.2.1 represents the neighbours that the TIA instance deployed in the physical computer on the right (big grey box) is able to reach. When TIA sends a message to request information about neighbours, those neighbours answer with information about their chassis and some other properties such as the interface where they are connected or the system capabilities. The available system capabilities in the LLDP protocol are: other, repeater, bridge, wlan, router, telephone, docsis and station. TIA uses those capabilities to deduce the type of device discovered. For example, if it contains the "bridge" capability enabled, TIA considers it a switch; if it contains the "station" capability enabled, it will be considered a computer; etc.

The reason to make this extension is as follows. TIA uses the LLDP daemon implementation for Linux. That implementation uses the existing networking configuration of the device to automatically set the LLDP system capabilities. Then, if a computer has a Linux bridge or any other kind of software-based switching technology, the LLDP daemon enables the "bridge" capability. This configuration would be useful in many use cases. However, when LLDP daemon enables the "bridge" capability, it also disables the "station" capability. This means that TIA will recognise that neighbour as a switch.

The extension provided by UWS solves this issue (from LLDPD version 1.0.15) by providing a new method in the LLDP client. This method provides a new functionality that allows the administrator to override the automatically settled capabilities. It also allows to provide more than one capability. Then, the physical computer with bridge capabilities could be reported with both "bridge" and "station" capabilities enabled.

3.2.3 Process slicing

Process slicing is a concept and methodology used in the management of processes within complex systems, primarily in virtualized or softwarized networking environments in 5G and 6G. Process slicing involves the isolation of specific resources, such as CPU and Input/Output (I/O) capabilities, to provide strict Quality of Service (QoS) guarantees and Service Level Agreement (SLA) capabilities for each individual process/thread running within a given system. This approach is motivated by the need to address the challenges presented by the increasingly complex computing scenarios in factories of the future, and to optimize resource utilization, security, and performance required in Industry 4.0/5.0 and beyond use cases in the 6G era, as envisioned in the 6G BRAINS project.

The creation of this innovative term has emerged as a response to the evolving field of computing. Traditional computing environments can struggle to efficiently manage and secure multiple concurrent processes with varying resource requirements. This is important since not all the processes need to be treated the same way, being especially pertinent with the rise of virtualization. It is crucial to ensure that each of the processes running is given the needed quantity of resources based on their priority.

Process slicing serves several important purposes:

- **Resource Optimization:** It ensures efficient utilization of system resources by allocating them based on predefined rules and priorities, preventing resource contention, and bottlenecks that can negatively impact performance.
- **Quality of Service:** By offering strict QoS guarantees, process slicing enables organizations to meet the performance and reliability expectations of their clients and users. It provides assurances that critical processes receive the resources they require.
- **Security:** Through fine-grained control, process slicing enhances system security. It allows administrators to isolate and contain potentially harmful or unauthorized processes, thereby protecting the system's integrity.
- **Predictive Maintenance:** The use of forecasting in process slicing allows for proactive measures to prevent or mitigate potential issues, ensuring the system's continuous operation and minimizing disruptions.

Our proposed architecture is composed of a set of agents that perform the necessary steps to achieve complete management of processes. These agents are Process Inventory Agent (PIA), Process Monitoring Agent (PMA), Process Forecasting Agent (PFA), Process Controlling Agent (PCA), System Orchestrator (SO).

PIA is responsible for collecting information about the whole running processes over the virtualized component. This agent inventories the processes, and maintains the architecture informed of the currently running processes.

PMA is responsible for monitoring the industrial processes in real time. It receives the processes inventoried by PIA and then gathers and calculates metrics. This agent ensures the security and reliability of virtualized processes by keeping track of the metrics of the processes.

PFA is responsible for predicting metrics one step ahead of the last metric received by PMA. PFA applies statistical models to carry out the forecasting task. By predicting an approximation of the following metrics, this agent enables the process management layer to take proactive measures to prevent or mitigate any potential harm.

PCA is responsible for executing intents or sets of intents meticulously created by the Security Orchestrator.

The SO receives information from PIA, PMA, and PFA and is responsible for detecting any anomalies in the industrial process. When an anomaly is detected, the SO prepares a set of intents to be executed by PCA to mitigate security risks and prevent further anomalies.

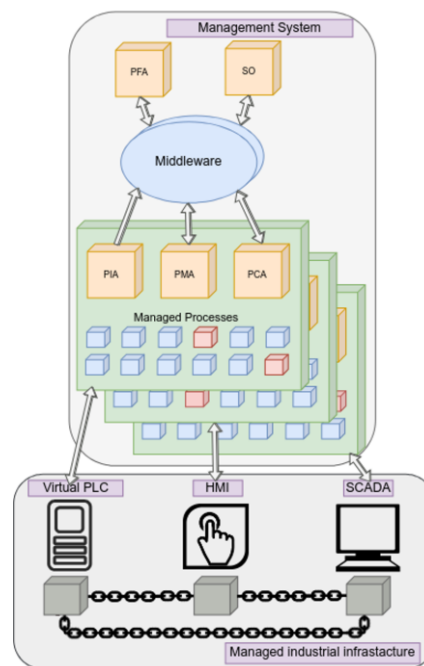


Figure 3.2.3.1: Process slicing architecture

3.3 E2E Network Slice MANO

3.3.1 ONAP-based Network Slice MANO

The role of ONAP ETSI NFV-MANO¹ in 6G-Brains is to provide the management interface for deployed 5G network services and maintain the underline infrastructure. For example, when a Network Slice is to be instantiated, the E2E SO and E2E slice manager collaborate based on intent-based network MANO in allocating the resources based on the available resources. Each Network Slice subnet is composed of one or multiple VNFs, and these VNFs are connected by Virtual Links (VLs). ONAP interfaces with OSS/BSS (northbound) or, for this scenario, using the IVA and with multiple cloud/on-prem infrastructure (OpenStack, Azure, k8s). Allows the design, creation, orchestration, monitoring and life-cycle management of VNFs and network services. When the SO receives the service order, it is decomposed into the network requirements and request management operations such as create, update, delete to ETSI NFV and MEC services.

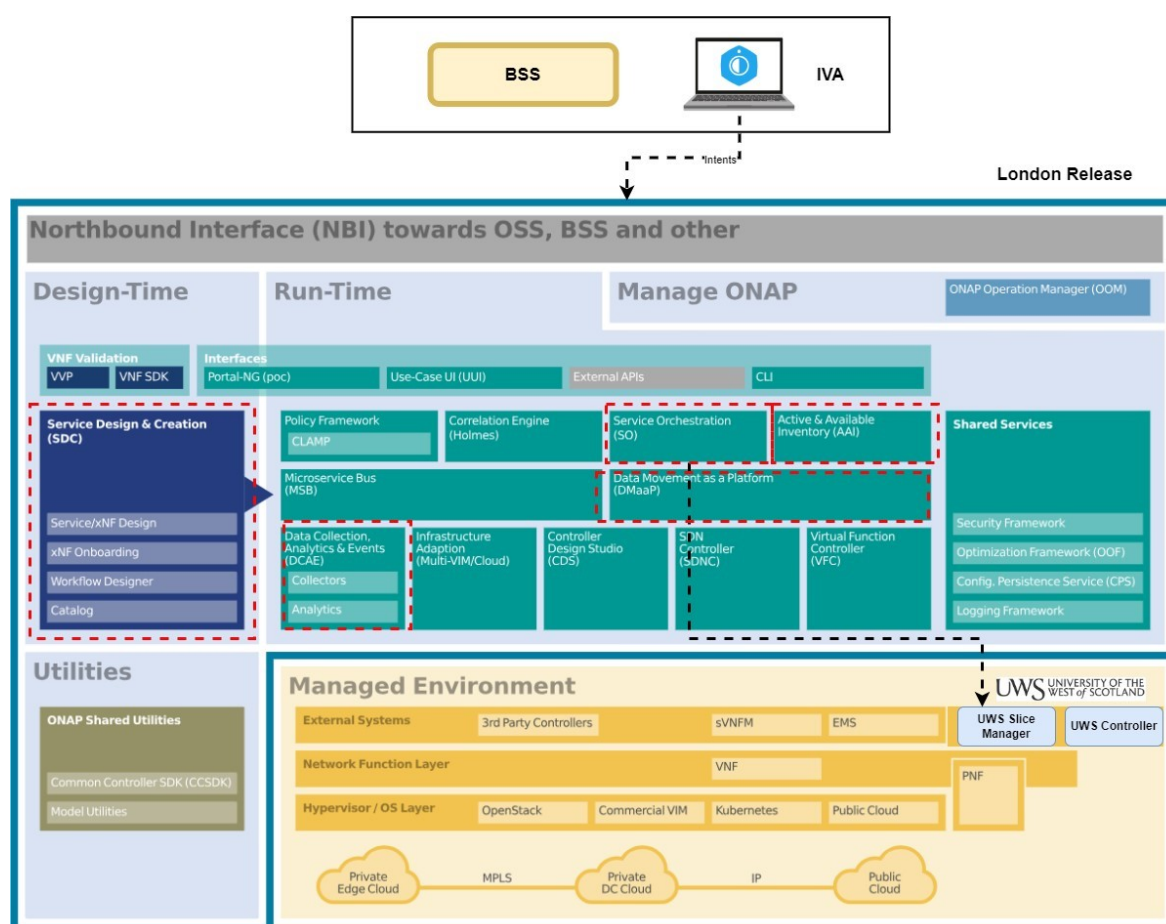


Figure 3.3.1.1: ONAP High Level Architecture Integration

Figure 3.3.1.1, shows the ONAP High Level Architecture, where we can highlight some components that were used in the developed solution. The framework can be divided into

¹ <https://www.onap.org/>

two main groups, the Design Time, and the Run Time. On Design time we can highlight the Service Design and Creation (SDC) component that allows the operator to on-board and design the services that compose the network. On the Run time we can find the controllers, inventory, adaptors and data collection components, that allow to perform the life-cycle of the network.

Starting by the service design, the operator can on-board and design the service according with their requirements. Those services will then be distributed to the run-time components to later perform the fulfilment of the service. From the catalogue, external systems can request the deployment, activation, modification and decommissioning of those services.

Also, from the architecture we can see that ONAP provides an external API allowing the interconnection with IVA and on Service Orchestrator some connectors were implemented allowing ONAP communicate with the external system for slice management implemented by UWS.

3.3.2 Autonomous and Intent-based Network Slice MANO

Autonomous and intent-based Network Slice Management is a feature or capability of ONAP that aims to provide autonomous and intent-driven management and orchestration of Network Slices.

Here is a breakdown of the key components:

- **Autonomous Management:** various network management tasks, such as configuration, monitoring, and optimization.
- **Intent-Based Management:** Intent-based networking (IBN) approach allows network operators to define high-level intents or policies for their networks, rather than dealing with low-level configurations. ONAP supports intent-based networking by enabling operators to specify the desired behaviour of Network Slices and then automatically configuring the underlying network elements to fulfil those intents.
- **Network Slice:** A Network Slice is a logically separate portion of a network infrastructure that is dedicated to a specific use case or service.

Network Slicing management goes through different phases:

- **Preparation** which includes the Network Slice design, Network Slice capacity planning, evaluation of the network functions and existing infrastructure, intent definition.
- **Commissioning** includes the creation of the Network Slice Instance (NSI), network service. During the NSI, the resources are allocated and configured to satisfy the Network Slice requirements.
- **Operation** phase includes the activation, supervision, performance reporting (network monitoring), resource planning, modification, and deactivation of an NSI.
- **Decommissioning** phase includes removing the NSI specific configuration and terminate all resources. After decommissioning phase, the NSI does not exist mode.

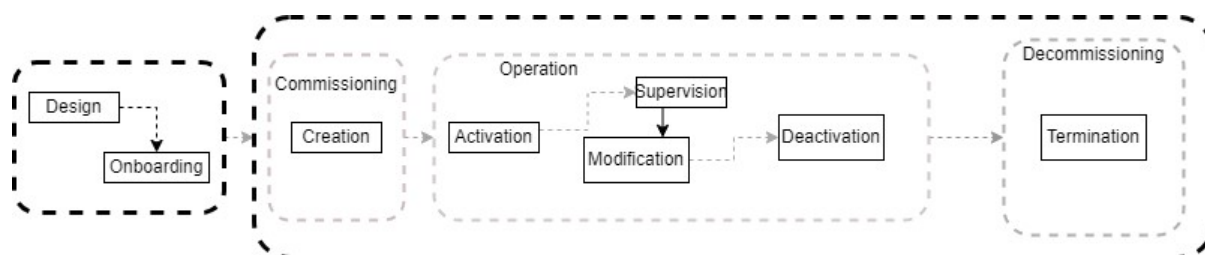


Figure 3.3.2.1: Network Slicing Phases

3.3.2.1 Network Slicing Design/Preparation Phase

Starting by the preparation phase, we can verify that ONAP already offers architectural options for Network Slicing for RAN and Transport. However, it also allows the creation and or customization of the existing architectural options. In context of 6G-BRAINS new templates were created in order to allow further integration with external slicing management system developed by UWS.

With this and using SDC new models were created where properties were assigned to the services allowing the operator to request slicing services with specific requirements in order to meet the desired QoS.

Figure 3.3.2.1.1, shows the SDC portal with the templates available to the operator as well as the properties assigned to the service, such as maxBandwidth, minBandwidth, priority, resources, and others.

Those templates are made available via API for external systems to request the instantiation of a new service.

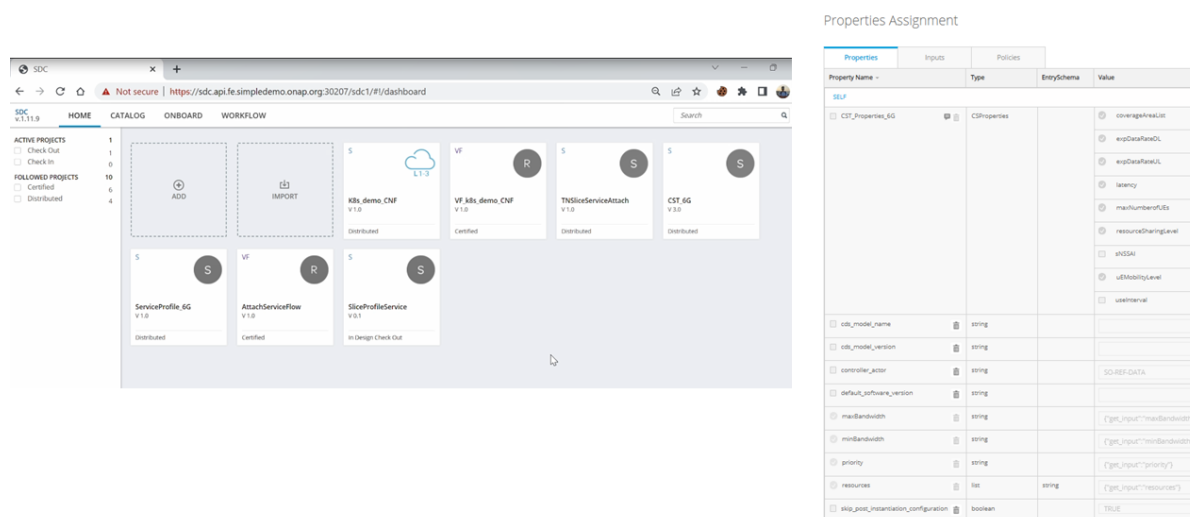


Figure 3.3.2.1.1: Service Design and Control for Network Slice Templates Modelling

<input type="checkbox"/> spProp	<input type="checkbox"/> SliceProfile	<input type="checkbox"/> activityFactor <input type="checkbox"/> areaTrafficCapDL <input type="checkbox"/> areaTrafficCapUL <input type="checkbox"/> cSAvailabilityTarget <input type="checkbox"/> cSReliabilityMeanTime <input type="checkbox"/> coverageAreaTAList <input type="checkbox"/> expDataRate <input checked="" type="checkbox"/> expDataRateDL <input checked="" type="checkbox"/> expDataRateUL <input checked="" type="checkbox"/> latency <input type="checkbox"/> maxNumberOfPDUSession <input checked="" type="checkbox"/> maxNumberOfUEs <input type="checkbox"/> msgSizeByte <input type="checkbox"/> overallUserDensity <input type="checkbox"/> pLMNidList <input type="checkbox"/> resourceSharingLevel <input checked="" type="checkbox"/> sNSSAI <input checked="" type="checkbox"/> sST <input type="checkbox"/> survivalTime <input type="checkbox"/> transferIntervalTarget <input type="checkbox"/> uEMobilityLevel
---------------------------------	---------------------------------------	--

Figure 3.3.2.1.2: Service Characteristics Template

Figure 3.3.2.1.2 we can observe the properties of the service profile model. Several characteristics are defined in the template, which can be populated by the requester during the service order request.

3.3.2.2 Network Topology Discovery/ Preparation Phase

Regarding the operation phase, different workflows were developed, for the distinct operations of the life cycle of the Network Slice. As requirement for the creation of a Network Slicing, ONAP must know the existing infrastructure (RAN, Core, and Transport). For that, a discovery mechanism was developed that allows to retrieve all the existing topology. The discovery mechanism will update the inventory with the current topology available in the infrastructure, creating the resources and their relationships. Based on the model retrieved using the thirty-party controller, the ONAP A&AI model was adapted to support the model.

Figure 3.3.2.2.1 shows the main relationships between the objects stored in inventory during the Topology Synchronization process. The model aggregates all objects under the entity customer, which owns the network topology. The customer can have different service instances. In this case, a service instance was created that relates to the topology and is

composed by multiple network resources, including VNF, PNFs, Switchs, Physical and logical interfaces and all logical links between the different resources.

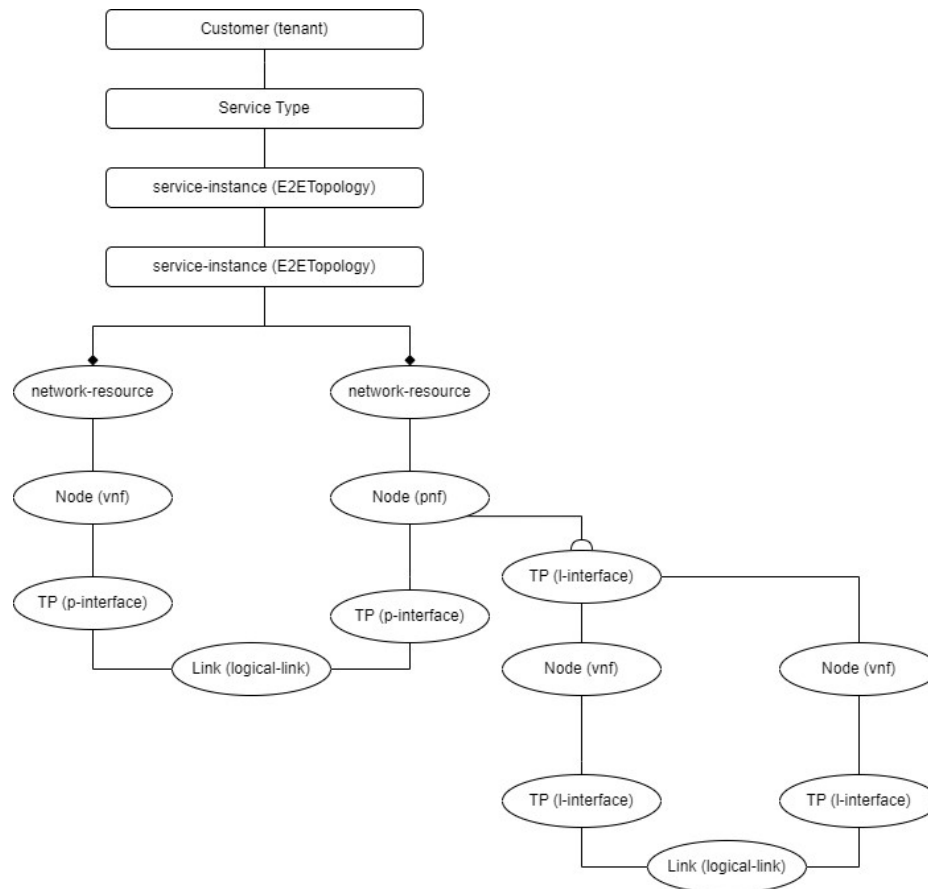


Figure 3.3.2.2.1: Network Service Topology Model Design

Figure 3.3.2.2.2 part of the developed workflow that has the responsibility for request to the external controller the topology and then process the collected data and update the inventory with most recent topology and their relationships.

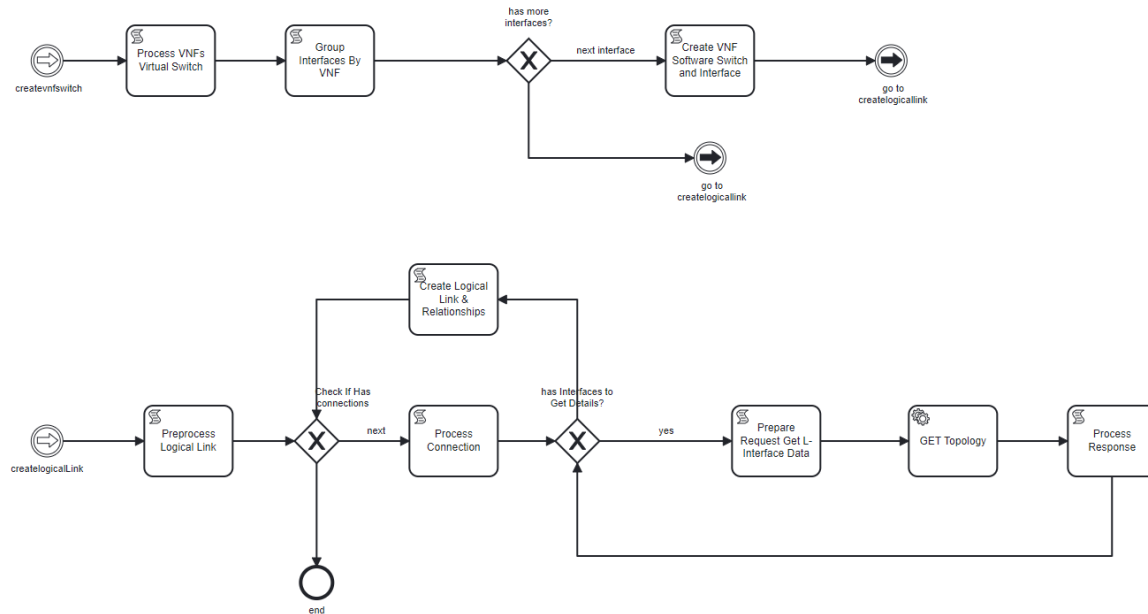


Figure 3.3.2.2.2: Part of Topology Discovery Workflow

On Figure 3.3.2.2.3, is shown in a graphical way the result from the topology discovery on A&AI. This shows the result of topology discovery mechanism where we can find out that a service instance is related with all discovered resources. Some of those resources are: generic virtual network functions (VNFs), physical network functions (PNFs), where these are related to other topology objects. During the topology discovery process, ONAP processes the response from the external controller where the different resources are mapped to the supported models.

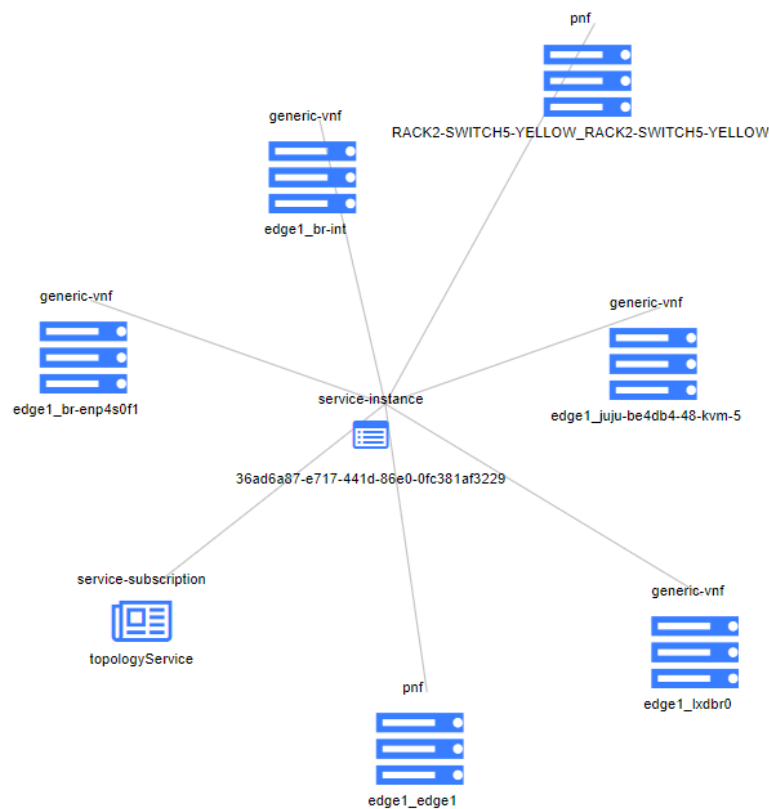


Figure 3.3.2.2.3: AAI view of Discovered Topology Result

3.3.2.3 Network Slicing Creation Phase

As explained previously the operation phase is composed by a set of phases. Next phase is the network slicing creation phase responsible for the creation of a logical network slice, composed by a set of network resources and its configurations.

The network slicing phase was designed in order to contemplate a well-defined data structure. On Figure 3.3.2.3.1 we can observe the different resources and services and their relationships. Starting by the communication service that represents the top communication service, where details about the customer and the type of service to be created are stored. The communication service is then composed of a service profile, that represents the slice service instance, which contains the network characteristics of the slice. Then a set of allotted resources are associated with the network slice, that make use of the resources discovered during the topology discovery phase, i.e., aggregates the interfaces that composes the network slice. Finally, the slice service profile contains a set of policies that defines the network flow.

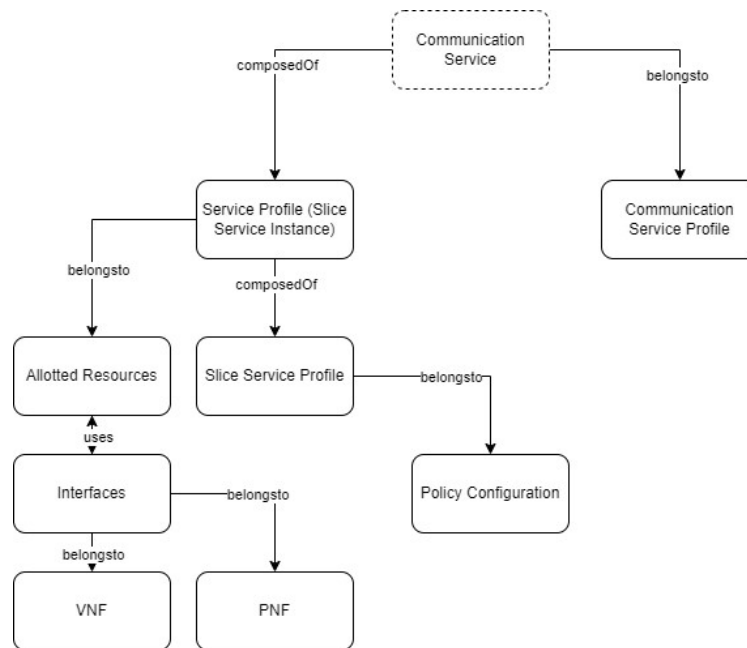


Figure 3.3.2.3.1: AAI model structure for slice creation

The first phase is the network slice creation: which results on an intent initiated by an external entity. In the current use case, the virtual voice assistant is integrated with ONAP, allowing an user to initiate the operations via voice commands. Figure 3.3.2.3.2 represents the BPMN work-flow containing the main steps realized by the ONAP service orchestrator for network slice creation. As it is possible to understand looking for the BPMN workflow, the process is composed by different operations such as the decomposition phase, the validation of the available resources into the inventory, the creation of the different services. The creation of the network service slice is realized by the NSMF, where we can see a call on the workflow.

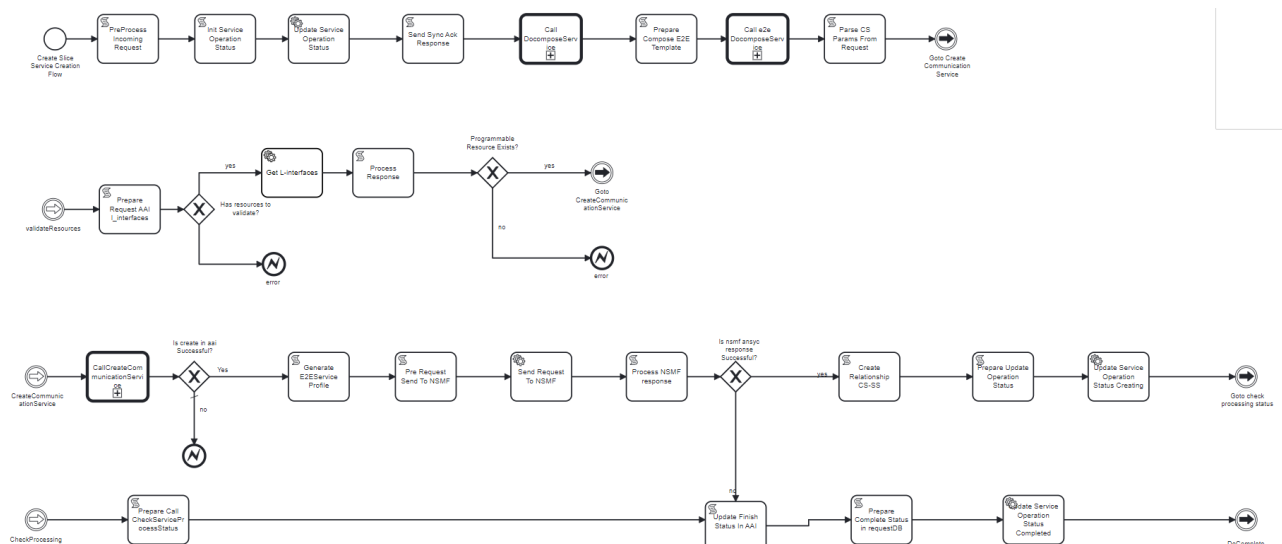


Figure 3.3.2.3.2: Network Slice Creation Workflow

On Figure 3.3.2.3.3. we can see the part of the BPMN logic to create the service slice via NSMF. The NSMF is called by the Network Slice Creation Workflow and is responsible for creating the

allotted resources, the slice service profile and to trigger the creation of the network slice using the external UWS slice manager.

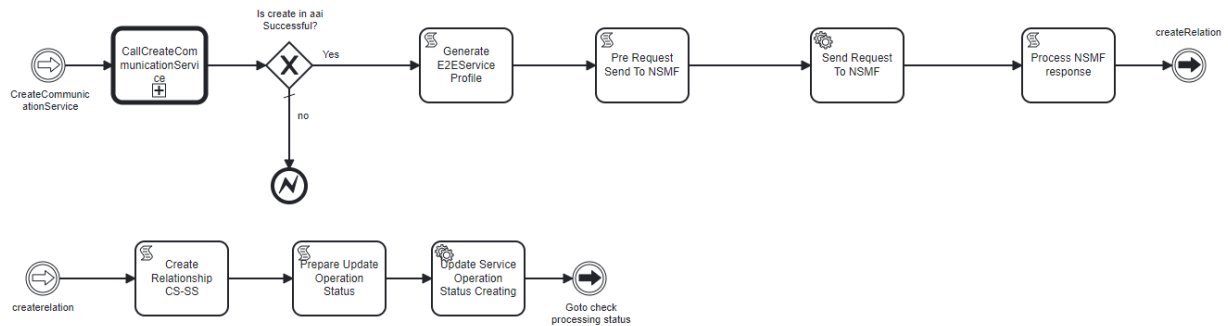


Figure 3.3.2.3.3: Create Custom Communication Service and Send to NSMF

During the commissioning process several objects are created and maintained into the inventory, so that any BSS system can access ONAP to view the service instance data in relation to Service Slice created. Those objects are arranged in a hierarchical manner as we can verify on Figure 3.3.2.3.1.

On Figure 3.3.2.3.4 is represented a flowchart representing the high-level logic for the creation of the network slice.

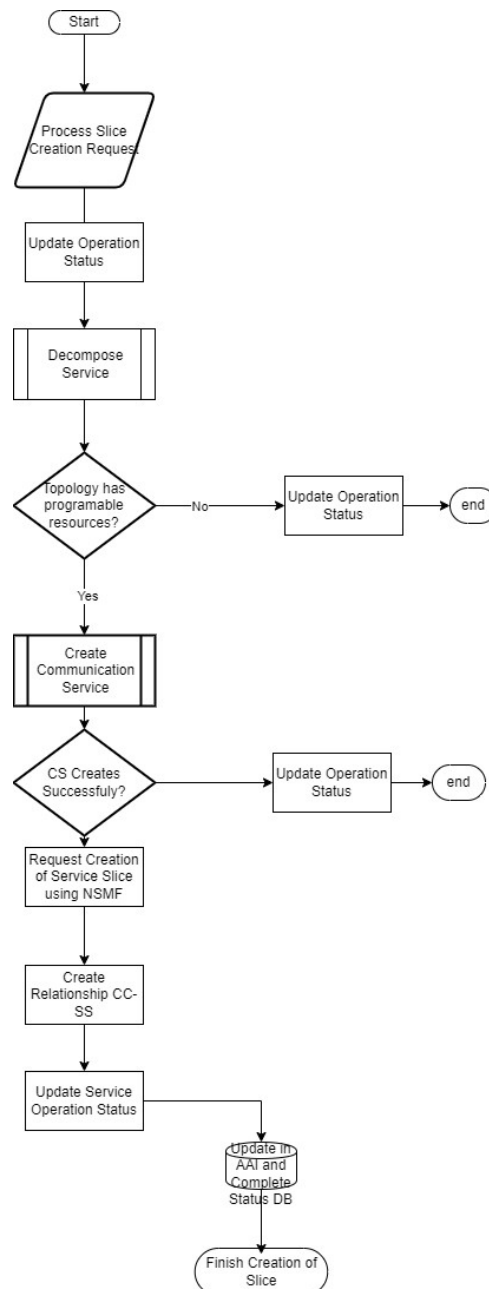


Figure 3.3.2.3.4: Network Slice Creation Logic

It starts by the process of decomposing the service based on the received request, then a validation against the existing available topology is realized to validate if a network slice can be created and if the resources are eligible. Then a communication service is created following the creation of a Service Slice by the NSMF. During the creation of a service slice phase, it is requested to UWS Slice Manager the creation of the network slice in the infrastructure.

Figure 3.3.2.3.5 shows the result of the created objects for the network slice. This data is returned by ONAP AAI and can be accessed externally by other systems such as an BSS system.

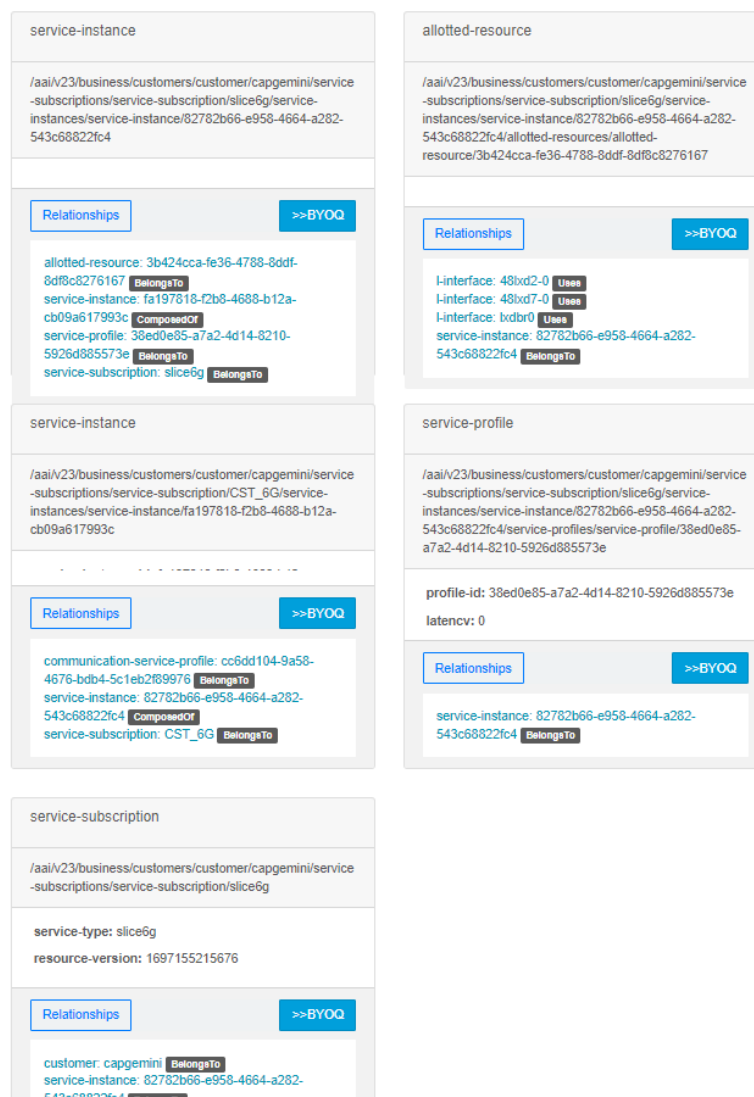


Figure 3.3.2.3.5: Result After creating a network slice via ONAP

3.3.2.4 Network Slicing Attachment Phase

The next phase is the **network slice attachment**, that consists of the attachment of traffic flows to an existing slice. On ONAP side was implemented a workflow that handles the requests to attach traffic rules applied to a network slice id previously created. Those traffic rules are associated to a given port or IP address with certain policies, creating a network flow.

This operation is realized requesting the network slice attachment via the exposed REST APIs from UWS slice manager. In result, on ONAP side, it will be created a configuration and all the relations with the slice service. Figure 3.3.2.4.1 shows part of the BPMN workflow for slice attachment.

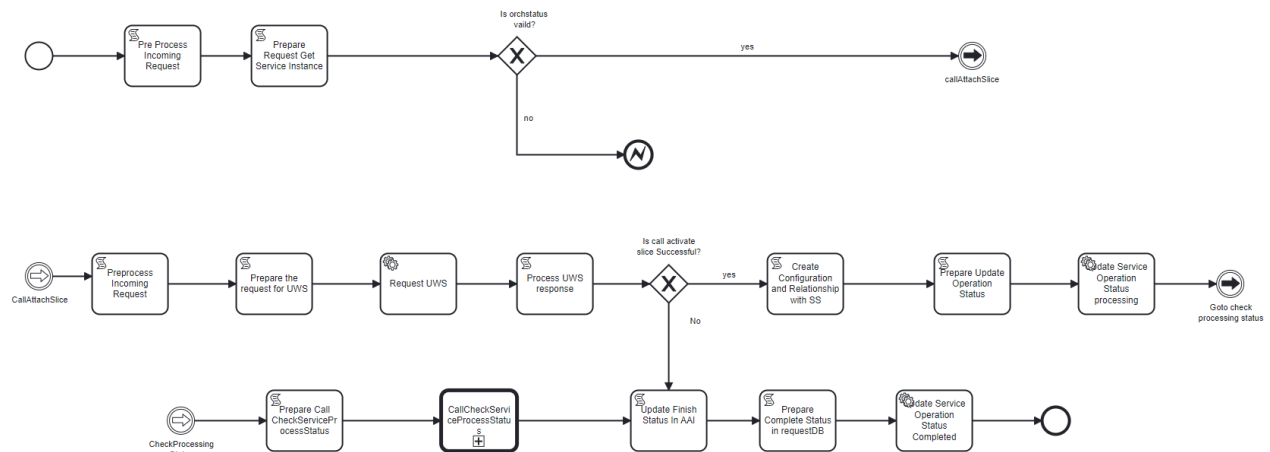


Figure 3.3.2.4.1: ONAP BPMN Workflow for Slice Attachment

During this process, all these configurations and identifiers are stored in ONAP AAI for the lifecycle management of the network slice. The operation status is also kept on inventory for later perform its maintenance.

3.3.2.5 Network Slicing Decommission Phase

The **deletion phase of the network slice** consists into the decommission of the network slice starting by requesting the deletion of the slice to UWS Slice manager and finally delete all resources, services, and relationships on inventory. Figure 3.3.2.5.1 shows part of the BPMN workflow for network slice deletion.

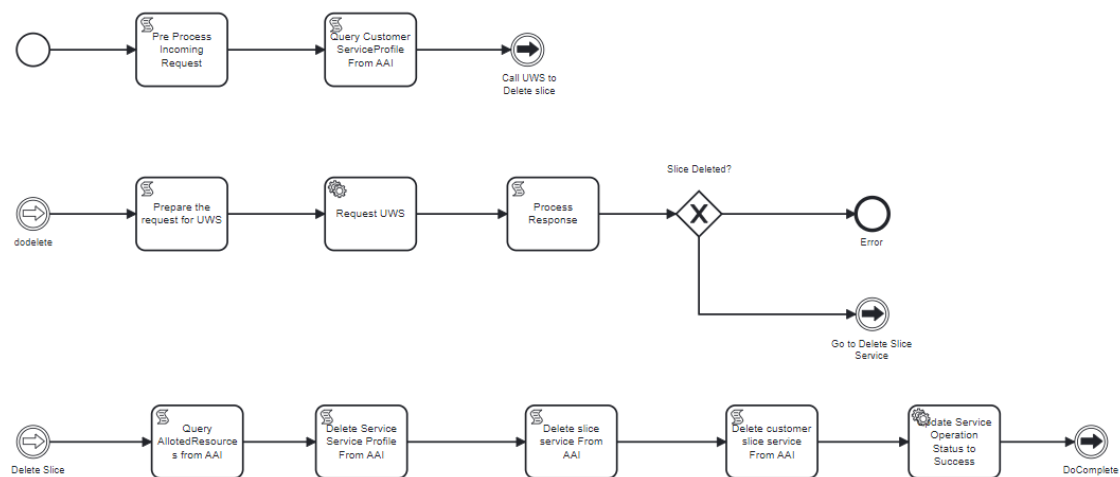


Figure 3.3.2.5.1: Delete Slice BPMN Workflow

3.3.2.6 Autonomous Policy Management and Metrics Collection MANO

In context of 5G networks, the non-RT RIC plays a crucial role in monitoring RAN usage for several reasons, including optimizing the network performance by monitoring RAN usage patterns, traffic load and resource utilization. This data helps in optimizing the network by

allocating resources and adjusting. Also, it facilitates the allocation of resources among different Network Slices based on their specific requirements.

ONAP MANO layer supports functionalities such as non-Real-time RAN Intelligent Controller (RIC) as specified by the O-RAN Alliance. Thus, Capgemini developed a control loop where RAN metrics are being collected via ONAP VES Collector, such as THP, latency and then via an rAPP, which consists in an analytics application, using a deterministic algorithm that verifies whether the SLAs defined by the operator are being satisfied or not.

Figure 3.3.2.6.1 shows the architecture regarding this control loop. At top is located management layer where the ONAP framework implements the O-RAN Non-RealTime RIC, and data collection and analytics functionalities, controllers, and adaptors for interconnection with underlying RAN components are available.

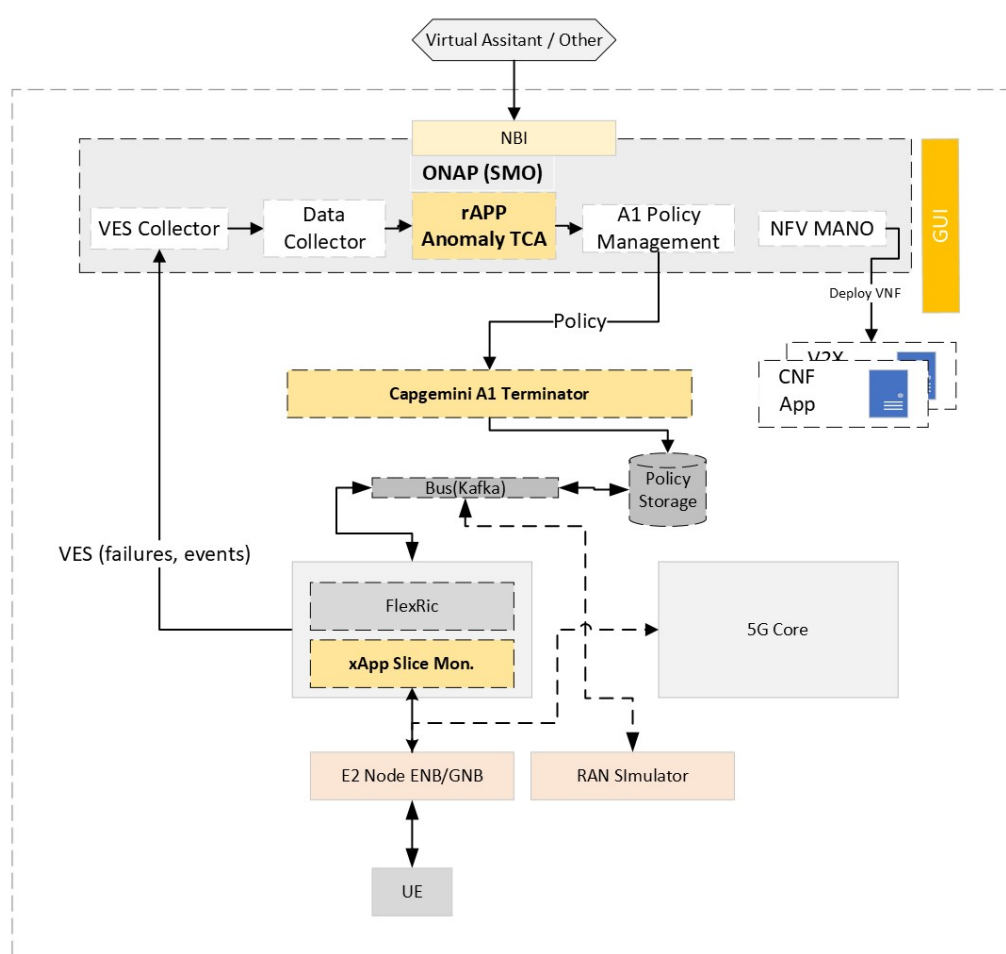


Figure 3.3.2.6.1: Control Loop for RAN Optimization Using ONAP as Non-RT-RIC

In the current scenario, the ONAP Non-RT RIC is responsible for continuously monitor the RAN network usage. If an SLA violation is detected, the non-RT RIC makes configuration changes and updates the xAPP in the Near-RT RIC that is responsible for RAN slice assurance.

Through the project capgemini faced several difficulties on validating phase (to validate whether the loop was working in an efficient manner, as intended) so we build a setup, composed mainly with open-source components, including ONAP framework, FlexRic (Near-

RT RIC and xAPP for slicing monitoring), OpenAirInterface RAN and a 5G Core (Open5Gs). This meant that we build our 5G network, with USRP X310 for implementing the enb/gnb, 5G terminals with SIM cards and respective cables and connectors. This was not forecasted initially in the project technical annex B, but it turned to be fundamental to ensure that the orchestration component of task 5.3 worked as intended.

Due the lack of interconnection between the ONAP Non-RT RIC and FlexRic we build an A1 Terminator following the O-RAN specifications [28] allowing its interconnection.

Finally on Non-RT RIC side analysis rApp was developed, based on TCA (Threshold Cross Analytics), allowing to analyse the collected metrics and trigger policies according to the defined SLAs.

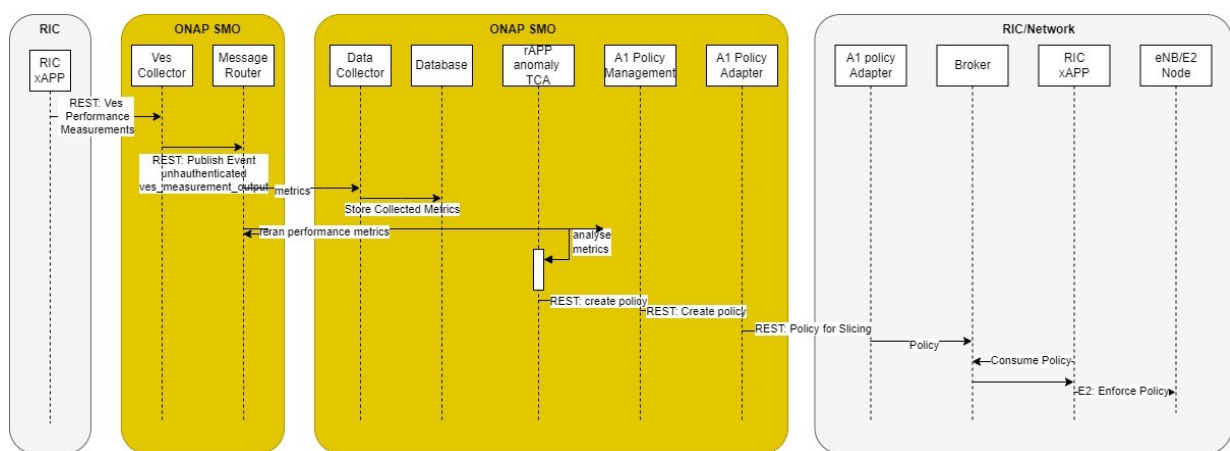


Figure 3.3.2.6.2: Flow for RAN Measurements and Policy Enforcement

On Figure 3.3.2.6.2 is represented the messages and interaction on this control loop, where the metrics are being collected and sent to VES collector, then they are stored, and finally analysed. A policy is triggered when the defined threshold is crossed, and it is propagated to the RIC/Network via A1 interface. A policy is translated and sent to ENB/GNB via E2 interface.

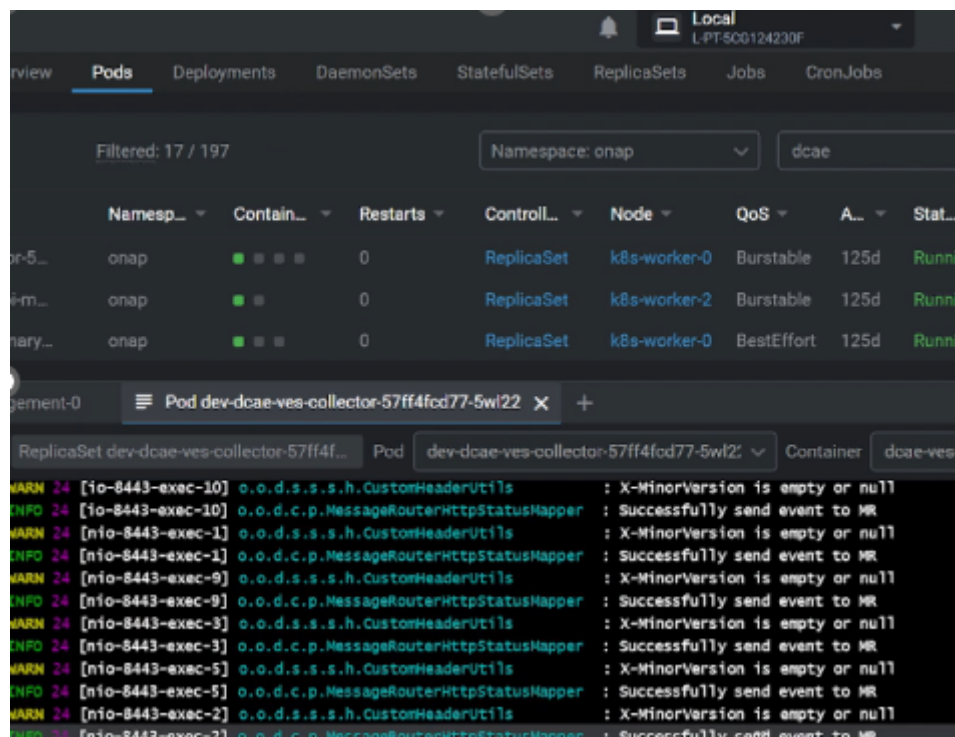


Figure 3.3.2.6.3: VES Collector Data Collection

On Non-RT RIC the monitoring data is collected and stored using the data collector for later be used by the rAPP. Figure 3.3.2.6.3, shows the logs of the RAN data that is collected and continuously checked by the rAPP that based on a defined SLA will trigger a policy with the recommendation for slice creation.

Following on Table 3.3.2.6.1 there is represented a policy example created by the rAPP that is send to RIC via A1.

PUT	/v1/a1-p/policytypes/SLICING/policies/<policieId> <i>Send policy</i>
Parameters	<i>No parameters</i>
Body	application/json <pre>{ "scope": { "sliceId": "URLLC", "cellId": ["cell1"] }, "statement": { "latency": "10", "ulthptPerSlice": "20", "dlthptPerSlice": "30" } }</pre>

<pre> } }</pre>
Response

Table 3.3.2.6.1: A1 Policy Example for Slicing Enforcement

Once the policy is received in the RIC, the requirements are translated on RAN configurations by the slicing xAPP. Finally, it is sent an request to E2 Agent, that will enforce the creation of a slice in the eNB as we can see on Figure 3.3.2.6.4. In this example, 2 DL slices were create using the slice scheduler algorithm EDF.

```
[S1AP] Found usable eNB ue_slap_id: 0x1321f1 1253873(10)
[E2-AGENT]: RIC SUBSCRIPTION REQUEST rx
[MAC] add DL slice id 0, label s1, slice sched algo EDF, ue sched algo proportional_fair_wbcrq_dl
[MAC] add DL slice id 2, label s2, slice sched algo EDF, ue sched algo round_robin_dl
[E2-AGENT]: CONTROL ACKNOWLEDGE sent
```

Figure 3.3.2.6.4: Logs on E2 Agent showing the creation of Slices

3.4 MA-DRL

3.4.1 Final version of the MA-DRL platform

3.4.1.1 Overview and architecture of the final TheRLib MA-DRL platform

Figure 3.4.1.1.1 illustrates the architecture of the TheRLib MA-DRL platform final version:

- The TheRLib core Python libraries contain the implementation of RL algorithms, the main interfaces of RL Environments with a Markov Decision Process (MDP) API, the utilities to define Neural Networks (NN) architectures, and to connect to the dashboard while defining the training's main parameters.
- The Therdash server, based on Apache and Flask technologies, with the help of the experiment database, allows the user to track its trainings outcomes.
- (NEW) The TheRLib-deploy Python package provides a lightweight Python environment to embed the decision-making of an agent trained in the DRL-platform, with the minimum required software dependencies.

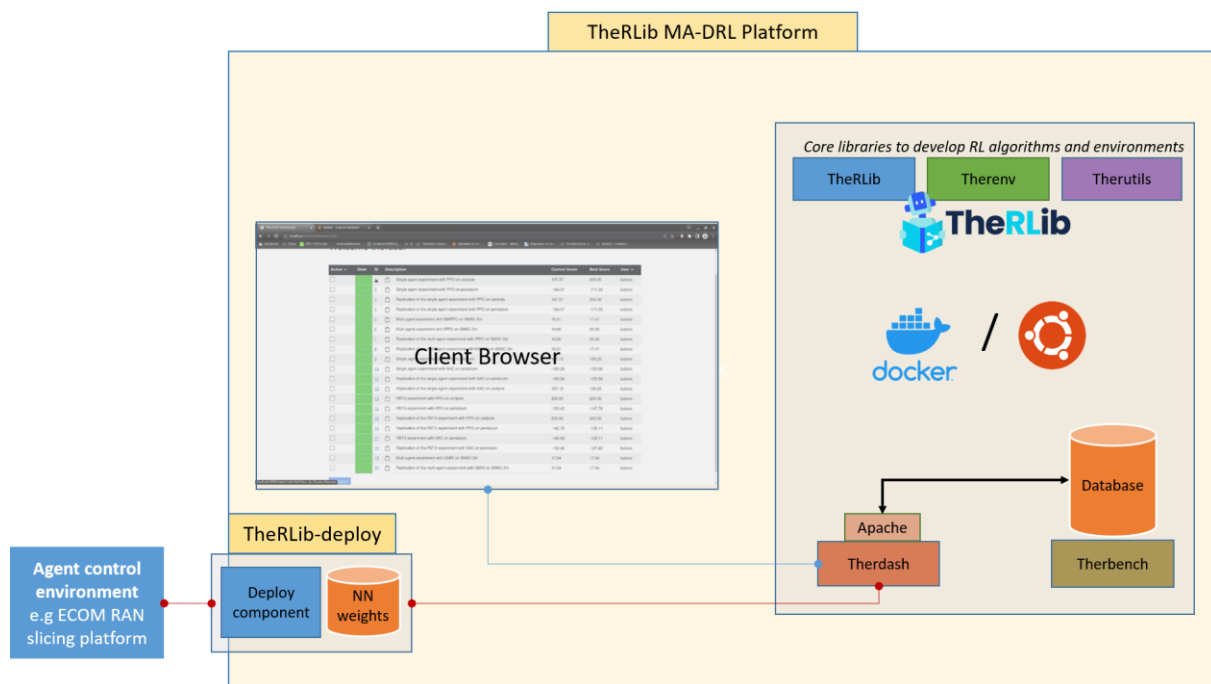


Figure 3.4.1.1.1: TheRLib MA-DRL Platform architecture

Native system install has been preferred to Docker for both integration use-cases.

- RAN Link (UBRU) requires training on a MATLAB-based environment that is not deployed on a Docker.
- RAN slicing requires interfacing to ECOM tools that are available on the native system. The integration to ECOM's RAN slicing RL pipeline in the testbed leverages the TheRLib-deploy package. It is to provide decisions from the trained agent without embedding the whole MA-DRL platform.

To align the usage conditions of the MA-DRL platform instantiations, we transition from the Docker deployment to the native system deployment on the instance of MA-DRL of the integration testbed.

Thus, an automated installation script now allows installing the MA-DRL platform on the native system. It has been developed and tested on Ubuntu 22.04 LTS which is the distribution targeted in the UBRU and UWS testbed MA-DRL platform instances. Besides, the installation script is leveraged for installing the platform on Ubuntu 20.04 LTS, with additional instructions documented on a dedicated README.

The Docker deployment of the MA-DRL platform is still used to test and validate updates of the software. Figure 3.4.1.1.2 shows the semi-automated CI/CD pipeline implemented to generate a new TheRLib release. The process originates from the current version of the development TSG Gitlab of TheRLib. Two Python scripts are run:

- *make_release.py* selects and packages the stable code of the core Python libraries, the Therdash dashboard code and the reference training scripts. Thus, they are packaged in a single archive with the installation scripts and READMEs.
- *run_benchmark.py* runs all the reference training scripts to generate initial benchmark experiment data for the dashboard.

The outputs of these scripts are then tested with the Docker image building. This handles the necessary steps to deploy the TheRLib release with the benchmark data on a Ubuntu 22.04 LTS container. A quick user test of the Docker image allows validating that the dashboard runs well and loads the benchmark data, and that the release allows running the reference training scripts. This step is especially useful to detect regressions with the Python software dependencies, as some dependencies can be made obsolete when the targeted version is not available anymore with the Python package manager.

When these tests are passed with success, the new release is ready to be shared. For now, each of these steps must be run manually. A fully automated version with Gitlab-CI is foreseen out of the scope of the 6G-BRAINS project.

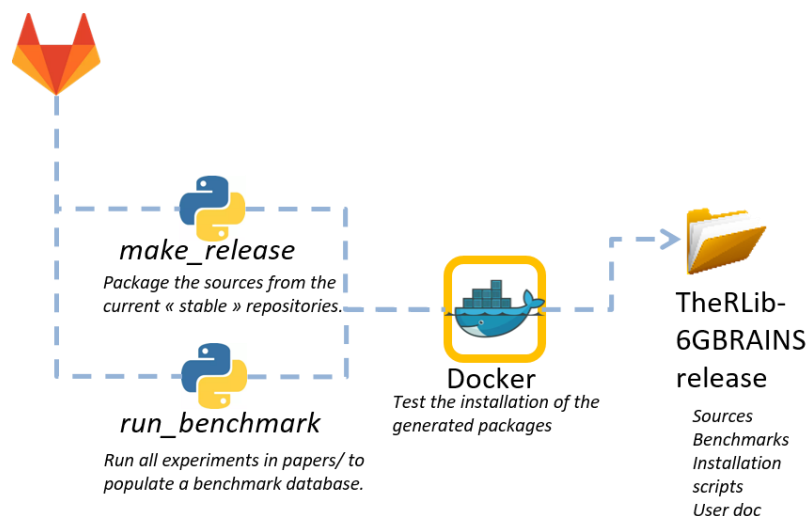


Figure 3.4.1.1.2: Semi-automated CI-CD pipeline

In the final release of the MA-DRL platform, the Therdash dashboard is improved with the following features:

- In the control panel view, it is now possible to sort experiments by status, identifier, or title. It is also possible to choose the number of experiments displayed on a page.
- In the experiment view, it is now possible to zoom in and out the metric plots.

3.4.1.2 Updated training scripts architecture

The software architecture of the training scripts and the algorithms implementation has been reworked as shown by Figure 3.4.1.2.1. It is now friendlier for non RL experts users. Whereas the previous structure of the *train.py* scripts incorporated the main structure of the algorithm's implementation on a given environment, a dedicated *Trainer* class within **therlib** now handles this task. An Agent class also embeds the agent-environment interactions between the Deep RL agent and a training or evaluation environment and relies on a generic neural network Model class for the simple vector-shaped observation spaces.

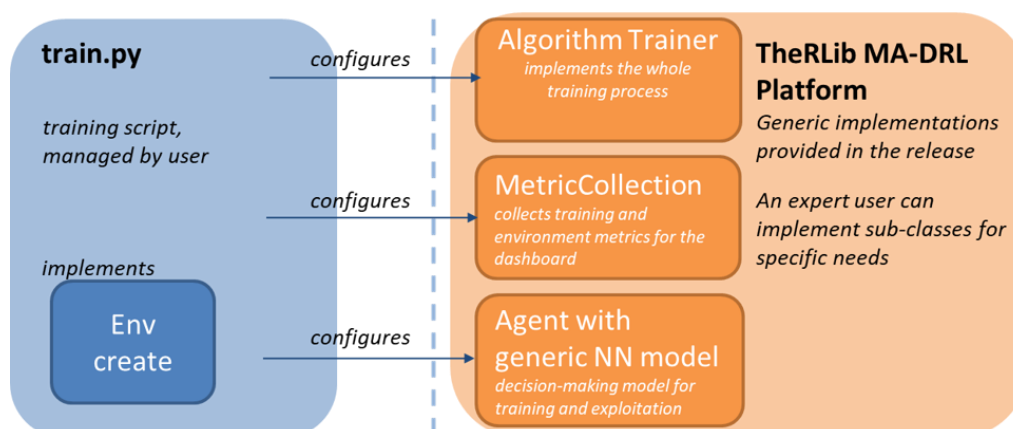


Figure 3.4.1.2.1: Overview of the new training architecture

Figure 3.4.1.2.2 presents the new way of writing a training script. In the previous release, the training script contained most of the algorithm implementation, as well as the definition of the training parameters. Now, the training script instantiates the *Trainer*, *Environment*, *MetricCollection*, *Agent*, etc. components from the parameters loaded with the script parser. The main modification a non-expert user needs to make to a training script is writing the `make_env` function that creates an instance of the training environment.

For both main RL algorithms provided in the MA-DRL platform, namely Soft Actor Critic (SAC) and Proximal Policy Optimization (PPO), the provided example training script now handles both Discrete and Continuous action spaces and can be run on all Gym's classic control environments.


```

19 def make_env(args):
20     def _init():
21         env = gym.make(args['--env_name'])
22         return env
23     return _init
24
25
26 def make_env_agent_and_trainer(args, exp_id, agent_id, device):
27     runnable_env = make_env(args)
28     env = runnable_env()
29     # We create the actor-critic model
30     model = make_classic_control(
31         env.observation_space, env.action_space,
32         hidden_dim=args["--hidden"],
33         separate_backbone=args["--separate_backbone"],
34         value_distillation=args["--value_distillation"]
35     ).to(device)
36     # We create the agent
37     agent = ActorCriticAgent(agent_id, model)
38     # We create the trainer
39     trainer = PPOTrainer(
40         exp_id, agent_id, agent,
41         path = os.path.join(args['--exp'], 'agent_0'),
42         train_seed = args["--train_seed"],
43         num_envs = args["--num_envs"],
44         actor_clip_param = args["--actor_clip_param"],
45         critic_coef = args["--critic_coef"],
46         entropy_coef = args["--entropy_coef"],
47         clip_grad = args["--clip_grad"],
48         norm_adv = args["--norm_adv"],
49         clip_critic = args["--clip_critic"],
50         gamma = args['--gamma'],
51         lambda_pi = args['--lambda_pi'],
52         lambda_value = args['--lambda_value'],
53         critic_clip_param = args['--critic_clip_param'],
54         clip_grad_critic = args['--clip_grad_critic'],
55         value_epochs = args['--value_epochs'],
56         ppo_epochs = args['--ppo_epochs'],
57         mini_batch_size = args['--mini_batch_size'],
58         lambda_KL = args['--lambda_KL'],
59         distillation_period = args['--distillation_period'],
60         horizon = args['--horizon'],
61         budget = args['--budget'],
62         save_interval = args['--save_interval'],
63         value_distillation = args['--value_distillation'],
64         auxiliary_epochs = args['--auxiliary_epochs'],
65         checkpoint = args['--checkpoint'],

```

Figure 3.4.1.2.2: Overview of a training script

The new *MetricCollection* object provides a generic way to define the data collection and logging during the training. It is used to define which training metrics are logged such as the algorithm's loss and its sub-terms. It also handles collecting data from the environment(s) during the training, such as the episodic reward (always collected) and the episode length in environment steps. Additional metrics can be collected from the info dictionary of the environment's step API. The *MetricCollection* implementation handles several instances of environments being run in parallel processes during the training, as in usual PPO trainings. An example of the *MetricCollection* instantiation is described in Figure 3.4.1.2.3 below.

```

71
72 def make_metric_collection(args):
73     # Losses
74     loss_keys = [
75         'loss', 'critic_loss', 'actor_loss', 'entropy_loss'
76     ]
77     if args['--value_distillation']:
78         loss_keys += ['auxiliary_loss', 'KL_loss']
79     losses = [
80         LossMetric(loss_key) for loss_key in loss_keys
81     ]
82     return MetricCollection(
83         num_envs=args["--num_envs"],
84         score_buffer_size=50,
85         collect_episode_length=False,
86         losses=losses,
87         score_metric="episode_reward"
88     )
89

```

Figure 3.4.1.2.3: Overview of the MetricCollection parameters and instantiation

3.4.1.3 MATLAB Interface

The final MA-DRL release comes with an interface to MATLAB environments defined with the MATLAB Reinforcement Learning Toolbox. The Matlab Interface is implemented within the **therenv** library and interacts with a MATLAB RL environment through MATLAB Engine API for Python.

An illustration of this MATLAB interface is shown below in Figure 3.4.1.3.1. The code is separated into the Python code to interact with TheRLib (highlighted in green) and the MATLAB code designed to interact with the MATLAB rLEnvironment (highlighted in orange). The parsing of information within the interface is shown by the direction of the arrows. Further detail is also provided about the functions used and the expected arguments and returns to provide traceability to the interface.

The python code illustration for the MATLAB interface is shown below in Figure 3.4.1.3.1 and how it interacts with TheRLib. TheRLib's train script requires minimal alteration, just replacing the loading of a gym environment with the calling of the MATLAB interface Environment wrapper. This will allow the MATLAB rLEnvironment to be stored and read in a Gym environment format which TheRLib is expecting. The Environment wrapper initialises the MATLAB Engine API and then uses this to initialise the MATLAB rLEnvironment. A parser extracts the object and observation space information and stores them in the required Gym format. The TheRLib's train script interacts with the MATLAB rLEnvironment through the standard step and reset functions, which in turn use the MATLAB Engine API to call the equivalent functions and parse information as required.

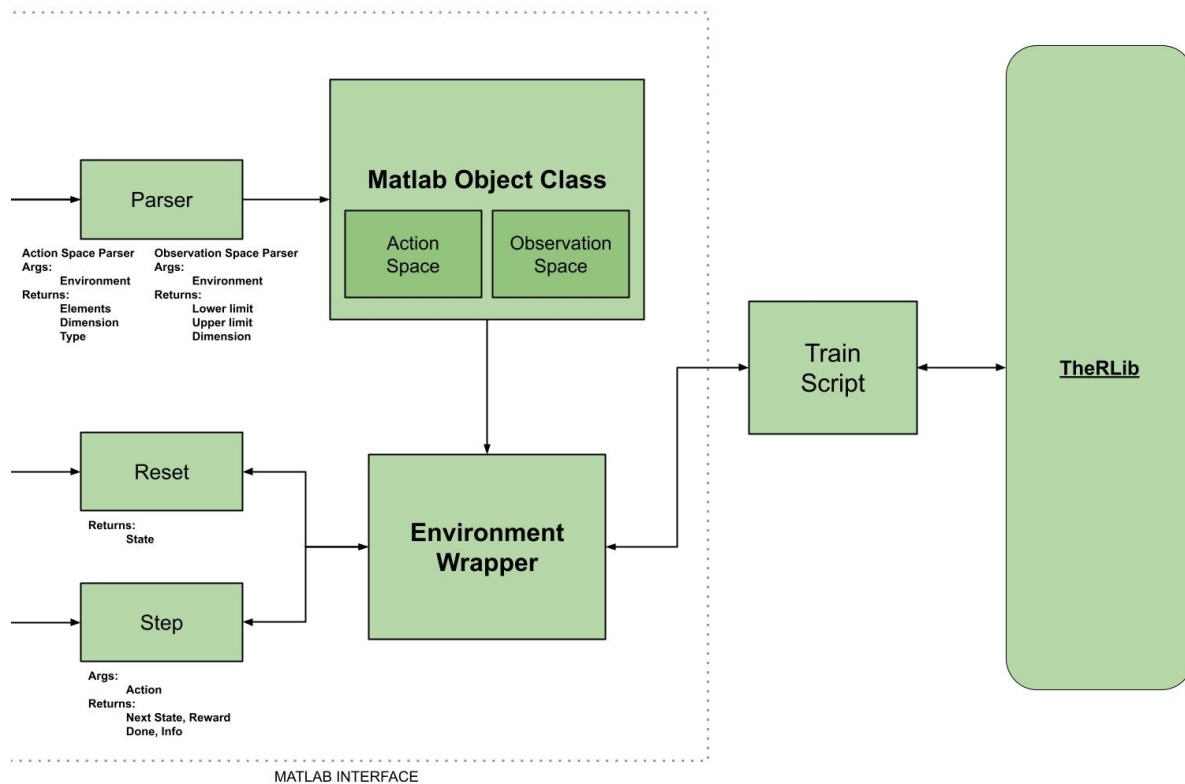


Figure 3.4.1.3.1: MATLAB Interface Python code illustration

The MATLAB code illustration for the MATLAB interface is shown below in Figure 3.4.1.3.2. It is within the MATLAB code where the error handling for the interface can be done as otherwise all errors are parsed to python through the MATLAB Engine API as the same. The functions `env_step` and `env_reset` check for the existence of step and reset functions within the environment with the expected format for input and output. Similarly, `getObservationInfo` and `getActionInfo` parses the required information of the action and observation spaces to the python code. As MATLAB's RL toolbox allows for the use of infinity as a limit, the `limit_check` script checks for and applies limits as appropriate based on other features in the action and observation spaces. This combination, when called through the MATLAB Engine API, allows for the interaction between TheRLib's train script and MATLAB's `rlEnvironment`. The full flow of information is shown in Figure 3.4.1.3.3.

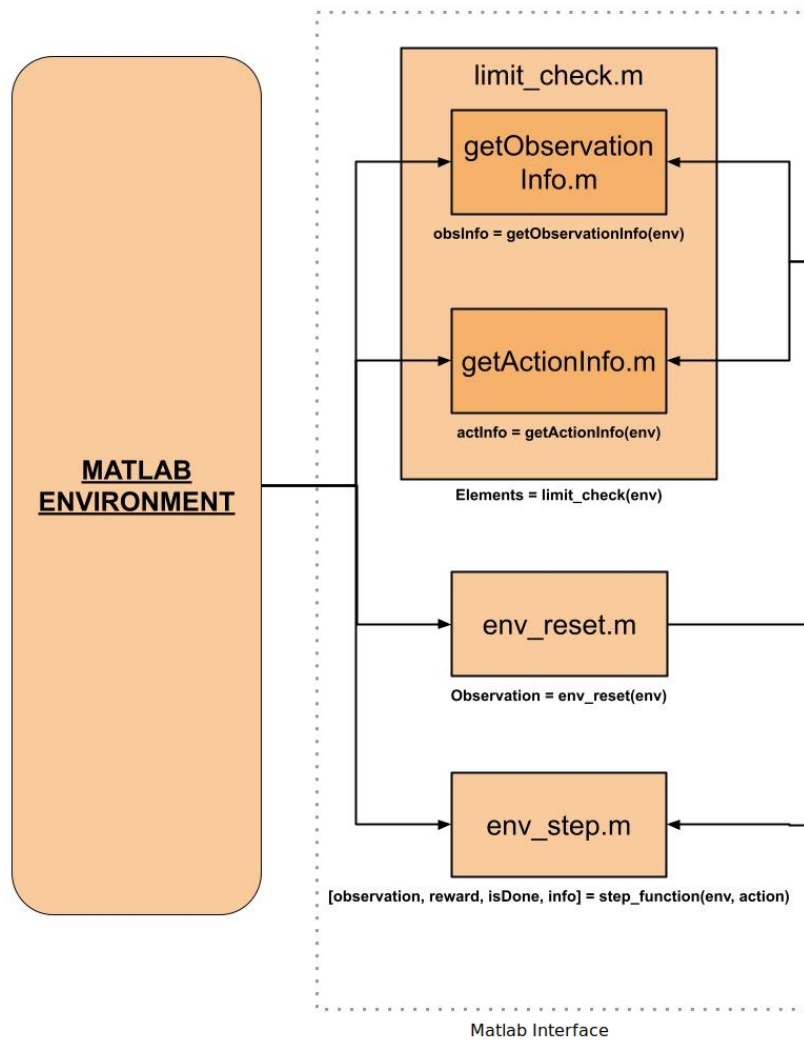


Figure 3.4.1.3.2: MATLAB Interface MATLAB code illustration

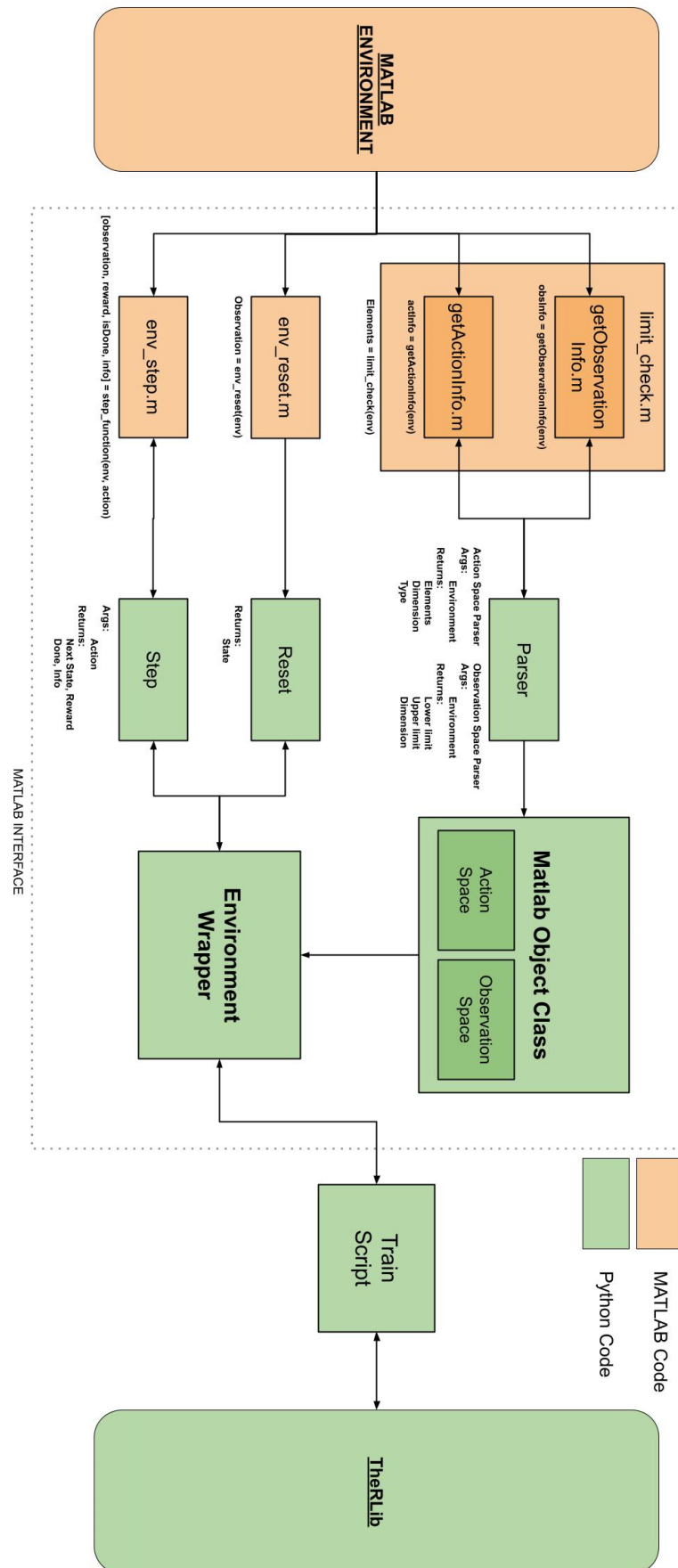


Figure 3.4.1.3.3: Illustration of Overview of the MATLAB Interface

As the MATLAB RL toolbox comes with an implementation of the CartPole environment that was also used in the MA-DRL platform example benchmarks, comparison experiments with the SAC algorithm have been conducted on MATLAB's (in red) and Gym's (in blue) Python-based CartPole environments, with the results illustrated by Figure 3.4.1.3.4. We can see that the agent achieves similar performances on the episodic reward, but the MATLAB rEnvironment takes around 60% more training wall time compared to the python Gym counterpart. The offset in the training time at zero steps is caused by the time to initialise the MATLAB Engine API and environment.

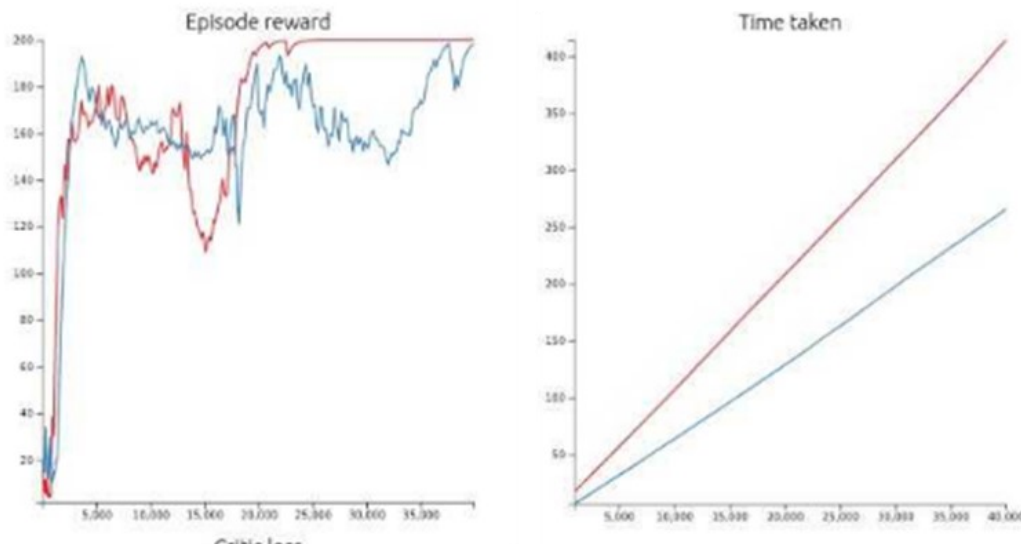


Figure 3.4.1.3.4: Episode Reward and Time Taken for a CartPole experiment within a MATLAB Environment (red) and a Gym Environment (blue)

3.4.1.4 TheRLib-deploy deployment component

The new TheRLib-deploy Python package extends the capabilities of the MA-DRL platform to deploy trained agents in their operating environment. In the envisioned use-cases, we suppose that it is more advantageous, or that some constraints obligate, to embed the trained agent close to the Control Plane, where the CPU, GPU and RAM resources are limited. Figure 3.4.1.4.1 describes the abstract usage scenario: the RL-trained policy is exploited in the deployment environment (green), whereas it has been trained or is undergoing the training process in the training environment (orange). The deployment environment holds the TheRLib-deploy package ("Lightweight Client") and the control process that requires the actions provided by the RL Agent. The training environment holds the MA-DRL platform that performs the training, and especially, the Therdash Flask server, with which TheRLib-deploy can interact.

To be initialised or updated, the TheRLib-deploy code can query a particular version of the agent trained for its deployment usage, such as RAN slicing or RAN link control. Typically, the best agent during the training history is used. This is implemented through generic commands to explore, analyse, and fetch data from the experiment database.

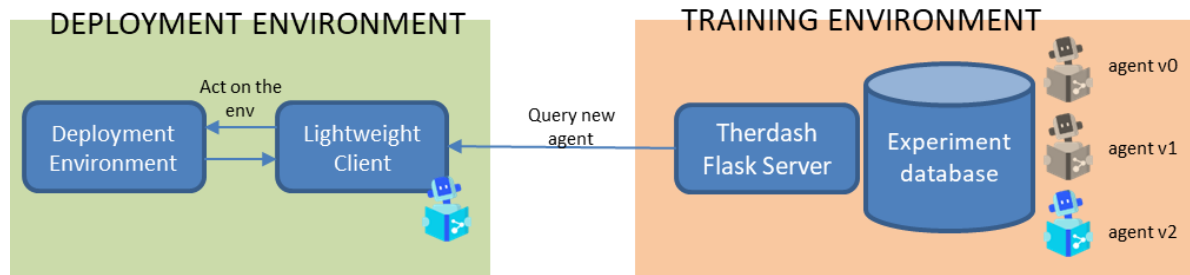


Figure 3.4.1.4.1: Main principles of the TheRLib-deploy component

The deployment component requires few software dependencies on the TheRLib code: it only needs the *Agent* and *Model* implementation and acts as a client to the Therdash Flask server. Thus, it can be implemented on a light Python virtual environment based on torch, numpy and the requests package of the Python standard library.

The following figures illustrate a proof of concept of this deployment component implemented on a Python notebook.

In Figure 3.4.1.4.2, the code connects to the Therdash Flask server to query basic information on a training experiment leveraged for this example. This training will serve as a source for our deployed agent. With the “options” route, all parameters of the training are collected as a JSON text. Basic validation can be performed from this information: for example, verify that the training environment name matches the deployment environment.

Then in Figure 3.4.1.4.3, the “weights” route is to get all neural network weights available for this training. The information is returned in the JSON format, as a dictionary. The first-level key is the agent identifier: for a single-agent training we will only get the “agent_0” key, whereas for a multi-agent training we will get “agent_0”, “agent_1”, ..., “agent_n”. The second-level key is the number of agent-environment steps for a RL training, or of training epochs for an Imitation Learning (IL) training, when the weights have been saved. As in the notebook example a single-agent IL experiment is used, the code extracts the available weights by training epoch.

Most of the time, using the latest neural network weights would lead to the best results in deployment. But in RL and in Deep Learning in general, it is possible that the performance (in terms of reward) of the agent drops during or at the end of the training. This phenomenon is called catastrophic forgetting. To help provide countermeasures, the API allows querying the training metrics with the “experiment_data” route, as illustrated in Figure 3.4.1.4.4. Then, it is possible to check the value of these metrics at the training epochs of the neural network weights to validate their performance and choose the best one. In this example, the “score” metric considered corresponds to the mean episode reward over a set of 100 test episodes. More complex validation methods can be envisioned, like running the test procedure on a realistic simulation.

When the best agent has been chosen from the available metric information, the “weights_byte_data” route is used to get the neural network weights as a base64-encoded binary string. It is then decoded and loaded with Pytorch from the deployment component to

create or update the corresponding TheRLib agent and provide decisions for the deployment environment.

```
In [18]: # We set some general parameters
server_address = "http://localhost:8066" # address of the therdash Flask server
experiment_id = 7 # the training experiment that holds our training weights
agent_id = 0 # mono-agent experiment. To respect TheRLib conventions, agent_id is set to 0
device = "cpu" # CPU will be used for NN inference

In [19]: # We load the training experiment options. This holds some information on the
# environment and model parameters that we need to create/validate our agent.
exp_options = requests.get("{}options?exp_id={}".format(
    server_address, experiment_id
))
exp_options_json = json.loads(exp_options.content)
exp_options_json

Out[19]: {'--bc_epochs': 30,
 '--beta_max': 0.999,
 '--beta_min': 0.9,
 '--buffer_capacity': 10000,
 '--buffer_folder_path': './data_collection_buffer',
 '--checkpoint': None,
 '--clip_grad': None,
 '--decay': 0,
 '--description': 'BC on CartPole from dataset generated with PPO',
 '--entropy_coef': 0.0,
 '--env_name': 'CartPole-v1',
 '--eval_period': 2,
 '--exp': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole',
 '--exp_conf': 'bc_options.json',
 '--help': False,
 '--hidden': 256,
 '--lr': 0.0001,
 '--max_steps': None,
 '--mini_batch_size': 10,
 '--normalize_actions': False,
 '--num_envs_test': 5,
 '--num_episodes_test': 10,
 '--optim_epsilon': 1e-08,
 '--optimizer': 'Adam',
 '--save_interval': 1,
 '--save_start': 0,
 '--test_period': 2,
 '--test_seed': 0,
 '--train_seed': 123,
 '--user': None,
 '--validation_buffer_name': 'buffer_data_eval',
 '--version': False,
 '--weight_interval': 5}
```

Figure 3.4.1.4.2: Deployment pipeline: connect to Therdash and query experiment options


```
In [21]: # We get the sorted identifiers (here training epoch number) for our available
# weights
weights_files = requests.get("{}weights?exp_id={}".format(
    server_address, experiment_id
))
weights_files_json = json.loads(weights_files.content)
print(weights_files_json)
weights_ids = sorted([int(k) for k in weights_files_json["agent_0"].keys()])
print(weights_ids)

{'agent_0': {'0': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight0.pt', '10': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight10.pt', '15': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight15.pt', '20': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight20.pt', '25': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight25.pt', '30': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight30.pt', '5': '/home/t0147188/EXPERIMENTS/6gbrains/BC-cartpole/agent_0/weights/weight5.pt'}}
[0, 5, 10, 15, 20, 25, 30]
```

Figure 3.4.1.4.3: Deployment pipeline: query which neural network weights are available

```
In [22]: # We get all the experiment metric data
experiment_data = requests.get("{}experiment_data?exp_id={}".format(
    server_address, experiment_id
))
experiment_data_json = json.loads(experiment_data.content)
experiment_data_json = experiment_data_json["agents"]["agent_0"]

In [23]: # And we extract the 'score' indicator associated to our weights ids.
data_per_weight_id = {
    k: experiment_data_json[str(k)]
    for k in weights_ids
}
# We hack with json to pretty-print our data dict
print((json.dumps(data_per_weight_id, indent=2, default=str)))
score_per_weight_id = {
    k: data_per_weight_id[k]["score"]
    for k in weights_ids
}
# We print the text scores. 500 is the best we can achieve on CartPole
print(score_per_weight_id)
```

```
{
  "0": {
    "bc_loss": 0.6095314900577068,
    "entropy_loss": 0.654677318572998,
    "episode_reward": 18.4,
    "loss": 0.6095314900577068,
    "score": 18.4,
    "validation_bc_loss": 0.6965541842579842,
    "validation_entropy_loss": 0.6882856411337852,
    "validation_loss": 0.6965541842579842
  },
  "5": {
    "bc_loss": 0.5094067673385143,
    "entropy_loss": 0.5456465177237988,
    "episode_reward": 247.1,
    "loss": 0.5094067673385143,
    "score": 247.1,
    "validation_bc_loss": 0.5197382824718952,
    "validation_entropy_loss": 0.5572844988405704,
    "validation_loss": 0.5197382824718952
  },
  "10": {
    "bc_loss": 0.4971006271094084,
    "entropy_loss": 0.5130267409980297,
    "episode_reward": 492.0,
    "loss": 0.4971006271094084,
    "score": 492.0,
    "validation_bc_loss": 0.4971951330751181,
    "validation_entropy_loss": 0.5134524602293968,
    "validation_loss": 0.4971951330751181
  },
  "15": {
    "bc_loss": 0.4938003252893686,
```

Figure 3.4.1.4.4: Deployment pipeline: query the metric data associated to the neural network weights

```
In [24]: # Now we pick our latest weights, and query the dashboard to get their binary
# data
weights_id = weights_ids[-1]
weights_byte_data = requests.get(
    "{}/weights_byte_data?exp_id={}&agent_id={} &weight_id={}".format(
        server_address, experiment_id, agent_id, weights_id
    ))
weights_bytes_json = json.loads(weights_byte_data.content)
weights_bytes_json
```

```
Out[24]: {'status': 'success',
'weights': 'UESDBAACAGAAAAAAAAAAAAAAAAAAAAQBIAYXJjaGl2S29scYXRhLnBrEzCDgBaWlpawlpw\ntlpaWlpawoACfXEAKfgFAAA
AZXBvY2ZhaU5eWNAQAABsb3NzcQjOWBAAAABtb2RlbF9zdGFM0ZV9k\nawlM0cQNjY29sbGVjdGlvbnMKt3JkZXJlZERpY3QkcQQpUnEFKfgPAAAAAY3J
pdG1jlJAud2VpZ2ZhbnCQZjdG9yY2guX3V0aWxzCl9yZWJ1aWxkX3RlbnNvc192MgpXBygoWAcAAABzdG9yYmldlcQhjdG9y\nnY2gKRmxvYXRTdG9yY
WdlCnEjWAEAAAAwcqPYBgAAGN1ZGE6MHELTQAEDEHMuUsATQABSWSgcQ1L\nnBsEbhnEOingEKVjxd3RXefJxEvgNAAAAAY3JpdG1jlJAUymIhc3ESa
AcokKgIa1YAQAADfxElGlnAAAAAY3VkYTowcRRNAAF0cRVRSwBNAAKGfcRZLAYVxf4loBC1ScRh0cR1ScRpYdWAAAGNYaXRPy4y\nnlndlaWdodHe
baAcOKGgaIA1YAQAADAJxHFgGAAAAAY3VyTowcR1NAAF0cR5RSwBLAU0AAAYZxH00A\nnAuSBhnEgiWgEKVjIXRxIljXIlgNAAAAAY3JpdG1jlJAUymI
hc3EkaAcOKGgaIA1YAQAADINjVGglnAAAAAY3VyTowcSZLAXRxlJ1FLAESBHXeoSwGfCSmjAaqUPunEqdHERUnEsWA4AAABHY3Rvcil4Wldnl\ndWadod
HFTAdcnKGoTaDlVA0ADAADRvl1oGADAAY3VyTowcSQNADAAR0cTRRSwRNADAFLRT7YMIHCFSGG6ncTK7AdOnInFzdHF0InFlWADAADAARHY3Rvcil4WldmlnY
```

```
In [25]: # Then we decode what we received, knowing that we sent our base64-encoded
# bytes as an ASCII string on the server-side.
# And we deserialize with torch.
if weights_bytes_json["status"] == "success":
    wba_decoded = io.BytesIO(decodebytes(
        bytes(weights_bytes_json["weights"], encoding='ascii')
    ))
    state_dict = torch.load(wba_decoded)
```

Figure 3.4.1.4.5: Deployment pipeline: download and decode the adequate neural network weights

3.4.1.5 MA-DRL algorithms

3.4.1.5.1 DRL algorithmic support in the platform

The MA-DRL platform introduces a new type of algorithms: the Imitation Learning (IL) approach. Agents trained with RL need to explore numerous strategies before finding the one that optimize the long-term return.

In numerous application cases, it may be long to train an agent with RL because of the execution time of the training environment. It may also be hard to keep the training environment up and stable long enough. IL can prove useful to exploit expert demonstrations (or an expert baseline implementation) to train a decision-making agent. In the MA-DRL platform, the Behavioural Cloning (BC) algorithm is implemented to learn from an expert.

Figure 3.4.1.5.1.1 describes the process of a BC experiment. It is comprised of two phases:

1. Data collection where we build a training and validation database by running the expert agent on instances of the targeted environment. This step is conducted in TheRLib by filling the *Offline buffer* data structure. Alternatively, as in ECOM’s integration, it is conducted externally.
2. Behavioural Cloning, where we run the BC trainer that implements supervised learning on the previously generated *Offline buffer*.

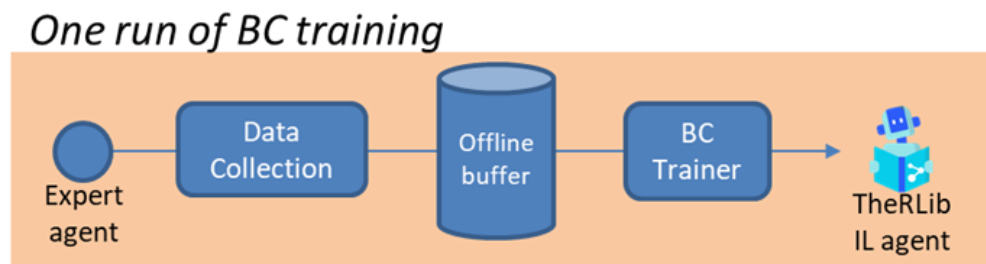


Figure 3.4.1.5.1.1: Behavioural Cloning experiment process

As shown by Figure 3.4.1.5.1.2, the BC experiment is visualised in Therdash in the same fashion than RL experiments, with minor differences:

- The X-axis values represent training epochs instead of learning environment steps. In each training epoch, minibatches are collected with the Offline buffer to optimize the BC loss.
- In the MA-DRL provided example, the training experiment score along the training epoch is computed in a test phase, where test episodes are played using the BL-trained agent on the targeted environment. This is an arbitrary choice and other approaches can be set up when the environment is not accessible during the BC training.

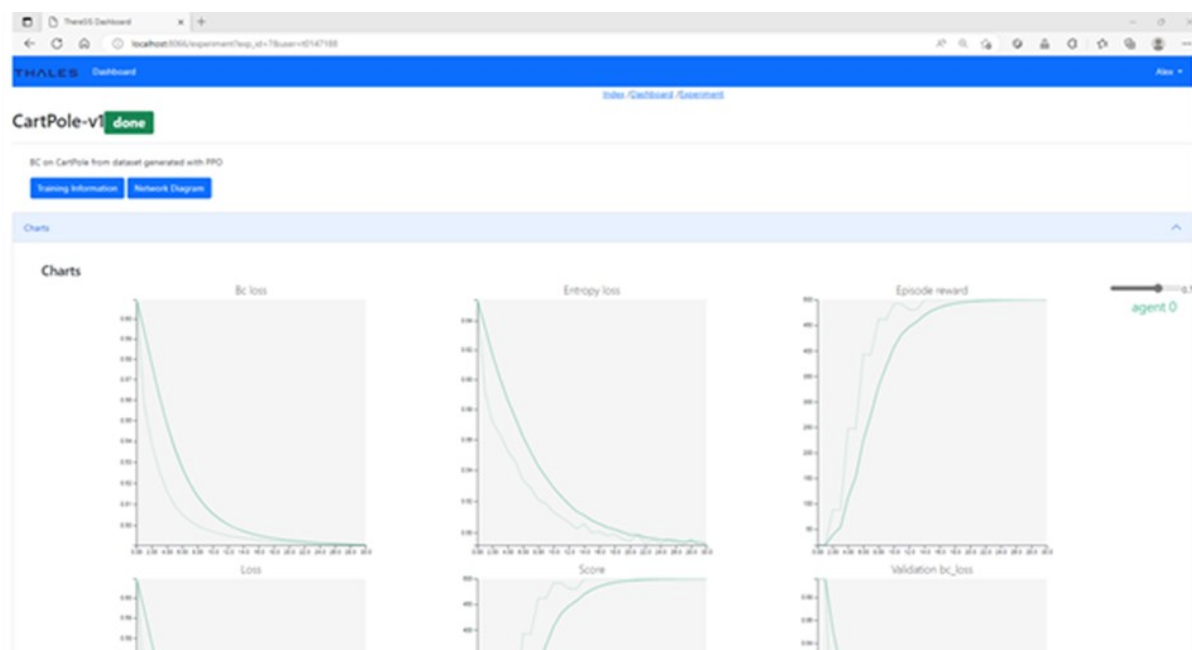


Figure 3.4.1.5.1.2: Behavioural Cloning experiment dashboard view

The final MA-DRL platform also adds to its Reinforcement Learning (RL) library the Deep Q Network (DQN) algorithm. In the TheRLib implementation of DQN, the same architectural principles were used than for the PPO and SAC algorithms: using dedicated Trainer, Agent and generic Model classes.

Since it has been introduced by [8], the DQN algorithm has undergone improvements in the definition of the objective loss and the overall training process. These algorithmic

improvements are explained and benchmarked in [23]. The TheRLib DQN implements the following, that can be toggled on and off and configured as training script parameters:

- Double Q Learning, as introduced in [24].
- Prioritized Experience Replay.
- Multi-step returns.

Besides, Prioritized Experience Replay and multi-step returns are also usable with the SAC implementation which shares a similar architecture than DQN, and leverages an experiment replay memory.

3.4.1.5.2 AI algorithms for RAN Slicing

Please refer to Section 3.1.1 for a brief description, or D5.2 Section 3.2 for the full description of the Hyper Autoscaling (HAS) algorithm.

To integrate the TheRLib MA-DRL platform with the RAN Slicing use case, an offline learning approach was initially envisioned due to the harsh real-time and stability constraints of ECOM's OAI platform. The main steps for a proof-of-concept of this approach were:

- Collecting of a training dataset of agent-environment interactions using ECOM's Dynamic Autoscaling agent and OAI RF-SIM environment.
- Collecting of a validation dataset of agent-environment interactions using ECOM's Dynamic Autoscaling agent and OAI USRP environment.
- Training of a TheRLib agent using Imitation Learning (IL) algorithms with the collected datasets on the TheRLib MA-DRL platform.
- Testing of the TheRLib IL agent on OAI USRP environment with the TheRLib lightweight deployment component.

As this plan could not be implemented in due time, DQN training was performed on a simplified Gym environment of the OAI simulation, as reported in Section 4.5.

3.4.1.5.3 RAN Radio Link Control DRL algorithms

Matlab Network Simulation of the Reinforcement Learning Radio Link Control is described in Deliverable D5.2 Section 6.3. The Markov Decision Process (MDP) Radio Link Control model using Matlab Reinforcement Learning toolbox is described in this D5.3 Section 5.3.7. How they relate to each other is illustrated in Figure 3.4.1.5.3.1. Each was modelled independently with internally generated MCS up/down/same actions and BLER result. These need to be integrated with each other with 12 instances of the Matlab MDP for each UE.

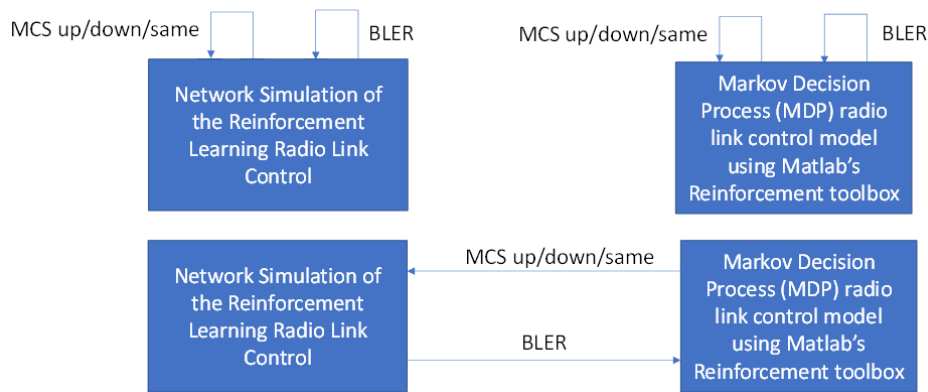


Figure 3.4.1.5.3.1: Relationship between Network Simulation, Matlab's Markov Decision Process Model

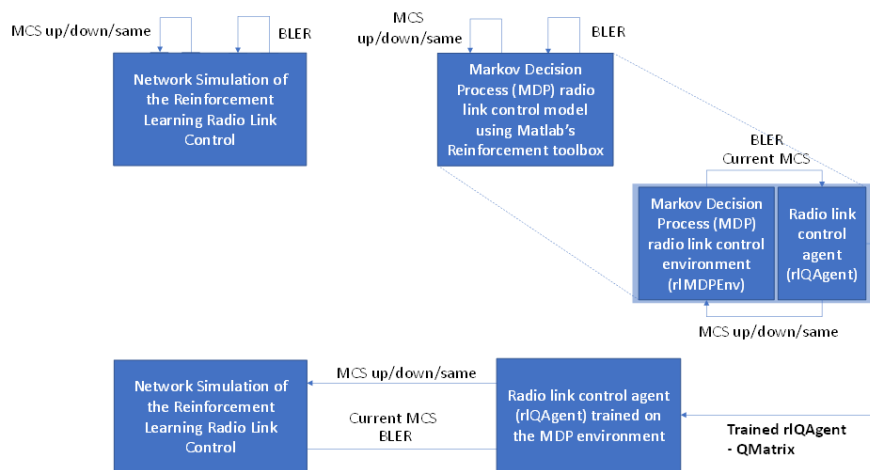


Figure 3.4.1.5.3.2: Relationship between Network Simulation, Matlab's Markov Decision Process Model and TheRLib MA-DRL platform

The high-level architecture of a pure-MATLAB integration is as shown in Figure 3.4.1.5.3.2 and in more detail in Figure 3.4.1.5.3.3. The “Markov Decision Process (MDP) radio link control model using Matlab's Reinforcement toolbox” is broken down into the “environment” and “agent” objects that are used by the MATLAB RL toolbox and the trained “agent” is used for with the Network Simulation. The small-dimensional agent trained on the MDP model, which is leveraged to cope with performance and scale because 12 instances of the Agent, is required for each of the 12 User Equipment in the Network Simulation.

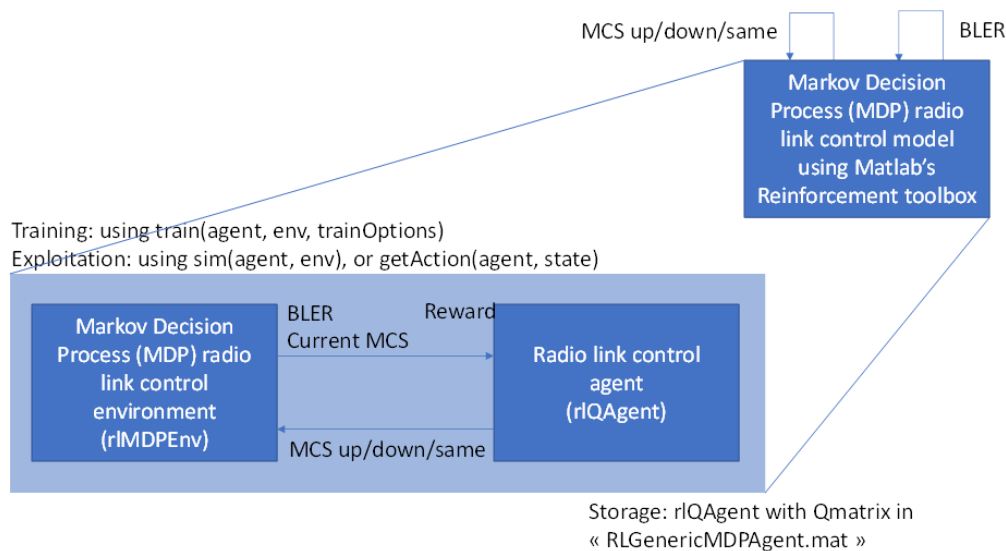


Figure 3.4.1.5.3.3: Detail of Matlab MDP model

TheRlib can also be used to train and deploy the agent using Markov Decision Process as shown in Figure 3.4.1.5.3.4. Note that TheRlib interacts with the MDP “environment” introduced in the previous slide and manages the training process instead of the MATLAB RL toolbox. When interacting with the Network Simulation, the “TheRlib MA-DRL Deployment component” code is in a simple script that loads a serialized TheRlib agent from a data file, instantiates the MATLAB RL environment and performs the agent-interaction environment. The merit of this approach is that an instance of TheRlib agent has a much smaller footprint and computes much more efficiently when using higher-dimensional states than the MATLAB RL toolbox version of it, so can be used to train and deploy many of them such as the 12 UEs required in the network simulation.

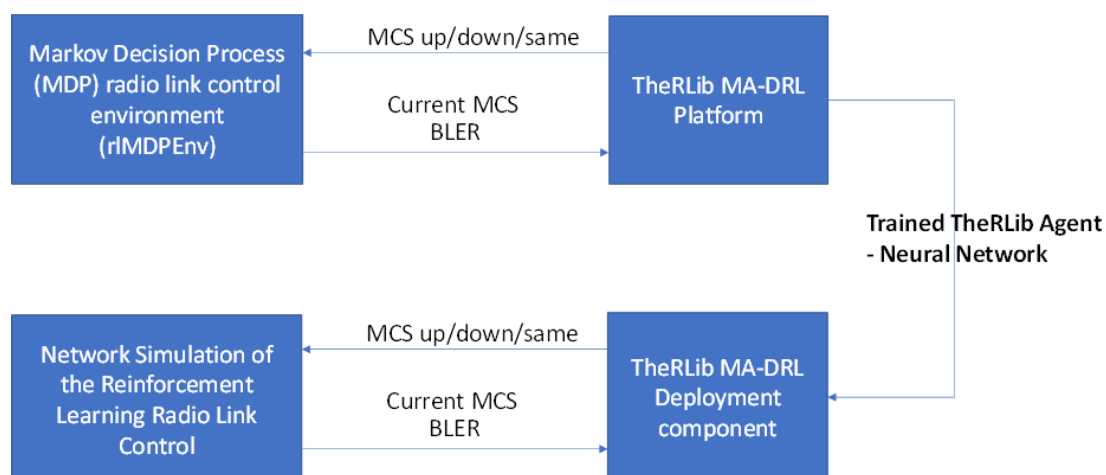


Figure 3.4.1.5.3.4: Applying TheRlib MA-DRL platform for both training and deployment of MDP

TheRlib can be used to train and deploy the agent using the soft actor critic Neural Network model, or alternatively the DQN model, as opposed to the Markov Decision Process model as shown in Figure 3.4.1.5.3.5. Here the merit is that multiple states can be incorporated such as for example MCS/CQI states for Radio Link Control and Resource Block Capacity states for a Slice Capacity Control system. Training can be performed with the MDP radio link control

environment, or a refined version with more complex states, if it provides the RL Environment interface, i.e., step and reset functions.

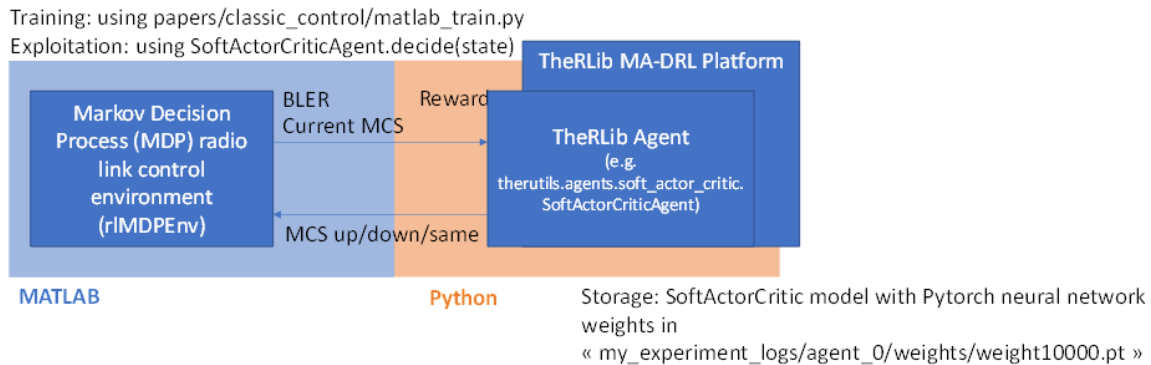


Figure 3.4.1.5.3.5: Applying TheRLib MA-DRL platform for Neural Network training and deployment

Alternatively, training can be performed with the Network Simulation if it provides the RL Environment interface, i.e., step and reset functions, as shown in Figure 3.4.1.5.3.6.

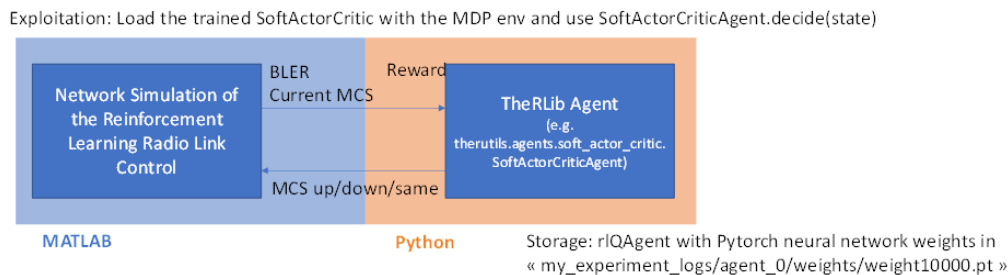


Figure 3.4.1.5.3.6: Applying TheRLib MA-DRL platform for Neural Network training and deployment

Please note that the work of applying MATLAB's RL toolbox to Radio Link Control model was performed as research informed teaching MSc student project by Prince Kwaku Boakye [21] under the guidance and supervision of 6G BRAINS researchers Kareem Ali and John Cosmas, with the objective of automating the RL design process that will be performed in WP 5.

The work of integration of Matlab Radio Link Control to TheRLib and Network simulation is being performed as research informed teaching BEng student project by John Miguel under the guidance and supervision of 6G BRAINS researchers Kareem Ali and John Cosmas, with the objective of automating the RL design process that performed in WP 5.

4 Final system prototyping

This section describes the details of the final integration of the sub-systems, components and enablers delivered as outcomes in WP5, across the various tasks with the goal of providing End-to-End Network Slicing capabilities in 5G and beyond infrastructures.

4.1 Overall approach

The final integration is conducted over a physical testbed deployed in UWS premises. The main technical integration approach includes the following:

- Integration for E2E Network Slice control enablers across RAN and non-RAN segments,
- Integration for E2E Network Slice management and orchestration especially between the ONAP-based framework, and the Slice Control Agent (SCA) architecture.
- Integration of TheRLib MA-DRL platform to support AI-based RAN slice control.

The secondary integration is performed between TheRLib MA-DRL platform and the MatLab-based RAN link control.

4.2 Testbed for integration

Figure 4.2.1 depicts a high-level overview of the testbed deployed at UWS premises. This testbed is designed to integrate the components and slice enablers, both software and hardware, provided by the 6G BRAINS partners involved in WP5 to achieve an E2E Network Slicing. It comprises the four network segments illustrated in the figure: RAN, Edge, Core and transport plus a cloud management layer on top of the physical cloud infrastructure (i.e., the 6G data plane).

The testbed infrastructure is composed of the following equipment:

- “One Plus 8T” 5G Smartphones as **User Equipment (UE)**
- Regarding the **Radio Access Control (RAN)**, Ettus USRP B210 devices are acting as base stations providing continuous RF coverage from 70 MHz to 6 GHz, full duplex operation, USB 3.0 high speed connectivity, GNURadio and BTS compatibility via the open-source USRP hardware driver (UHD) among other features.
- The **transport** segments use 10 Gbps Netgear switches to interconnect Edge nodes, Core nodes and management computers.
- Regarding the **Edge and the cloud-based Core** segments, they are composed by nodes with the following technical specifications: Dell T56810 with Intel Xeon E5-2630 v4 CPU, 10 cores with hyper-threading, 32768 MBytes of RAM and a 512 GBtyes SSD HD.

Concerning the provisioning of physical resources and cloud management, the following tools and frameworks are deployed in dedicated computers for management and system administration purposes (see management and control layer in Figure 4.2.1):

- **MaaS** version 2.9.33 (Metal as a Service) as bare metal cloud providing on-demand servers with super-fast deployment of physical and virtual nodes from scratch.²
- **Juju** version 2.9 as orchestration framework enabling the deployment, integration, and life cycle management of cloud applications.³
- **OpenStack** Wallaby distribution as cloud infrastructure management framework providing Infrastructure as a Service (IaaS). Multiple OpenStack applications are deployed for networking management, images service, block storage management, compute node management, message broker middleware, or identity service just to mention a few.⁴

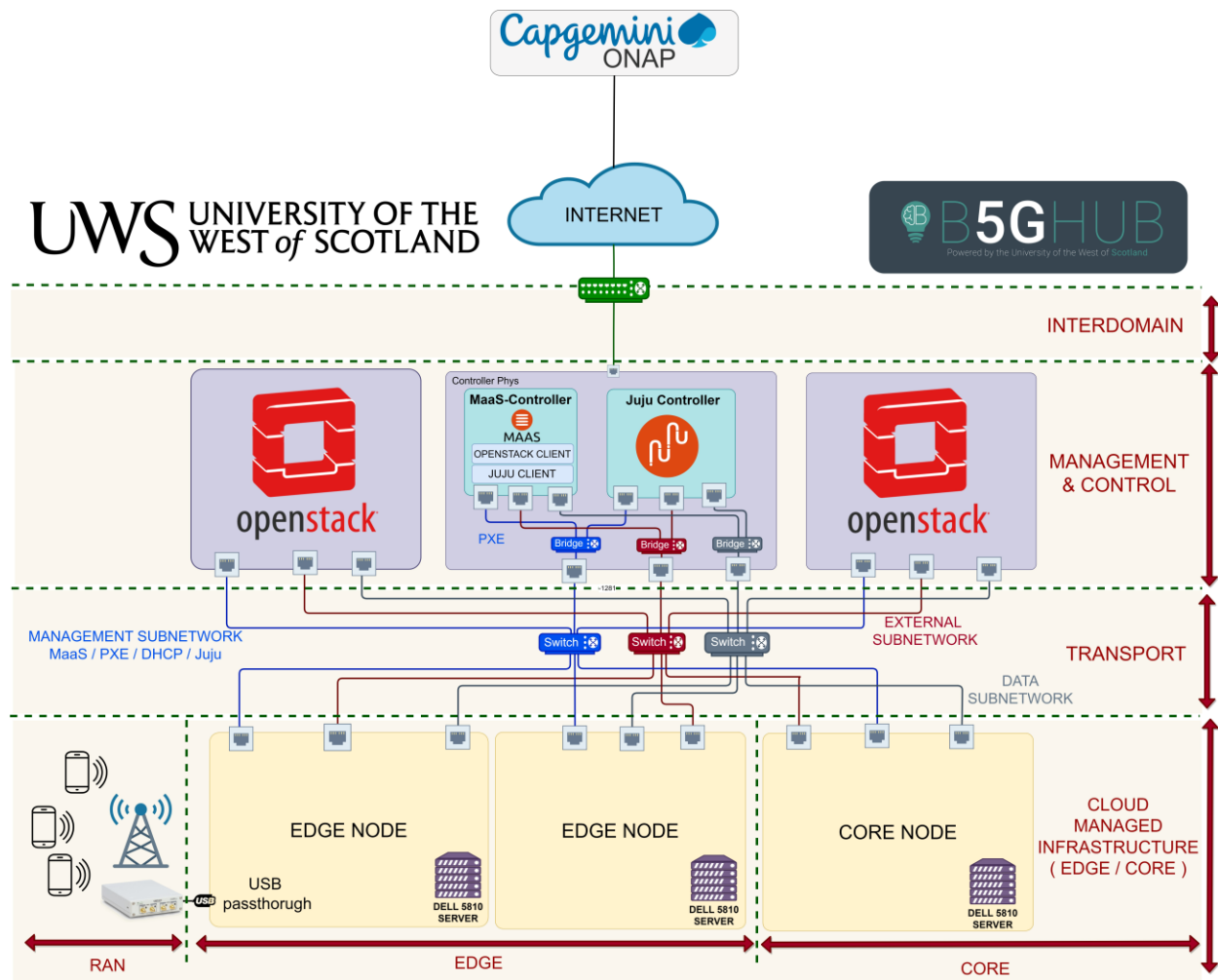


Figure 4.2.1: High level view of the 6G-BRAINS testbed for WP5 integration

Because of the complex nature of the coexisting technologies involved in the infrastructure, the following improvements and innovations have been implemented in order to get a fully operative testbed:

² <https://maas.io/>

³ <https://juju.is/>

⁴ <https://www.openstack.org/>

- Enhancement of the source code of the Openstack nova-compute application, introducing new features:
 - To allow the setup of on-demand escalated privileges when deploying virtual instances based on LXC. This feature is needed to install and deploy Open Air Interface RAN (OAI-RAN) in LXC containers managed by Openstack.
 - To allow USB devices pass-through from physical hosts to Linux containers (LXC) managed by Openstack. This new capability is required to enable connectivity between the antenna (the USRP device) and the software Radio Access Network (OAI-RAN running in a LXC instance).
- Implementation of a new Linux system service named **“USB-USRP Monitoring”**. The function of this service is to verify the connection between the USRP and the LXC container running the RAN is established. That is, that the USB pass-through is being performed correctly. The service detects when a USRP is connected/disconnected to the physical host and automatically connects/disconnects it to the RAN according to the set up configuration.

The following is a comprehensive description of the WP5 6G BRAINS components, RAN, and CORE network functions deployed on the Edge and Core nodes of the UWS testbed.

First, a more detailed inspection of the node hosting the 6G CORE shows that OAI Core Network (CN) has been deployed, providing a 3GPP-Compliant Standalone (SA) CN implementation (see Figure 46). It supports different core network functions such as UPF, AMF, SMF or AUSF. The 6G CORE enables the user data forwarding via the User Plane Function (UPF) and provides a set of control and management functions for authentication, user mobility, handover management, or user registry just to mention a few.

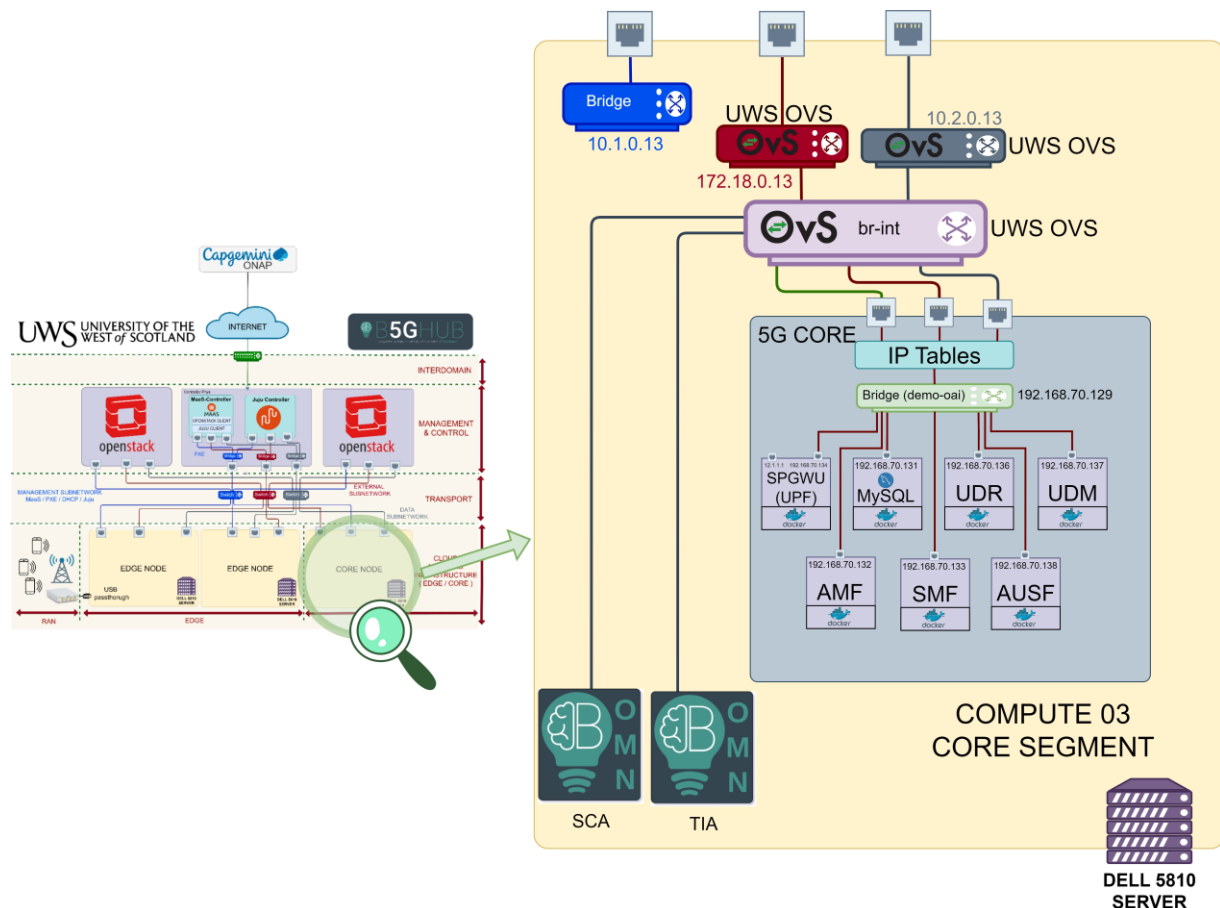


Figure 4.2.2: CORE compute node for 6G-BRAINS integration

Secondly, taking a more detailed look at the computer deployed on the Edge segment (see Figure 4.2.3) where the RAN is installed, we notice the following:

- OAI-RAN is deployed in a Linux container (LXC) instance managed by OpenStack, providing a 3GPP compatible gNB Radio Access Network (RAN).
- The antennas (USRP devices) are connected to the RAN via the new USB pass-through capability previously described.
- There are two more LXC instances deployed along with the RAN LXC:
 - One container hosting FLEX-RIC API and the Reinforcement-Learning pipeline provided by Thales.
 - One container with a database server storing the AI-MODEL needed for the RL pipeline.

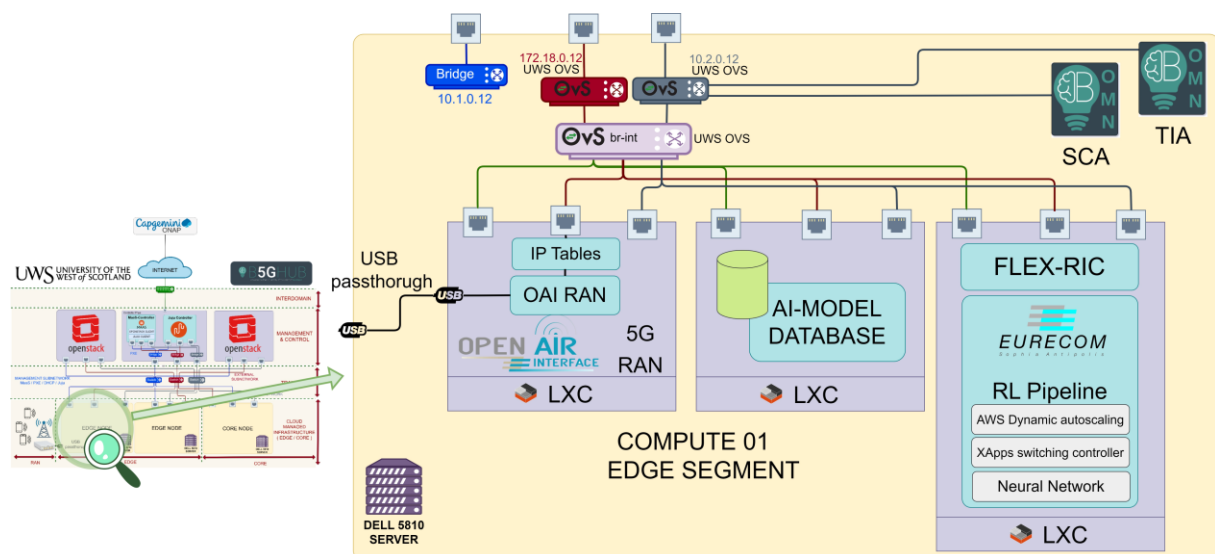


Figure 4.2.3: RAN compute node deployed in Edge segment for 6G-BRAINS integration

Finally, as depicted in Figure 4.2.4, a Virtual Machine hosting the Thales RLib is allocated in a compute node deployed in the Edge network segment. It can be noticed in Figure 48 the three main architectural subcomponents of RLib:

- Thales Training Workflow
- Thales Deployment Workflow
- Model Monitoring

Finally, it is worthy to highlight that an Slice Controller Agent (SCA) instance is present in all the nodes composing the 6G BRAINS testbed. As further described in the following section, the SCA component plays an key role in the 6G BRAINS integration for providing End-to-End Network Slicing. The SCA provides an technology-independent abstract view of the 6G data plane enabling extended programmability for different underlying technologies coexisting in the data plane.

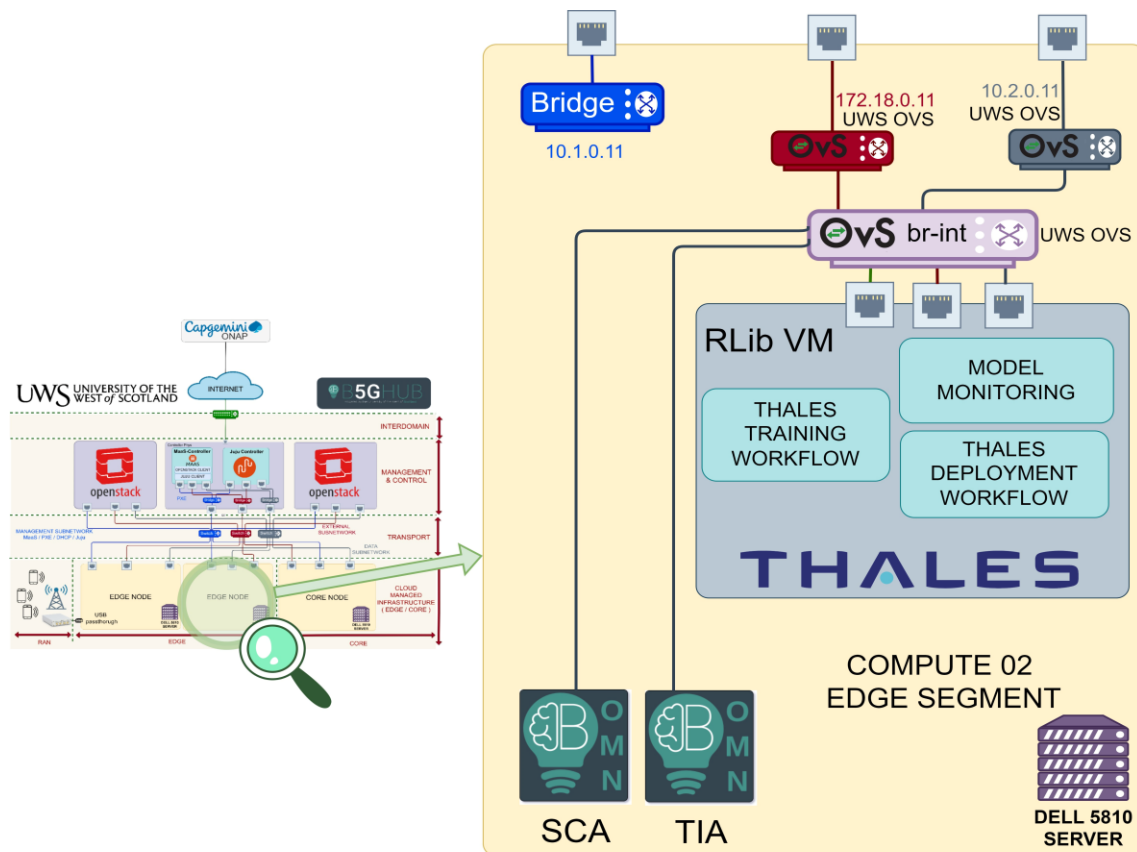


Figure 4.2.4: Compute deployed in Edge segment hosting THALES RLib 6G-BRAINS component

4.3 Integration for E2E Network Slice control enablers

Different programmable networking devices have been explored, improved, tested, and evaluated at various stages of WP5. These devices are considered "slice enablers" since they possess the ability to provide network flows with specific, tailored QoS configurations at the data-plane level.

The next step in the process is integrating these devices with the Slice Control Agent (SCA), which will enable the separation of control and data planes. This separation is key to optimizing the system for simplicity, performance, and resilience. The SCA was first introduced in Deliverable D5.2 (section 7.2.1). It supports interactions with a variety of software and hardware programmable technologies via a north-bound unified and common API. The SCA utilizes a data-plane abstraction service, which simplifies the architecture's upper layers by creating queuing schemes, QoS policies, and advanced filtering expressions. It does this while using a dedicated slice enabler in its south-bound interface to handle technology-dependent logic.

As a result of the SCA integration, the upper management layers of the 6G Brains architecture can seamlessly utilise and merge various Network Slice enablers as required. This facilitates the design and implementation of E2E slices throughout the network architecture, regardless of the slicing technology present in each network segment. This process is executed in a consistent, integrated, and transparent manner, eliminating the need for manual configurations for diverse technologies. Figure 4.3.1 depicts the logical architecture of the SCA

and its associated Slice Enablers. The bottom part of the figure illustrates the distribution of these Slice Enablers across different network segments. Table 4.3.2 presents the current integration status of all the different Slice Enablers approached in 6G Brains, including those previously reported and the ones included in this deliverable.

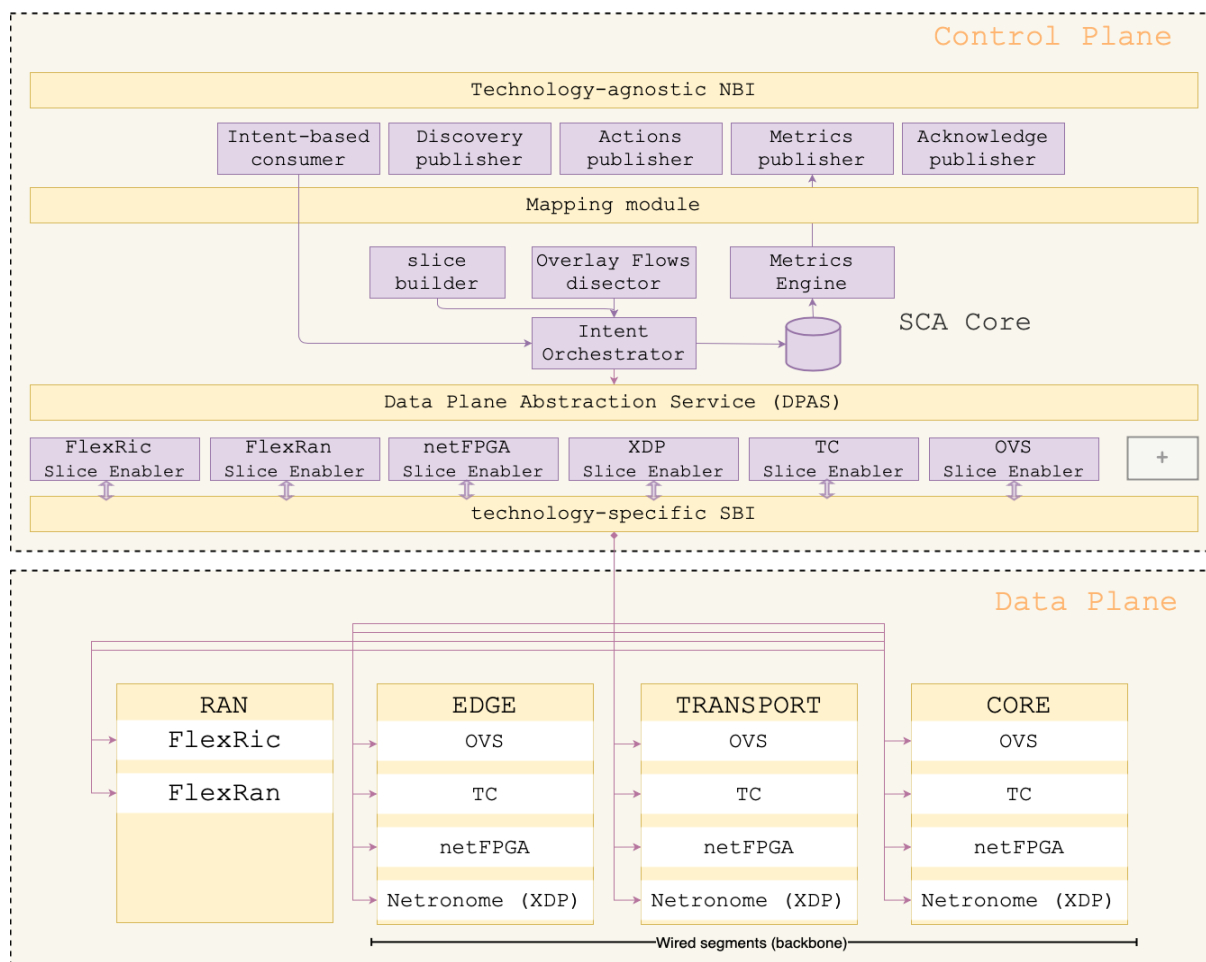


Figure 4.3.1: Slice Control Agent (SCA) - Logical Architecture

Slice control enabler	Reported in deliverable(s)	Adapted for 6G data-plane	Slice Enabler Implementation	E2E support	SCA integration
Traffic Control (TC)	D5.1	Done	Done	Done	Done
Open Virtual Switch (OVS)	D5.1, D5.2	Done	Done	Done	Done
P4-based NetFPGA (smartNIC)	D5.1	Done	Done	Done	Done
XDP-based Netronome (smartNIC)	D5.2	Done	Done	Done	Future work
FlexRIC	D5.1, D5.2	Done	Done	Done	Done

Table 4.3.1: Integration Status of Slice Control Enablers with the SCA

4.3.1 Integration of SCA and FlexRIC

As part of the progressive implementation throughout this project, this section introduces the integration of the final Slice Enabler approached in 6GBrains, the FlexRIC. This will facilitate the Slice control and monitoring of the Radio Access Network. Here, it is explained the details and solutions employed to integrate the SCA within a Slice Enabler in the Radio network segment. This aims to enable end-to-end control and monitoring, adhering to the approach previously adopted for other Slice enablers in the data-plane.

For this objective, UWS has developed a component designed to serve as an interface between the SCA and the RAN controller. The RMC (RAN Monitor & Control) has been created to incorporate the core functionalities of the FlexRIC —a flexible O-RAN compliant RIC and a RAN agent running on top of the OpenAirInterface (OAI) radio software stack— for the efficient management and coordination of various xApps. Through this approach, the complexity of managing the Radio Intelligent Controller (RIC) is abstracted, resulting in more streamlined operations. Note that while the SCA primarily utilizes the Control xApps from the catalogue, the RMC has been equipped with monitoring capabilities (Monitoring xApps). This allows it to, if required, be accessed by other components of 6GBrains for the purpose of monitoring the Radio Segment.

Figure 4.3.1.1 shows how the architecture of the RMC component is deployed to allow the control of the RAN segment. With it, different xApps can be managed in parallel, as well as monitor the metrics received from the RAN. xApps are plug-and-play components that implement specific logic, often for RAN data analysis and control. These xApps can receive data and telemetry from the RAN and return control directives via the E2 interface. An xApp descriptor (e.g., a YAML or JSON file) provides information on the parameters required to manage the xApp, such as policies for deployment, deletion, and upgrade specifics.

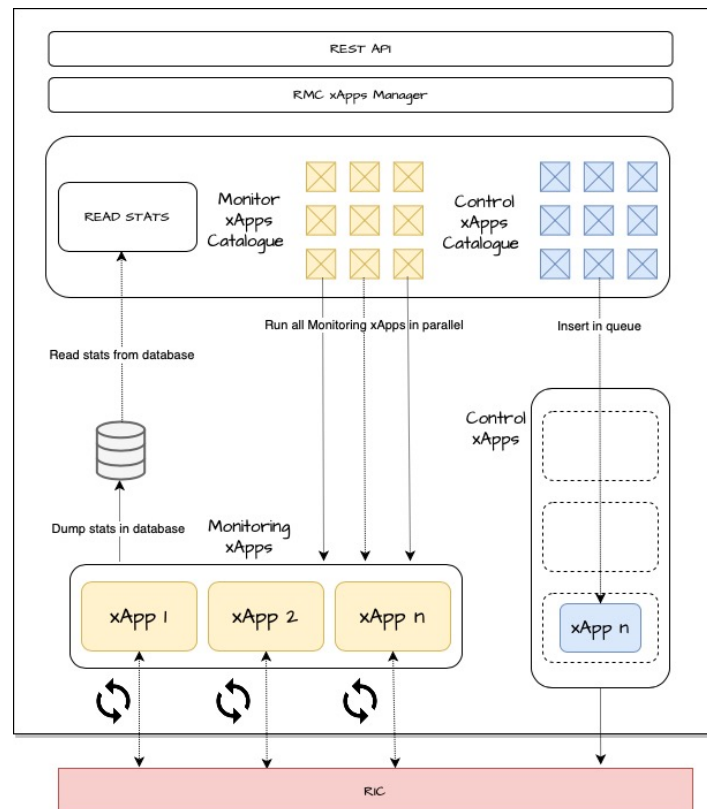


Figure 4.3.1.1: RMC architecture

The RMC is equipped with a catalogue of xApps specifically tailored for tasks like slice creation, update, and deletion (related to slicing control). Core features of the RMC can be easily accessed by the SCA via the interface shown at the top of the Figure 4.3.1.1 and the capabilities these functionalities offered are listed in Figure 4.3.1.2. The API exposes endpoints to allow seamless communication with other components. These endpoints represent specific functions, methods, or data collections within the API, serving as interaction points where requests are received, and responses are returned. Note that while the SCA primarily utilizes the Slice Control endpoints, the RMC has been equipped with monitoring endpoints for other purposes.

Monitor

GET	/monitor/topology/ran/	Get a JSON with the RAN topology retrieved by FlexRIC.	rmc.api.backend.get_ran_topology
GET	/monitor/stats/slices/	Get stats of all active slices in the RAN reported by FlexRIC.	rmc.api.backend.get_slice_stats
GET	/monitor/stats/slice/{slice_id}	Get stats from specific slice.	rmc.api.backend.get_slice_stat

Control

POST	/control/insert/slice/	Add a new Slice.	rmc.api.backend.insert_slice
POST	/control/insert/associate_ue/	Associate UE to an existing Slice.	rmc.api.backend.associate_ue
DELETE	/control/delete/slice/{nb_id}/{slice_id}	Delete an existing slice in a given E2 Node	rmc.api.backend.delete_slice
DELETE	/control/reset/slices/{nb_id}	Reset all existing slices in a given E2 Node	rmc.api.backend.reset_slices

Figure 4.3.1.2: RMC endpoints

The endpoints explained below are tuned to the slice control panel since the SCA is responsible only for the management of the RAN.

4.3.1.1 Add a new slice related to a base station node

POST	<IP>:<port>/api/slice/control/insert/slice/
Parameters	No parameters
JSON example	<pre>{ "e2_node": 0, "num_slices": 1, "slice_sched_algo": "NVS", "slices": [{ "sense": "dl", "id": 0, "label": "slice-0", "ue_sched_algo": "PF", "type": "SLICE_SM_NVS_VO_RATE", "slice_algo_params": { "mbps_rsvd": 60, "mbps_ref": 120 } }] }</pre>
Response	200 -> slice created

Table 3.4.2.1: Add a new slice related to a base station node

POST	<IP>:<port>/api/slice/control/insert/associate_ue/
Parameters	No parameters
JSON example	<pre>{ "e2_node": 0, "num_ues": 1, "ues": [{ "rnti": 0, "assoc_dl_slice_id": 2, "assoc_ul_slice_id": 2 }] }</pre>
Response	200 -> UE successfully associated to the slice
	400 -> Bad request
	401 -> Unauthorized. Please check you authentication credentials.

	403 -> Forbidden
	404 -> Not found

Table 3.4.2.2: Attach new UE to a slice

POST	<IP>:<port>/api/slice/control/delete/slice/nb_id/slice_id
Parameters	No parameters
JSON example	<pre>{ "e2_node": 0, "num_dl_slices": 1, "delete_dl_slice_id": [5] }</pre>
Response	200 -> Slice successfully deleted
	400 -> Bad request
	500 -> Internal server error

Table 3.4.2.3: Deleting an existing slice related to a base station node

POST	<IP>:<port>/api/slice/control/delete/slices/nb_id
Parameters	No parameters
JSON example	<pre>{ "e2_node": 0, }</pre>
Response	200 -> Slices deleted
	400 -> Bad request
	500 -> Internal server error

Table 3.4.2.4: Deleting all slices related to a base station node

4.4 Integration for E2E Network Slice management and orchestration

This section we detail the System Integration between the different components being developed in the context of the WP5 of 6G BRAINS towards the creation of a common stack for E2E Management and Orchestration. In specific, this section details the integration between ONAP and UWS Network Slice Manager and the integration of ONAP and the Virtual Industrial Assistant

4.4.1 Integration of ONAP and Network Slice Manager

4.4.1.1 UWS Slice Manager API

This subsection provides the basic actions that the UWS Slice Manager supports. All the actions contain different examples with description messages and responses.

The description of the data model used to represent the network topology is described in Section 7.3.1 of Deliverable 5.2.

4.4.1.1.1 Get Topology

GET	/rest/CAPGEMINI/api/topology <i>Get the existing topology elements from the infrastructure</i>
Parameters <i>No parameters</i>	
Response <div> 200 ok <pre> { "elements": [{ "iface": "enp4s0f1", "deviceHostName": "edge1", "resourceId": "3B93F4B0", "resourceType": "DEVICE_PORT", "hostName": "br-enp4s0f1", "programmableResourceId": null, "deviceType": "SOFTWARE_SWITCH", "mac": "e4:11:5b:11:cb:78", "ipv4": null, "ipv4Mask": null, "ipv6": null, "ipv6Mask": null, "portId": "5", "portUuid": null, "networkId": null, "state": "UP" }, { "iface": "enp4s0f1", "deviceHostName": "edge1", "resourceId": "4B0E5E27", "resourceType": "DEVICE_PORT", "hostName": "edge1", "programmableResourceId": "910F1AB4", "deviceType": "PHYSICAL_HOST", "mac": "e4:11:5b:11:cb:78", "ipv4": null, "ipv4Mask": null, "ipv6": null, "ipv6Mask": null, "portId": "5", </pre> </div>	

```

        "portUuid": null,
        "networkId": null,
        "state": "UP"
    },
    {
        "iface": "481xd7-0",
        "deviceHostName": "edge1",
        "resourceId": "84978EA6",
        "resourceType": "DEVICE_PORT",
        "hostName": "br-enp4s0f1",
        "programmableResourceId": null,
        "deviceType": "SOFTWARE_SWITCH",
        "mac": "c2:b9:b5:4a:ca:21",
        "ipv4": null,
        "ipv4Mask": null,
        "ipv6": null,
        "ipv6Mask": null,
        "portId": "30",
        "portUuid": null,
        "networkId": null,
        "state": "UP"
    },
    {
        "iface": "481xd7-0",
        "deviceHostName": "edge1",
        "resourceId": "F5D2C360",
        "resourceType": "DEVICE_PORT",
        "hostName": "edge1",
        "programmableResourceId": "35FB70F6",
        "deviceType": "PHYSICAL_HOST",
        "mac": "c2:b9:b5:4a:ca:21",
        "ipv4": null,
        "ipv4Mask": null,
        "ipv6": null,
        "ipv6Mask": null,
        "portId": "30",
        "portUuid": null,
        "networkId": null,
        "state": "UP"
    },
    {
        "iface": "eth0",
        "deviceHostName": "edge1",
        "resourceId": "6104F0C9",
        "resourceType": "DEVICE_PORT",
        "hostName": "juju-be4db4-48-1xd-7",
        "programmableResourceId": "9CBB1D98",
        "deviceType": "VIRTUAL_HOST",
        "mac": "00:16:3e:68:b0:8a",
        "ipv4": "10.10.0.139",
        "ipv4Mask": "23",
        "ipv6": "fe80::216:3eff:fe68:b08a",
        "ipv6Mask": "fe80::216:3eff:fe68:b08a",
        "portId": "29",
        "portUuid": null,
        "networkId": "c67c3342-d327-a53d -c3a2-a3a4d267a89a,",
        "state": "UP"
    },
    {
        "srcResourceId": "4B0E5E27",
        "dstResourceId": "3B93F4B0",
        "resourceId": "E0312557",

```

```

        "resourceType": "CONNECTION"
      },
      {
        "srcResourceId": "F5D2C360",
        "dstResourceId": "84978EA6",
        "resourceId": "DCC11C10",
        "resourceType": "CONNECTION"
      },
      {
        "srcResourceId": "F5D2C360",
        "dstResourceId": "6104F0C9",
        "resourceId": "5B55F056",
        "resourceType": "CONNECTION"
      }
    ]
  }
}

```

Table 3.4.2.1: Get Topology**4.4.1.1.2 Create Slice****NOTE:**

- *maxBandwidth* and *minBandwidth* are passed in bits.
- The *sliceId* value in the response is a string (it will have the same value as *sliceName*).

POST <code>/rest/CAPGEMINI/api/slice/</code> <i>Create a new slice instance</i>	
Parameters <i>No parameters</i>	
Body	application/json
<pre> { "sliceName" : "uws-capgemini-slice1", "maxBandwidth" : "125000000", "minBandwidth" : "25000000", "priority" : "2", "resources": ["programmableResourceId1", "programmableResourceId2", ..., "programmableResourceIdN"] } </pre>	
Response	application/json
200	ok
<pre> { "sliceId": "uws-capgemini-slice1", "sliceName" : "uws-capgemini-slice1" } </pre>	
500	error
<pre> { "description" : "Slice cannot be created. Duplicated sliceName" } </pre>	

```

500    error

    {
      "description" : "Slice cannot be created. Unexpected error"
    }

```

Table 3.4.2.2: Create Slice

4.4.1.1.3 Get Slice

GET <code>/rest/CAPGEMINI/api/slice/{sliceId}</code>	
<i>Get specific slice instance given a slice ID (sliceId) and its' state.</i>	
Parameters	
<ul style="list-style-type: none"> sliceId: unique identifier of the specific slice instance (e.g., uws-capgemini-slice1) 	
Response	application/json
200 ok { "sliceId" : "uws-capgemini-slice1", "sliceName" : "uws-capgemini-slice1", "maxBandwidth" : "125000000", "minBandwidth" : "25000000", "priority" : "2", "resources": ["programmableResourceId1", "programmableResourceId2", ..., "programmableResourceIdN"] "health" : "Ok" }	
404 fail { "description": "Slice 'uws-capgemini-slice1' not found!" }	

Table 3.4.2.3: Get Slice

4.4.1.1.4 Get All Slices

GET <code>/rest/CAPGEMINI/api/slices</code>	
<i>Get all existing slice instances and their state.</i>	
Parameters	
<i>No parameters</i>	
Response	application/json
200 ok	

```
[
  {
    "sliceId" : "uws-capgemini-slice1",
    "sliceName" : "uws-capgemini-slice1",
    "maxBandwidth" : "125000000",
    "minBandwidth" : "25000000",
    "priority" : "2",
    "resources":
    ["programmableResourceId1","programmableResourceId2",...,"programmableRe
sourceIdN"]
    "health" : "Ok"
  },
  {
    "sliceId" : "uws-capgemini-slice2",
    "sliceName" : "uws-capgemini-slice2",
    "maxBandwidth" : "125000000",
    "minBandwidth" : "25000000",
    "priority" : "3",
    "resources":
    ["programmableResourceId1","programmableResourceId2",...,"programmableRe
sourceIdN"]
    "health" : "Ok"
  }
]
```

Table 3.4.2.4: Get All Slices

4.4.1.1.5 Attach Service to Slice

NOTE:

- *l4Protocol* valid values are: 17 for udp, 6 for tcp.
- *encapsulationType* valid values are: "geneve", "vxlan", "gtp".
- All fields are strings.
- The ID of the service attached is created by the *UWS Slice Manager* and returned in the response. It will be used to detach the service from the slice.

<p>POST <code>/rest/CAPGEMINI/api/attach/{sliceId}</code></p> <p><i>Attach resource to an existing slice</i></p>	
<p>Parameters</p> <ul style="list-style-type: none"> • <i>sliceId</i>: unique identifier of the specific slice instance (e.g., <i>uws-capgemini-slice1</i>) 	
<p>Body</p> <pre>{ "srcIP": "192.168.100.10", "dstIP": null, "l4Proto": "17", "srcPort": null, "dstPort": "5005", "encapsulationType1": null, "encapsulationID1": null, "encapsulationType2": null,</pre>	application/json

<pre> "encapsulationID2": null } } } </pre>	
Response	application/json
200 ok	<pre> { "sliceId": "uws-capgemini-slice1", "sliceName" : "uws-capgemini-slice1", "serviceAttached" : "05E3D221" } </pre>
404 fail	<pre> { "description": "Slice id uws-capgemini-slice1 not found" } </pre>

Table 3.4.2.5: Attach Service to Slice

4.4.1.1.6 Get All Attached Services

GET /rest/CAPGEMINI/api/attached <i>Get all existing attached flows</i>	
Parameters <i>No parameters</i>	
Response	application/json
200 ok	<pre> [{ "05E3D221": { "srcIP": "192.168.100.10", "dstIP": null, "l4Proto": "17", "srcPort": null, "dstPort": "5005", "encapsulationType1": null, "encapsulationID1": null, "encapsulationType2": null, "encapsulationID2": null, "sliceId": "uws-capgemini-slice1" } }, { "73D6AE56": { "srcIP": "192.168.100.11", "dstIP": null, "l4Proto": "17", "srcPort": null, "dstPort": "5006", "encapsulationType1": null, </pre>


```

        "encapsulationID1": null,
        "encapsulationType2": null,
        "encapsulationID2": null,
        "sliceId": "uws-capgemini-slice2"
    }
}
]

```

Table 3.4.2.6: Get All Attached Services

4.4.1.1.7 Detach Service

DELETE <code>/rest/CAPGEMINI/api/slice/{sliceId}/service/{serviceId}</code>	
<i>Detach the service with serviceId from slice instance with sliceId</i>	
Parameters <ul style="list-style-type: none"> • sliceId: unique identifier of the specific slice instance (e.g., uws-capgemini-slice1) • serviceId: unique identifier of the specific service (e.g., 05E3D221). Provided in attach method. 	
Response	application/json
200 ok	<pre> { "description": "service with Id '05E3D221' has been detached from sliceId 'uws-capgemini-slice1'" } </pre>
404 error	<pre> { "description": "slice instance with sliceId 'uws-capgemini-slice1' not found" } </pre>

Table 3.4.2.7: Detach Service

4.4.1.1.8 Delete Slice

DELETE <code>/rest/CAPGEMINI/api/slice/{sliceId}</code>	
<i>Remove the slice Instance by id {sliceId}</i>	
Parameters <ul style="list-style-type: none"> • sliceId: unique identifier of the specific slice instance (e.g., uws-capgemini-slice1) 	
Response	application/json
200 ok	

	<pre> { "description": "sliceId 'uws-capgemini-slice1' has been deleted" } </pre>
404	<pre> error { "description": "slice instance with sliceId 'uws-capgemini-slice1' not found" } </pre>

Table 3.4.2.8: Detach Service

4.4.1.1.9 Integration of ONAP and Network Slice Manager

For the integration between ONAP and the UWS Network Slice Manager 3 main components of ONAP were used. Firstly, the SDC for the Service Design and Configuration of the Slice Service. Then SO was used, to create new workflows (see section 3.3.2) using the Camunda for the interaction and operations involving the UWS Network Slice Manager. Finally, the AAI component was used for updating the inventory regarding the components that were part of the E2E Network Slice created in the UWS domain.

In Figure 4.4.1.1.9.1 a diagram showing the overall architecture for this integration is presented. UWS Slice Manager expose APIs allowing to use their functionalities such as to retrieve the existing topology, perform operations regarding the network slice lifecycle management via the APIs. The APIs from UWS Slice Manager are being used on ONAP NSMF, allowing, as explained, the discovery of the current topology and allowing to request the creation of new slices, attachment, and deletion. ONAP NSMF as a client implements the APIs defined section 4.4.1.

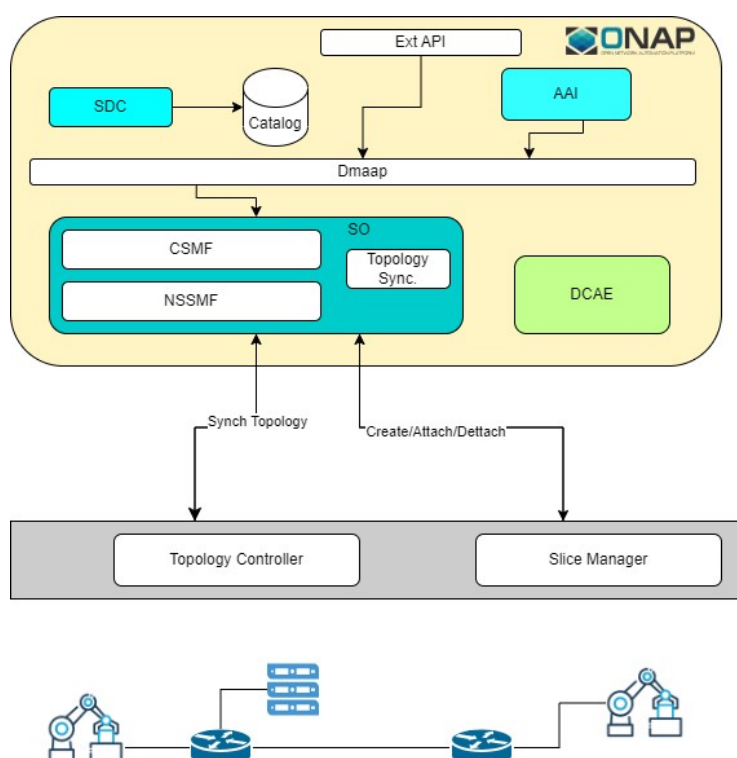


Figure 4.4.1.1.9.1: High Level Architecture Integration ONAP - Topology Controller and Slice Manager

The network slice from the point of view of the management layer, is viewed as an aggregation of resources (physical and virtual interfaces) that are related with an Network Slice instance. From Figure 4.4.1.1.9.2 we can see a topology where physical and virtual interfaces from the infrastructure compose a network slice (represented by the blue line).

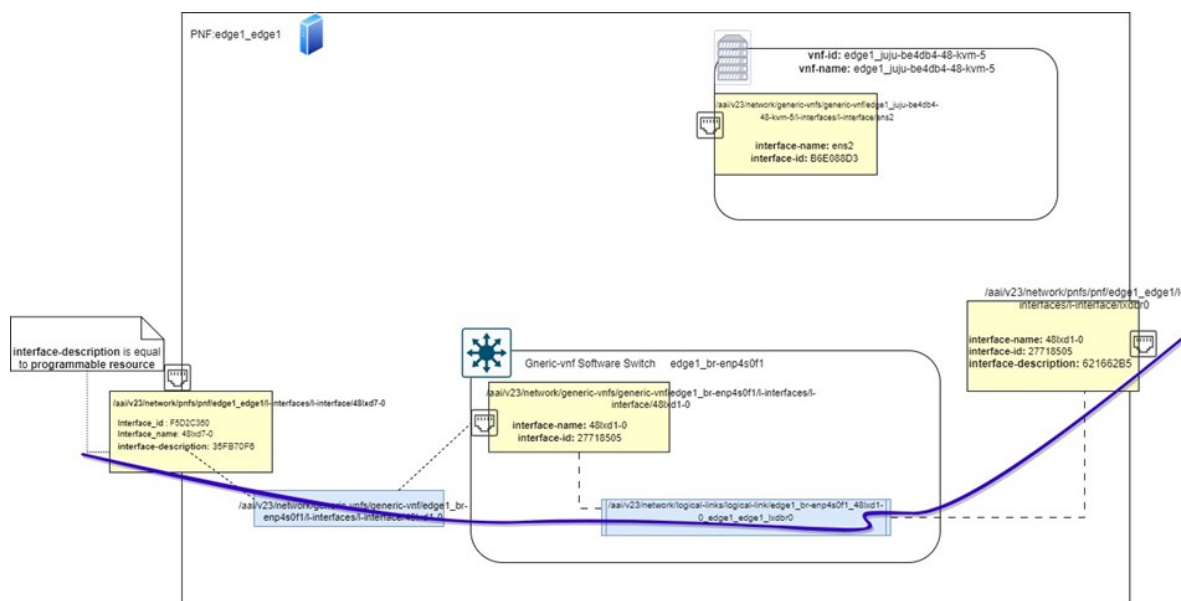


Figure 4.4.1.1.9.2: Topology Slicing View

4.4.2 Integration of ONAP and Virtual Industrial Assistant

Creating a Network Slice via voice commands with a virtual assistant involves combining natural language processing (NLP) and integration with relevant network management systems. Before requesting the network slice, the virtual voice assistant performs several actions such as voice command recognition, where the user verbally requests the creation of a network slice (such as “Create a Network Slice with priority “1” for sector 1 in the factory”). To build the request to create the slice, the topology was logically grouped by sectors in order to facilitate the user to know in which area of the factory is to create a slice. Also, the virtual assistant must retrieve via the catalog REST API the service identifiers available, as well the properties and required fields to send on the creation request.

Below, is represented the request that the virtual voice assistant sends to ONAP NBI.

POST	/nbi/api/v4/serviceOrder <i>Request the creation of a Network Slicing</i>
Parameters	<i>No parameters</i>
Body	application/json

```

{
  "externalId":"slice_service_01",
  "priority":"1",
  "description":"slice_service",
  "category":"Consumer",
  "requestedStartDate":"",
  "requestedCompletionDate":"",
  "relatedParty":[
    {
      "id":"capgemini",
      "role":"ONAPcustomer",
      "name":"capgemini"
    }
  ],
  "orderItem":[
    {
      "id":"1",
      "action":"add",
      "service":{
        "name":"slice_service_01",
        "serviceState":"active",
        "serviceSpecification":{
          "id":"5f506936-bcef-47ac-b97f-6cd8f6e2d1e2"
        }
      },
      "serviceCharacteristic":[
        {
          "name":"maxBandwidth",
          "valueType":"string",
          "value":{
            "serviceCharacteristicValue":"200"
          }
        },
        {
          "name":"minBandwidth",
          "valueType":"string",
          "value":{
            "serviceCharacteristicValue":"100"
          }
        },
        {
          "name":"priority",
          "valueType":"string",
          "value":{
            "serviceCharacteristicValue":"1"
          }
        },
        {
          "name":"sST",
          "valueType":"string",
          "value":{
            "serviceCharacteristicValue":"1"
          }
        },
        {
          "name":"sliceName",
          "valueType":"string",
          "value":{
            "serviceCharacteristicValue":"criticalService"
          }
        },
        {
          "name":"resources",

```

```

        "valueType": "string",
        "value": {
            "serviceCharacteristicValue": "35FB70F6,621662B5,96282965"
        }
    },
    {
        "name": "expDataRateDL",
        "valueType": "string",
        "value": {
            "serviceCharacteristicValue": "100"
        }
    },
    {
        "name": "expDataRateUL",
        "valueType": "string",
        "value": {
            "serviceCharacteristicValue": "100"
        }
    },
    {
        "name": "maxNumberOfUEs",
        "valueType": "string",
        "value": {
            "serviceCharacteristicValue": "1000"
        }
    }
]
}
}
]
}

```

Response

201

```

{
  "id": "6531038c9c1f853f98f34a11",
  "href": "serviceOrder/6531038c9c1f853f98f34a11",
  "externalId": "slice_service_01",
  "priority": "1",
  "description": "slice_service",
  "category": "Consumer",
  "state": "acknowledged",
  "orderDate": "2023-10-19T10:23:08.515Z",
  "completionDateTime": null,
  "expectedCompletionDate": null,
  "requestedStartDate": null,
  "requestedCompletionDate": null,
  "startDate": null,
  "@baseType": null,
  "@type": null,
  "@schemaLocation": null,
  "relatedParty": [
    {
      "id": "capgemini",
      "href": null,
      "role": "ONAPcustomer",
      "name": "capgemini",
      "@referredType": null
    }
  ],
  "orderRelationship": null,

```

```

"orderItem": [
  {
    "orderMessage": [],
    "id": "1",
    "action": "add",
    "state": "acknowledged",
    "percentProgress": "0",
    "@type": null,
    "@schemaLocation": null,
    "@baseType": null,
    "orderItemRelationship": [],
    "service": {
      "id": null,
      "serviceType": null,
      "href": null,
      "name": "slice_service_01",
    }
  }
]

```

Table 4.4.2.1: Service Order Request Add Slice

4.5 Integration of TheRLib MA-DRL platform to support RAN slice control

In Deliverable 5.2, we aimed to develop an OpenAI Gym wrapper for FlexRIC- and OAI-based testbeds. However, this task is unfruitful for two reasons. One, it is almost impossible to safely start and stop FlexRIC, OAI, and their related entities, programmably. Two, the sequential nature of Gym’s rollout and update do not support the near-RT control-loop requirement of Open RAN, i.e., sub-10ms. In particular, preliminary experiments showed that model update takes around 10 seconds, meaning that if we stop rollout to execute update, we would miss around 1000 control inferences, an unacceptable number in production.

Thus, in this Deliverable 5.3, we switch to implement a Gym environment that abstracts OAI and FlexRIC, asking that: *“If one only models key elements of a real RAN system, will the resulting RL policy work on that real RAN?”* To answer this question, we implement a queue-based abstraction of a one UE – one Base Station downlink connection. Briefly stated, we will train a policy that seeks optimal allocation of a UE budget given simple traffic and channel profiles, illustrated in In particular, the traffic arrival is Pareto-based, the channel variations is periodic, and the MCS scheme is 3GPP’s TS 38.214 Table 1.

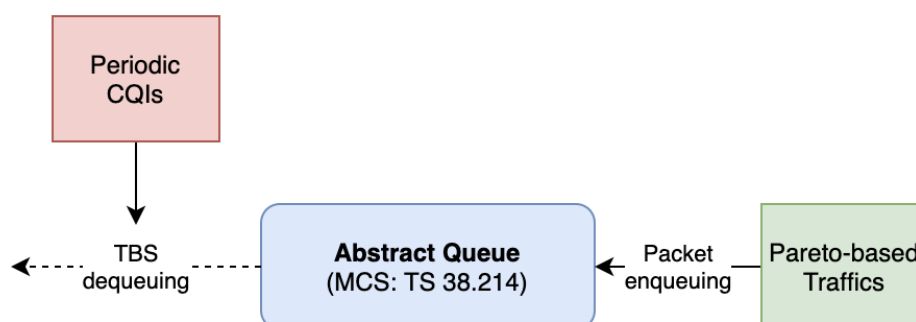


Figure 4.5.1: The inner mechanism of our abstracted FlexRIC-OAI simulation

After training with TheRLib, we found that the “abrupted” nature of this simulation makes training very hard to converge in comparison to training on the real RAN (*subsection 5.3.2*),

where packets are more “systematically” handled. Also, when onboarding on the real RAN, the performance differences (between training and testing) is huge: approx. 23% difference. These results are shown in Figure 4.5.2.

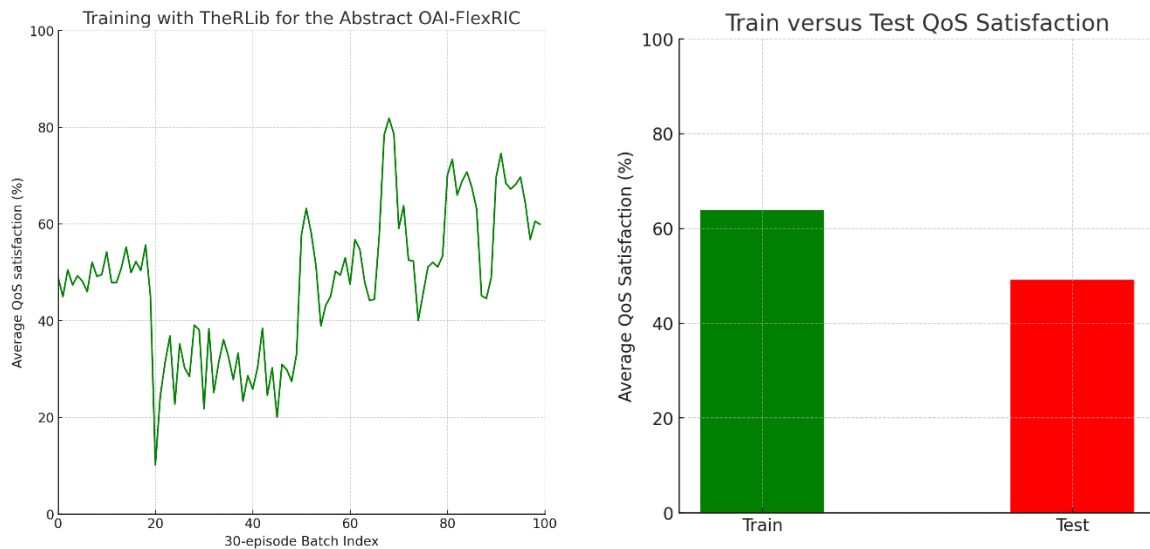


Figure 4.5.2: Differences between Training Expectation and Actual Tested results

Our conclusion is that, abstracting key elements of a RAN system is not sufficient to make its RL policies predictable. This raises a deeper question: “*Will a minor difference between a RAN system and its simulator also cause such train versus production performance gap?*” If the answer is positive, then one should **not** use RAN simulators to prepare network controllers for Industrial 5G use cases. We will explore this question in *subsection 5.3.2*.

4.6 Integration of TheRLib MA-DRL platform and the MatLab-based RAN link control

4.6.1 Overview

UBRU has implemented a MATLAB-based Network Simulation of the Reinforcement Learning Radio Link Control, as described in Deliverable 5.2 Section 6.3. The goal of this research is to create a Markov Decision Process (MDP) radio link control model using Matlab’s Reinforcement toolbox and then firstly design interfaces to be able to connect UBRU training environment to the MAD-DRL platform and then secondly design interfaces to the network model developed and described in Deliverable 5.2 Section 6.3. To this end, we need to implement interfaces between the Matlab code of UBRU’s environment and the Python code of TSG’s DRL software platform. Binding with Python is well handled in the latest Matlab release, by calling Python code from MATLAB or the other way around. Two alternatives have been considered:

- (1) Calling the MATLAB simulation from the Python training code, with the Gym environment style as reported in D2.3 and D5.1. This way requires little to no adaptation to the RL training code of the MA-DRL platform, but requires that the MATLAB simulation is able to stop its execution loop and ask for input when the agent

(located in the UE MAC Layer) needs to take a decision. A sample of the interfacing code within the Python training code would look like:

```
state = matlab_env.reset()
next_state, reward, done, info = matlab_env.step(action)
```

- (2) Calling the Python DRL trainer from the MATLAB simulation, with an external trainer interface. This way turns around the classical Gym interface by initiating the decision from the environment. The environment requests an action to the Python trainer when it is needed, and regularly send its interaction data to the trainer that processes it to update its model. This requires little to no adaptation to the MATLAB simulation, but an adaptation to the Python trainer to be able to provide decisions on-demand while updating its training process. A sample of the interfacing code within the MATLAB simulation code would look like:

```
python_trainer.log_transition(state, action, next_state, reward, done, info)
action = python_trainer.request_action(state)
```

Option 1 has been chosen and implemented in TheRLib's MATLAB interface described in Section 3.4.1.3, leveraging environment interfaces already provided by the MATLAB Reinforcement Learning toolbox.

4.6.2 Modulation and coding scheme (MCS) and channel quality indicator (CQI)

To maximise spectral efficiency and data transmission speeds, the 5G Radio Access Network (RAN) architecture uses the Modulation and Coding Channel Quality Indicator (CQI), a crucial measure. CQI offers useful data on the wireless channel quality between the user equipment (UE) and the base station (BS). Because of this, the base station is able to dynamically modify the modulation and coding schemes (MCS) to suit the various channel circumstances that various UEs may encounter [20].

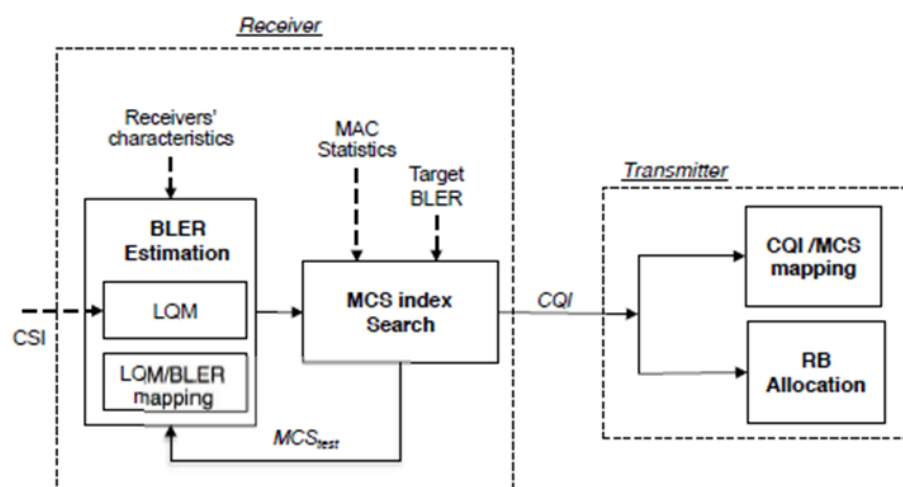


Figure 4.6.2.1: Conventional Choosing the best Modulation and Coding Scheme (MCS) from Channel Quality Indicator (CQI)

4.6.2.1 Modulation and coding scheme

In the 5G Radio Access Network (RAN) design, the Modulation and Coding Scheme (MCS) is a key parameter that controls how many meaningful bits may be conveyed by a single symbol (Resource Element - RE) in the wireless communication channel. The modulation scheme and the coding rate are two essential components that the MCS establishes. The coding rate affects the ability of the transmitted data to adjust for errors whereas the modulation technique determines how information is encoded into the carrier signal. Higher MCS levels provide faster data rates, but they also call for more dependable channel conditions [20]. The MCS is a higher resolution version of the CQI, and there is an algorithmic mapping between them at the gNB but they essentially convey the same information.

The MCS levels supported by 5G NR, which include QPSK, 16 QAM, and 64 QAM, will be taken into account in this study. A varied amount of bits can be transmitted depending on the modulation strategy. The right MCS must be chosen in order to maximise data flow and achieve dependable communication [20]. The Table 4.6.2.1.1 shows MCS index and their corresponding spectral efficiency.

MCS Index I_{MCS}	Modulation Order Q_m	Target code Rate $R \times [1024]$	Spectral efficiency
0	2	30	0.0586
1	2	40	0.0781
2	2	50	0.0977
3	2	64	0.1250
4	2	78	0.1523
5	2	99	0.1934
6	2	120	0.2344
7	2	157	0.3066
8	2	193	0.3770
9	2	251	0.4902
10	2	308	0.6016
11	2	379	0.7402
12	2	449	0.8770
13	2	526	1.0273
14	2	602	1.1758
15	4	340	1.3281
16	4	378	1.4766
17	4	434	1.6953
18	4	490	1.9141
19	4	553	2.1602
20	4	616	2.4063
21	6	438	2.5664
22	6	466	2.7305
23	6	517	3.0293
24	6	567	3.3223
25	6	616	3.6094
26	6	666	3.9023
27	6	719	4.2129
28	6	772	4.5234
29	2	reserved	
30	4	reserved	
31	6	reserved	

Table 4.6.2.1.1: MCS Index Table 5.1.3.1-3 in [20]

4.6.2.2 Channel quality indicator (CQI)

The adaptive selection of MCS relies heavily on the Channel Quality Indicator (CQI). The user equipment (UE) and base station (gNB) wireless channel quality is measured using the CQI metric. It quantifies a number of channel characteristics, including path loss, interference, fading, and signal-to-noise ratio. How effectively the channel can sustain dependable data transmission at various MCS levels is shown by the CQI number [20].

The CQI input from UEs will be gathered during the reinforcement learning agent's training phase to offer important data about the channel quality. With the use of this data, the MCS levels will be dynamically adjusted for each UE, ensuring effective use of the available radio resources, and preserving the required level of service [20]. The different CQI Indexes, as well as their associated modulation, coding rate, and efficiency, are displayed in the CQI table below.

CQI index	modulation	code rate x 1024	efficiency
0	out of range		
1	QPSK	78	0.1523
2	QPSK	120	0.2344
3	QPSK	193	0.3770
4	QPSK	308	0.6016
5	QPSK	449	0.8770
6	QPSK	602	1.1758
7	16QAM	378	1.4766
8	16QAM	490	1.9141
9	16QAM	616	2.4063
10	64QAM	466	2.7305
11	64QAM	567	3.3223
12	64QAM	666	3.9023
13	64QAM	772	4.5234
14	64QAM	873	5.1152
15	64QAM	948	5.5547

Table 4.6.2.2.1: 4-bit CQI Table 5.2.2.1-2 in [12]

The objective is to adapt the Reinforcement Learning Markov Decision Process (MDP) Example Model in Matlab to produce a Reinforcement Learning MDP 5G Radio Link Control Model in Matlab.

4.6.2.3 Adaptive MCS selection using MDP with block error rate

The reinforcement learning-based methodology was used to integrate MCS, CQI, and BLER. The MDP environment, which models the 5G RAN setup, was interacted with by the reinforcement learning agent. In order to choose the proper MCS levels for each UE in the system, the agent obtains knowledge from the CQI feedback and BLER data. The agent seeks to maximise cumulative rewards by optimising the MCS selection based on real-time channel circumstances, which equates to obtaining greater data rates while ensuring dependable and error-free communication [20].

The core of our approach lies in the **move_function**, which simulates state transitions in the MDP. The function accepts an action, representing the MCS to be used, and returns the next state, the received reward, and a flag indicating whether a terminal state has been reached. Here, the reward R is set to zero if a block error occurs.

If no block error has occurred then the training episode is not terminated early and is allowed to continue for the whole frame of 20 slots when it is terminated, which is shown in the figure below. Whereas if a single block error has occurred then the training episode is terminated, since less than 1 in 20 block errors or $< 5\%$ is allowed. Less than 2 in 20 block

errors or < 10% is allowed then 2 block errors must have occurred for the training episode to be terminated.

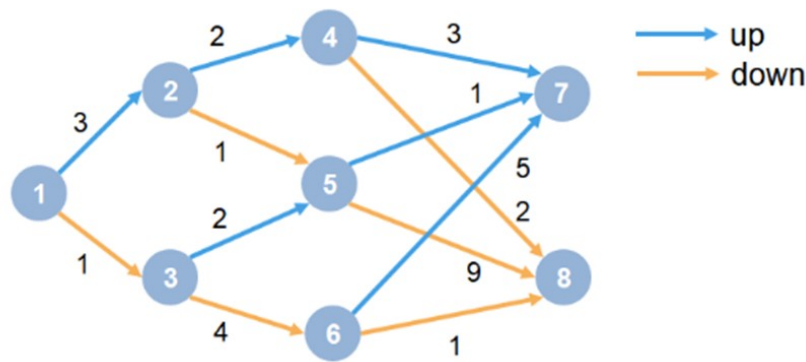
SimCount = 250				
slotCnt = 1	state = 1	Action Name = up	NoPBER = 0	Reward = 0.2344
slotCnt = 2	state = 2	Action Name = up	NoPBER = 0	Reward = 0.3770
slotCnt = 3	state = 3	Action Name = up	NoPBER = 0	Reward = 0.6016
slotCnt = 4	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 5	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 6	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 7	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 8	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 9	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 10	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 11	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 12	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 13	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 14	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 15	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 16	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 17	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 18	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770
slotCnt = 19	state = 5	Action Name = down	NoPBER = 0	Reward = 0.6016
slotCnt = 20	state = 4	Action Name = up	NoPBER = 0	Reward = 0.8770

Figure 4.6.2.3.1: MCS, CQI & BLER

4.6.3 Modification and Adaptation of Reinforcement Learning Generic Markov Decision Process (MDP) Example Model for Radio Link Control in Matlab

4.6.3.1 Original MDP model

The initial Markov Decision Process (MDP) model used for radio link control was based on a relatively simple existing design, incorporating 8 states and 2 possible actions—termed as 'UP' and 'Down' [8]. These two actions corresponded to improving the cumulative rewards obtained by the agent when navigating between state 1 and the Terminal states 7 and 8. The agent receives a reward equal to the value on each transition in the graph. The training episode ends when states 7 or 8 are reached.



```

MDP = createMDP(8,["up","down"]);

MDP.T(1,2,1) = 1;
MDP.R(1,2,1) = 3;
MDP.T(1,3,2) = 1;
MDP.R(1,3,2) = 1;

% State 2 transition and reward
MDP.T(2,4,1) = 1;
MDP.R(2,4,1) = 2;
MDP.T(2,5,2) = 1;
MDP.R(2,5,2) = 1;

% State 3 transition and reward
MDP.T(3,5,1) = 1;
MDP.R(3,5,1) = 2;
MDP.T(3,6,2) = 1;
MDP.R(3,6,2) = 4;

% State 4 transition and reward
MDP.T(4,7,1) = 1;
MDP.R(4,7,1) = 3;
MDP.T(4,8,2) = 1;
MDP.R(4,8,2) = 2;

% State 5 transition and reward
MDP.T(5,7,1) = 1;
MDP.R(5,7,1) = 1;
MDP.T(5,8,2) = 1;
MDP.R(5,8,2) = 9;

% State 6 transition and reward
MDP.T(6,7,1) = 1;
MDP.R(6,7,1) = 5;
MDP.T(6,8,2) = 1;
MDP.R(6,8,2) = 1;

% State 7 transition and reward
MDP.T(7,7,1) = 1;
MDP.R(7,7,1) = 0;
MDP.T(7,7,2) = 1;
MDP.R(7,7,2) = 0;

% State 8 transition and reward
MDP.T(8,8,1) = 1;
MDP.R(8,8,1) = 0;
MDP.T(8,8,2) = 1;
MDP.R(8,8,2) = 0;

```

Figure 4.6.3.1.1: Original MDP Model

The core of our approach lies in the **move_function**, which simulates state transitions in the MDP. The function accepts an action, representing the MCS to be used, and returns the next state, the received reward, and a flag indicating whether a terminal state has been reached.

```

173 function [S,R,isTerm] = move_(obj,ActionName)
174 % Brunel Work added to keep count of number of slots |
175 obj.slotCnt = obj.slotCnt + 1;
176 %move Summary of this method goes here
177 R = 0;
178 S0 = obj.CurrentState_;
179 isTerm = isTerminalState(obj);
180 if isTerm
181 R = 0;
182 S = S0;
183 else

```

Figure 4.6.3.1.2: Reference Simulation for BLER

This function is embedded deep inside successive function calls from the top-level program RLCReinforcementlearning.m, as shown in Figure 4.6.3.1.3.

```

AbstractMDP.move_
GenericMDP.move
rMDPEnv.step
@(a)step(env,a)
MATLABFunctionHandleSimulator.step_
MATLABSimulator.step
MATLABSimulator.simInternal_
MATLABSimulator.sim_
AbstractSimulator.sim
runEpisode
SeriesTrainer.run
TrainingManager.train
TrainingManager.run
train
RLCReinforcementLearning

```

Figure 4.6.3.1.3: move_ function embedded deep inside successive function calls from the top level program

The agent is trained, and the results are presented using the Reinforcement Learning Episode Manager, as shown in Figure 4.6.3.1.4, which outputs the Episode Reward against Episode number as the agent is being trained.

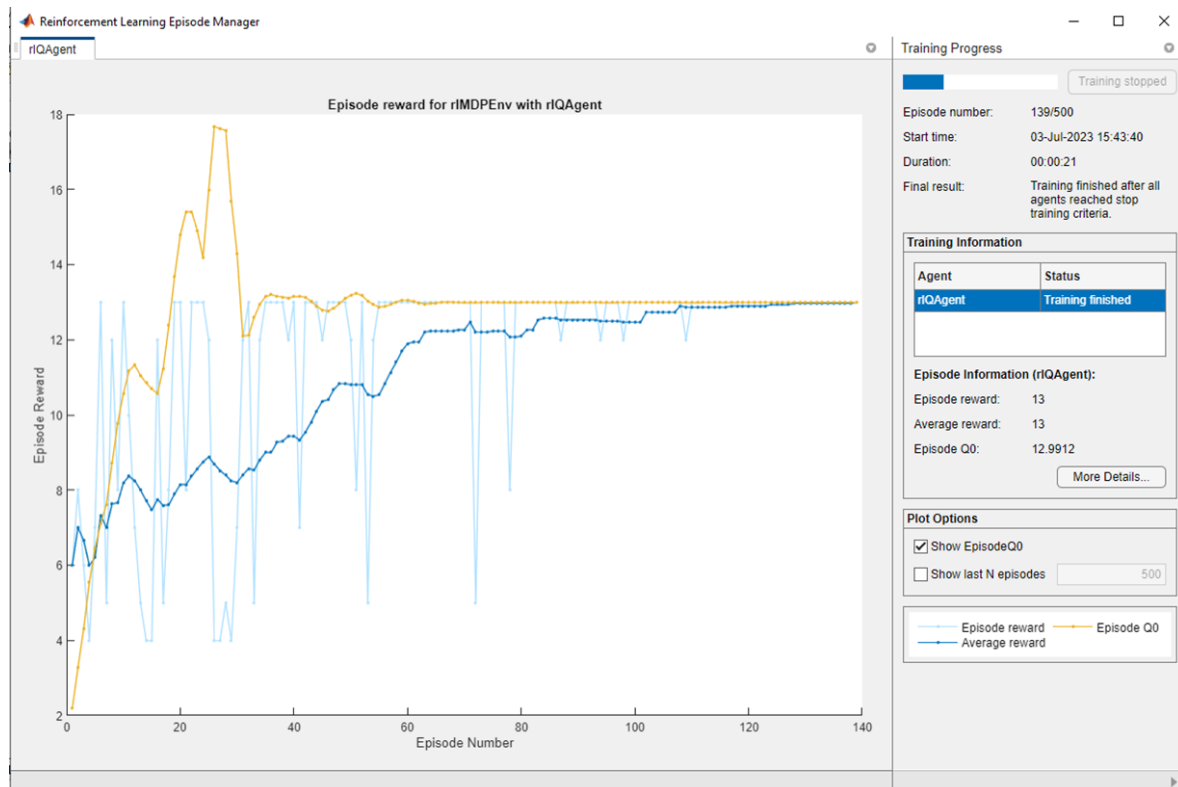


Figure 4.6.3.1.4: Reinforcement Learning Episode Manager

4.6.3.2 Limitations of the original MDP Model

While effective as a basic model for representing link control decisions, the original MDP was limited in its ability to:

1. Access to the top-level program file: `RLGenericMDPExample.m` is available and editable in the Current Folder of the development environment. However, all other programs, which it calls are only available to be viewed but are restricted from being edited by Brunel University strict access restrictions. They are available in `c:/Program Files/Matlab/R2022a/toolbox/rl/rl/` and its subfolders.

2. There are only 2 state transitions “up” and “down”.
3. The Training episode ends when terminal states 7 or 8 are reached and success is recorded.
4. There is no fixed period during which training is occurring to represent for example 20 slots in a frame.
5. There is no probability that the Transport Block Error Rate (BLER) has been experience.
6. There is probability of an error E occurring in any Modulation and Coding Scheme state.
7. There is no slot Count slotCnt that counts the 20 slots in a frame and terminates the frame once the 20th slot has been transmitted.

4.6.4 Linux Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Matlab Development Environment

Ubuntu Linux 20.04 was selected as the system operating system to use for the purpose of achieving the goals of this research because it contained all the necessary drivers and applications for both the TheRLib MA-DRL platform and the Matlab Reinforcement Learning toolbox. Matlab was successfully downloaded, and installation was completed on the system with the required version R2023a. This is made possible because Brunel University has a site license for all Matlab tools which allows students to download and install Matlab on their PCs. The creation of a Markov Decision Process (MDP) radio link control model was the objective, and in order to accomplish this objective, additional packages had to be installed in Matlab. It was possible to successfully install all the additional packages that were required, and some of the most important ones were the Deep Learning HDL toolbox and the Reinforcement Learning toolbox. Since the Linux Machine was not Brunel University registered computer this allows Super User status for researchers to change the read-write-execute mode of files, which otherwise would not have been accessible without the support of University IT support.

4.6.5 Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab

The Markov Decision Process (MDP) approach in Reinforcement Learning (RL) for the 5G Radio Access Network (RAN) setup entails characterising a series of states, actions, transition probabilities, and rewards. Resource distribution and interference patterns are only two examples of the various 5G RAN configurations that may be represented in the state space. The RL agent's choices to increase RAN performance are included in the action space. The possibility of changing to a new state depending on activities is shown by transition probabilities, which reflect environmental uncertainty. State-action combinations are given numerical rewards via the reward function, which directs the agent towards more advantageous RAN configurations [12].

When MDP is used in RL, the agent discovers the best course of action through interactions with the outside world. For the purpose of making wise judgements, the agent experiments with various activities, monitors results (rewards), and updates its knowledge. The agent

develops methods to lessen interference, efficiently manage resources, and improve overall RAN performance over time. By using an adaptive method, the agent may configure the 5G RAN in a way that is effective and intelligent and improves wireless communication capabilities while dynamically responding to changes in the wireless environment [24].

As part of the methodology for this project we created a Markov Decision Process (MDP) environment in MATLAB to address the 5G Radio Access Network (RAN) Radio Link Control configuration problem using Reinforcement Learning (RL). The process can be summarized into below four key steps.

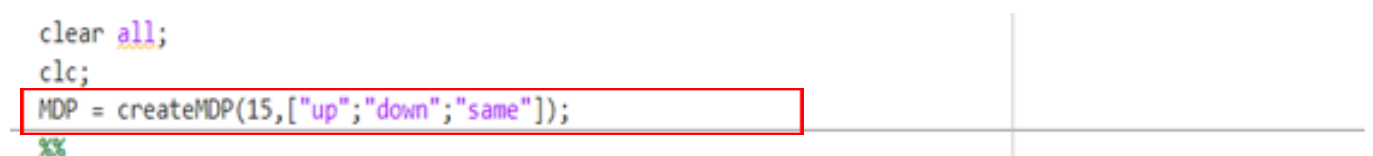
1. Creating the MDP Environment
 - a. Creating the MDP Environment
 - b. Defining the State Transition Matrix
 - c. Specifying the Reward Matrix
 - d. Incorporating the Transport Block Error Ratio (BLER) in the MDP States
 - e. Continuous Decision Making with Terminal States
2. Creating the Q-Learning Agent
3. Integrate the MDP Environment with Q-Learning Agent
4. Training and Validating the Q-Learning Agent
 - a. Train Q-Learning Agent
 - b. Validate Q-Learning Result

4.6.5.1 Creating MDP Environment

The first step in working with a Markov Decision Process (MDP) involves establishing the fundamental elements that define the environment in which the agent will operate. This foundational step has several sub-steps:

A. Defining the state space

The createMDP function in MATLAB (Figure 4.6.5.1.1) is used to construct an MDP model in the initialisation phase. This function allows the user to specify the number of states and actions required for the MDP framework. For the purposes of this project, the MDP is designed with 15 states that correspond to the 15 CQI indexes and their corresponding Modulation and Coding Scheme (MCS) states and three possible actions: 'UP,' 'DOWN,' and 'SAME.' These actions signify the choices available to the Reinforcement Learning (RL) agent at each state. In this context, each state within the 5G Radio Access Network (RAN) is representative of a specific configuration or situation [12].



```
clear all;
clc;
MDP = createMDP(15, ['up'; 'down'; 'same']);
%%
```

Figure 4.6.5.1.1: Defining the State Space

In the Markov Decision Process (MDP) environment depicted by the circles, the agent faces a series of decision points, represented by states. At each state, the agent can choose between

going up, down or staying same, determining its next move. The agent starts from state 1 and receives immediate rewards equal to the values on each transition in the graph. The training objective is to learn an optimal policy that allows the agent to collect the maximum cumulative reward by making informed decisions at each state during its interactions with the environment [12].

B. Defining the state transition matrix

The state-action transition matrix (**MDP.T**), Figure 4.6.5.1.2, is a three-dimensional matrix where each element $\text{MDP.T}(i, j, k)$ specifies the transition probability from state 'i' to state 'j' when taking action 'k' [8].

In our model, the transition probabilities were deterministic. For instance, a transition probability of $\text{MDP.T}(1,2,1) = 1$ indicates that if the system is in state 1 and the action 'UP' is taken (indexed as 1), the system will transition to state 2 with a probability of 1. Actions are indexed as 1 for "UP," 2 for "DOWN," and 3 for "SAME." States are indexed numerically starting from 1.

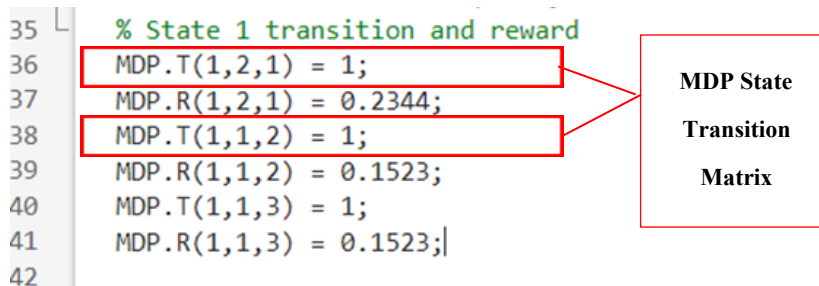


Figure 4.6.5.1.2: Specifying the Transition Matrix

C. Specifying the reward matrix

The reward matrix (**MDP.R**) is another three-dimensional matrix where each element $\text{MDP.R}(i, j, k)$ specifies the reward obtained when transitioning from state 'i' to state 'j' through action 'k' [14]. For example, $\text{MDP.R}(1,2,1) = 0.2344$ implies that a transition from state 1 to state 2 through action 'UP' would yield a reward of 0.2344. The reward matrix, which gives the RL agent rapid feedback for each state-action transition, is essential component of the MDP environment. It shows the benefits the agent gains from doing particular actions and changing states. The agent's behaviour may be influenced and made to make choices that result in greater rewards by altering the reward matrix.

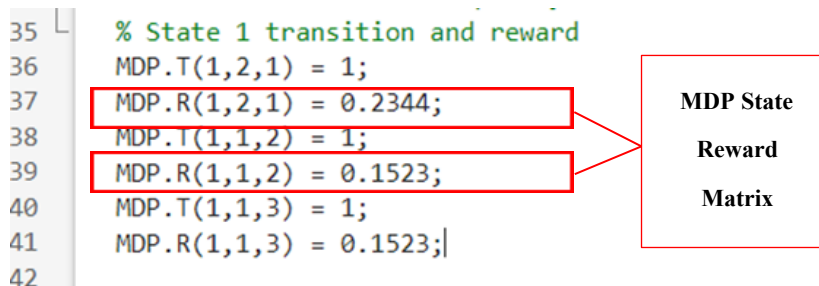


Figure 4.6.5.1.3: Reference Simulation for Reward Matrix

- **MDP.T(1,1,2) = 1;** and **MDP.R(1,1,2) = 0.1523;** These lines specify that from state 1, taking action 2 ("down") leads to a transition to state 1 (itself) with a probability of 1 and a reward of 0.1523.
- **MDP.T(2,3,1) = 1;** and **MDP.R(2,3,1) = 0.3770;** These lines specify that from state 2, taking action 1 ("up") leads to a transition to state 3 with a probability of 1 and a reward of 0.3770.
- **MDP.T(2,2,3) = 1;** and **MDP.R(2,2,3) = 0.2344;** These lines specify that from state 2, taking action 3 ("same") leads to a transition back to state 2 (itself) with a probability of 1 and a reward of 0.2344.

This was designed to transition between Modulation and Coding Scheme (MCS) by single state transitioning between states namely [up, down, same] and starting from State 1. The state transition diagram for this is shown in Figure 4.6.5.1.4.

The agent was modified to also start from the best state identified by the largest Reward in the QMatrix to ascertain if performance in training the agent is improved. This was realised by modifying in top level RLCReinforcementLearning.m file the function

```
env.ResetFcn = @() 1;
```

to

```
env.ResetFcn = @() MDP.StartState;
```

and inserting the following code in SeriesTrainer.m in lines 76 - 83:

```
if any(QTableMatrix)           % if Qmatrix is not all zero
    hvalue = max(QTableMatrix(:,2))
    location = find(QTableMatrix(:,2) == hvalue)
    env.ResetFcn = location
end
```

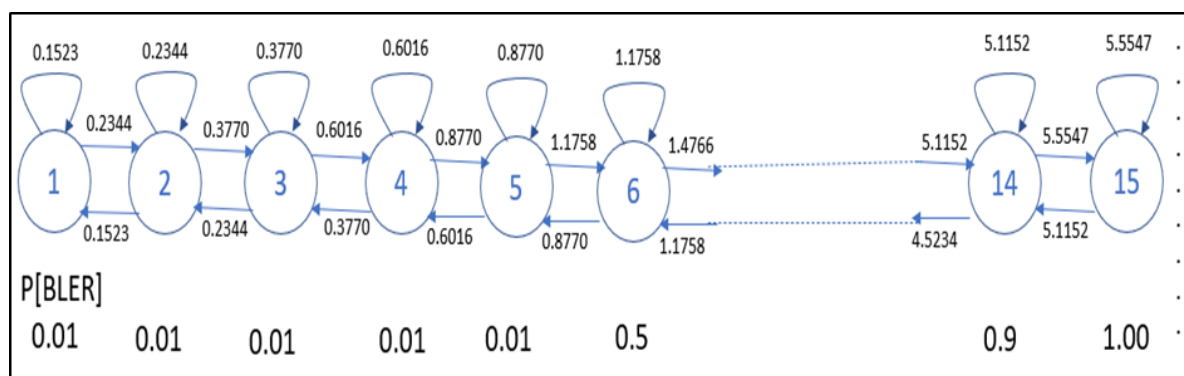


Figure 4.6.5.1.4: MDP against BLER for single state transitioning [12]

An alternative scheme was designed to transition between Modulation and Coding Scheme (MCS) by up to two state transitioning between states namely [up, down, same, up2, down2] and starting from State 1. The state transition diagram for this is shown in Figure 4.6.5.1.5.

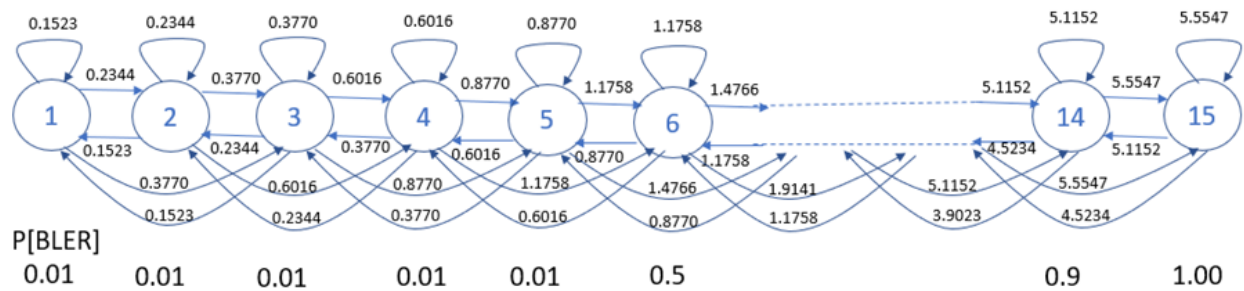


Figure 4.6.5.1.5: MDP against BLER for up to two state transitioning[13]

D. Incorporating block error rate (BLER) in the MDP states

In our MDP model for optimizing 5G RAN configuration, we incorporate the Block Error Rate (BLER) as an essential state-dependent parameter. It displays the likelihood that a block of sent data will be incorrectly received. The MATLAB code assigns BLER values to each of the 15 states in our model, stored in the array **MDP.E**

Validating Q-Learning Results

139	MDP.E(1) = 0.01;
140	MDP.E(2) = 0.01;
141	MDP.E(3) = 0.01;
142	MDP.E(4) = 0.01;
143	MDP.E(5) = 0.01;
144	MDP.E(6) = 0.5;
145	MDP.E(7) = 0.5;
146	MDP.E(8) = 0.5;
147	MDP.E(9) = 0.5;
148	MDP.E(10) = 0.5;
149	MDP.E(11) = 0.6;
150	MDP.E(12) = 0.7;
151	MDP.E(13) = 0.8;
152	MDP.E(14) = 0.9;
153	MDP.E(15) = 1.0;

BLER
incorporated
in MDP States

Figure 4.6.5.1.6: Reference Simulation for Block Error Rate

Key Interpretations:

1. **Low BLER States:** The states with **MDP.E** values of **0.01** represent configurations or conditions where the BLER is very low, indicating high reliability.
2. **Moderate BLER States:** The states with **MDP.E** values around **0.5** to **0.9** signify moderate to high levels of block errors, possibly requiring corrective actions or reconfigurations.
3. **High BLER State:** The state with an **MDP.E** value of **1.0** is an extreme case where every block is erroneous and likely signifies a severe issue requiring immediate attention.

These BLER values are used to guide the decision-making process of our reinforcement learning agent. By associating each state with a specific BLER, the model can more effectively learn to make configurations that optimize for lower error rates, improving the overall performance and reliability of the 5G RAN. To maintain the necessary degree of dependability in this study, a BLER threshold, normally around 10%, will be established. The link adaptation algorithm will dynamically change the MCS levels for each UE in response to CQI feedback to

make sure that the BLER stays below the specified threshold even under a variety of radio situations [20].

E. Continuous decision making without terminal states

In our study on 5G Radio Access Network (RAN) Radio Link Control configuration using reinforcement learning, we have designed MDP model that intentionally lacks specific terminal states. This design choice reflects the continuous operational nature of radio link control network management task in a 5G environment. 5G RANs are continuously operational and require ongoing optimization for parameters to set the Modulation and Coding Scheme. Hence, an MDP without terminal states is more representative of these operational needs. Also, the absence of terminal states permits the reinforcement learning agent to adapt its policy in real-time, enabling more responsive and robust 5G RAN configurations.

We successfully built the MDP environment for the 5G RAN setup issue by completing the above stages. The environment is now prepared for integration with reinforcement learning methods, including Q-learning, in order to develop an RL agent and teach it how to configure the 5G RAN in the best way possible. Through interactions with the MDP environment, the agent will learn in order to maximise cumulative rewards and create an effective and intelligent 5G RAN setup.

4.6.5.2 Creating Q-learning agent

In the realm of reinforcement learning, a Q-Learning Agent aims to learn the optimal policy to navigate through a given Markov Decision Process (MDP) environment. By iteratively updating a Q-table based on rewards and state transitions, the agent learns to make decisions that maximize cumulative rewards over time [19].

Using the `getObservationInfo` and `getActionInfo` methods, we extract the observation and action specifications from the MDP environment (`env`) in this phase. Then, using the specified observation (`obsInfo`) and action (`actInfo`) parameters, we generate a Q table using `rlTable`. The Q table, observation specifications, and action specifications are then used to form a Q-value function (`qFunction`). Then, using `rlOptimizerOptions`, we set the learning rate of the Q-value representation to 1. The learning rate determines how frequently the agent adjusts its Q-values in response to fresh encounters as it learns [19].

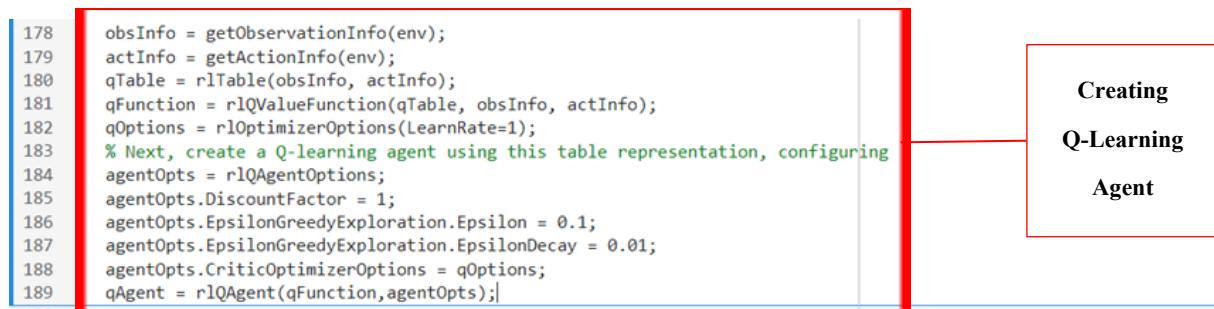


Figure 4.6.5.2.1: Simulation of creating Q-learning agent.

The Q-learning agent parameters are then configured using `rlQAgentOptions`. The agent regards short-term and long-term benefits equally, therefore we set the discount factor

(DiscountFactor) to 1 [19]. In order to balance exploration with exploitation, the agent will employ an epsilon-greedy exploration strategy, in which it will randomly behave with probability Epsilon (which is initially set at 0.9). The exploration rate is gradually decreased over time by the EpsilonDecay parameter (set at 0.01). Finally, we define the Q-value function optimizer options (qOptions), which are the critic optimizer settings for the agent. In this final step, we create the Q-learning agent (qAgent) using the previously defined Q-value function (qFunction) and agent options (agentOpts) [19].

4.6.5.3 Integration of MDP environment with Q-learning agent

Finding the best policy in an MDP environment may be done using Q-learning, a model-free reinforcement learning technique. According to the agent's experiences, the Q-learning algorithm repeatedly updates the Q-values, which are assessments of the quality of each action-state combination. As part of the integration, Q-value updates are made while learning is taking place utilising the state transition and reward matrices specified in the MDP environment [9][10].

The Markov decision process in the Q-learning algorithm consists of five crucial components: (1) S is a collection of states, (2) A denotes a collection of actions, (3) P denotes a transition probability function, (4) R denotes a reward function, and (5) γ denotes the discount rate for potential rewards. In this project, the Block Error Rate (BLER) values serve as the Reward, the User Equipment serves as the Environment, and the gNodeB, acting as the Base Station, operates as the Agent in the Reinforcement Learning cycle [27]. The Channel Quality Indicator (CQI) values reflect the while the BLER values represent the Reward. Choosing the appropriate Modulation and Coding Scheme (MCS) based on the relevant CQI is the Agent's activity [18].

Three layers make up the MATLAB 5G New Radio Intercell Interference Downlink Model: the physical (PHY), the medium access control (MAC), and the radio link control (RLC). The gNodeB and User Equipment, respectively, as represented by the hNRGNB.m and hNRUE.m auxiliary classes, are where the procedure occurs. The Environment (User Equipment) is influenced by the Agent's (Base Station's) choice of an MCS that matches a CQI. A 10% throughput threshold (expressed as 100) is used to assess if an experiment was successful or unsuccessful. Finding the best mapping method between CQI and MCS is the goal of the gNB. The scheduler for resource allocation, the Modulation and Coding Scheme value computation, and the selection of the Hybrid Automatic Repeat Request (HARQ) procedure are all done inside the gNB. Each Resource Block Group (RBG) in hNRSheduler.m's Scheduler input generation process requires the translation of CQI to MCS, with each UE doing the translation at each RBG. CQI and PMI information are sent between the UE and gNB MAC layers via the link between the gNB and UE.

4.6.5.4 Training and validating the Q-learning agent

The success of a Q-Learning Agent hinges on effective **training** and **validation**. In the training phase, the agent interacts with the environment to update its Q-table, thereby learning the optimal policy to maximize cumulative rewards. Subsequently, the validation phase tests the agent's learned behaviour to ensure its efficacy and reliability [19].

a. Train Q-learning agent

Using `rlTrainingOptions`, we build the training choices for the Q-learning agent in this stage. `MaxStepsPerEpisode` has been set to 20, suggesting that each episode will have a cap of 20-time steps. There is a cap of 250 episodes (`MaxEpisodes`) throughout the duration of the training session. When the agent earns an average cumulative reward larger than the provided `StopTrainingValue` (set to 100) across a window of `ScoreAveragingWindowLength` (set to 10) consecutive episodes, we specify "AverageReward" as the stopping criterion to halt training [19].

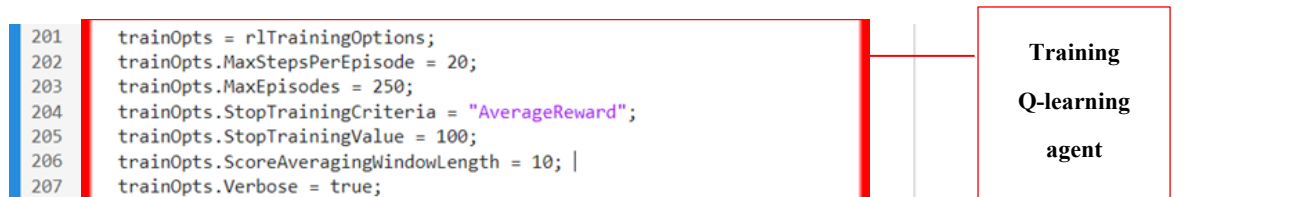


Figure 4.6.5.4.1: Simulation of training Q-learning agent

The Q-learning agent interacts with the MDP environment while the training process is underway, updating its Q-values and figuring out the best configuration strategy for the 5G RAN. When the stopping requirements are satisfied or the maximum number of episodes defined in the training choices is reached, the training process will automatically come to an end. The resultant `trainingStats` will include details on the training performance, including the total incentives earned during the training episodes [19].

b. Validating Q-learning results

In this section, we assess the performance of the Q-learning agent in the 5G Radio Access Network (RAN) configuration problem by validating its learned policy and cumulative reward achieved during simulation. We start by simulating the trained Q-learning agent in the training environment to evaluate its performance. The agent successfully finds the optimal path, resulting in a cumulative reward equal to `Sum(data, Reward)`, as shown in the following code snippet [19]:

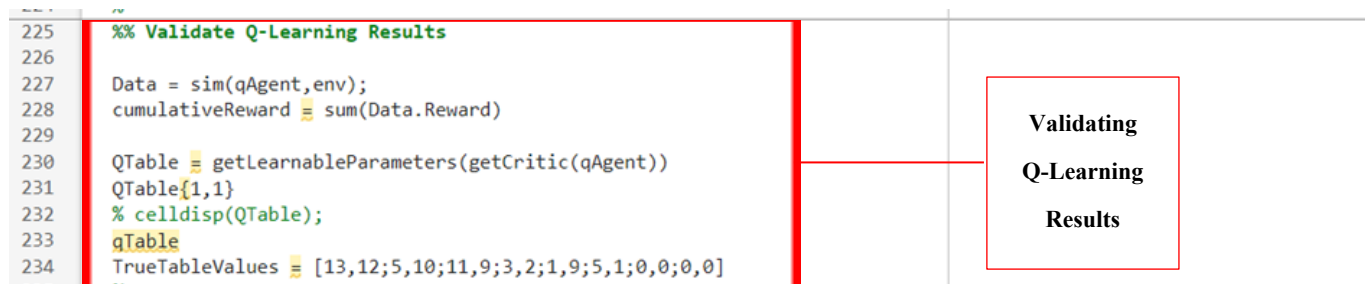


Figure 4.6.5.4.2: Simulation of Validate Q-Learning Results

4.6.5.5 Adaptive MCS selection using MDP with block error rate

The cumulative reward gained demonstrates the agent's learnt policy's success in effectively configuring the 5G RAN to maximise rewards. We examine the Q table of the Q-learning agent and contrast it with the predicted Q-values to further verify the training. The `getLearnableParameters` function is used to acquire the Q table of the trained agent, and the

TrueTableValues matrix contains the expected Q-values as shown in above figure. By comparing the Q table with the true table values, we verify that the agent has successfully learned an optimal policy that aligns with the expected Q-values [19].

The reinforcement learning-based methodology will be used to integrate MCS, CQI, and BLER. The MDP environment, which models the 5G RAN setup, will be interacted with by the reinforcement learning agent. In order to choose the proper MCS levels for each UE in the system, the agent will get knowledge from the CQI feedback and BLER data. The agent seeks to maximise cumulative rewards by optimising the MCS selection based on real-time channel circumstances, which equates to obtaining greater data rates while ensuring dependable and error-free communication [20].

The core of our approach lies in modifying the **move_ function**, which simulates state transitions in the MDP. The function accepts an action, representing the MCS to be used, and returns the next state, the received reward, and a flag indicating whether a terminal state has been reached. Here, the reward R is set to zero if a block error occurs, which is determined as shown in the Figure 4.6.5.5.1.

```

SimCount = 5
slotCnt = 1    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 2    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 3    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 4    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 5    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 6    state = 1 Action Name = same    NoPBER = 0    Reward = 0.1523
slotCnt = 7    state = 1 Action Name = up      NoPBER = 0    Reward = 0.2344
slotCnt = 8    state = 2 Action Name = up      NoPBER = 0    Reward = 0.3770
slotCnt = 9    state = 3 Action Name = up      NoPBER = 0    Reward = 0.6016
slotCnt = 10   state = 4 Action Name = up      NoPBER = 0    Reward = 0.8770
slotCnt = 11   state = 5 Action Name = same    NoPBER = 0    Reward = 0.8770
slotCnt = 12   state = 5 Action Name = up      NoPBER = 0    Reward = 1.1758
slotCnt = 13   state = 6 Action Name = down    NoPBER = 1    Reward = 0.0000
  
```

Figure 4.6.5.5.1: MCS, CQI & BLER

4.6.5.6 Implementation of state transition logic abstract MDP

In our RL implementation, the **move_ function** plays a crucial role. Originating from an abstract class **AbstractMDP**, this function is modified to implement the specific logic for MCS selection. The function takes in the current state and action as inputs and returns the new state, reward, and a flag indicating whether the simulation has reached a terminal state.

```

172 end
173 function [S,R,isTerm] = move_(obj,ActionName)
174 % Brunel Work added to keep count of number of slots |
175 obj.slotCnt = obj.slotCnt + 1;
  
```

Figure 4.6.5.6.1: Reference simulation of the function signature

The three additional parameters required for this modified **move_()** function is

1. Transport Block Error Rate (BLER) to signify if a Transport Block has experienced an error or not.
2. Probability of an error E occurring in any Modulation and Coding Scheme state

3. Slot Count slotCnt that counts the 20 slots in a frame and terminates the frame once the 20th slot has been transmitted.

These parameters are declared in toolbox/rl/rl/+rl/+env/GenericMDP.m

```
properties
    %E
    % added this Prob of an error E occurring in any MCS state
        E

    % NoPBER
    % added this to record the total No PBER in a Frame
    NoPBER

    % slotCnt
    % added this to record the slotCnt number
    slotCnt
end
```

These parameters need to be accompanied with corresponding get.param() method otherwise errors occur on the command line.

```
% added slotCnt to count slot number
function slotCnt = get.slotCnt(obj)
    % Get slotCnt
    slotCnt = obj.slotCnt;
end

% added NoPBER for the total No PBER in a Frame
function NoPBER = get.NoPBER(obj) % Get NoPBER
    obj.NoPBER = obj.NoPBER;
end

% added E probability of BLER in a slot
function E = get.E(obj) % Get E
    E = obj.E;
end
```

In toolbox/rl/rl/+rl/+train/SeriesTrainer.m additional code is inserted in function run(this) to keep a track of what episode is being processed:

```
function run(this)
.....
for simCount = 1:N
fprintf("SimCount = %d \n",simCount); % to know which frame/episode we are in
% book keeping train data for logging
.....
end
```


Initialization:

At the beginning of each call to the `move_` function, certain initialization steps are undertaken. The time slot count (`obj.slotCnt`) is incremented by one to keep track of the number of steps taken in the environment. The initial reward `R` is set to zero, and the current state `S0` is fetched from the object's property.

```

175     obj.slotCnt = obj.slotCnt + 1;
176     %move Summary of this method goes here
177     R = 0;
178     S0 = obj.CurrentState_;
179     isTerm = isTerminalState(obj);

```

Figure 4.6.5.6.2: Reference Simulation for initialization

Terminal State Check:

The function first checks if the current state is terminal using the helper function `isTerminalState(obj)`. In the context of a terminal state, the episode concludes, and the agent starts a new episode.

State Transition Logic:

The next state is calculated based on a pre-defined state transition matrix. A random number generator is used in conjunction with this matrix to determine the next state probabilistically.

```

185     % called in GridWorld and GenericMDP. Hence, it is not
186     % necessary to check the validity of the action here.
187     ST = getStateTransition(obj,ActionName);
188     S0_idx = state2idx(obj,S0 );
189     T = ST(S0_idx,:);
190     T = T./sum(T);
191     cumsumT = cumsum(T);
192     % If the last element is smaller than 1, rand can be

```

Figure 4.6.5.6.3: Reference Simulation of State Transition logic

Reward Calculation:

The reward for the transition is fetched using another helper function `getRewardTransition_(obj, ActionName)` which likely refers to a reward matrix that specifies the reward for each state-action pair.

```

198     RT = getRewardTransition_(obj,ActionName);
199     R = RT(S0_idx,S1_idx);
200     % simulate block error and record total number of PBER BRUNEL WORK

```

Figure 4.6.5.6.4: Reference Simulation of Reward Calculation

Simulating Block Error and Its Effects:

The function simulates the occurrence of block errors. A random number is drawn and compared against a block error rate specific to the current state. If the random number is smaller, it simulates the occurrence of a block error, resetting the reward to zero and incrementing a counter for the number of block errors (obj.NoPBER).

```

200 % simulate block error and record total number of PBER BRUNEL WORK
201 if rand < obj.E(S0_idx)
202     R = 0;
203     obj.NoPBER = obj.NoPBER + 1;
204 end

```

Figure 4.6.5.6.5: Reference Simulation of BLER

Logging and Debugging:

For tracking and debugging purposes, crucial information is printed at each step. This includes the time slot, current state, action taken, total number of block errors, and the reward.

Updating the Environment:

The state of the environment and the termination flag are updated based on the logic and counters discussed above.

In Conclusion the core logic in the **move_** function is designed to consult the current MCS (defined as part of the state space) and the action selected by the RL agent. It then determines the new MCS for the next time slot based on probabilistic state transitions. The reward (**R**) is calculated based on various metrics such as throughput, latency, and error rates, serving as an essential signal for the RL agent to learn optimal MCS selection over time. The customization of the **move_** function allows us to embed the logic for MCS selection deeply into the RL training loop. By doing so, we enable the RL agent to explore and exploit different MCS options dynamically, adapting to varying network conditions for optimal performance.

5 Final system testing and performance evaluation

This section presents the experimental testing, validation, and performance evaluation of the final system.

5.1 KPIs and testing scenarios

5.1.1 Summary of KPIs

To this end, a set of relevant KPIs have been employed for the testing, in response to the concerned use cases in industrial applications in line with D2.1. Selected highlighted KPIs can be categorised into three major groups.

The first group is concerned with QoS as this is the primary purpose for applying Network Slicing solutions to meet the QoS requirements of applications, and these KPIs mainly include typical QoS metrics such as the perceived throughput at UE, one-way delay/latency and reliability (measured packet loss percentage/ratio), and QoS satisfaction (measured bitrate as feedback to compare with QoS requirements e.g., in video streaming).

The second group is to evaluate the efficiency and capacity of the Network Slicing operations over RAN or backhaul/backbone networks, and these KPIs mainly include RAN slicing control loop latency (measured between the control logic abstraction location and the corresponding RAN function) for analysing the optimal locations of deploying control logic, backhaul/backbone Network Slicing capacity (measured overall combined bandwidth for the simultaneous backhaul/backbone Network Slices), and scalability of Network Slicing (measure time of creating an increasing number of Network Slices at a concerned level e.g., at the Slice Control Agent level).

The third group is to evaluate the E2E performance of Network Slicing, and these KPIs mainly include E2E service creation (commissioning) time, E2E service decommissioning time, and E2E network topology discovery time.

In addition, through Matlab simulation, some metrics on the reinforcement learning process can be measured, e.g., the time required for the reinforcement learning state machine to reach a stable optimal state depends on the complexity of the transition between states. If state transitions are restricted to up/down/same, then the MDP Q-Matrix can be trained within 100 episodes, whereas if the state transitions are expanded to up/down/same/up2/down2 then the MDP Q-Matrix can be trained within 1500 episodes with a marginal improvement efficiency/throughput and on how quickly the optimum state is reached.

5.1.2 Summary of testing scenarios

The testing scenarios conducted within the project encompass various aspects of the Network Slicing and management taking in consideration the use cases defined in D2.1:

- The first testing scenarios are related to the RAN slicing part, starting by a quantitative comparison of control logic topologies (centralized, decentralized, and distributed)

where the focus is on evaluating trade-offs related with control loop latency and resource consumption associated with each topology. Next, a Dynamic RAN Slicing testing scenario allows to evaluate the performance of the network and the user performance when the network is dynamically sliced. Here we observe 6 scheduling scenarios obtaining different resource utilization.

- Yet regarding RAN slicing, a dynamic and heterogeneous setup of video streaming was implemented in context of an automated factory in order to evaluate the impact of the discrepancies between RAN systems and their simulators on the performance of Reinforcement Learning (RL) policies, evaluate the need for robust RL policies against resource competition and finally the potential performance enhancement in RAN slicing by considering UE positions.
- The second part focuses on evaluating Network Slicing operations in the data plane using UWS-OVS Software. It assesses adaptive Slice Management performance by measuring the time taken to create varying numbers of slices. Additionally, it evaluates the performance of the UWS-OVS data plane agent in complex scenarios involving diverse traffic types with differing Quality of Service (QoS) needs, all varying for the network resources concurrently.
- Next, the evaluation of E2E slice creation and deployment is divided in two main areas: Empirical analysis quantified on the duration for deploying a slice between two distinct UE. And, at management layer, specifically within the MANO framework, where the process of creating and decommissioning Network Slices using E2E procedures was examined. This evaluation involved the topology discovery through UWS controller and subsequent inventory updates. Additionally, it involved the creation of Network Slice Instances (NSI) in response to topology discovery or triggered by the Voice Assistant, where time measurements for procedure execution were taken. Furthermore, the evaluation measured the deletion time of a Network Slice instance, aiming to assess the efficiency and promptness of the decommissioning process.
- Finally, on the third part, was realized an empirical evaluation of the Reinforcement Learning (RL) model for optimizing the configuration of a 5G Radio Access Network (RAN). The evaluation was realized using two scenarios:
 - The first Reinforcement Learning testing scenario consists of using the Matlab Reinforcement Learning Toolbox to learn the optimum CQI/MCS state given the definition of the probability of BLER (specified by parameter E to emulate the network) for each CQI/MCS state, for example:
 - $MDP.E(1) = 0.01;$
 - $MDP.E(2) = 0.01;$
 - ...
 - $MDP.E(15) = 1.0;$
 - The second Reinforcement Learning test scenario consists of learning the optimum CQI/MCS state for each of the 12 User Equipment's in the Network Simulation described in D5.2 section, 6.3 RAN link DRL algorithms. Realized by directly interconnecting 12 instances of the Matlab Reinforcement Learning RLC system to each of the UEs in the network simulation or performing this via

much more efficient TheRLib. These programs have provided a good insight into the pathway towards realising Reinforcement Learning for O-RANs via FlexRIC.

5.2 Validation and performance evaluation results

This section discusses the empirical experiments carried out in order to validate and evaluate the 6G BRAINS components delivered as outcomes of Work Package 5.

5.2.1 FlexSlice RAN slicing testing and evaluation

We have prototyped the FlexSlice framework on top of a platform comprising components from OpenAirInterface (core network, gNB as E2-Node, UEs) and FlexRIC (nearRT-RIC, xAPP). Specifically, the control logic abstraction is developed in xApp, nearRT-RIC, and E2-Node, and messages in between are encapsulated into SLA SM and Slice SM. Our evaluation includes two aspects: and (1) quantitative comparison of three control logic topologies, and (2) network and user performance when RAN is dynamically sliced.

A. Quantitative comparison of three control logic topologies

In this part, we quantify the trade-off between three distinct control logic topologies (i.e., centralized, decentralized, and distributed) in terms of control loop latency and resource consumption. In their respective cases, control logic abstraction is deployed in the xApp, nearRT-RIC, and E2-Node to handle the service requirements in SLA SM (received through service APIs), which are then mapped to the slice configuration in Slice SM (sent via RAN APIs).

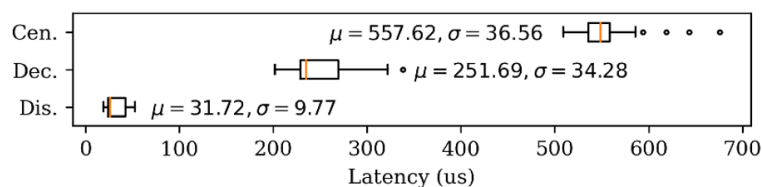


Figure 5.2.1.1: Reference Simulation of Reward Calculation

Control Logic Topology	Control Logic Location	Baseline Memory Usage	Extra Memory Usage	Scaled to N E2-Nodes
Centralized	xApp	3.41MB	0.23MB	1
Decentralized	NearRT-RIC	3.50MB	0.10MB	1
Distributed	E2-Nodes	1.19GB	0.26MB	N

Figure 5.2.1.2: Memory consumption comparison.

First, the control loop latency is measured between the control logic abstraction location and the corresponding RAN function in Figure 5.2.1.1. We can see that the distributed topology has the lowest mean value and least variation, the reason behind this is that there is no need to leverage the protocol stack, i.e., Stream Control Transmission Protocol (SCTP), between E2-Node and nearRT-RIC as well as between xAPP and nearRT-RIC for communication. Such benefit is essential to realize time-critical business logic into the real-time programmable RAN

UP functions for deterministic behaviour. In contrast, the decentralized and centralized topologies take 8x and 17x more latency, due to the extra one-hop and two-hop operations, respectively.

We then measure the memory usage of xApp, nearRT-RIC, and E2-Nodes in Figure 5.2.1.2 before and after introducing the control logic abstraction. First, the extra memory incurred by realizing the control logic abstraction in nearRT-RIC (decentralized topology) is minimal. This is because the nearRT-RIC was originally designed to communicate with xApp and E2-Node via SLA SM and Slice SM, so no extra message handling is required. Also, when we compute the ratio of the extra memory usage to the baseline memory usage, the smallest result occurs in the distributed topology. This is because an E2-Node already requires a larger baseline memory to accommodate all RAN functions. But when a single business logic would like to control N E2-Nodes ($N > 1$), this extra memory usage for control logic abstraction needs to be deployed in every E2-Node due to the distributed nature.

To sum up, there are pros and cons to deploying control logic in different locations. First, the centralized topology enables the xApp to directly define its requirements for all RAN-Nodes, and it can naturally control multiple E2-Nodes without additional resources (cf. Figure 5.2.1.2), but at the cost of a higher control loop latency (cf. Figure 5.2.1.1). In contrast, the decentralized topology requires minimal resources and can control multiple underlying E2-Nodes without further resources (cf. Figure 5.2.1.2). Also, the control loop latency is reduced compared with the centralized one (cf. Figure 5.2.1.1). Finally, the distributed topology shows minimal control loop latency with the smallest variance to enable deterministic business logic; however, control logic abstraction needs to be performed at each E2-Node.

B. Network and user performance when RAN is dynamically sliced

In this part, we aim to show the capability of FlexSlice for dynamic RAN slicing when applying different algorithms and changing parameters on-the-fly. Specifically, a single E2-Node is used to serve UE1 and UE2, and its maximum number of schedulable RBs is $R=106$. Moreover, due to radio condition, the maximum cell capacity is about 130Mbps, and we send fixed 120Mbps downlink User Datagram Protocol (UDP) traffic to each UE. It is worth emphasizing that this scenario is created to observe the impacts on dynamic RAN slicing even when RAN UPs is fully loaded.

Moreover, six scenarios are described in Figure 5.2.1.3 to serve both UE1 and UE2, with the number of slices varying between zero (Scenario 1), one (Scenarios 2 and 5) and two (Scenarios 3, 4, and 6). Also, both Network Virtualization Substrate (NVS) and Early Deadline First (EDF) algorithms apply different parameters in these scenarios: NVS uses the $\$cap\$$ parameter to identify the slice radio resource share in percentage, while EDF uses the d parameter to indicate the maximum delay and the n parameter to denote the number of RBs provided during this period.

Scen. 1	Scen. 2	Scen. 3	Scen. 4	Scen. 5	Scen. 6
No slicing	S1 (NVS,cap=70): UE1, UE2	S1 (NVS,cap=70):UE1 S2 (NVS,cap=30):UE2	S1 (NVS,cap=50):UE1 S2 (NVS,cap=50):UE2	S1 (EDF,d=2,n=150): UE1, UE2	S1 (EDF,d=2,n=150):UE1 S2 (EDF,d=20,n=620):UE2

Figure 5.2.1.3: Description of six scenarios (Here s1 and s2 represent slice 1 and slice 2, respectively).

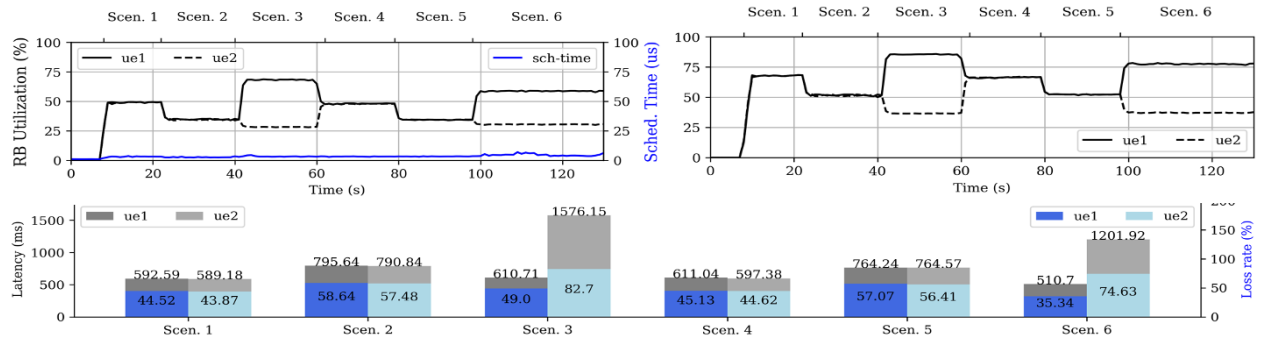


Figure 5.2.1.4: Network and User performance in a fully loaded scenario with a centralized topology.

As follows, we go through the results of each scenario in terms of RB utilization percentage and MAC scheduler processing time at E2-Node, the perceived throughput at UE, and both one-way delay and packet loss percentage as shown in Figure 5.2.1.4:

- **Scenario 1:** At 5 sec, both UE1 and UE2 consume 50% of RBs because no slicing is employed, and the PF algorithm is used at the user-level. Since the sum of UP traffic (240Mbps) is much higher than the cell capacity (130Mbps), the one-way delay and loss rate are high, but seem fair to both UEs.
- **Scenario 2:** At 22 sec, we use the NVS algorithm and set 70% of the radio resources to slice 1, so that each UE takes 35% of the radio resources. Moreover, both one-way delay and loss rate increase accordingly (around 1.3x) due to traffic overload of a finite queue size (i.e., first-in, first-out blocking queue).
- **Scenario 3:** At 40 sec, the second slice is set up with 30 % of the radio resource for UE2; thus, UE1 will occupy all 70 % of the radio resource for slice 1. As for the one-way delay in Figure 5.2.1.4, it is inversely proportional to *cap* values, but UE1 has a larger latency than in Scenario 1. This is because the NVS algorithm selects one slice at a time, so higher throughput does not imply lower delay (cf. Figure 5.2.1.4).
- **Scenario 4:** At 62 sec, both slices use *cap* = 50, so compared to Scenario 1, both UEs use a similar number of RBs and have similar throughput and loss rates. Due to the above characteristic of the NVS algorithm, the one-way delay is slightly higher than Scenario 1.
- **Scenario 5:** At 79 sec, the EDF algorithm is applied, and the second slice is removed. Note that the EDF algorithm will preserve $n = 150$ RBs within the maximum delay of $d = 2$ slots. So about 70% of RBs will be allocated to Slice 1, similar to Scenario 2; however, this scenario has a lower one-way delay because the preserved resources are fixed periodically, which is not the case for the NVS algorithm.
- **Scenario 6:** At 98 sec, the second slice is set up, and it requests about 30% of RBs in a relaxed deadline $d = 20$. In Figure 5.2.1.4, UE1 takes about 60% of RBs, while UE2

occupies 30% of RBs. It is worth noting that about 10% of RBs are unused, since we do not over-provide the values of n , which is smaller than the maximum number of schedulable RBs per slot. We also see that UE1 now has a lower latency and loss rate than Scenario 1, because it has a smaller deadline and thus takes precedence over UE2 most of the time, while both UEs are only handled by the PF algorithm in Scenario 1.

In short, the FlexSlice framework enables dynamic RAN slicing no matter what algorithms or parameters are changed on-the-fly. Also, the redesigned radio resource scheduler has a very small footprint (cf. Figure 5.2.1.4), even when RAN UPs is fully loaded.

5.2.2 Advanced RAN slicing testing and evaluation

In this subsection, we will utilize a single scenario to answer the following 3 questions:

1. *What is the impact of a minor difference between real RAN systems and their simulators on the production performance of an RL policy?*
2. *Does making RL policies robust against resource competition necessary?*
3. *Will taking into account UE positions improve the performance of RAN slicing?*

Scenario Description: Let us consider the preparation of a video streaming slice (from AGVs to the Edge) to support automated factory coordination. The uplink specification of this slice, together with its training/emulation artifacts, is described in Table 5.2.2.1. We note that in answering these questions, one is safe to assume that: (a) there are no differences between training artifacts and production workloads; (b) algorithm-wise, RAN slicing is similar in both uplink and downlink. Thus, we will conduct our study utilizing the downlink RAN slicing capability of OpenAirInterface [22] and FlexRIC [23], with Quectel-based UEs as the AGVs.

	Use Case Specs	Emulation Artifacts
Requirements	20mbps throughput & 50ms RTT (every 100ms) Evaluated every second with a 260-slot budget	
Traffic Profile	DASH on BBR (MSS: 1460 bytes)	BBR-based iPerf (MSS: 1460 bytes)
Channel Profile	Indoor Factory <10m/s	TR 38.901's InF Gauss-Markov Mobility

Table 5.2.2.1: Uplink specification of the AGV video streaming slice

Traffic-Channel Description: While our traffic emulation is straightforward, our channel emulation needs more details. Specifically, we utilize an NS-3-implemented channel simulation of TR 38.901 [24], with the following parameters:

- **Topology:** factory with dense clutter and high antenna
- **Signal Propagation:** “standard” line-of-sight, 3D alpha-beta-gamma path loss, log-normal shadows
- **Spectrum Matrix:** Rician fast fading with volume-to-surface ratio delay spread, other large-scale parameters are TR 38.901's defaults; all small-scale parameters are NS-3 indoors defaults.

Answering Q1: We will compare the learning curves and testing results of 3 training regimes: (i) on the exact OpenAirInterface at sub-2ms control loop, (ii) on a slightly modified OpenAirInterface (changed MCS mappings) at sub-2ms control loop, and (iii) on the exact OpenAirInterface at artificially pushed 10ms control loop. All are included in Figure 5.2.2.1.

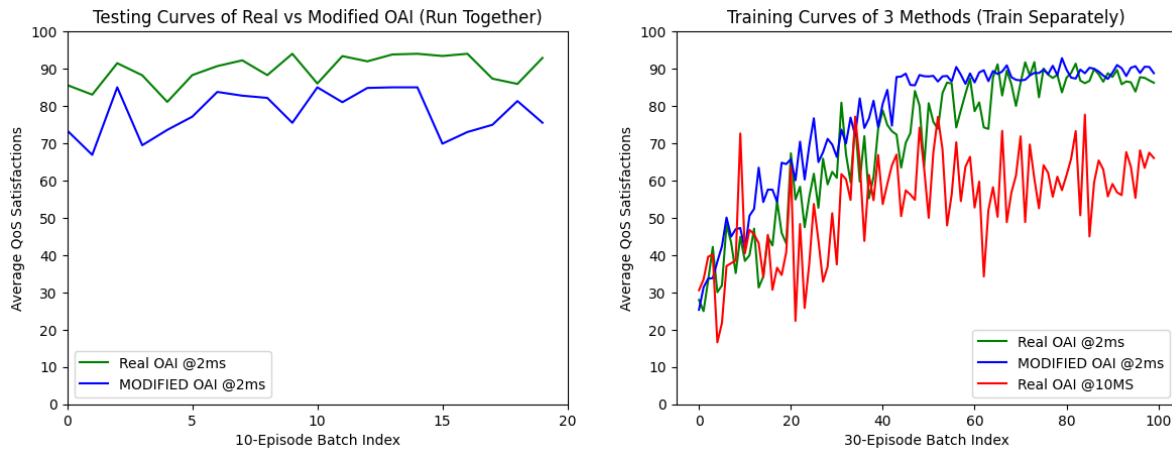


Figure 5.2.2.1: Learning and testing curves of the 3 training regimes; the regime with 10MS control-loop is excluded from testing due to bad training performance.

These results show that, in order to prepare RAN control policies for industrial 5G scenarios, one should **not** use RAN simulators as even a minor difference can lead to non-trivial performance gaps (~16%). This is due to the compounding effect of multiple coupled-and-complex rules written inside real RAN software. Additionally, we observe the unstable nature of RL when executing at high control-loop time, which justifies the control-loop time optimization we did in subsection 3.1.3.

Answering Q2 and Q3: We will compare the testing results of 4 control policies: (i) competition-robust DQN versus standard DQN, and (ii) proportional-fair intra-slice scheduling versus directional sub-slicing under Rate-based NVS; both are shown in Figure 5.2.2.2.

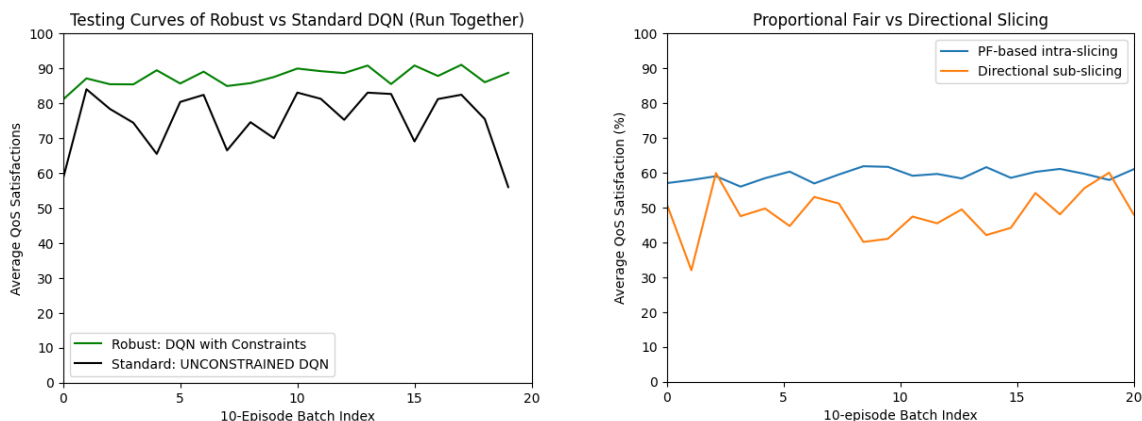


Figure 5.2.2.2: Comparing DQN variations (RAN inter-slicing) and difference RAN intra-slicing

While the left result confirms the importance of making RL policies aware of resource “cuts”, the right result shows a much more stable performance of Proportional Fair scheduling in comparison to (distance-based) Directional (sub-)Slicing. Nevertheless, we believe Directional Slicing is still a promising approach, as it is comparable, performance-wise, to the long-established Proportional Fair scheduler. However, having more parameters to tune (i.e., RSRP ranges and resource points), optimizing Directional Slicing for production remains a challenge.

5.2.3 OVS software-based Network Slicing enabler testing and evaluation

a. Adaptive slice management performance

5G and Beyond networks are dynamic and complex environments subject to continuous changes, one of the most challenging features that a Network Slice control framework must address is the flexibility to adapt to such complex scenarios. This implies a quick response in managing Network Slices and provisioning the appropriate network resources needed to enforce the slices in the data plane and meet the diverse requirements demanded by users and verticals. Figure 5.2.3.1 provides a breakdown of the time spent by the UWS-OVS in creating different number of slices when ranging exponentially the number of slices from 2 to 256. The given results refer to the time consumed since the NBI of the SCA receives an intend-based message from the Slice Manager until the slice has been successfully enforced in the data plane and an ACK message is sent back to the management plane.

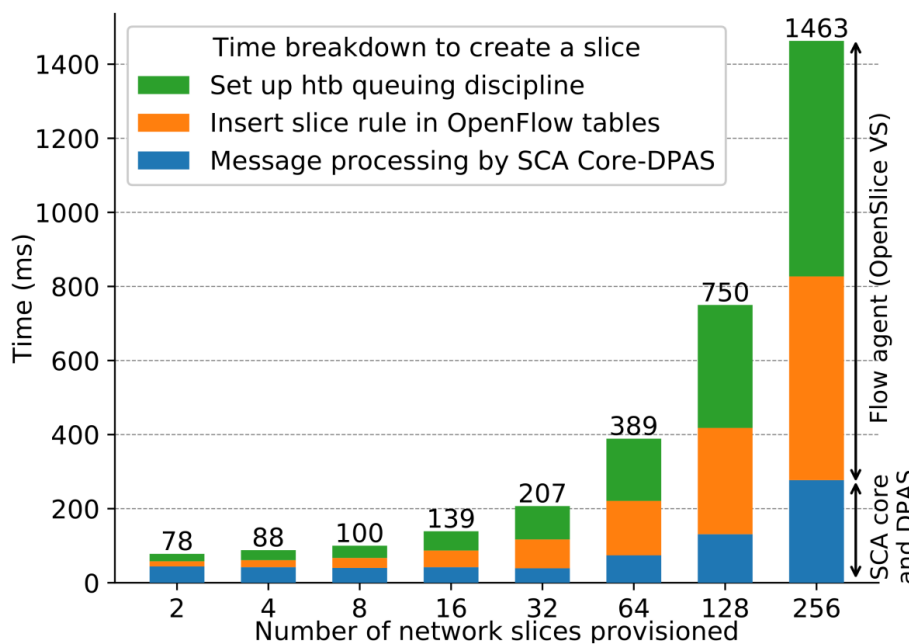


Figure 5.2.3.1: Empirical analysis of the scalability of the time required by the SCA and UWS-OVS data plane agent to create a Network Slice.

Three architectural components are involved in this task: the SCA core and the DPAS sub-modules in the SCA and the UWS-OVS flow agent. It can be observed that the major workload relies upon the Flow Agent. The Flow Agent is responsible for the insertion of the Open Flow rule matching the slice traffic in the slice definition and action table (the orange stacked bar in Figure 5.2.3.1) and setting up the htb queuing discipline where the slice traffic will be

forwarded and the QoS requirements will be enforced (the green stacked bar in Figure 5.2.3.1). Meanwhile, the time required by the SCA component to process the intend-based message has less impact on the overall time spent to create a slice (the blue stacked bar in Figure 5.2.3.1).

The gathered results indicate that the proposed solution needs 1463 ms to create 256 slices, resulting in an average time of only 5.7 ms per slice. Similar experiments have been carried out to evaluate the time to delete and modify Network Slices, obtaining better results than those when creating new Network Slices (an average overall time of 21% lower). For the sake of brevity, these results are not shown. To delete or modify a Network Slice, the SCA core does not have to process a full intend-based message, instead it just delegates to the UWS-OVS data plane agent the required action based on the slice ID reported by the external policy enforcer of the management plane, which is faster. Similarly, regarding the UWS-OVS data plane agent side, releasing network resources or modifying previously provisioned network resources is quicker than provisioning them from scratch.

Hence, these results prove that the proposed solution is able to adapt on demand to evolving 5G and beyond network traffic, adding, modifying or decommissioning in an efficient way (just a few milliseconds) the slicing policies imposed by the upper cognitive management layers.

b. Performance under heterogeneous traffic

In Deliverable 5.1, a first prototype of the UWS-OVS was tested separately against five different use cases sending homogeneous traffic matching the different ITU traffic categories or a mix of them (mMTC, eMBB and URLLC). In this deliverable, a final version of the UWS-OVS data plane agent has been evaluated in a more challenging heterogeneous scenario in which the previous five use cases are simultaneously handled by the 6G infrastructure.

The prototyped UWS-OVS data plane agent has been evaluated in a challenging heterogeneous scenario where traffic from 5 use cases demanding diverse QoS requirements is competing simultaneously for the available network resources. Table 5.2.3.1 provides a breakdown of the heterogeneous traffic profile of the experiments conducted. The configuration of this scenario is a realistic example of 5G and Beyond traffic which is expected in real-world 5G and Beyond infrastructures. As it can be noticed, the UWS-OVS 6G-BRAINS component has been tested with 432 slices sending traffic from a total of 280,920 users (IoT devices). In terms of bandwidth, the network handles 1,404,000 packets per second (PPS), reaching a combined bandwidth of 13.83 Gbps whereas it has to ensure the users QoS requirements committed in the agreed Service Level Agreements (SLAs) in terms on bandwidth and delay.

Use case	Testbed configuration (Verticals and IoT devices per vertical)	IoT devices	PPS	Bandwidth	Priority
2 Industrial Automation	8 Smart Factories with 2000 Robots	16,000	64,000	0.18 Gbps	2 (Highest)
5 Manufacturing Industry	8 Smart Factories with 15 AR Units	120	396,000	4.75 Gbps	3
3 Smart Cities	8 Smart Cities with 3000 Sound Sensors	24,000	384,000	4.61 Gbps	4
4 Smart Buildings	8 Smart Buildings with 100 IP Cameras	800	320,000	3.84 Gbps	5
1 Smart Agriculture	400 Smart Farms with 600 Climate Sensors	240,000	240,000	0.45 Gbps	6 (Lowest)
	432 Slices (1 slice per vertical)	280,920	1,404,000	13.83 Gbps	

Table 5.2.3.1: Testbed set up for empirical evaluation of UWS-OVS with heterogeneous traffic

In this kind of scenario, priority is a key QoS parameter since the infrastructure is dealing with heterogeneous traffic in complex scenarios where it is required to guarantee delay-sensitive and highly reliable services. In congested networks with high traffic load, packets are buffered before being forwarded, causing an additional delay. Moreover, if the buffer becomes full, packets are dropped. This fact might compromise the fulfilment of the QoS parameters specified in the SLA in terms of latency and reliability. Prioritization mechanisms address this issue by forwarding high priority traffic (demanding low latency and/or high reliability) before other traffic with less strict QoS needs. Table 5.2.3.2 provides a proposal on assigning priorities to different types of network traffic. These values have been used to set the priority for the traffic of each vertical business in the experiments carried out in the testbed (see the fifth column in Table 5.2.3.1).

Type of 5G traffic	Slice priority
Network control and management	0 (Highest)
Critical communications	1
Delay-sensitive and high reliable URLLC traffic	2
Delay-sensitive and high reliable bulk URLLC/eMBB traffic	3
High latency mMTC/eMBB traffic	4
High latency eMBB traffic	5
High latency and packet loss tolerant mMTC traffic	6
Best effort	7 (Lowest)

Table 5.2.3.2: Example of proposed priorities mapping different network traffic profiles

Figure 5.2.3.2 displays the average delay per slice for every use case when the 6G BRAINS infrastructure processes traffic from all 5 use cases simultaneously. The data correspond to network traffic captured for 10 seconds in the UWS testbed. There are several conclusions that can be drawn from this graph. Firstly, an average delay below 1 ms is achieved for all Network Slices. Secondly, as expected, the network traffic with the highest priority (URLLC category) gets the shortest delay (around 200 μ s). Furthermore, it can be observed that the slices of use cases with higher priority (2 and 5) obtain a rather stable delay in comparison with the remaining use cases (the whisker box's height is lower). This fact suggests that the network is honouring the priority parameter specified when these slices, with more severe QoS requirements, were enforced in the data plane by the UWS-OVS agent. Therefore, their traffic is treated with higher priority by the UWS-OVS flow agent, leading in a better delay performance and stability. To conclude the analysis of Figure 5.2.3.2, it can be noticed that the delay interval of the slices with less priority (use case 1 with mMTC traffic profile) is larger than the rest of the use cases. This interval measures the stability of the delay along the time (jitter). Results indicate that lowest priority traffic is buffered longer and forwarded with a more irregular pattern depending on the network congestion. It depends on how other slices with higher priority are consuming network resources. In the worst-case scenario, the jitter varies roughly 250 μ s (between 650 and 900 μ s) for use case 1 which is a more than acceptable value for this kind of use cases. Concerning reliability, the packet loss ratio is also 0% for all use cases so no graph is provided in this regard.

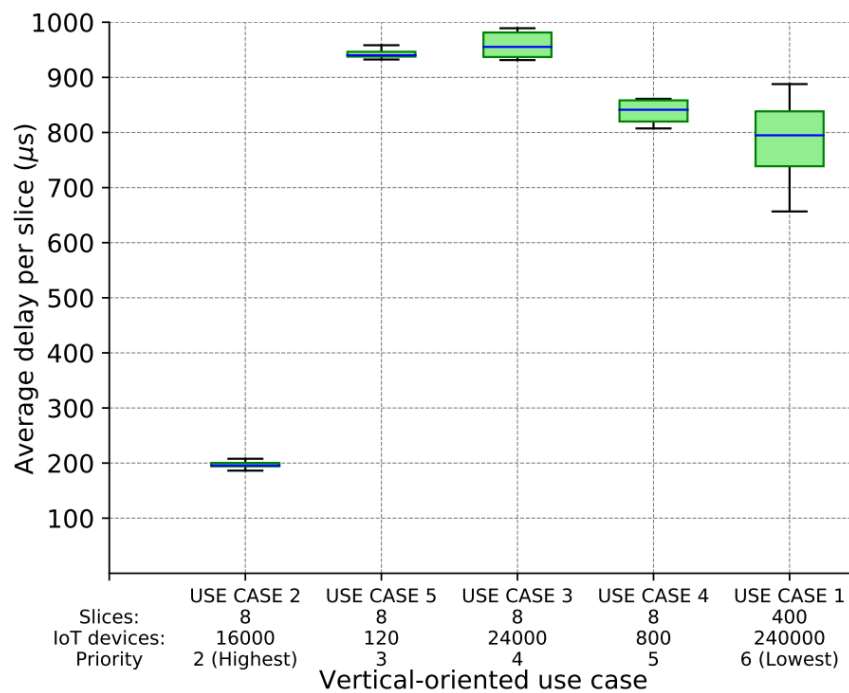


Figure 5.2.3.2: Average delay per slice and use case (in μ s) when processing heterogeneous network traffic from 280,920 IoT devices and 432 slices by the UWS-OVS data plane agent.

Finally, Figure 5.2.3.3 depicts the system behaviour in terms of bandwidth when dealing with the network traffic transmitted simultaneously from the 5 different use cases. The results correspond to a 10-second interval where the average bandwidth per slice has been computed at 0.25 second intervals. It can be observed that, for all use cases, the proposed framework is able to deliver a guaranteed bandwidth service within the boundaries agreed in the SLAs between the verticals and the service provider. Regarding the number of packets, the infrastructure can manage around 1,2500,000 packets for this experiment while the backhaul/backbone Network Slicing capacity (measured average overall bandwidth obtained over the backhaul/backbone network) is 12.36 Gbps. Therefore, it can be concluded that the proposed UWS-OVS data plane agent has been empirically validated against a large-scale infrastructure. It is suitable to deliver networking slicing with guaranteed QoS requirements in 5G and Beyond networks dealing with simultaneous and heterogeneous traffic. The UWS-OVS solution prototyped as 6G BRAINS component has shown very promising results in terms of the number of slices supported, and in terms of the bandwidth, delay, jitter, and packet loss able to be warranted in 6G networks.



Figure 5.2.3.3: Average bandwidth per slice for each use case in a scenario with heterogeneous 5G IoT traffic.

The measured results correspond to a 10-second interval where 5G network traffic is simultaneously sent for 5 vertical-oriented use cases. The combined total bandwidth achieved, i.e., backhaul/backbone Network Slicing capacity, is 12.36 Gbps

5.2.4 XDP hardware-based Network Slicing enabler testing and evaluation

To evaluate the effectiveness of the proposed XDP Hardware-based SmartNIC slicing enabler, the framework was subjected to a constant bandwidth rate of 25Gbps. This rate represents the theoretical limit of the utilized Network Card (Netronome Agilio CX 2x25GbE SmartNIC). A comparison was made between the Network Card's performance under default conditions (one hardware queue and one socket) and its performance when the combination of XDP

hardware offload and XDP at the driver level was used. This comparison is depicted in Figure 5.2.4.1 to Figure 5.2.4.5.

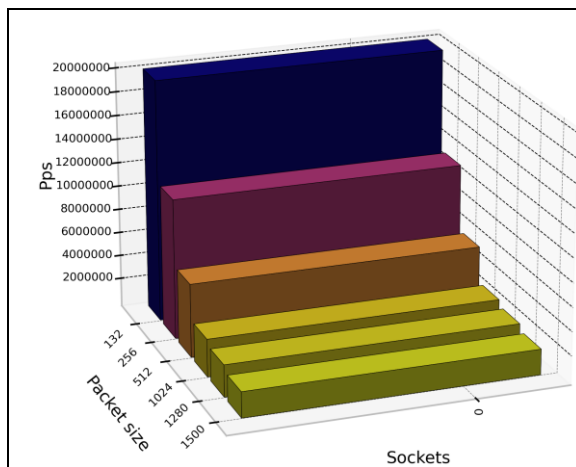


Figure 5.2.4.1: Packet Per Second, 1 Socket

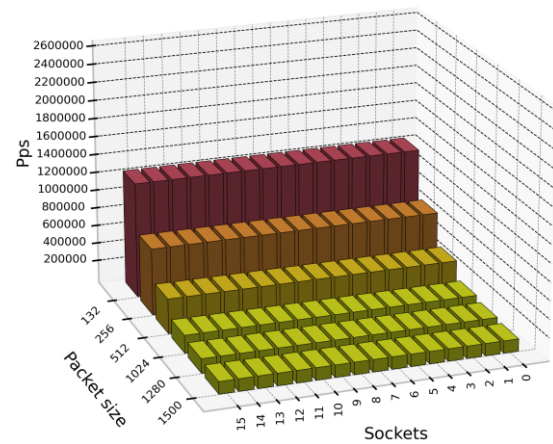


Figure 5.2.4.2: Packet Per Second, 16 Socket

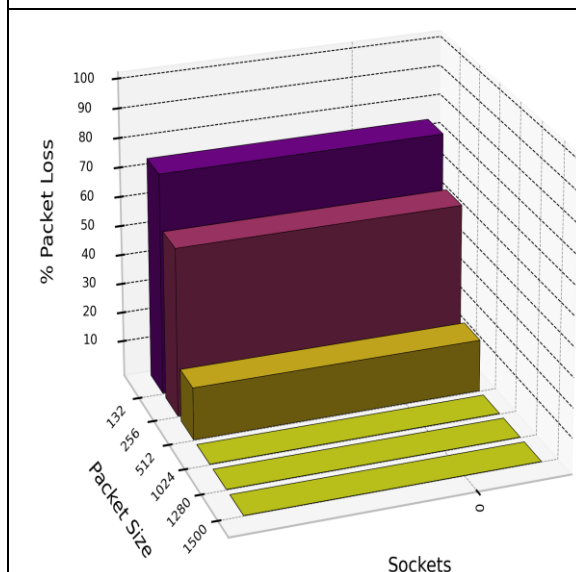


Figure 5.2.4.3: Packet Loss, 1 Socket

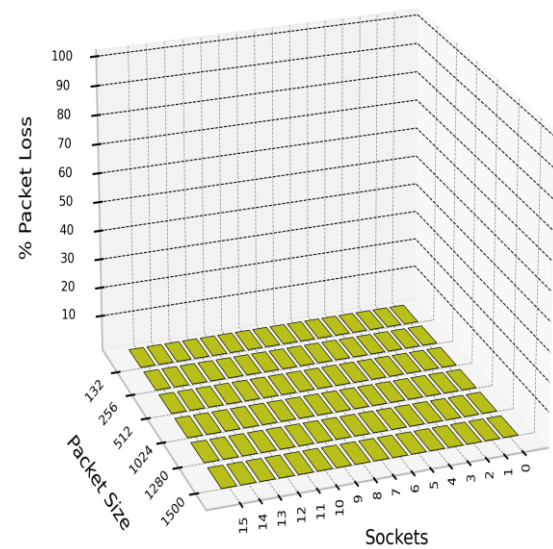


Figure 5.2.4.4: Packet Loss, 16 Sockets

The proposed framework possesses the capability to classify and divide incoming network traffic into several hardware queues (16 in the demonstrated example). These queues are then bound to an equivalent number of sockets at the driver level. This strategy enables segmentation of network traffic into diverse channels, which can then be forwarded directly into user space. This process bypasses the restrictions imposed by the Linux Kernel Network Stack. As demonstrated in Figure 5.2.4.2 and Figure 5.2.4.4, performance markedly improves with an increased number of sockets. The system achieves a maximum packet rate of 1,200,000 packets per second per socket, with zero packet loss, as measured from the Network Interface Card (NIC) to user space. For a more comprehensive understanding and in-

depth detail of the proposed solution, the most recent outcome of the 6G Brains project, published in [27], can be referred to. This solution has enabled the 6G Brains project to significantly enhance Network Slicing performance results for the wired segments of the infrastructure, demonstrating notable progress compared to previously reported achievements.

5.2.5 E2E Network Slicing testing and evaluation

5.2.5.1 E2E Network Slice control (UWS)

This subsection provides an empirical validation of the spent time of deploying an E2E Network Slice between two different UEs. The scenario consists of 4 different zones and 2 tenants per zone. 8 antennas model x310 are connected to the testbed where each antenna belongs to a different tenant. The experiments were done randomly by choosing 2 UEs among all the available in the scenario (2048 UEs). Once both UEs were chosen, the framework calculated the shortest path between both UEs gathering the information regarding all the network interfaces where the Slice Manager needed to enforce each of the Network Slice Instances (NSIs). Once that information was collected, the Slice Manager created all the NSIs and sent them to each SCA deployed in distributed fashion across the infrastructure. Thus, there are four possible combinations for the evaluated scenarios:

- Both UE belong to the same tenant and are in the same zone.
- Both UE belong to the same tenant and are not in the same zone.
- Both UE do not belong to the same tenant and are in the same zone.
- Both UE do not belong to the same tenant and are not in the same zone.

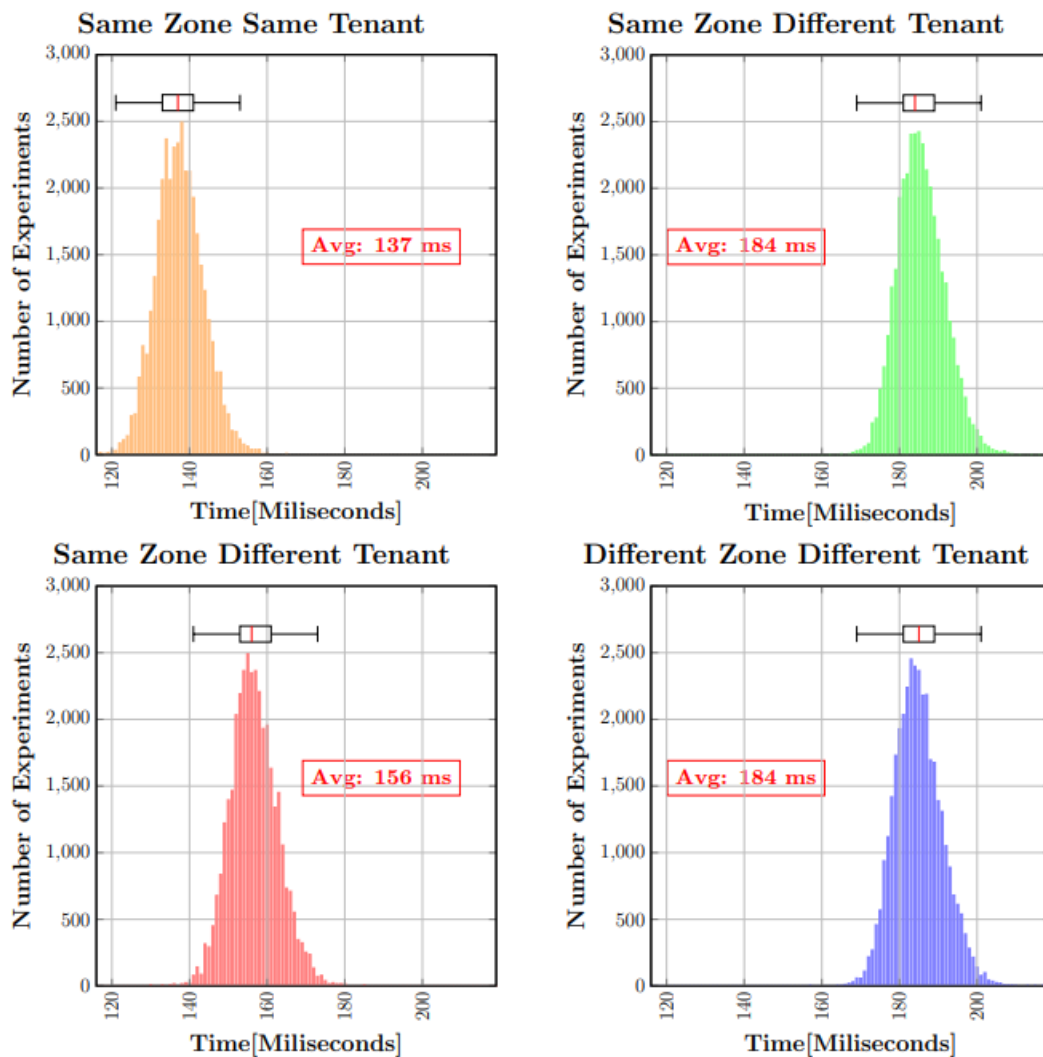


Figure 5.2.5.1.1: Slice manager E2E Network Slice creation time

As shown in Figure 5.2.5.1.1, the obtained results for the four combinations indicates a stable behaviour of the Slice Manager regardless of the location and tenant of the 2 UEs acting as endpoints of the slice. The average time to enforce an E2E slice ranges between 137 and 184ms, demonstrating the feasibility of the proposed solution for 5G and Beyond networks.

5.2.5.2 E2E Network Slice management and orchestration

This subsection presents the outcomes of different experiments focusing on the MANO layer, responsible for lifecycle management of a Network Slice on operator's side, encompassing tasks like commissioning and decommissioning, which is integrated with UWS Topology Controller. The first scenario involves the topology discovery, wherein ONAP, facilitated by the UWS Topology Inventory Agent, retrieves the current available topology, and subsequently updates its inventory, with all network resources. The second scenario, is dedicated to the Network Slicing commissioning, aiming to execute and evaluate the creation of the Network Slice via UWS Slice Manager. The testing starts from the NBI of ONAP, creating

the different service instances (CSI, NSI), all the relationships in the inventory and the requesting time from ONAP SBI to UWS Controller.

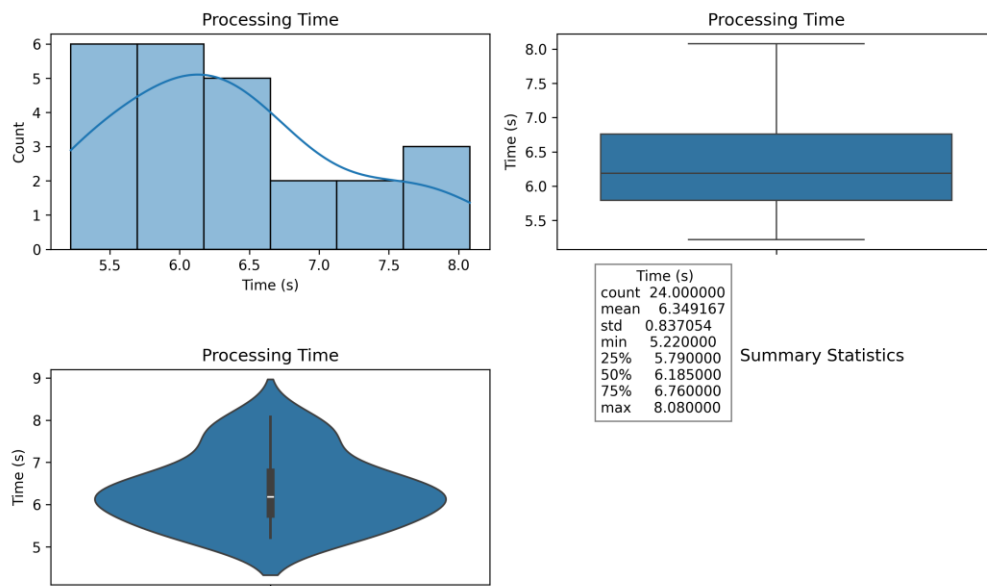


Figure 5.2.5.2.1: ONAP Topology Synchronization Process Times

Figure 5.2.5.2.1, indicates a consistent and stable performance trend. The average processing time observed across multiple instances remains steady at approximately 6.35 seconds. This suggests a reliable timeframe for the execution of topology discovery tasks, in retrieving and updating the available network topology information through the ONAP-UWS topology agent integration.

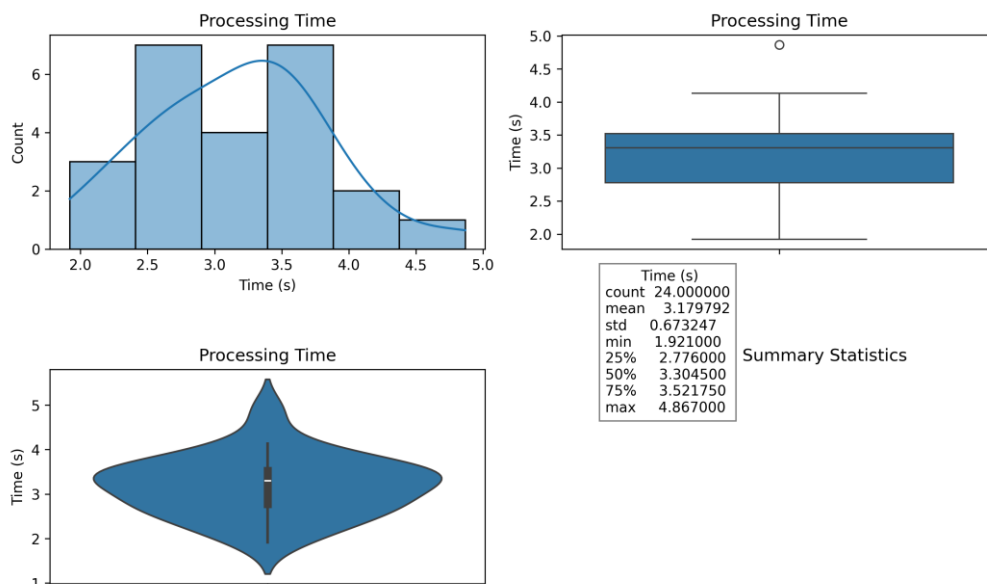


Figure 5.2.5.2.2: ONAP Slice Creation Processing Times

Figure 5.2.5.2.2, the graph illustrates the processing time of slice commissioning, demonstrating consistent performance with an average task execution time of 3.18 seconds. Considering their values and the absence of real-time operational demands on the MANO

layer, this performance signifies an efficient time frame for the task, indicating a satisfactory level of performance.

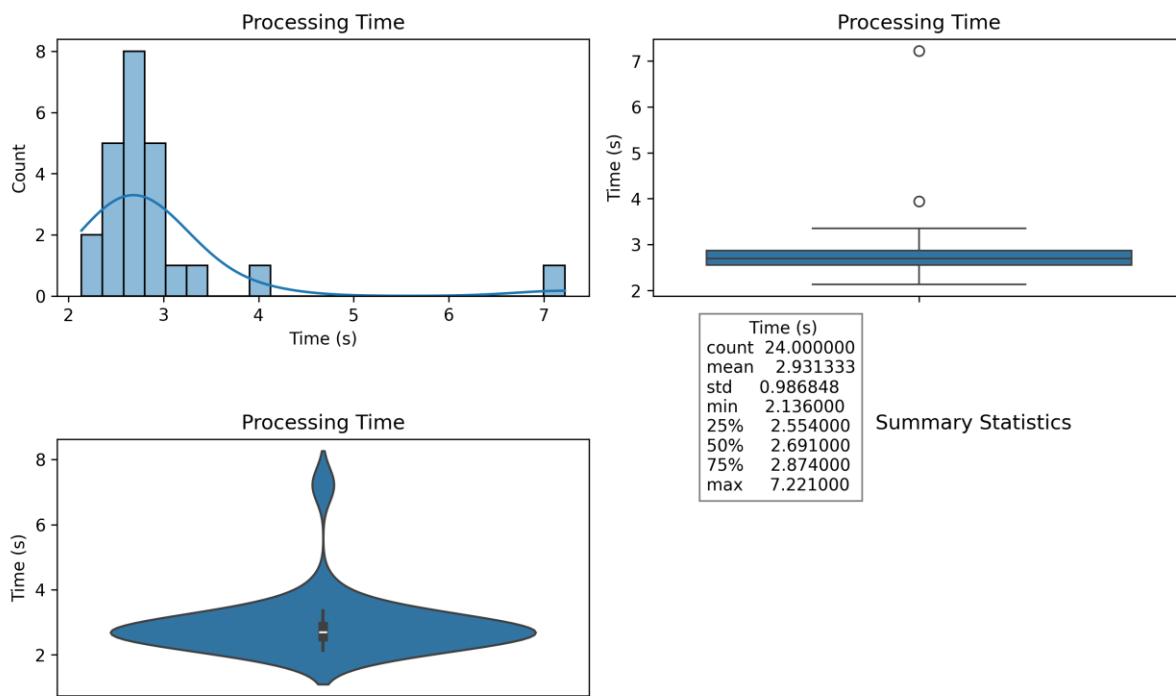


Figure 5.2.5.2.3: ONAP Slice Deletion Processing Times

Regarding the slice deletion, which includes the validation of existing service slices, UWS slice manager request to delete the network slice and finally the deletion of all objects in the inventory, we can observe on Figure 5.2.5.2.3 that in average it took 2.9 seconds to realize the operation.

Summarily, these obtained values are good, especially considering that we are not currently operating within a real-time framework.

5.2.6 Process slicing testing and evaluation

Experiments have been conducted in process slicing applied to a mitigation tool produces promising results in two different experiments, emulating that a cyberattack process was launched and how the proposed process slicing can mitigate this by managing the computing resources required by that process. In the first experiment, the CPU percentage was decreased by 91%, and the gradient of Nonvoluntary Context Switches was lowered by 88% before the attack was eventually terminated. In the second experiment, there was a decrease of 66% in both Read Bytes and System Read Calls gradients. These results demonstrate the potential of process-level approaches in enhancing cyber-security in industrial environments. Figure 5.2.6.1 shows that the metrics change along the time with the different states from: process being a possible threat, process priority changed, process stopped, and process killed.

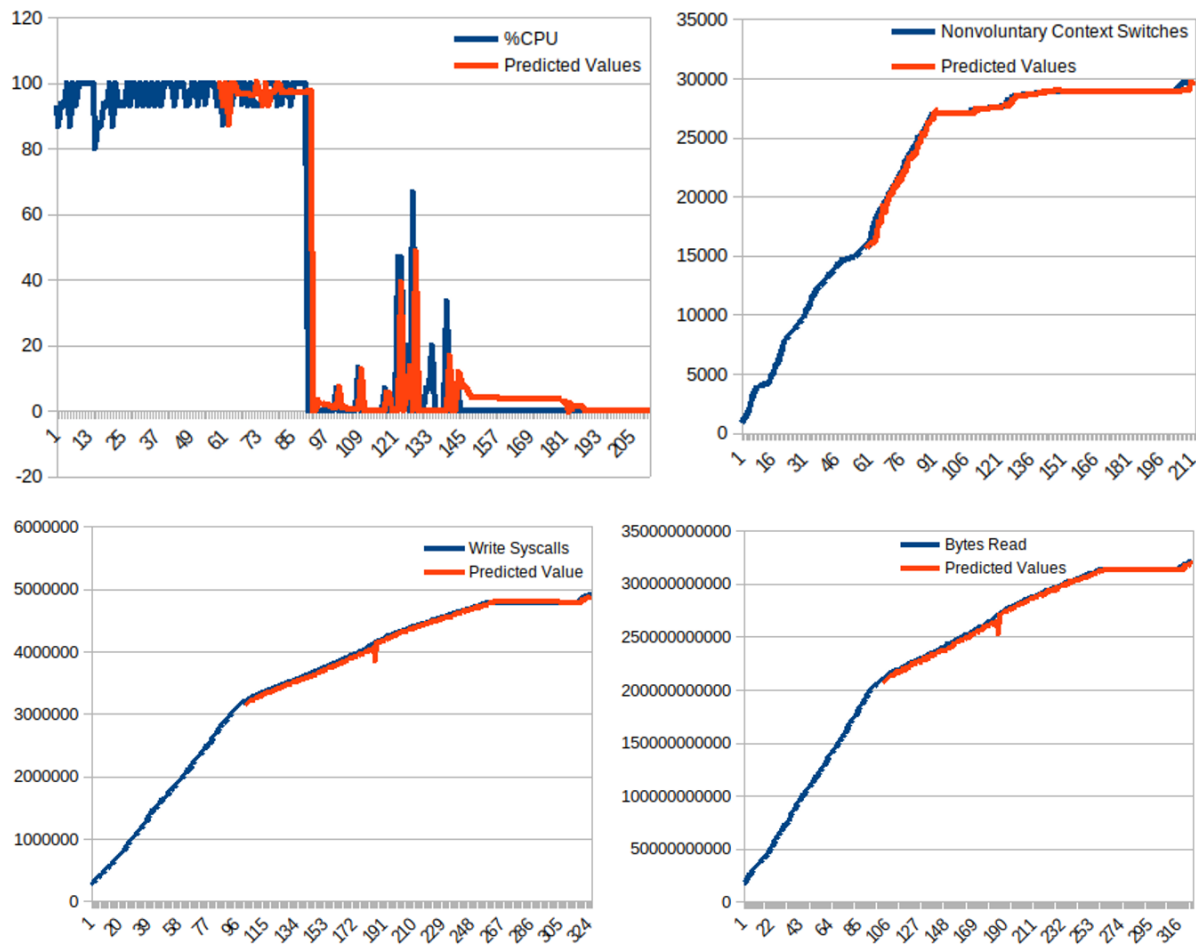


Figure 5.2.6.1: Metrics data for computing resources.

In conclusion, process slicing is a methodology created because of the need for efficient, secure, and predictable process management in complex computing environments. It offers a structured approach to resource isolation and allocation, ensuring that each process operates optimally while safeguarding the overall system's performance and security. An example of process slicing applied to security is shown to demonstrate its capacity for managing single processes.

5.2.7 AI-based RAN radio link control testing and evaluation

5.2.7.1 Results and analysis

The subsequent section focuses on the empirical evaluation of our proposed model that leverages Reinforcement Learning (RL) for optimizing the configuration of a 5G Radio Access Network (RAN). The overarching aim of this empirical exercise is to showcase how advanced machine learning techniques like RL can lead to more efficient and dynamic allocation of network resources, consequently improving the overall performance metrics of a 5G network.

For an intricate and multifaceted technological infrastructure like 5G RAN, traditional optimization methods often fall short due to high complexity, dynamic network conditions, and the need for real-time decision-making. Reinforcement Learning (RL), in contrast, is well-suited for handling such complexities. Our specific RL model, embodied by the Episode

Manager, utilizes an RL agent named 'rlQagent' operating within a Markov Decision Process Environment ('rIMDPEnv'). The agent's task is to dynamically configure the network's modulation and coding scheme (MCS), an essential aspect of 5G RAN that has far-reaching implications for network latency, throughput, and reliability.

The centrepiece of our results is a graphical representation that maps the episode rewards against the episode numbers. The 'Episode Reward' on the y-axis serves as a quantitative measure of how well the rlQagent is performing its task of 5G RAN configuration in each episode, whereas the 'Episode Number' on the x-axis indicates the sequence of interactions the agent has had with its environment.

Two experiments were performed the first for single state transitioning between MCS/CQI states and the second with up to two state transitioning between MCS/CQI states to determine if average efficiency is improved, the time to establish optimisation of average efficiency is obtained faster.

5.2.7.2 Result overview for single state transitioning

Single state transitioning between states restricts the transitions between states to [up, down, same].

5.2.7.3 Preliminary Exploration Phase (Episode 1-20)

This phase captures the initial learning stages where the rlQagent is still gathering knowledge about the rIMDPEnv. The decisions made here are mostly exploratory, as the agent starts to interact with the environment and receives its initial sets of rewards. This section can delve into the role of randomness, initial learning rates, and why it is essential for the agent to explore before exploiting its knowledge.

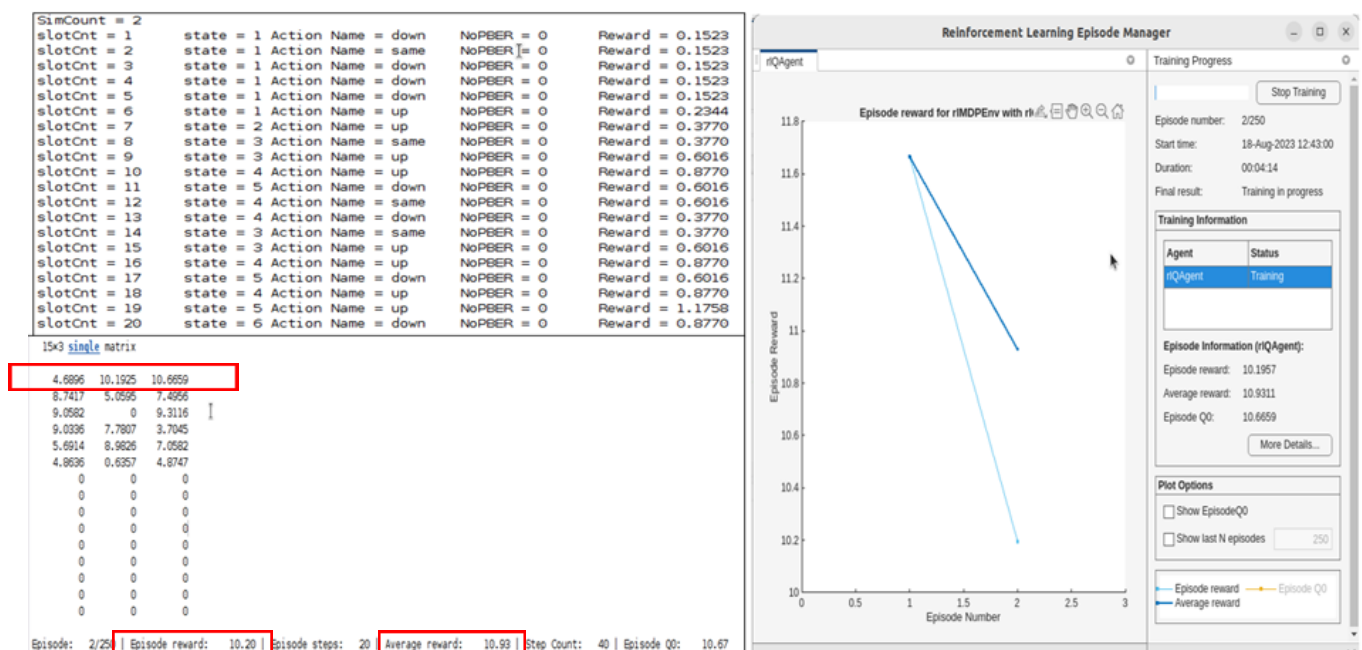


Figure 5.2.7.3.1: Single State Transitioning Episode 2

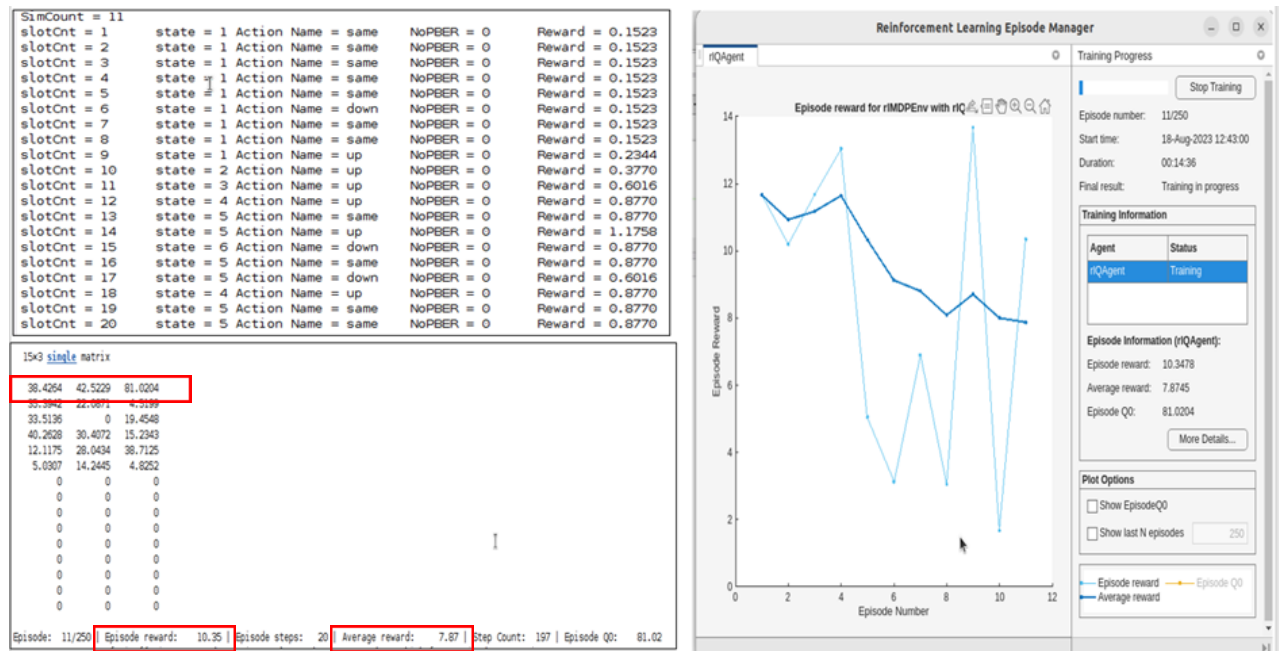


Figure 5.2.7.3.2: Single State Transitioning Episode 11

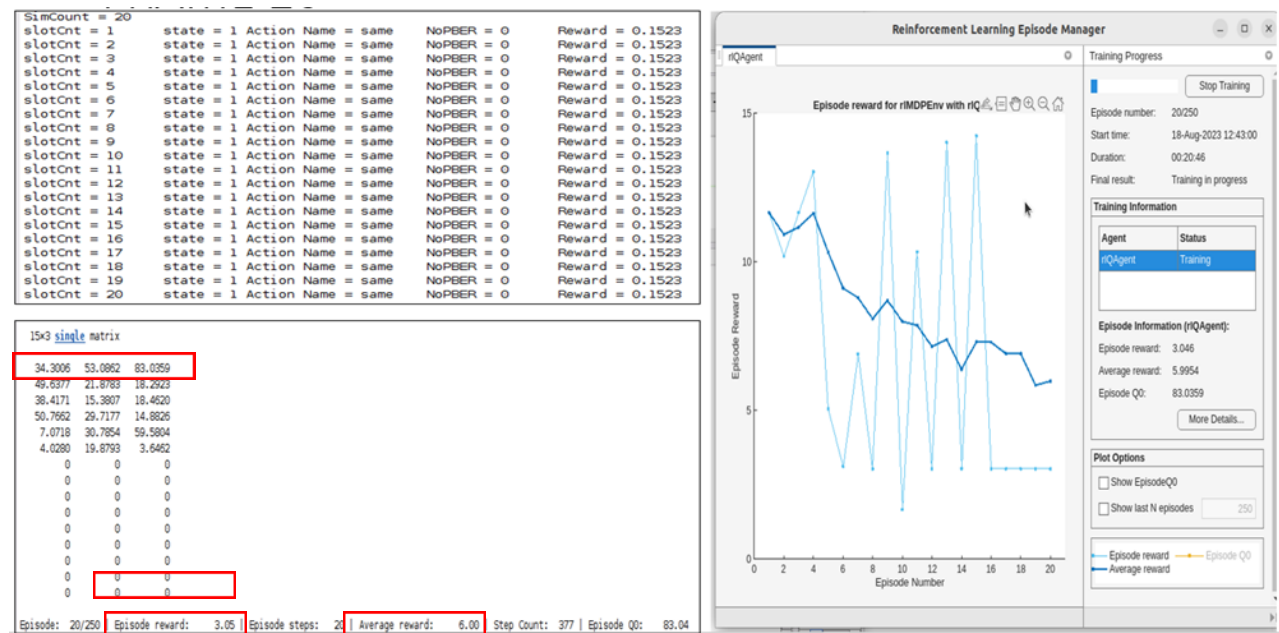


Figure 5.2.7.3.3: Single State Transitioning Episode 20

In this initial phase, the rIQAgent is in the early stages of understanding the rMDPEnv, which is evidenced by its choice of Modulation and Coding Scheme (MCS). During these episodes, the agent predominantly opts for MCS=1, a conservative and sub-optimal choice when compared to the ultimate finding of MCS=5 as the optimum level for our 5G network configuration.

Also, during this preliminary phase, it is observed that the episode rewards are consistently smaller than the average reward obtained in later stages. This is indicative of the agent's incomplete understanding of the environment, where actions are less about optimizing for

immediate gains and more about gathering data for future decision-making. The low initial rewards are an expected part of the learning curve and serve as a foundation upon which the agent can build more nuanced strategies in the subsequent phases.

5.2.7.4 Intermediate adaptation phase (Episode 21-100)

This phase represents a critical transition in the agent's learning process. Shifting from the initial exploration to a more balanced strategy, the rIQAgent starts to incorporate prior experience into its decision-making. Here, the agent begins to exploit its existing knowledge base to make more informed choices, although it continues to explore new strategies, fine-tuning its actions based on the iterative feedback received from the rIMDPEnv.

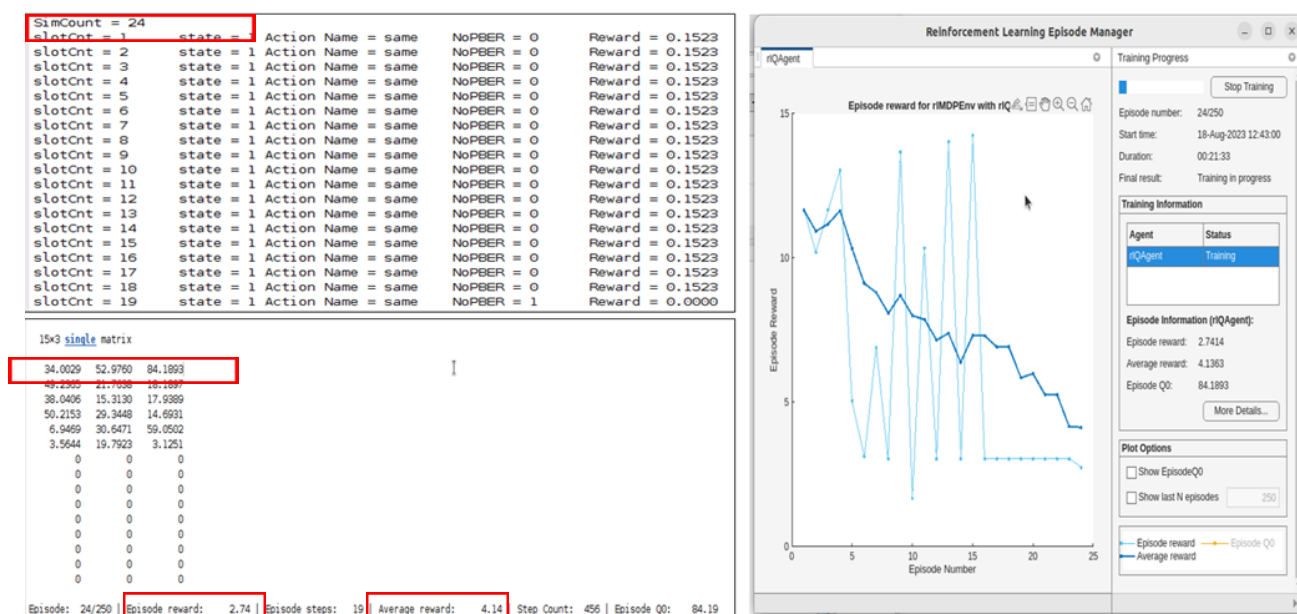


Figure 5.2.7.4.1: Single State Transitioning Episode 24

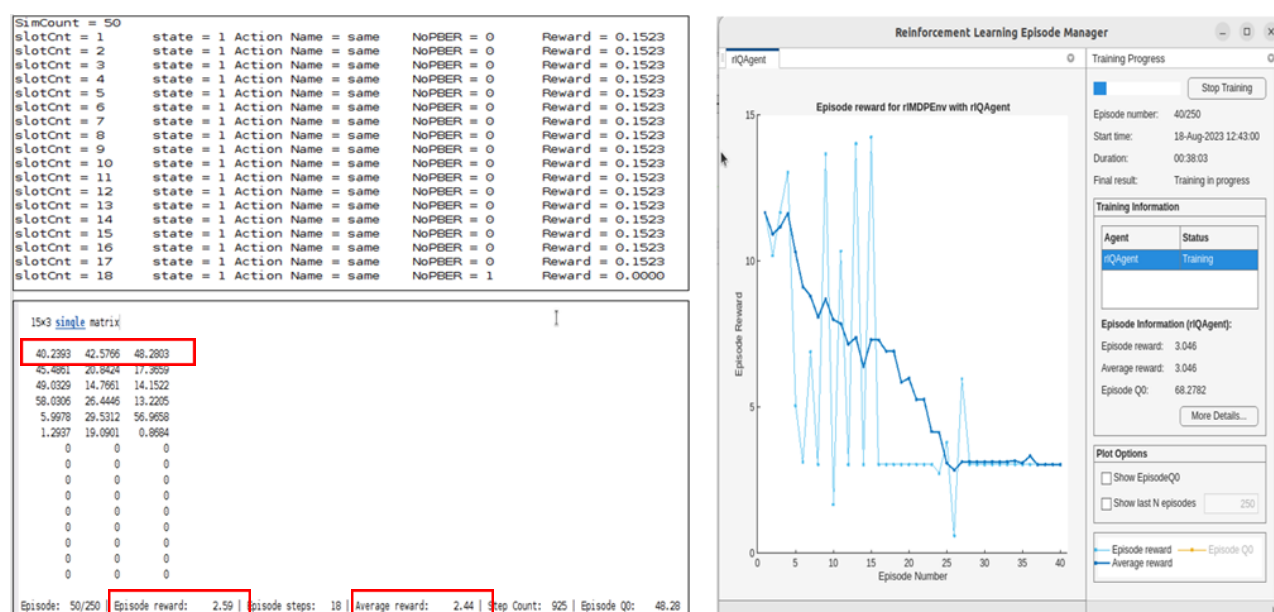


Figure 5.2.7.4.2: Single State Transitioning Episode 50

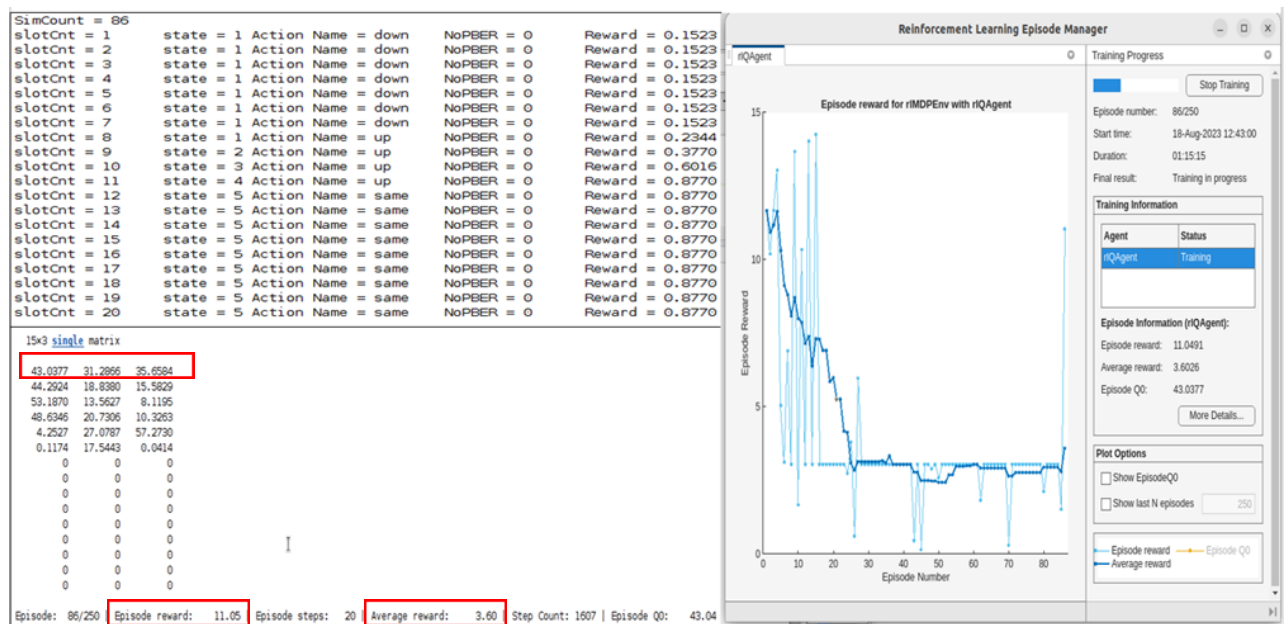


Figure 5.2.7.4.3: Single State Transitioning Episode 86

Intriguingly, even in this phase, the system predominantly opts for MCS=1. This repetitive selection suggests that while the agent is in a state of adaptation, it has not yet fully converged to the optimal MCS level. It may also indicate that the learning rate, exploration parameters, or other aspects of the RL model may need further calibration to push the agent toward making more optimal selections.

Nevertheless, this phase serves as a bridge, guiding the rIQAgent from initial exploration towards more targeted, optimized actions. The continued selection of MCS=1 serves as an instructive lens through which to examine the agent's learning curve, highlighting the intricate balance between exploration and exploitation that characterizes this intermediate stage. Further analysis and refinement may be required to propel the system into making more optimal choices in the final Converged Optimization Phase.

5.2.7.5 Converged optimization phase (Episode 100-250)

By the time the learning trajectory reaches this phase, the rIQAgent is mostly in an exploitation mode, leveraging its accumulated knowledge to make decisions that are highly targeted and optimized. The agent has spent considerable time understanding its environment (rMDPEnv) and has transitioned from a phase of exploration and adaptation to one focused squarely on optimization.

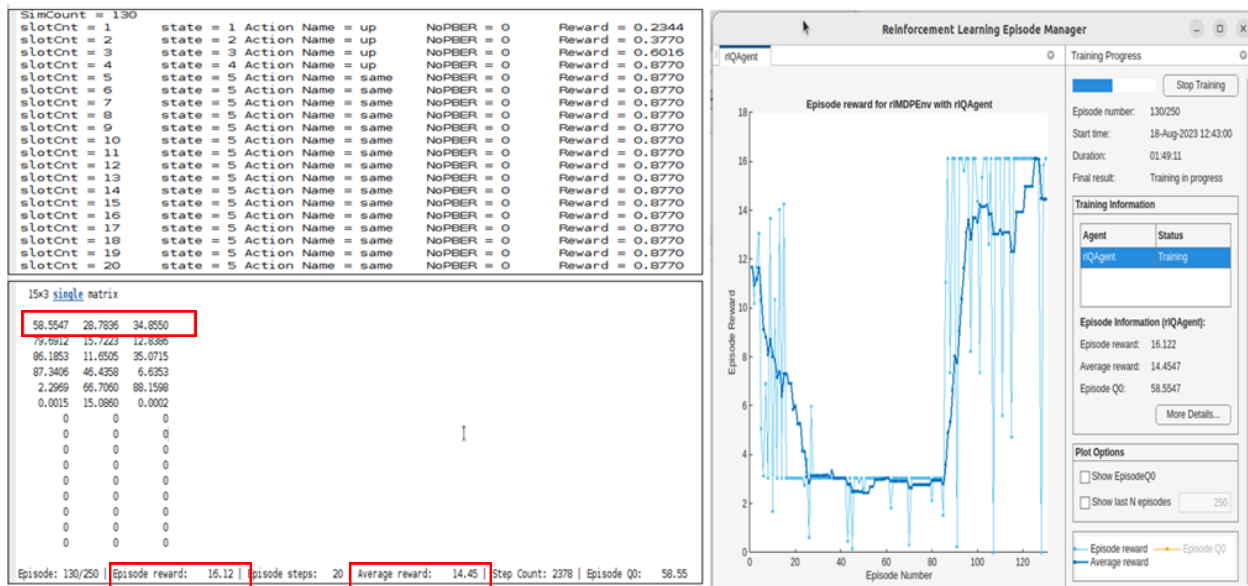


Figure 5.2.7.5.1: Single State Transitioning Episode 130

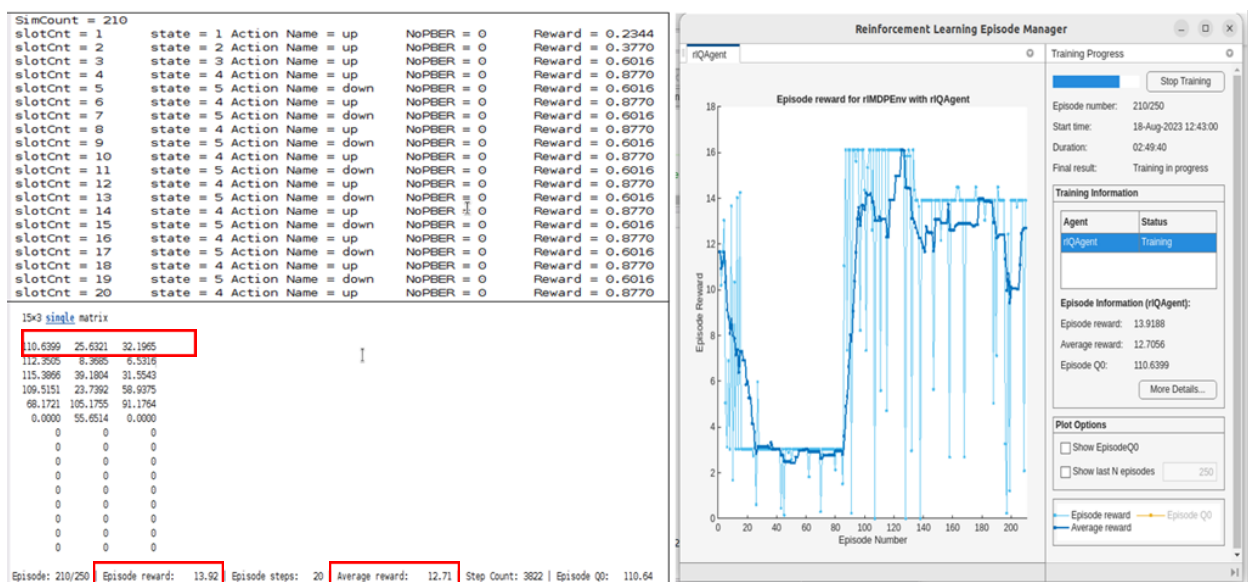


Figure 5.2.7.5.2: Single State Transitioning Episode 210

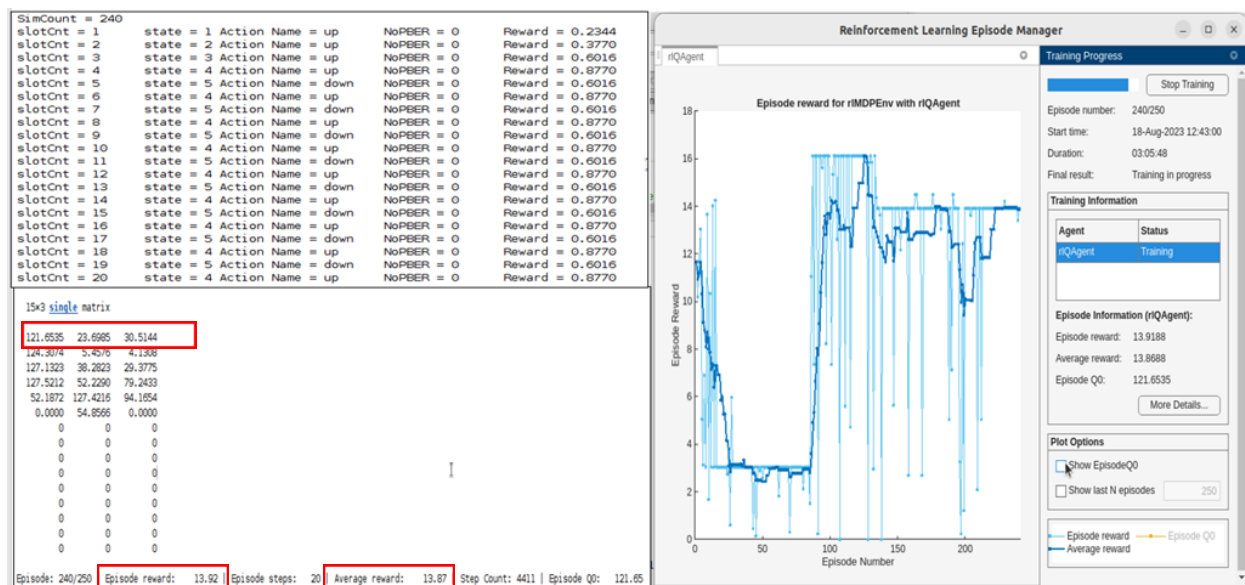
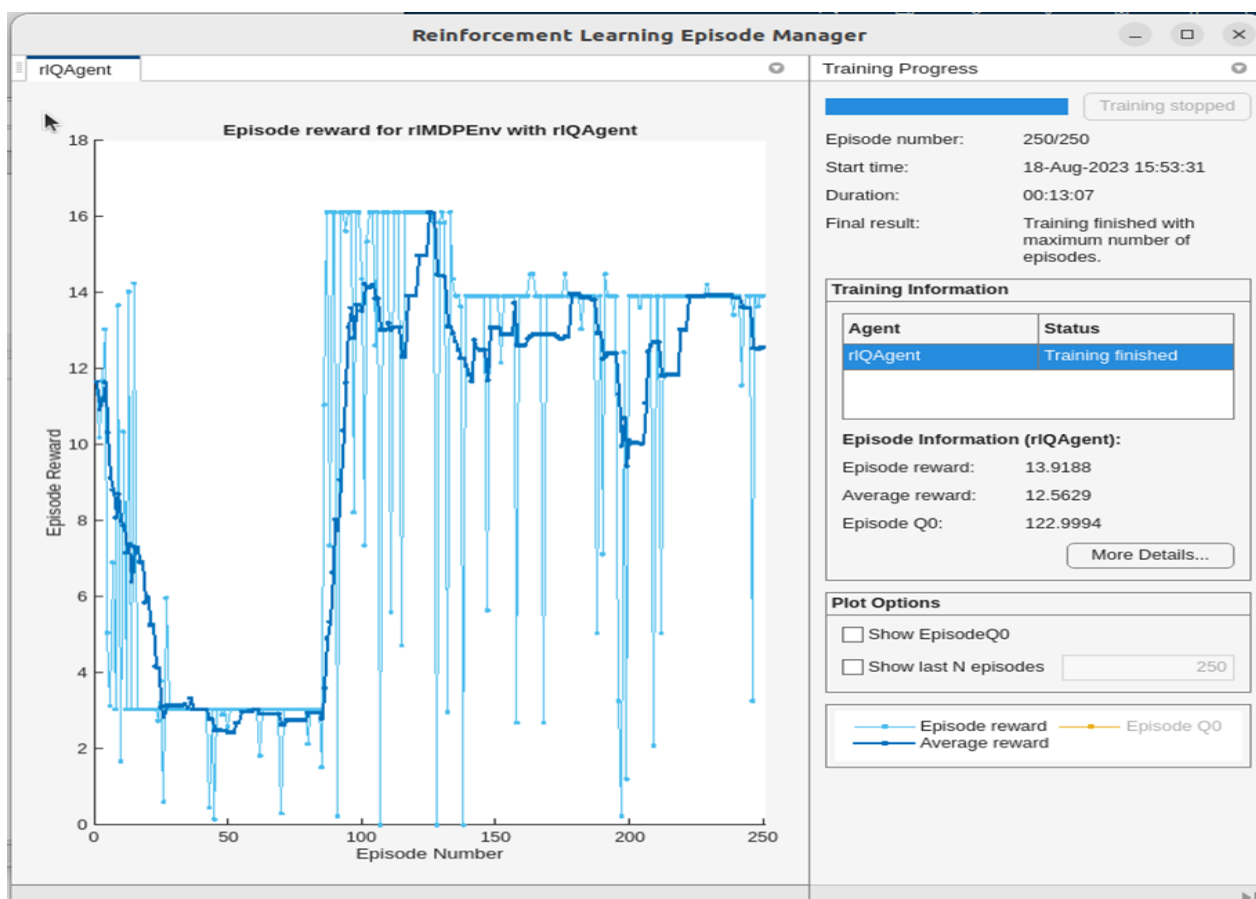


Figure 5.2.7.5.3: Episode 240



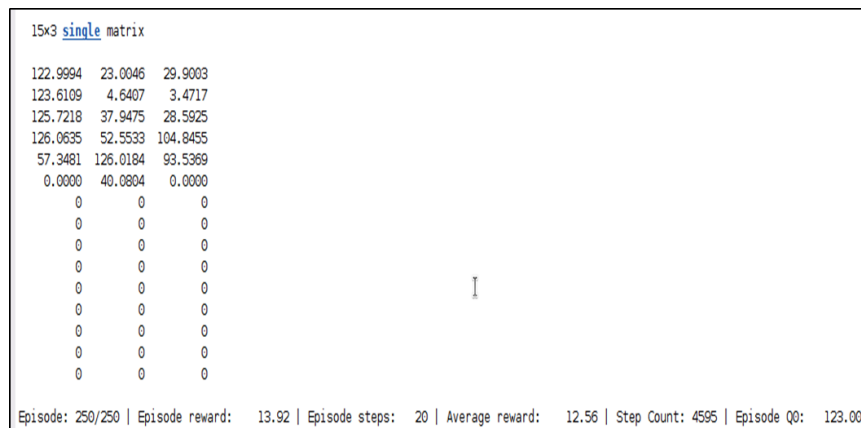


Figure 5.2.7.5.4: Single State Transitioning Episode 250

A significant milestone reached during this Converged Optimization Phase is the consistent selection of MCS=5. This is particularly noteworthy as MCS=5 is the theoretically expected optimum level for 5G RAN configuration in our model. The shift from MCS=1 in the earlier phases to MCS=5 here indicates that the agent has successfully learned to approximate the optimal policy. Furthermore, the average is 12.5629 for an episode (as shown in Figure 43) which consists of 20 slots, therefore the average reward per slot = $12.5629/20 = 0.6282$, which is an efficiency between MCS/CQI 4 and 5. It required around 250 episodes for the Reinforcement Learning state machine to reach a stable optimal state, which corresponds to 250 5G frames of 10ms duration or 2.5 seconds in real time.

The number of episodes This highlights the efficacy of reinforcement learning in achieving highly effective and dynamic configurations for complex systems like 5G RAN.

Furthermore, the episode reward during this phase is consistently higher than the average reward accumulated over the entire span of 250 episodes. This is indicative of the agent having learned a near-optimal or optimal policy for 5G RAN configuration. The elevated rewards serve as empirical validation of the model's robustness and its capability to perform exceptionally well in a real-world application.

5.2.7.6 Result overview for up to two state transitioning

Up to two state transitioning between states restricts the transitions between states to [up, down, same, up2, down2].

The average is 12.6564 for an episode (as shown in Figure 44) which consists of 20 slots, therefore the average reward per slot = $12.6564/20 = 0.6328$, which is again an efficiency between MCS/CQI 4 and 5. It required around 2000 episodes for the Reinforcement Learning state machine to reach a stable optimal state, which corresponds to 2000 5G frames of 10ms duration or 20 seconds in real time. This is considerably larger time required than for the single state transitioning system.

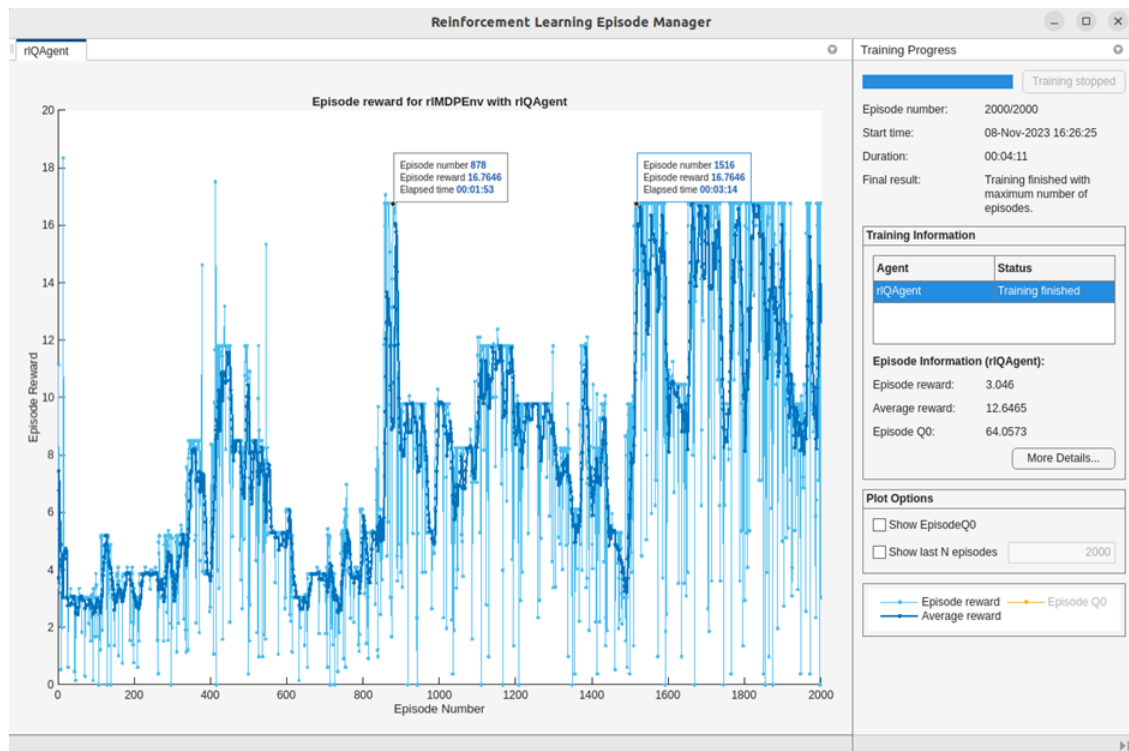


Figure 5.2.7.6.1: Up to two State Transitioning Episode 2000

5.2.7.7 Result overview for starting from best identified state

Recall that for agent starting training from state 1, the average reward is 12.5629 for an episode (as shown in Figure 5.2.7.5.4) which consists of 20 slots, therefore the average reward per slot = $12.5629/20 = 0.6282$

Now for an agent starting training from the best state identified so far, the average reward is 14.8942 for an episode (as shown in Figure 5.2.7.7.2 which consists of 20 slots, therefore the average reward per slot = $14.8942/20 = 0.7447$

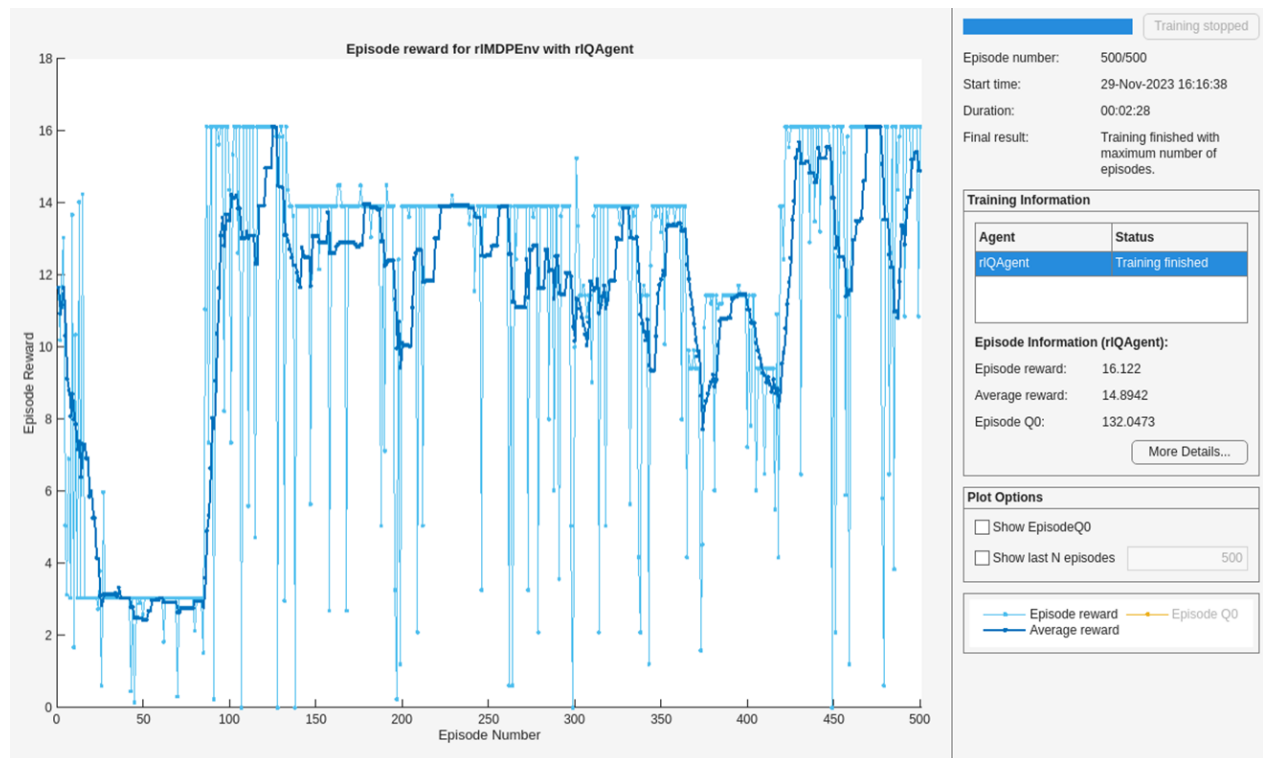


Figure 5.2.7.7.2: Episode reward for rIMDPEnv with riQAgent

5.2.8 Interfacing Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab to the 5G Network Model

The 5G network in which we are required to apply Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control has 4 User Equipment each connected to one of three gNBs, which totals to 12 User Equipment, as shown in Figure. This requires 12 instances of the Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control model.

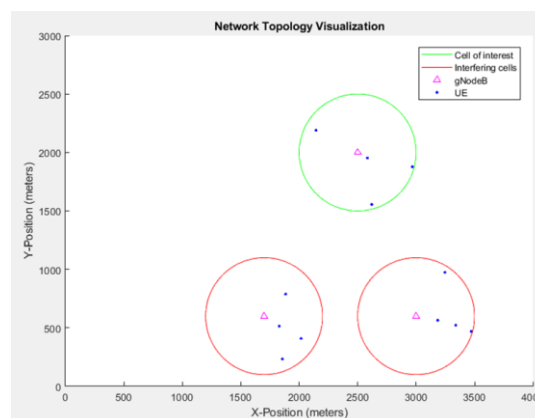


Figure 5.2.8.1: Matlab's Network Topology Visualisation of Intercell Interference Model

The BLER should be applied to RL model from network model. The QMatrix should be processed by the RL model and the MCS Action should be processed by the RL model and applied to the Network model.

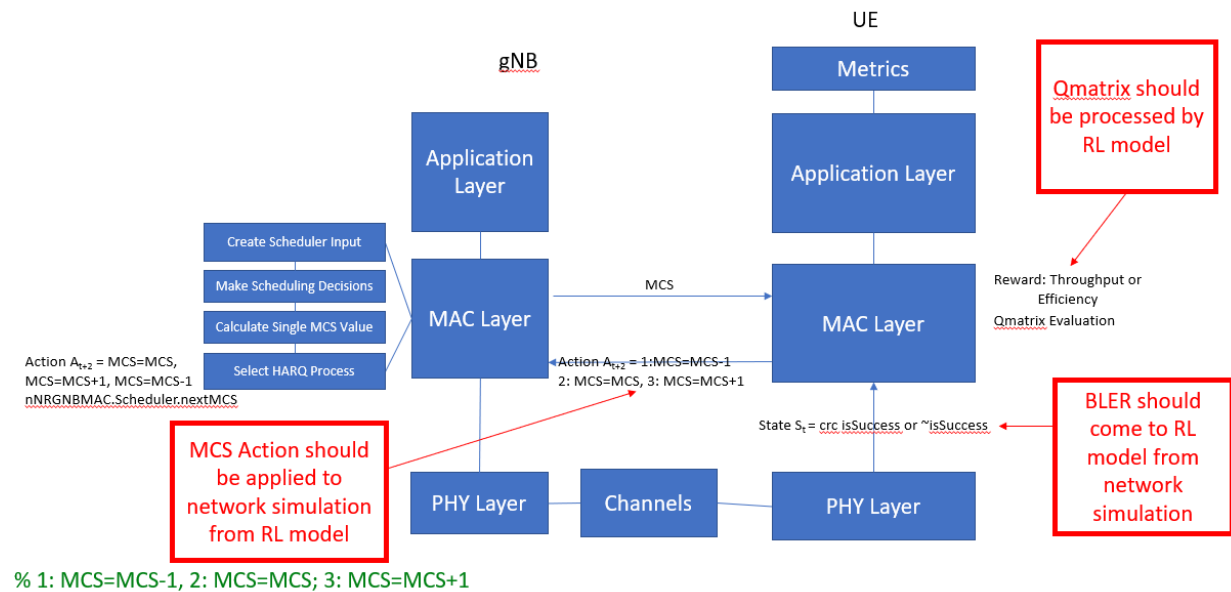


Figure 5.2.8.2: DRL Link Control Markov Decision Process

The issue now becomes how to run one Matlab network model and 12 Matlab RL models concurrently within a Linux operating system environment passing parameters between them.

5.2.9 Interfacing Reinforcement Learning Markov Decision Process (MDP) 5G Radio link Control Environment in Matlab to TheRLib

Deep Reinforcement Learning (DRL) trainings have been conducted with TheRLib in Thales premises on a modified version of the Radio Link Control MDP environment described in Section 4.6.5. To avoid modifying the core logic of MATLAB Reinforcement Learning toolbox's MDP move_function, the environment used for training with TheRLib does not model the BLER phenomenon but only the transitions between the different MCS states. The environment is used by TheRLib through the MATLAB interface as a MDP environment with discrete states and discrete actions.

To be better handled by the neural-network based DRL algorithms, the states are pre-processed with one-hot encoding. One-hot encoding transforms a single categorical variable with d distinct values, to d binary variables, each indicating the presence (1) or absence (0) of the dichotomous binary variable. It is a common approach in Deep Learning to process categorical variables with a neural network.

Figure 5.2.9.1 shows the outcome of training an agent on the Radio Link Control MDP environment with the Soft Actor Critic algorithm (SAC). The value shown in the X axis represents the number of environment steps, with each training episode lasting 20 steps (2000 episodes in total). The episodic reward reaches high values until around 13k steps, then a phenomenon of catastrophic forgetting in the DRL training is observed: the agent performance keeps dropping until the end of the training. Thus, with the default set of hyper-parameters tested, the SAC algorithm does not seem reliable for the RLC environment.

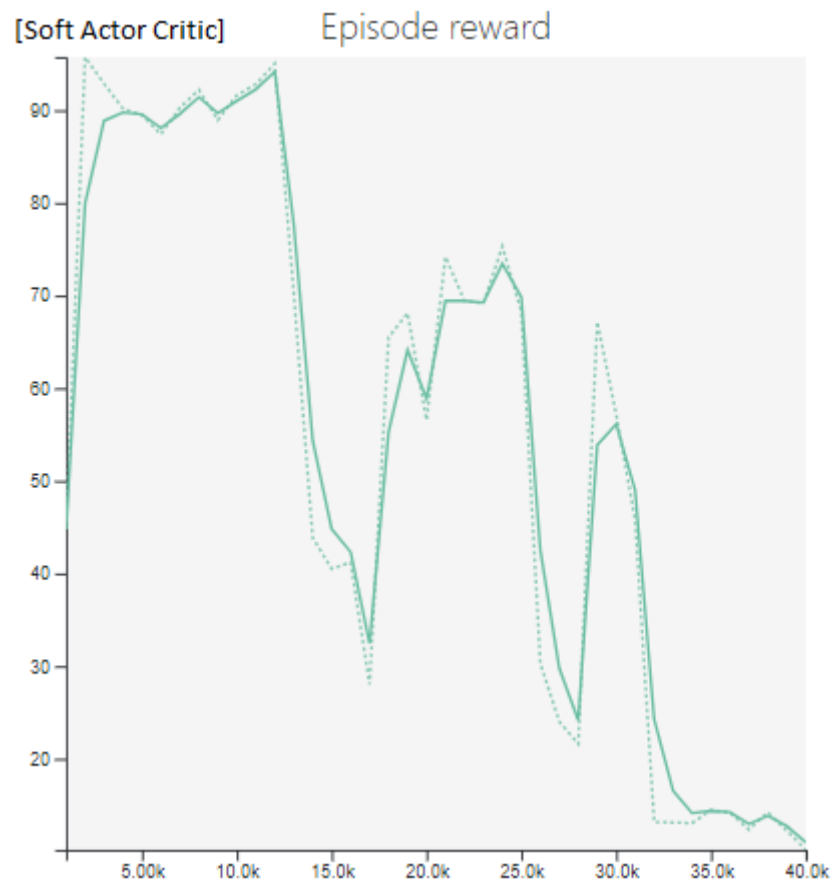


Figure 5.2.9.1: Training curve with the Soft Actor Critic (SAC) algorithm

Figure 5.2.9.2 illustrates the outcome of training an agent on the Radio Link Control MDP environment with the Proximal Policy Environment (PPO) algorithm. Like the SAC training curves, the X axis represents the number of environment steps throughout the training. However, with PPO 8 parallel environments are used to interact with, thus the total number of agent-environment interactions of this training counts $8 \times 20k$ steps (or 8×1000 episodes) processed in parallel processes. With the PPO algorithm and its default set of parameters, we can observe that the performance of the agent during the training is much more stable than with SAC.

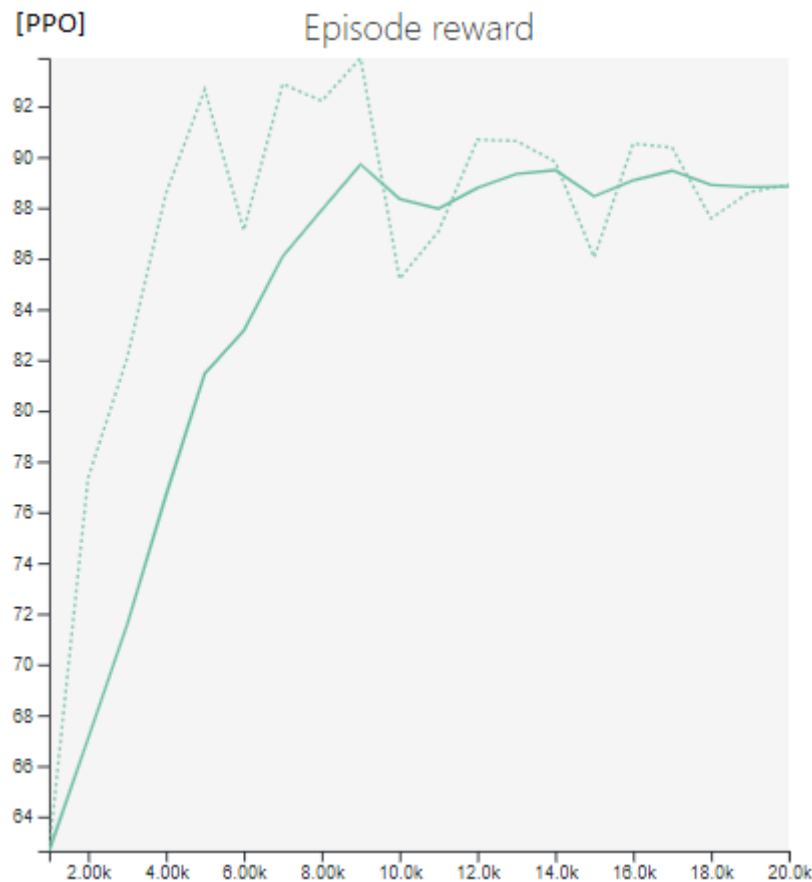


Figure 5.2.9.2: Training curve with the Proximal Policy Optimisation (PPO) algorithm

The values of episodic reward are higher than those reported in 5.2.7: this may be explained with the absence of the BLER in the simplified environment used for these tests. In fact, BLER modelling in the RLC MDP environment leads to fewer rewards and an earlier episode termination.

To leverage the scalability enabled with the DRL training, we can envision using the two-state transitioning scheme and adding more information such as the BLER status to enrich the agent's state.

This realisation of the Markov Decision Process (MDP) Radio Link Control model using TheRLib shows how this small footprint solution which is scalable to cope with large number of User Equipment (UE) (in our case 12) deployed in the network, can be used to successfully train a RL agent. However, work continues to complete linking TheRLib RL agent to the Matlab Network Simulation to show them collaboratively working together. It should reveal what is the optimum Modulation and Coding Scheme (MCS) for each of the UE in the network simulation, which is as yet unknown. This whole process provides the project partners with invaluable experience for the next stage of our learning process, namely: how to interconnect the RL agent with the actual UEs and RLC FlexRIC interfaces to the O-RAN.

6 Conclusions

Network Slicing has become a cornerstone technology in 5G and Beyond networks towards 6G in meeting the diverse requirements from various applications. In the context of the 6G BRAINS project, this work has successfully designed and prototyped advanced E2E Network Slicing over RAN and backhaul/backbone. The experimental results have validated that the Network Slicing system is fully functional and working in support of various application scenarios including those concerned in the project.

During the process of achieving the above, several tasks have been undertaken. Firstly, AI-based RAN Slicing has been achieved to optimise the usage of the radio resources. Secondly, hybrid software- and hardware-based backbone Network Slicing has been realised to achieve performance warranty at the wired segments as well and to enable the E2E Network Slicing, together with the RAN Slicing. Thirdly, LiFi Network Slicing and process slicing have been studied to further support industrial use cases as emphasised in the project. Fourthly, corresponding intent-based, enterprise-grade network management and orchestration has been implemented to effectively oversee and orchestrate the network slicing infrastructure. Fifthly, a new deep learning platform has been created to support the development of the AI algorithms related to RAN slicing and radio link control.

In summary, the work reported has successfully delivered the corresponding objectives in the scope of this work package.

7 References

- [1] Bosch in Singapore. Industrial 5G. <https://www.bosch.com.sg/news-and-stories/5g-in-action/>, 2021
- [2] Amazon Web Services. Deploying DISH's 5G Network in AWS Cloud. <https://aws.amazon.com/blogs/industries/telco-meets-aws-cloud-deploying-dishs-5g-network-in-aws-cloud/>, 2022.
- [3] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. *MIT press*, 2018.
- [4] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient Surgery for Multi-task Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [5] Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine. EPOpt: Learning Robust Neural Network Policies Using Model Ensembles. In *International Conference on Learning Representations (ICLR)*, 2017.
- [6] Jaden B. Travník, Kory W. Mathewson, Richard S. Sutton, and Patrick M. Pilarski. Reactive Reinforcement Learning in Asynchronous Environments. *Frontiers in Robotics and AI*, 2018 (5).
- [7] O-RAN Working Group 2. O-RAN AI/ML Workflow Description and Requirements 1.03. Technical report, 2022
- [8] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *nature* 518.7540 (2015): 529-533.
- [9] Hessel, Matteo, et al. "Rainbow: Combining improvements in deep reinforcement learning." *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. No. 1. 2018.
- [10] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. No. 1. 2016.
- [11] "Train Reinforcement Learning Agent in MDP Environment" Available at: Train Reinforcement Learning Agent in MDP Environment - MATLAB & Simulink - MathWorks United Kingdom, seen 7/9/2023
- [12] Sutton, R.S. and Barto, A.G., 2018. Reinforcement learning: An introduction. MIT press.
- [13] K. A. Prof. John Cosmas, "6GBRAINS-T5.4 Brunel Powerpoint report," 2023
- [14] Santos, E.C., 2017. A simple reinforcement learning mechanism for resource allocation in lte-a networks with markov decision process and q-learning. *arXiv preprint arXiv:1709.09312*.
- [15] ETSI TS 138 214 V16.2.0 (2020-07) "5G; NR; Physical layer procedures for data" (3GPP TS 38.214 version 16.2.0 Release 16)

- [16] Tang, L., Tan, Q.I., Shi, Y., Wang, C. and Chen, Q., 2018. Adaptive virtual resource allocation in 5G Network Slicing using constrained Markov decision process. *IEEE Access*, 6, pp.61184-61195.
- [17] Wilhelmi, F., Bellalta, B., Cano, C. and Jonsson, A., 2017, October. Implications of decentralized Q-learning resource allocation in wireless networks. In 2017 IEEE 28th annual international symposium on personal, indoor, and mobile radio communications (PIMRC) (pp. 1-5). IEEE.
- [18] Available at: <https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da> [Accessed 27 July 2023].
- [19] MATLAB, 2023. Train Reinforcement Learning Agent in MDP Environment. [Online] Available at: https://uk.mathworks.com/help/reinforcement-learning/ug/train-reinforcement-learning-agent-in-mdp-environment.html?s_tid=srchtitle_site_search_2_MDP [Accessed 26 July 2023].
- [20] Techplayon, 2020. 5G NR Modulation and Coding Scheme – Modulation and Code Rate. [Online] Available at: <https://www.techplayon.com/5g-nr-modulation-and-coding-scheme-modulation-and-code-rate/> [Accessed 27 July 2023].
- [21] Prince Kwaku Boakye “5G RAN Configuration and Control Using Reinforcement Learning” MSc Wireless Computer Communication Networks Dissertation Thesis, September 2023.
- [22] Navid Nikaein, Mahesh K. Marina, Saravana Manickam, Alex Dawson, Raymond Knopp, and Christian Bonnet. OpenAirInterface: A flexible platform for 5G research. *ACM SIGCOMM Computer Communication Review*. 2014 (5).
- [23] Robert Schmidt, Mikel Irazabal, and Navid Nikaein. FlexRIC: an SDK for next-generation SD-RANs. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2021.
- [24] Andrea Ramos, Yanet Estrada, Miguel Cantero, Jaime Romero, David Martín-Sacristán, Saúl Inca, Manuel Fuentes, and José Monserrat. Implementation and Calibration of the 3GPP Industrial Channel Model for ns-3. In *Workshop on NS-3 (WNS3)*. 2022.
- [25] P. Salva-Garcia, R. Ricart-Sanchez, E. Chirivella-Perez, Q. Wang, and J. M. Alcaraz-Calero, “Xdp-based smartnic hardware performance acceleration for next-generation networks,” *Journal of Network and Systems Management*, 2022.
- [26] Kernel, “Af xdp - af xdp socket (xsk),” https://www.kernel.org/doc/html/latest/networking/af_xdp.html, accessed: July 15, 2022.

- [27] R. Ricart-Sanchez, P. Salva-Garcia, Q. Wang, and J. M. Alcaraz-Calero, O. Herrera-Ruiz “An eBPF-XDP hardware-based Network Slicing architecture for 6G front- to back-haul networks” IEEE Transactions on Network and Service Management, 2023.
- [28] O-RAN Working Group 2: The Non-Real-Time RAN Intelligent Controller (Non-RT RIC) and A1 Interface Workgroup, 2021

[End of document]