Network Slice Mobility and Service Function Chain Migration across Multiple Administrative Cloud Domains

Rami Akrem Addad
Network Slice Mobility and Service Function Chain Migration across Multiple Administrative Cloud Domains

Rami Akrem Addad

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Electrical Engineering, at a public examination held at the lecture hall AS1 of the school on 26 January 2024 at 12.

Aalto University
School of Electrical Engineering
Department of Communication and Networking
Supervising professor
Prof. Jukka Manner, Aalto University, Finland

Thesis advisor
Prof. Diego Leonel Cadette Dutra, Federal University of Rio de Janeiro, Brazil

Preliminary examiners
Prof. Jussi Kangasharju, University of Helsinki, Finland
Prof. Fabrizio Granelli, University of Trento, Italy

Opponent
Prof. Erkki Harjula, University of Oulu, Finland.

Aalto University publication series
DOCTORAL THESES 13/2024

© 2024 Rami Akrem Addad

ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2024

Finland
**Abstract**

**Author**
Rami Akrem Addad

**Name of the doctoral thesis**
Network Slice Mobility and Service Function Chain Migration across Multiple Administrative Cloud Domains

**Publisher** School of Electrical Engineering

**Unit** Department of Communication and Networking

**Series** Aalto University publication series DOCTORAL THESES 13/2024

**Field of research** Networking Technology

**Manuscript submitted** 11 November 2022  
**Date of the defence** 26 January 2024

**Permission for public defence granted (date)** 6 April 2023  
**Language** English

- Monograph  
- Article thesis  
- Essay thesis

**Abstract**

The maturing 5G network technology sees growing commercial deployments, with a shifting focus to service delivery. 5G networks, a common platform for diverse services, utilize network slicing for service isolation. Cloud-native services, composed of inter-dependent micro-services, are allocated to network slices spanning multiple areas, domains, and data centers.

Due to mobility events caused by mobile end-users, slices with their assigned resources and services need to be re-scope and re-provisioned. This requires slice mobility, which involves a slice moving between service areas. Slice mobility requires the inter-dependent service and resources to be migrated to reduce system overhead and to ensure low-communication latency by following end-user mobility patterns. Recent advances in computational hardware, Artificial Intelligence, and Machine Learning have attracted interest within the communication community, with increased research interest in self-managed network slices. However, migrating a service instance of a slice remains an open and challenging process given the needed coordination between inter-cloud resources, the dynamics, and the constraints of inter-data center networks.

In this regard, this dissertation defines and enables smooth network slicing mobility patterns while maintaining both system and network resources stable. Specifically, we design, implement, and evaluate our proposed migration framework. Then, we design and define different network slice mobility patterns with their corresponding grouping methods and relevant mobility triggers. Next, we introduce various SFC migration strategies as an underlay technology enabler for network slice mobility patterns. After that, we propose an agent for automating the triggers selection process for enabling various network slice mobility patterns. Finally, we develop a network-aware agent capable of selecting accurate bandwidth values while ensuring fast and reliable service migration, thus enabling slice mobility while matching network and system requirements. In each section of this dissertation, the research results are evaluated and validated under different configurations in real-world settings or simulated environments.

This dissertation provides recommendations for improving and extending the notion of mobility in network slices while also highlighting the various outstanding questions and suggesting future challenges and research directions.

**Keywords** 5G, Network Slicing, Software Defined Networking, Network Function Virtualization, Service Function Chain, Multi-access Edge Computing, Network Softwarisation, Deep Reinforcement Learning


**ISSN (printed)** 1799-4934  
**ISSN (pdf)** 1799-4942

**Location of publisher** Helsinki  
**Location of printing** Helsinki  
**Year** 2024

**Pages** 235  
Preface

The research work for this doctoral dissertation has been carried out in the Department of Communications and Networking (ComNet), School of Electrical Engineering, Aalto University, Espoo, Finland, in collaboration with Nokia Bell Labs.

It is with great pleasure that I can show my gratitude and express my appreciation to all those who have contributed to the realization of this dissertation. First of all, praise and gratitude be to God Almighty for the graces and blessings that he bestowed on me so that I could complete my dissertation. I would like to express my sincere gratitude to my supervisors, Prof. Tarik Taleb, Prof. Jukka Manner, and Prof. Raimo Kantola, for allowing me to pursue a doctoral degree under their supervision. I am very grateful for the mentoring provided by them, and that they were always available and willing to offer help, guidance, and insights.

I would like to extend my gratitude to my thesis advisor, colleague, and friend, Prof. Diego Leonel Cadette Dutra. His positivity, flexibility, patience, understanding, technical expertise, and leadership have been an inspiration to me. Big thanks go to M.Sc. Hannu Flinck for allowing me to work on various collaborative research projects between Aalto University and Nokia Bell-Labs. His guidance, helpful comments, innovative ideas during discussions, and technical knowledge have been precious. I would also like to thank Dr. Miloud Bagaa for his help and initial support. A special thank you goes to pre-examiners, Prof. Jussi Kangasharju and Prof. Fabrizio Granelli, and opponent Prof. Erkki Harjula for their valuable remarks, recommendations, and constructive comments.

I want to express my appreciation to my colleagues at MOSA!C Lab and all the personnel at the ComNet Department, especially Aalto IT Service Desk members, for their support, friendship, and the exciting working environment. I also want to thank all the co-authors of the papers that
form the basis of this thesis, and also those who have contributed to the publications through pre-reviewing processes but are not included in the publications.

I extend my sincere gratitude to all my teachers and professors for their pivotal role in my academic journey, from elementary school to my thesis defense. A special thanks to my friend and mentor, Dr. Khaled Zeraoulia, whose guidance has been invaluable. I also appreciate the unwavering support of Dr. Nassima Toumi and M.Sc. Mehdi Rezzoug, whose structural and linguistic assistance significantly contributed to the quality of my work. Thank you all for being instrumental in my academic success.

I would like to thank my family members and friends for their love and unconditional support. My deepest gratitude to my mother, Fatima, grandmother, Kheira, aunt, Faiza, and uncle, Mohamed, for trusting me on my lifelong journey and giving me the most important things family ever can – roots, honesty, and dignity.

Finally, I want to dedicate this thesis to the soul of my aunt, Aicha. This achievement would have been impossible without you taking care of me when I was young. May the Almighty God welcome you in his vast paradise.

Helsinki, December 27, 2023
Rami Akrem Addad
# Table of Contents

Preface 5

Table of Contents 7

List of Publications 10

Author's Contributions 12

List of Abbreviations 15

List of Symbols 19

List of Figures 22

1. Introduction 24
   1.1 Motivation ................................. 24
   1.2 Scope and Research Problem Area .............. 26
   1.3 Research Methodology .......................... 28
   1.4 Contributions to Knowledge .................. 29
   1.5 Structure of the Dissertation ............... 30

2. Enablers and Theoretical Frameworks for Network Slice Mobility 32
   2.1 Enabling Technologies and Paradigms ........ 32
      2.1.1 Virtualization Approaches .................. 32
      2.1.2 Virtual Machines vs Containers ............ 33
      2.1.3 Migration .................................. 34
      2.1.4 Downtime .................................. 35
      2.1.5 Total Migration Time ....................... 35
      2.1.6 Live Iterative Migration ................... 35
      2.1.7 Checkpoint and Restore Mechanisms ...... 36
CHAPTER 0. TABLE OF CONTENTS

2.1.8 Software Defined Networking .................................. 36
2.1.9 Service Function Chains .......................................... 38
2.1.10 Live Service Function Chain Migration ....................... 39
2.1.11 Network Slicing .................................................. 39

2.2 A Primer on Reinforcement Learning ............................... 40
   2.2.1 Reinforcement Learning Background ............................ 40
   2.2.2 Reinforcement Learning Classes and Types .................... 40

2.3 Summary .............................................................. 42

3. A Framework for Enabling Network Slice Mobility ................. 43
   3.1 Towards Fast Service Migration in 5G and Beyond Networks ... 43
      3.1.1 Main Architecture and Problem Formulation .................. 44
      3.1.2 Stateful Service Migration Based on a Predefined Mobility Pattern ............................................. 47
      3.1.3 Stateful Service Migration Based on Undefined Mobility Pattern .............................................. 49
      3.1.4 Experimental Evaluation ........................................ 49
   3.2 Benchmarking the ONOS Intent Interfaces to Support Network Slice Mobility Patterns ............................... 55
      3.2.1 Methodology .................................................... 56
      3.2.2 Experimental Evaluation ........................................ 57
   3.3 MIRA!: An SDN-based Framework for Network Slicing Mobility and Ultra-Low Latency 5G Services .................. 61
      3.3.1 Framework Design .............................................. 62
      3.3.2 Experimental Evaluation ........................................ 65
   3.4 Summary .............................................................. 67

4. AI-based Network Slice Mobility Patterns in Next Generation Mobile Systems .............................................. 69
   4.1 Network Slice Mobility Patterns and Enabling Triggers ....... 69
      4.1.1 Network Slice Mobility Use Cases ............................. 70
      4.1.2 Key Enabling Network Slice Mobility Triggers .............. 72
      4.1.3 Grouping Attributes ............................................ 74
      4.1.4 Network Slice Mobility Patterns ................................ 76
      4.1.5 Evaluation Setup & Experimental Results .................... 80
   4.2 AI-based Triggers selection for Network Slice Mobility ....... 83
      4.2.1 Envisioned Architecture ........................................ 84
      4.2.2 Environment Description and Type of Collected Data ........ 86
      4.2.3 System Model .................................................. 88
CHAPTER 0. TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4 Design of the Slice Mobility Decision Maker Agent</td>
<td>91</td>
</tr>
<tr>
<td>4.2.5 Experimental Evaluation</td>
<td>96</td>
</tr>
<tr>
<td>4.3 Summary</td>
<td>103</td>
</tr>
<tr>
<td>5. AI-assisted Service Function Chain Migrations for Network</td>
<td>104</td>
</tr>
<tr>
<td>5.1 Service Function Chain Migration Strategies in 5G and Beyond Networks</td>
<td>104</td>
</tr>
<tr>
<td>5.1.1 Service Function Chain Migration Strategies</td>
<td>105</td>
</tr>
<tr>
<td>5.1.2 Implementation Details and Key Technological Prerequisites</td>
<td>107</td>
</tr>
<tr>
<td>5.1.3 Experimental Evaluation</td>
<td>108</td>
</tr>
<tr>
<td>5.2 AI-based Network-aware Service Function Chain Migration in 5G and Beyond Networks</td>
<td>114</td>
</tr>
<tr>
<td>5.2.1 Proposed Architecture</td>
<td>115</td>
</tr>
<tr>
<td>5.2.2 System Model</td>
<td>118</td>
</tr>
<tr>
<td>5.2.3 Design of the DRL Algorithms Trainer</td>
<td>121</td>
</tr>
<tr>
<td>5.2.4 Experimental Evaluation</td>
<td>128</td>
</tr>
<tr>
<td>5.3 Summary</td>
<td>134</td>
</tr>
<tr>
<td>6. Conclusion and Future Research</td>
<td>135</td>
</tr>
<tr>
<td>6.1 Conclusion</td>
<td>135</td>
</tr>
<tr>
<td>6.2 Future research</td>
<td>137</td>
</tr>
<tr>
<td>References</td>
<td>139</td>
</tr>
<tr>
<td>Publications</td>
<td>149</td>
</tr>
</tbody>
</table>
List of Publications

This dissertation consists of an overview of the following publications, referred to in the text by their roman numerals, ordered according to their relevance to the context of the author’s doctoral studies and the dissertation timeline.


VI R. A. Addad, D.L.C. Dutra, M. Bagaa, T. Taleb, and H. Flinck, “Towards studying Service Function Chain Migration Patterns in 5G
Networks and beyond,” In 2019 IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, Dec 2019.


Author’s Contributions

Publication I: “Towards A Fast Service Migration in 5G”

The author of this dissertation was the main author of the paper. He wrote the paper, designed the evaluation testbed, and analyzed the experimental results. Prof. Dutra helped in defining the evaluation testbed and interpreting the obtained results. Dr. Bagaa and M.Sc. Flinck assisted in the writing process and contributed to the architecture definition. Prof. Taleb supervised the work, validated the proposed architecture, and enhanced the quality of the manuscript through revisions, suggestions, and modifications.

Publication II: “Fast Service Migration in 5G Trends and Scenarios”

The author of this thesis had the responsibility of formulating the idea, deriving the results, and writing the manuscript. M.Sc. Flinck assisted in the writing process and refined the work through his guidance and proposals. Prof. Dutra assisted in designing the experiments and analyzing the results discussed in the article. Prof. Taleb supervised the work, improved the paper readability through his remarks and observations from both technical and linguistic perspectives, and refined the initially proposed architecture.

Publication III: “Benchmarking the ONOS Intent interfaces to ease 5G service management”

The idea for the article was proposed by Prof. Dutra. M.Sc. Mehdi Namane carried out the initial version, first tests, implementations, and val-
The author of the thesis was responsible for redesigning the evaluation, collecting data, refining the writing, and repeating the analysis. Prof. Taleb supervised the research and provided valuable comments on the document.

**Publication IV: “MIRA!: A SDN-based Framework for Cross-Domain Fast Migration of Ultra-Low Latency 5G Services”**

Prof. Taleb conceived the idea of MIRA! framework. The author implemented a proof of concept, tested various features, and conducted the experiments which led to the writing of the article. Prof. Dutra supervised the evaluation and assessment part. The author, together with Dr. Bagaa and M.Sc. Flinck, revised the paper after acceptance.

**Publication V: “Network Slice Mobility in Next Generation Mobile Systems: Challenges and Potential Solutions”**

Prof. Taleb proposed the idea of the network slice mobility patterns. The author implemented the test-bed, gathered, analyzed, and interpreted the results. M.Sc. Flinck assisted the author in defining the structure of the manuscript, helped in the writing process, and refined the technical content of the proposed patterns. Prof. Dutra guided the analysis and discussion of the results. The final manuscript benefited from the valuable comments and suggestions of Prof. Taleb.

**Publication VI: “Towards studying Service Function Chain Migration Patterns in 5G Networks and beyond”**

The candidate was the leading author, responsible for designing and implementing SFC migration patterns as well as writing the paper. The test-bed developments, measurements, and performance evaluation were conducted by the author with the assistance of Prof. Dutra. Prof. Taleb and M.Sc. Flinck supervised the research and provided valuable guidance on the document.
CHAPTER 0. AUTHOR’S CONTRIBUTIONS

Publication VII: “Toward Using Reinforcement Learning for Trigger Selection in Network Slice Mobility”

The author of this dissertation was the primary author of this paper and was responsible for the main idea, system modeling, simulation development, and configuration as well as results analysis. Prof. Taleb supervised the work. He also contributed to enhancing the quality of the manuscript by providing valuable suggestions, refining the envisioned architecture, and making revisions. Prof. Dutra and M.Sc. Flinck assisted in the writing process by making revisions, improving the quality of the paper, and enhancing the technical details related to the envisioned architecture.

Publication VIII: “AI-Based Network-Aware Service Function Chain Migration in 5G and Beyond Networks”

The author of this dissertation was the primary author of this article. He modeled, designed, and implemented agents for bandwidth allocation in SFC service migration. The author also evaluated, analyzed, and validated the results. Prof. Taleb, Prof. Dutra, and M.Sc. Flinck assisted in the writing process by making revisions, enhancing the quality of the paper, and suggesting new approaches. Besides, Prof. Taleb supervised the work during the research journey and contributed to refining the design of the system architecture.
List of Abbreviations

3GPP  Third Generation Partnership Project
5G    Fifth Generation
A2C   Advantage Actor-Critic
A3C   Asynchronous Advantage Actor-Critic
AI    Artificial Intelligence
AMF   Access Management Function
API   Application Programming Interface
AR    Action Refinement
BAE   Bandwidth Allocator and Exploitation
BSS   Business Support System
C-SMDM Core-Slice Mobility Decision Maker
CAPEX CApital EXpenditure
CDN   Content Delivery Network
CI    Confidence Interval
CLI   Command Line Interface
CRIU  Checkpoint/Restore In Userspace
CV    Coefficient of Variation
DAC   Deep Reinforcement Learning Algorithms Comparator
DAT   Deep Reinforcement Learning Algorithms Trainer
DDPG  Deep Deterministic Policy Gradient
DL    Deep Learning
DN    Data Network
DNN   Deep Neural Network
DM    Discretization Module
DoS   Denial of Service
DQN   Deep Q-Network
DRL   Deep Reinforcement Learning
E2E   End-to-End
EM    Element Management
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>eMBB</td>
<td>enhanced Mobile Broadband</td>
</tr>
<tr>
<td>EPS</td>
<td>Evolved Packet System</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>FRD</td>
<td>Fog Reference Design</td>
</tr>
<tr>
<td>GMT</td>
<td>Group Mobility Trigger</td>
</tr>
<tr>
<td>GPSI</td>
<td>Generic Public Subscription Identifier</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>GRE</td>
<td>Generic Routing Encapsulation</td>
</tr>
<tr>
<td>HN</td>
<td>Host Node</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>IDS</td>
<td>Intrusion Detection System</td>
</tr>
<tr>
<td>IPS</td>
<td>Intrusion Prevention System</td>
</tr>
<tr>
<td>ITU-T</td>
<td>International Telecommunication Union — Telecommunication</td>
</tr>
<tr>
<td>KSN-A</td>
<td>Kernel-Smart Network-Aware</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel Virtual Machine</td>
</tr>
<tr>
<td>LE</td>
<td>Learning and Exploration</td>
</tr>
<tr>
<td>LTS</td>
<td>Long-Term Support</td>
</tr>
<tr>
<td>LXC</td>
<td>Linux Container</td>
</tr>
<tr>
<td>MC</td>
<td>Metric Collector</td>
</tr>
<tr>
<td>MEAO</td>
<td>Mobile Edge Application Orchestrator</td>
</tr>
<tr>
<td>MEC</td>
<td>Multi-Access Edge Computing</td>
</tr>
<tr>
<td>MEP</td>
<td>Mobile Edge Platform</td>
</tr>
<tr>
<td>MEPM</td>
<td>Mobile Edge Platform Management</td>
</tr>
<tr>
<td>MGO</td>
<td>MIRA! Global Orchestrator</td>
</tr>
<tr>
<td>MlIoT</td>
<td>massive Internet of Things</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MME</td>
<td>Mobility Management Entity</td>
</tr>
<tr>
<td>mMTC</td>
<td>massive Machine-Type Communications</td>
</tr>
<tr>
<td>NETCONF</td>
<td>Network Configuration Protocol</td>
</tr>
<tr>
<td>NF</td>
<td>Network Function</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Function Virtualization</td>
</tr>
<tr>
<td>NFVI</td>
<td>Network Function Virtualization Infrastructure</td>
</tr>
<tr>
<td>NFVO</td>
<td>Network Function Virtualization Orchestrator</td>
</tr>
<tr>
<td>NGMN</td>
<td>Next Generation Mobile Network Alliance</td>
</tr>
<tr>
<td>NS</td>
<td>Network Slicing</td>
</tr>
<tr>
<td>NSM</td>
<td>Network Slice Mobility</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NSSAI</td>
<td>Network Slice Selection Assistance Information</td>
</tr>
<tr>
<td>OAM</td>
<td>Operations, Administration and Maintenance</td>
</tr>
<tr>
<td>ONF</td>
<td>Open Network Foundation</td>
</tr>
<tr>
<td>ONOS</td>
<td>Open Network Operating System</td>
</tr>
<tr>
<td>OPEX</td>
<td>OPerational EXpenditure</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSS</td>
<td>Operation Support System</td>
</tr>
<tr>
<td>OVS</td>
<td>Open vSwitch</td>
</tr>
<tr>
<td>PCEP</td>
<td>Path Computation Element Configuration Protocol</td>
</tr>
<tr>
<td>PG</td>
<td>Policy Gradient</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAN</td>
<td>Radio Access Network</td>
</tr>
<tr>
<td>RAT</td>
<td>Resource Availability Trigger</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit</td>
</tr>
<tr>
<td>RESTful</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RH</td>
<td>Request Handler</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>RM</td>
<td>Request Manager</td>
</tr>
<tr>
<td>ROT</td>
<td>Request Overload Trigger</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Question</td>
</tr>
<tr>
<td>rsync</td>
<td>remote synchronization</td>
</tr>
<tr>
<td>RT</td>
<td>Reliability Trigger</td>
</tr>
<tr>
<td>SBA</td>
<td>Service Based Architecture</td>
</tr>
<tr>
<td>SCT</td>
<td>Service Consumption Trigger</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SDO</td>
<td>Standards Development Organization</td>
</tr>
<tr>
<td>SFC</td>
<td>Service Function Chain</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLT</td>
<td>Service Load Trigger</td>
</tr>
<tr>
<td>SMDM</td>
<td>Slice Mobility Decision Maker</td>
</tr>
<tr>
<td>SN-A</td>
<td>Smart Network-Aware</td>
</tr>
<tr>
<td>ST</td>
<td>Security Trigger</td>
</tr>
<tr>
<td>STD</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>SUPI</td>
<td>Subscription Permanent Identifier</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TD</td>
<td>Temporal Difference</td>
</tr>
<tr>
<td>TE</td>
<td>Training and Exploration</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>tmpfs</td>
<td>temporary file system</td>
</tr>
<tr>
<td>TPU</td>
<td>Tensor Processing Unit</td>
</tr>
<tr>
<td>TSE</td>
<td>Trigger Selector and Exploitation</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>UE</td>
<td>User Equipment</td>
</tr>
<tr>
<td>URLLC</td>
<td>Ultra-Reliable and Low-Latency Communication</td>
</tr>
<tr>
<td>VIM</td>
<td>Virtualized Infrastructure Manager</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VNF</td>
<td>Virtualized Network Function</td>
</tr>
<tr>
<td>VNFM</td>
<td>Virtualized Network Function Manager</td>
</tr>
<tr>
<td>VXLAN</td>
<td>Virtual eXtensible Local Area Network</td>
</tr>
</tbody>
</table>
List of Symbols

$\mathcal{A}$ The set of actions available for a given agent

$a_t$ Agent’s action at time $t$

$bw_c$ The total available bandwidth

$bw_i$ The current selected bandwidth value

$bw_{ddpg}(X, z)$ A function used to centralize the output around $X$

$\mathcal{C}$ The set of Containers/MEC apps

$c$ A given Container/MEC app $c \in \mathcal{C}$

$d_s$ The dump size input feature

$f_c$ The number of MEC apps $c \in \mathcal{C}$ features

$\mathcal{F}_n$ The number of selected features depending on MEC hosts and MEC apps

$f_n$ The number of MEC $n \in N$ features

$g(n, j)$ Function that returns the percentage of a given resource $j \in \mathcal{K}$ in $n \in N$

$G_t$ The expected discounted reward over the discrete time steps
\( \mathcal{K} \) The set of resources, e.g., CPU, RAM, DISK

\( \mathcal{N} \) Ornstein-Uhlenbeck noise used for exploration

\( N \) The set of MEC hosts

\( n \) A given MEC host \( n \in N \)

\( p_r \) The number of memory pages input feature

\( Q(s_t, a_t) \) Quality of the action in state \( s_t \) given the action \( a_t \)

\( \mathcal{R} \) The reward function

\( r_{t+1} \) Immediate reward at time \( t + 1 \)

\( \mathcal{R}_c \) Scalar value, to denote the types of resources used for each authorized operation returned by the function \( \Phi(c) \)

\( ReLU(z) \) An activation function used for training DNNs

\( S \) The set of states in a given environment

\( s_t \) State at time \( t \)

\( \mathcal{T} \) Migration downtime

\( \mathcal{T}_\eta \) The time of operation \( \eta \), i.e., \( \eta \) can be migration or scale up/down

\( \tanh(z) \) Hyperbolic tangent is an activation function used for training DNNs

\( \mathcal{X}_{c,n} \) Variable to introduce Container (MEC app) to MEC mapping
\( \alpha \)  
The learning rate

\( \beta \)  
Coefficient used to increase the influence of the resource usage parameter

\( \gamma \)  
The discount rate

\( \Delta \omega \)  
Changes in weights of the approximator (e.g., DNN)

\( \delta \)  
Coefficient used to increase the influence of the operation time parameter

\( \epsilon\)-Greedy  
Exploration/Exploitation trade-off policy

\( \theta \)  
Coefficient used to increase the influence of time parameter

\( \vartheta \)  
Coefficient used to increase the influence of the bandwidth value parameter

\( \pi(s_t, a_t) \)  
The policy that optimally maps states to actions

\( Q^\pi(s_t, a_t, \omega) \)  
Predicted quality of the action in state \( s_t \) given the action \( a_t \) using weights parameters \( \omega \)

\( \nabla Q^\pi(s_t, a_t, \omega) \)  
The gradient of the predicted Q value

\( \sigma(a_i) \)  
An activation function that outputs a multiclass probability distribution over target classes

\( \tau \)  
Coefficient used for soft updates of weights in the target networks (Actor, Critic)

\( \Phi(c) \)  
Function that returns the number of authorized operations for a Container/MEC app \( c \in C \) except for migration actions
List of Figures

Figure 1.1 Correspondence between the Research Questions and the publications.

Figure 2.1 Architectures of virtualization technologies: hypervisor-based vs container-based.

Figure 2.2 ONOS Intent framework mechanism.

Figure 3.1 Three layers cloud-based architecture.

Figure 3.2 Detailed live migration flowchart.

Figure 3.3 Proposed prototypes for both known and unknown path solutions.

Figure 3.4 Downtime in case of different migration approaches.

Figure 3.5 Total migration time experienced in case of different migration approaches.

Figure 3.6 Multi-to-Single-Point Intent - benchmarking results.

Figure 3.7 Single-to-Multi-Point Intent - benchmarking results.

Figure 3.8 Point-to-Point Intent - benchmarking results.

Figure 3.9 MIRA! framework architecture.

Figure 3.10 Total migration time comparison for Basic Solution and MIRA!.

Figure 3.11 Impact of CPU variations on total migration time for parallel migrations.

Figure 4.1 Full Network Slice Mobility pattern.

Figure 4.2 Partial Network Slice Mobility pattern (Slice Breathing).

Figure 4.3 Partial Network Slice Mobility pattern (Slice Splitting).

Figure 4.4 Partial Network Slice Mobility pattern (Slice Merging).

Figure 4.5 Network Slice Mobility Optimizer (Slice Shrinking).

Figure 4.6 Network Slice Mobility testbed setup.

Figure 4.7 Subsequent migrations vs parallel migrations in Network Slice Mobility.

Figure 4.8 Framework architecture for smart triggers selection in Network Slice Mobility.
Figure 4.9  States space representation and formation using the proposed modeling.
Figure 4.10  Rewards history for the SMDM agent using the DQN algorithm.
Figure 4.11  Rewards history for the SMDM agent using the A2C algorithm.
Figure 4.12  Rewards comparison in case of different RL algorithms.
Figure 4.13  Accuracies comparison in case of different RL algorithms.
Figure 4.14  Learning rate variations comparison for the SMDM-A2C agent.
Figure 4.15  Training Loss variations comparison for the SMDM-A2C agent.
Figure 5.1  Service Function Chain implementation.
Figure 5.2  Downtime comparison for the asynchronous SFC migration strategy under diverse network configurations.
Figure 5.3  Total migration time experienced for the asynchronous SFC migration strategy under diverse network configurations.
Figure 5.4  CPU consumption analysis in case of different SFC migration strategies.
Figure 5.5  Framework architecture for ML-based orchestration and allocation of bandwidth resources.
Figure 5.6  Design and principles of DRL Algorithms Selector (DAS) module.
Figure 5.7  SN-A agent training testbed setup.
Figure 5.8  DQN-based training and comparison.
Figure 5.9  DDPG-based training and comparison.
Figure 5.10  Comparing bandwidth action selection for DQN against DDPG and handcrafted values.
Figure 5.11  Downtime comparison in case of different DRL algorithms.
1. Introduction

This chapter delineates the literature and motivation, research scope, and specific research challenges presented in this dissertation. Then, the various research methodologies followed during the research journey are explained, and initial insight into our main contributions is provided. Finally, the dissertation structure is given, with the relationship between Research Questions (RQs), publications, and dissertation chapters indicated.

1.1 Motivation

The Fifth Generation (5G) of mobile networks will go beyond providing only high data rates for mobile users, as it will be a platform for a wider communication ecosystem and applications [1]. 5G network proposals offer services within three major axes: enhanced Mobile Broadband (eMBB), Ultra-Reliable and Low-Latency Communications (URLLC), and massive Machine-Type Communications (mMTC) [2]. The services of new vertical industries, e.g., automotive, e-health, public safety, and smart grids, impose unique requirements that will push the envelope for high performance, scalability, and availability. 5G has been re-architected from the ground up to achieve such ambitious goals compared to the previous mobile network generations. 5G networks adopted the principles of network softwarization and programmability through the use of Network Function Virtualization (NFV) and Software-Defined Networking (SDN) paradigms to cope with these stringent requirements, thus logically separating Network Functions (NFs) from the physical infrastructure [3]. This decoupling is the basis for deploying self-contained, programmable, and customizable networks [4].
The sharing of the same underlying infrastructure among isolated and self-contained networks and the highly dynamic and varying characteristics of the newly introduced services lead to several new mechanisms beyond network virtualization. Mechanisms such as Network Slicing (NS), new Quality of Service (QoS) models [5], and edge services offered by Multi-Access Edge Computing (MEC) [6–8] are prime examples. In the scope of this dissertation, the term slice refers to a subset of elements and their relationships. Moreover, an element may exist in multiple slices simultaneously. NS is a set of reserved resources (network, system, and storage) used to serve end-users while guaranteeing isolation, dynamic resource allocations, and user satisfaction. NS offers numerous benefits, particularly the ability to deliver highly customizable services to new industry sectors that have been unserved or inadequately served by earlier mobile technologies [9–13]. Furthermore, NS deployments in MEC environments provide effective service delivery with minimal delay for latency-sensitive services with different characteristics.

Among new industry use cases targeted by the 5G mobile systems, some scenarios exceed what the current device-centric mobility approaches can support [14, 15]. The mobility of low latency communication services, shared by a group of moving devices, e.g., autonomous vehicles that share sensor data, is a prime example of these cases [16]. Other use cases would be related to swarms of Unmanned Aerial Vehicles (UAVs) performing sensitive civilian or military surveillance operations and high-speed trains supporting rapidly-changing content and entertainment needs [17]. When these groups move from one area to another, and their ongoing mobile communication and service sessions run on at least one network slice, they may suffer a drop in QoS, if not a total disconnect. This may happen if their ongoing network service has to be served by instantiating a totally new network slice of the same type at the destination network, i.e., new radio resources, backhaul, computing, and storage. This leads to a network slice that has to support moving its computation, following its users’ mobility patterns. Additionally, the slice may need to adjust its resource allocation, adding more resources or releasing unused ones.

For that reason, there is a rich literature proposing algorithms and approaches tailoring mobility constraints for newly introduced paradigms like NS in 5G networks. For instance, the authors of [18] focused on extensions for NFV orchestration that provide tailored support for mobility and QoS/Quality of Experience (QoE) for network slices while ensuring ef-
sufficient utilization of the substrate network resources. This approach pre-
sented several ways to classify and select mobility management schemes 
based on the context of the service or type of a network slice. The au-
thors proposed slice-specific schemes by creating context-dependent con-
figurations of the instantiated NFs. Likewise, in [19], a new procedure 
to dynamically select the best Virtual Machines (VMs) susceptible to re-
duce energy consumption within a cloud data-center domain when shifted 
away, i.e., migrated, is proposed. The authors exploited over-utilized hosts 
for choosing VMs to reduce the number of migrations while preserving 
fair energy usage. Considering CPU utilization is proportional to power 
usage, the authors proposed a modeling based on the variations of CPUs 
across different hosts.

In this vein, these previous related works and the above requirements 
have motivated the work and led to defining the Network Slice Mobility 
(NSM) paradigm, its patterns, and its different enabling triggers as new 
concepts for next-generation networks.

1.2 Scope and Research Problem Area

NSM allows the motion of end-users to be followed with its different 
patterns, therefore enabling low communication latency, fast service de-
ivery, and increasing users’ QoE and QoS. However, migrating the service 
instances of a network slice, i.e., chained virtualized instances or Service 
Function Chain (SFC) migration, remains challenging and problematic in inter-cloud settings. A prime example is the considerable amount of 
network resource usage required to reshuffle virtualized instances con-
stituting the network slice. Therefore, there is an urgent need to support 
NSM through various SFC migration strategies and bandwidth allocation 
mechanisms, reducing the network overhead while ensuring the satisfac-
tion of the users.

In this dissertation, the broad aim is to support the mobility of network 
slices through employing SDN and NFV paradigms [20, 21] and explore 
the design principles of 5G and beyond systems. The main objective of 
the research is to enable a smooth NSM while optimizing the availability 
and usage of network resources. Our research area includes SFC migra-
tion strategies as underlying technologies and solutions supporting NSM 
and its associated patterns. Our investigation is focused on implementing
lower-level NSM components, i.e., chained virtualized instances or SFC, to ensure a seamless mobility procedure across MEC nodes leveraging Artificial Intelligence (AI) assistance. A set of AI techniques and methods are designed, implemented, and evaluated to select proper system-based triggers and bandwidth values, thus enabling various NSM patterns while guaranteeing the satisfaction of both network and system resources. Case-specific aspects, such as the signal strength and control plane entities in NSs, are not included in the scope of the thesis.

Taking into consideration the challenges mentioned above and the requirements related to NSM, its associated patterns and triggers, and the availability of network resources, the dissertation attempts to find acceptable answers to the following major RQ, i.e., RQ1: Can the current network slice paradigm be expanded to support different patterns of end-users’ mobility and network dynamics while maintaining network and system resources below the thresholds of the network operators?

The challenge to answer RQ1 leads us to seven distinct RQs, i.e., RQ2 to RQ8, each of which is more or less linked to another question.

1. Which enabling technologies are suitable for NSM? (RQ2)
2. What are the different patterns to enable NSM? (RQ3)
3. How should users be grouped to allow efficient resource allocation
CHAPTER 1. INTRODUCTION

and different user mobility patterns? (RQ4)

4. What are the triggers susceptible to assist NSM patterns? (RQ5)

5. Which SFC migration strategy is suitable for supporting the mobility of 5G network slices? (RQ6)

6. How can we automate the trigger selection process to optimize the system resources of the slices? (RQ7)

7. How can we reduce network overhead while performing different NSM patterns? (RQ8)

RQ2 and RQ3 represent underlying requirements for enabling NSM. Addressing RQ2 and RQ3 helps answer RQ4, RQ5, and RQ6. A favorable answer to RQ4 and RQ5 helps to tackle RQ7, while RQ8 is addressed via RQ6. Figure 1.1 provides a descriptive tree showing the correspondence between RQs and the publications.

1.3 Research Methodology

The research methodology followed to tackle the various research objectives in the collection of the publications forming the dissertation is oriented toward architecture and system design, quantitative analysis, and empirical experimentation. Publications I, II, and III are based on investigative research carried out to test and evaluate diverse migration approaches and northbound interfaces in the SDN paradigm, and also to select and adopt the right virtualization method. Although Publication II was the sixth scientific production in the chronological order, it is considered second within the context of the author’s doctoral studies and the dissertation timeline.

Publications from IV to VIII presented the design, implementation, and evaluation of several frameworks and prototypes. The results were validated by comparison with similar studies in the literature, including studies based on benchmarking, software and hardware performance, derived measurements, downtime, total migration time, and extensive simulations. In all publications, the system architecture designs mainly follow the standardizations of the European Telecommunications Standards Institute (ETSI) and the Third Generation Partnership Project (3GPP). Be-
sides, all publications end with the outline of contributions and achievements, open issues analysis, and key future research directions.

Finally, the proposed approaches and frameworks are fully developed in the MOSAIC Lab research group [22] and are publicly available as open-source projects on Github [23, 24] for the benefit of the research community. Table 1.1 summarizes the methodological approaches employed in the publications constituting the dissertation.

Table 1.1. Dissertation methodology through publications.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>PI</th>
<th>PII</th>
<th>PIII</th>
<th>PIV</th>
<th>PV</th>
<th>PVI</th>
<th>PVII</th>
<th>PVIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparative studies</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Extensive simulations</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System architecture</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Experimental evaluation</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Prototyping</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.4 Contributions to Knowledge

The research presented in this dissertation is centered on advancing the current state of knowledge about autonomic SFC management and control for NSM in 5G networks. These results and findings of the dissertation serve the existing literature by offering the following specific advancements in this field.

- The design and the evaluation of three different mechanisms to improve the end-user experience and reduce applications’ downtime leveraging container-based live migration paradigms.

- Four migration approaches based on the container technology to optimize disk migration and reduce total migration time based on user mobility patterns.

- The experimental evaluation of the Open Network Operating System (ONOS) Intent northbound interface to indicate the best practices in the development of complex behaviors and ONOS-aware distributed applications to assist NSM.

- “MIRA!”, a framework for managing reliable live migrations of virtual resources across different data centers belonging to multiple administrative domains;
• The introduction of multiple NSM patterns to optimally manage and use network slices.

• The definition of several key slice mobility triggers and User Equipment (UE) grouping methods to enable efficient NSM patterns.

• The design, proposition, and implementation of four SFC migration strategies for allowing the support of synchronized depending applications while maintaining the joint ETSI and 3GPP proposed architectures.

• The proposal, design, and implementation of a Deep Reinforcement Learning (DRL)-based agent for allowing a fine-grained selection of system-based triggers regarding the NSM patterns.

• The design and development of a network-aware agent based on two different DRL algorithms. This agent is capable of selecting accurate bandwidth values and adjusting the network usage while ensuring fast and reliable service migration.

Overall, this dissertation provides recommendations toward improving and extending the notion of mobility in network slices while maintaining the stability of both system and network resources. It also helps the research community monitor, further study, and make improvements in similar research fields and domain applications.

1.5 Structure of the Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 summarizes the necessary background information for the entire dissertation. This chapter defines the fundamental paradigms and concepts related to the research conducted in this dissertation. It also introduces and describes the various non-trivial techniques, methods, and technologies used to finalize the thesis. Publications I, II, III, VII, and VIII form the core content highlighted in this chapter.

Chapter 3 presents the design, implementation, and evaluation of different migration techniques and principles based on the container technology to both optimize downtime and total migration time with known and unknown users’ patterns taken into consideration. Besides, it presents a
benchmarking of the ONOS SDN controller due to the necessity of employing the SDN paradigm for enabling an inter-cloud migration. Finally, it describes the design and the implementation of the framework, i.e., MIRA!, that combines the results obtained from the previous works and manages reliable live migrations of virtual resources across different Infrastructure as a Service (IaaS). To validate the implementation, we perform a set of experimental evaluations under different configurations. This chapter is entirely based on the insights featured in Publications I, II, III, and IV.

In Chapter 4, we design and define different NSM patterns with their corresponding grouping methods and relevant mobility triggers. We also design and model two DRL-based algorithms to make the defined triggers intelligent and allow fine-grained selection of system-based triggers regarding the NSM patterns while conserving network slice characteristics. Finally, we present an extensive array of simulations and the results of assessments to validate the proposed methods and patterns. This chapter is based on the contributions in Publications V and VII.

In Chapter 5, four SFC migration strategies are proposed that consider both synchronizing the Virtualized Network Function (VNF) chain’s instances and the network resources consumption for allowing NSM. However, SFC migration remains challenging given the considerable amount of cross-cloud network resource usage and the constraints imposed by bandwidth limitations. For this purpose, we propose a DRL agent, based on two different algorithms, which selects the appropriate bandwidth values, in order to reduce network resource consumption. Finally, we conduct a set of experiments under different configurations and real-world deployments to determine the optimized SFC migration pattern within the 5G network. The content of this chapter relies on Publications VI and VIII.

Finally, the dissertation is concluded in Chapter 6. It summarizes the achieved objectives and the contributions to the research areas and points out the accumulated experience. Besides, it identifies the current open issues and introduces future research challenges and directions. Chapter 6 brings together all the papers involved in the thesis. Table 1.2 shows the relationship between publications and the dissertation chapters.

<table>
<thead>
<tr>
<th>Main Theme</th>
<th>Publication</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enablers and Theoretical Frameworks for NSM</td>
<td>I, II, III, VII, VIII</td>
<td>2</td>
</tr>
<tr>
<td>A Framework for Enabling NSM</td>
<td>I, II, III, IV</td>
<td>3</td>
</tr>
<tr>
<td>AI-based NSM Patterns in Next Generation Mobile Systems</td>
<td>V, VII</td>
<td>4</td>
</tr>
<tr>
<td>AI-assisted Service Function Chain Migrations for NSM</td>
<td>VI, VIII</td>
<td>5</td>
</tr>
</tbody>
</table>
2. Enablers and Theoretical Frameworks for Network Slice Mobility

In this chapter, we introduce enabling technologies, define the paradigms and the relevant nomenclatures, and overview the theoretical framework and method of the research. In Section 2.1, we present and compare various virtualization methods in related research works. We explain the reasoning behind the choice to use container-based virtualization rather than other solutions. Section 2.1 also includes an overview of the main concepts and paradigms used to achieve NSM and its various patterns. Section 2.2 presents a primer on Reinforcement Learning (RL), its various types and approaches, and the algorithms adopted in this dissertation. Finally, in Section 2.3, the chapter is summarized.

2.1 Enabling Technologies and Paradigms

In the following section, we present various virtualization approaches as well as their main characteristics. Next, we illustrate the different migration types and related terminologies concerning container-based virtualization. Finally, we highlight the main paradigms used in this dissertation to enable NSM.

2.1.1 Virtualization Approaches

Virtualization technologies allow physical devices to have several virtual instances, i.e., VMs, containers, and Unikernels, running simultaneously. Virtualization techniques are either hardware-level or Operating System(OS)-level. The hardware-level virtualization refers to VMs, while OS-level virtualization implies the use of containers or Unikernels. In the scope of this thesis, we focus exclusively on containers and VMs, while Unikernels are not considered.
The recent adoption of virtualization techniques helps to support cloud-native initiatives and improved network performance. Below, we highlight VMs and containers as virtualization enablers for next-generation network architectures.

**Virtual Machines:**

The use of VMs as a basic entity of cloud computing reduces both Capital Expenditure (CAPEX) and Operational Expenditure (OPEX) drastically. A VM is a software component that emulates an OS and runs on virtualization-based environments, like servers and their hypervisors. Besides, VMs allow high flexibility and elasticity and offer the possibility of deploying network services and functions which were, until recently, considered black boxes. Being hardware-level virtualization, each VM allocates CPUs, memory, and I/O devices using a hypervisor while remaining isolated from other VMs in the hosting physical server.

**Containers:**

Linux containers are a lightweight method for OS virtualization that leverages kernel sharing with the host. Each container has its own environment comprising CPU, memory, I/O blocks, network resources, naming, addressing, and resource management mechanisms. Most of the components needed by Linux containers are provided by the Linux kernel. These include namespaces and SELinux, both of which are used to ensure proper isolation between processes, containers, and host to containers separation.

In this dissertation, we employ the Linux Container (LXC), a lightweight virtualization technology which is integrated into the Linux kernel to enable the running of multiple containers on top of the same host [25]. The LXC container is built by mixing the Linux namespaces and CGroups to ensure a soft separation without virtualizing the hardware as VMs do. Compared to Docker, LXC [25] is a system-level container. In other words, it provides better flexibility when it comes to using system utilities, ensuring a similar powerful Linux system with less overhead than VMs.

### 2.1.2 Virtual Machines vs Containers

The main difference between container-based technologies and VM-based approaches is that the former provide lightweight virtualization mechanisms to run multiple workloads on a single OS, including VMs. Container-oriented implementations share the same OS kernel, thus, offering the
possibility to launch a much larger number of instances than VMs on the same hardware. For instance, a MySQL database service may be used once by a given VM, while containers offer the possibility of having several copies of the same service within a single VM. Figure 2.1 shows the difference between hypervisor-based virtualization technologies such as Kernel Virtual Machine (KVM) and container virtualization such as Linux containers.

![Figure 2.1. Architectures of virtualization technologies: hypervisor-based vs container-based.](image)

We acknowledge that hardware-based virtualizations, or VMs, are secure and efficient in terms of resource isolations. In this research, however, we rely on container-based virtualization for the following reasons:

- It has advantages in terms of management facilities;
- The rapid deployment and startup time [26];
- The replication [27–29], live service migration, and scaling methods are faster than traditional VMs [30];
- The computational resources for hosting standard virtualization technologies for MEC-enabled architectures in 5G and beyond networks are lacking.

### 2.1.3 Migration

Migration of computations refers to the process of moving one or a set of virtualization instances, i.e., VMs or containers, between two hosts, i.e., VMs, servers, or clouds. In this dissertation, we employ container migration. In fact, the simple migration principles apply to all virtualization
instances, i.e., VMs. There are three approaches for migrating virtual instances:

- **Cold Migration:** In this approach, the virtualization instance is turned off, then moved to another host. Later, the virtualization instance is restarted on the destination host.

- **Warm Migration:** In this approach, the virtualization instance is suspended instead of being wholly stopped as in a cold migration. The virtual environment is moved to another host, and the instance is resumed at the new destination.

- **Live Migration:** Live migration is the process through which we can transfer a “running” virtual instance between computer hosts, both its disk and current memory pages. The virtual instance may be running during the disk copy phase. In the second phase, the memory pages copy, the instance must be stopped, which creates a period where it will be unresponsive. This is known as downtime.

### 2.1.4 Downtime

Downtime is the period during which the services provided by the migrating instance, i.e., VM or container, are not available or no longer responding to user requests.

### 2.1.5 Total Migration Time

Total migration time is the period between launching the migration process and when the instance is made available on the destination server. Alternatively, it is the sum of the disk copy time, and the memory pages copy time, including the downtime. We can, therefore, derive the following formula:

\[
migration_{time} = disk_{time} + \sum_{k=1}^{k} iteration_{time}
\]  

(2.1)

Where \( k \) is the number of iterations. That is, \( k = 1 \) in the case of a single iteration, and is higher than 1 in the case of an iterative migration.

### 2.1.6 Live Iterative Migration

Since the number of memory pages and the available bandwidth are directly correlated with the downtime for the virtualization instance, we
can reduce the downtime using the concept of migration iteration. The iterative migration divides the copying of memory pages into several steps. Each step, except the last one, is possible without stopping the virtualization instance and considers only the changes relative to the previous iteration. In our proposed schemes, we named these intermediate steps the “Pre-Dump” phases, while the last one is named the “Dump” phase. During the Dump phase, the instance is stopped and a small amount of downtime is experienced due to the copying of a small number of memory pages to the destination host.

2.1.7 Checkpoint and Restore Mechanisms

The checkpointing procedure collects and saves the state of all processes of the container. This action occurs in the last iteration, the Dump-phase, of the iterative migration or during the copying of memory pages in the case of a basic live migration. The restore procedure re-spawns these processes from a previously generated file, the Dump file, during the checkpoint.

**Checkpoint/Restore In Userspace:**

CRIU [31] is an acronym for Checkpoint/Restore In Userspace. This open-source software allows checkpoint/restore processes in Linux systems. It can save the state of a running application so that its execution can later be resumed from the checkpoint time.

2.1.8 Software Defined Networking

SDN [32, 33] technology brings flexibility in the granularity of controlling network resources [34]. It is commonly identified as a key enabler of 5G networks since it can efficiently provide the on-demand reconfiguration of the critical networking resources [9, 35]. To take advantage of the flexibility provided by the SDN concept, a new network element, the SDN controller [36–38], needs to be deployed. The controller is a logically centralized entity responsible for storing the current network configuration requested by its operator and programming this configuration to the SDN-enabled switches of the infrastructure. The controller may also be involved in traffic flow processing as the switches may request it to handle packets that they do not know how to forward. Thus, SDN controllers may be based on OpenFlow [39], Network Configuration Protocol (NETCONF), Path Computation Element Configuration Protocol (PCEP), and similar protocols that separate traffic flow processing from the control and
management of the network devices performing the traffic flow processing, e.g., routing or switching.

SDN has gained significant momentum in recent years with the development of several open-source protocols and controllers for various purposes. In the following, we present the OpenFlow, a southbound protocol, and the SDN controller known as “ONOS”.

**OpenFlow:**

OpenFlow [39] was one of the first protocols to employ the SDN concept of extracting switch control to a centralized controller. OpenFlow provides a detailed view and control of the traffic flows passing through SDN-enabled switches. In this way, OpenFlow bypasses switch control restrictions that are “locked” into the proprietary OSs of network equipment. As a result, network operators can implement new strategies and protocols for traffic forwarding and routing.

An OpenFlow-enabled switch [40] includes a set of flow tables containing rules for identifying different traffic types, processing each packet of the flow (forwarding, dropping, adding/modifying VLAN tags and other header fields), and recording the number of packets processed for this flow. In the event that no rule matches a packet, the switch sends the packet’s header to the SDN controller. The SDN controller determines how the packet should be processed and pushes appropriate entries to create flow rules into the switch’s flow table. These new rules are then used for similar subsequent packets. This strategy is designed to mitigate the packet processing overhead on the SDN controller.

**Open Network Operating System:**

ONOS is an SDN controller conceived to support scalability, high availability, and performance [41]. It was designed for network service providers, and has the following key features: a distributed core for maximum scalability, modularity, southbound abstractions, and northbound abstractions. There are two main northbound abstractions: the first is the global network view, which provides the applications with a view of the network, the hosts, switches, and links, and the second is the Intent Framework, which enables the network administrator to manage the network with a high-level of abstraction by submitting Intents. An Intent could be, for example, to set up a connection between two particular hosts in the network. Intents will be handed over to the ONOS core, which is responsible
for translating them via “Intent Compilation” into “Installable Intents”, which are “Actionable Operations” to ONOS. The compilation process considers the network state and behavior requested in the Intent to generate one or more flow rules to implement the requested behavior in the ONOS managed network. These actions are then carried out by the “Intent Installation” process, which results in a set of flow rules installed on one or more selected switches in the network.

As shown in Figure 2.2, there are three different ways to push Intents on ONOS: (1) by creating an ONOS application, (2) using the Command Line Interface (CLI), or (3) through the Representational State Transfer (RESTful) interface. The first two access methods are equivalent, which means that the CLI will delegate the Intent creation according to the ONOS application while using the RESTful interface is equivalent to calling a Web Service with a particular request using the Hypertext Transfer Protocol (HTTP) protocol. This Web Service will relay this request to the ONOS Core after having transformed it into a request in a query format that can be understood by the latter. The primary generic Intents provided by ONOS are: “point-to-point”, “single-to-multi-point”, “multi-to-single-point” and “host-to-host”.

2.1.9 Service Function Chains

The SFC architecture specifications are addressed by the IETF SFC working group (RFC 7665) and the Open Network Foundation (ONF) [42, 43].
SFC is foreseen as a solution that will dynamically steer the network traffic and flows across multiple physical and logical infrastructures [44, 45]. Each SFC forms a set of NFs running inside either logical or physical nodes traversed in a certain strict order, leveraging NFV and SDN paradigms [46–48]. A notable example is the use of SFC to deliver a secure streaming service in which the intermediate services can be security functions, like the Intrusion Prevention System (IPS) or firewall, to protect from or prevent various attacks. It is possible to deploy several SFCs jointly: for example, two SFCs may be deployed in a connected car management use case. While the first SFC could be dedicated to the monitoring and control plane information, the second SFC could be used to apply different management actions, i.e., the data plane. It is clear that the new trend consists of instantiating NSs that contain one or more SFCs [49]. Thus, SFC is a key enabling paradigm for dynamic NS deployment in 5G and beyond networks.

2.1.10 Live Service Function Chain Migration

Like standard migration approaches, SFC migration operations permit the movability of services and applications amid different MEC nodes while ensuring low latency communications to end-users. However, SFC migration, in addition to ensuring service migration, guarantees the predetermined order of SFC components and their respective network and system dependencies. In this dissertation, we use various SFC migration strategies to handle the system requirements while we use ONOS as an SDN controller to steer network traffic between the SFC’s components.

2.1.11 Network Slicing

NS is an active research field within the academic community and among the different Standards Development Organizations (SDOs), such as the Next Generation Mobile Network Alliance (NGMN), 3GPP, and International Telecommunication Union — Telecommunication Standardization Sector (ITU-T) [50, 51]. In this dissertation, we define an NS as an End-to-End (E2E) logical network running on top of a common underlying, i.e., physical or virtual, network, with independent control and management, and flexibly programmable to meet Service Level Agreements (SLA) of a specific service [35]. An NS consists of computing and storage resources associated with virtual networks, possibly composed of multiple virtual
sub-network segments, which may span across multiple technological as well as administrative domains. Mobile users can connect to multiple slices simultaneously, depending on the type of services they are using [9].

2.2 A Primer on Reinforcement Learning

Hereafter, we present a brief introduction to RL and DRL, given their importance to the rest of the dissertation. Next, we outline the different types of RL and DRL algorithms, their advantages and limitations, and the algorithms used for answering RQs 7 and 8.

2.2.1 Reinforcement Learning Background

RL is one of the most important research directions of Machine Learning (ML), which has a significant impact on AI/ML development over the last twenty years [52]. Unlike supervised and unsupervised ML algorithms, RL techniques are independent of prior data [52]. RL algorithms or RL-based agents learn to perform complex tasks and take effective decisions through interaction with the environment on a trial and error basis. We define the interaction in terms of specific states/observations, actions, and rewards. Precisely, an RL agent interacts periodically with an environment “E”, observes the current state \( s_t \), then executes an action \( a_t \). Subsequently, the agent will observe a new state \( s_{t+1} \) and receives a corresponding reward \( r_{t+1} \) [53]. This process keeps repeating while adjusting the policy \( \pi(s_t, a_t) \) until the convergence phase, referred to as optimal policy [54]. A policy \( \pi \) has for objective to map states to actions, i.e., \( \pi : S \rightarrow A \), by maximizing the discounted reward over the discrete-time steps. The cumulative discounted reward \( G_t \) at each given time \( t \) is defined by:

\[
G_t = \sum_{m=0}^{\infty} \gamma^m r_{t+m+1} \tag{2.2}
\]

where \( \gamma \) is the discounting factor defined between [0-1]. The discount factor helps determine the importance of future rewards.

2.2.2 Reinforcement Learning Classes and Types

RL classes and algorithms are composed of value-based and policy-based methods, each of which has its advantages and inconveniences in terms of applicability, feasibility, and computation requirements. There is also a
hybrid approach in which both value and policy methods are combined [52]. While there is proof of convergence for RL methods, they have limited application in practice [55]. Admittedly, in complex and large-scale networks, state-action spaces became usually large, and RL struggles to represent them in current memory architectures. While RL has failed to find an optimal policy in a reasonable time, the emergence of the Deep Learning (DL) paradigm has caused a breakthrough in the ML area [56]. Therefore, DRL approaches combine basic RL methods with Deep Neural Networks (DNNs) to handle scalability issues effectively [57].

**Value-based Reinforcement Learning Algorithms:**
Among them, Q-learning, which is part of the value-based family, is one of the most prominent RL algorithms [58]. Q-learning uses a simple structure represented by a table dubbed Q-table. The algorithm, however, is limited and inefficient in practice. Consequently, Deep Q-Network (DQN) replaces the static Q-table with a DNN. The DNN computes the value of $Q(s_t, a_t)$ or a Q-value, which represents the quality of the selected action $a_t$ in the state $s_t$, and realizes an acceptable mapping from states to actions. Despite its good scalability regarding the number of states, DQN is often unstable and divergent. Therefore, experience replay memory was added to break the correlation between subsequent time-steps and allow a stable learning curve [59,60]. SARSA, Double DQN [55], Dueling DQN [61], Noisy DQN [62], and DQN with Prioritized Experience Replay [60] are prime examples of value-based RL/DRL algorithms.

**Policy-based Reinforcement Learning Algorithms:**
Policy-based methods directly learn the policy function that maps states to actions instead of computing action-value functions to each approximated state. Policy Gradients (PG) algorithms, such as REINFORCE and its variants [63], are efficient in high dimensional action spaces as well as continuous spaces compared to value-based algorithms [64]. Besides, policy-based methods can also learn from stochastic policies by outputting probabilities for each action. Therefore, policy-based methods handle the exploration/exploitation trade-off and eliminate the problem of the perceptual aliasing state where identical states require different actions [65]. Although policy-based methods can solve problems that value-based methods cannot, they usually converge on a local maximum rather than on the global optimum [66]. Consequently, the selection of RL/DRL algorithms must be related to the type of the problem and the computa-
Hybrid Reinforcement Learning Algorithms:
Based on the respective descriptions of the value and policy methods, we conclude that the selection of the algorithms relies mainly on the type of problem, learning time, and computational power. To this end, hybrid methods that combine both value-based and policy-based approaches can be a possible choice [67]. In those approaches, an agent measures the quality of actions through value-based methods while it optimizes the policy function leveraging the policy-based methods. This category incorporates many state of the art algorithms, including the Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C) [64, 68], Deep Deterministic Policy Gradient (DDPG) [69], and Proximal Policy Optimization (PPO) [70].

2.3 Summary

The main goal of this chapter was to provide the necessary background knowledge regarding the research area investigated in this dissertation. To this end, we have provided a general overview of the enabling technologies and paradigms, followed by the main definitions and nomenclatures adopted in the dissertation. We then finalized the chapter by presenting the necessary theoretical RL frameworks and their features.
3. A Framework for Enabling Network Slice Mobility

The focus of this chapter, which relies on Publications I, II, III, and IV, is the design, implementation, and evaluation of the NSM enabler framework, MIRA!, its components, and SDN-enabled modules. To achieve this objective, Section 3.1 introduces latency-sensitive optimized live migration approaches. In this section, different migration approaches to select the appropriate method for enabling NSM are evaluated and compared. Then, to comprehend how the ONOS SDN controller performs in terms of computational overhead, agility, and scalability, Section 3.2 benchmarks the northbound interface leveraging Intent-based networking. In Section 3.3, the various mechanisms responsible for handling resource consumption, networking challenges, and system control for guaranteeing short downtime and service continuity are proposed. Finally, in Section 3.4, the findings and conclusions are summarized.

3.1 Towards Fast Service Migration in 5G and Beyond Networks

NS enables the customization of network services and resources for the needs of different use cases. The same underlying infrastructure is shared among the different slices that add their own customized NFs and services to create a dedicated service network. The NS paradigm is expected to be a pillar for next-generation services and one of the 5G verticals enablers. These emerging services will run on top of MEC-enabled infrastructures via lightweight virtualization technologies, e.g., containers, as they require a configurable network with a high data rate and an E2E latency below the threshold specified per use case. However, neither the NS paradigm nor MEC mechanisms by themselves can ensure the required
QoE for the end-users because of the users’ unpredictable mobility patterns, e.g., the path and handover rate. This is underpinned by the high safety requirements posed on emerging 5G and beyond use-cases, such as autonomous vehicles, trains, or UAVs, calling for high reliability and low latency. The problem of reliable and seamless service consumption caused by user mobility patterns and the subsequent migration of services between edge clouds was addressed by the following:

- The introduction of an NFV compliant architecture to ensure high availability and to support ultra-low latency for real-time applications through service migration;

- The introduction of a complete framework that handles both the known and unknown mobility patterns while optimizing all steps of the migration process, mainly the disk copy and the memory page synchronization;

- The consolidation and evaluation of the proposed solutions via total migration time and downtime analyses while demonstrating the efficiency of the proposed solutions compared to prior works using similar underlying technology.

3.1.1 Main Architecture and Problem Formulation

Figure 3.1 depicts a three-layer cloud-based architecture for 5G networks. The architecture supports scalable and distributed 5G services deployments while meeting the 5G requirements in terms of low latency and high data rates. The proposed architecture complies with the ETSI-NFV standards and takes the orchestration and the management of the Core and MEC layers into account. The Core layer offers a centralized computational power that can accommodate data center capabilities from different vendors, e.g., Amazon, Microsoft, or private cloud solutions based on open-source projects such as OpenStack. Orchestrated by the top layer, the MEC layer features the Radio Access Network (RAN) with high spectral efficiency and bandwidth. In the NFV model, the Core and MEC layers form a distributed NFV Infrastructure (NFVI) that can be controlled by one or more Virtualized Infrastructure Managers (VIMs). In the envisioned architecture, containers host the components of VNFs managed by VNF Managers (VNFMs) that ensure the life-cycle management of VNF
instances distributed across multiple administrative domains (horizontal for MEC-to-MEC and vertical for Centralized Cloud-to-MEC). Moreover, this distributed computing model allows users in the “users layer” to be close to the computing capabilities according to their mobility. In our use-cases, the users may be onboard high-speed vehicles or UAVs, whereby their paths of movement can be either pre-determined/predictable [71–74] or random. The path-aware case allows us to trigger the migration process earlier, before reaching the edge of the cell, while in the path-oblivious solution, the actual migration must be completed before a given deadline, e.g., reaching the edge of the cell.

The main focus of our work is the implementation of live migration itself to ensure a seamless migration across edge clouds ignoring radio specific aspects of the use case, such as the signal strength received by each vehicle, user equipment (UE), or UAV. Furthermore, we noticed that the ETSI-NFV orchestrator (NFVO) architecture should be augmented to handle fine-grained live migration decisions. Based on whether the request is latency-sensitive or known/unknown mobility patterns, this new module of the NFVO decides the best target location for the migration. The NFVO ensures all security and communication requirements between the VNFM source, the VNFM destination, the VIM source, and the VIM destination. Given the “single domain” limitation expressed by both VNFM and VIMs, the NFVO coordinates either directly with the VIM to request migration action from a MEC source to a MEC destination or through both VNFM
and VIM elements.

The flowchart in Figure 3.2 details the proposed live migration strategies. We divide our solution into two main parts: the disk and the memory migration phases. Initially, the need for a migration action is detected by the management plane, which can be a part of the ETSI NFV's NFVO architecture. The management component can determine whether the service is serving an instance with an unknown mobility pattern, referred to by Step 0 in Figure 3.2. In the case of a known mobility pattern, a series of methods are used, with the color of the rhombus signifying an action in a given part of the proposed architecture:

- The memory migration check (Step 1 in Figure 3.2) is the initial test where the availability of the clone image (base image and the application) is verified in the target MEC host. If available, a cloning image process is started, followed by a memory migration;

- The partial migration check (Step 2 in Figure 3.2) is carried out in case of the non-availability of the clone image. Verification is done at the target MEC to find the base image. For example, in the case of Ubuntu, Trusty or Xenial is used for verification. Once found, cloning of the base image starts, followed by a copy of the application data from the source MEC to the target MEC and a memory migration;

- The full migration check (Step 3 in Figure 3.2) represents the final step, and the worst-case scenario as the entire file system (rootfs),
the application, and the memory need to be transferred because of their absence in the target MEC.

However, for end-users with unknown mobility patterns, we must rely on the latency test as our second solution requires a shared storage system, which may negatively impact latency-sensitive applications. Since the delay between the destination host of the container and the shared storage can be more significant than the application tolerance, the migration can harm the service’s QoS/QoE. To mitigate this issue, we can migrate latency-sensitive services to other MEC, even if that increases the migration time but offers a lower E2E delay.

In the following section, a detailed explanation of known/unknown mobility patterns is provided. The predefined mobility pattern and then the unknown mobility pattern scenario are discussed.

3.1.2 Stateful Service Migration Based on a Predefined Mobility Pattern

Hereafter, our solution for UEs with predefined mobility patterns is presented. As discussed, prior knowledge of the movement path allows us to anticipate different source and target MECs for any migration and the UEs’ movement paths. Moreover, we can implement the migration between MEC hosts without the use of shared storage. It is worth mentioning that all previous migration steps, including memory migration, partial migration, and full migration, work only for the predefined mobility pattern solution, as this is an optimization of the disk migration procedure.

In the case where both memory migration and partial migration checks fail, the predefined mobility pattern solution starts by copying the container’s file system and the user files from the current MEC host to the target MEC host using the remote synchronization utility (rsync) without service disruption. However, in case of a successful partial migration, the application copy is the only mandatory transfer. In the case of memory migration, however, no data transfer is required. A live iterative migration based on two different variants using the CRIU utility [31] to detail the memory pages migration is introduced in the next section.

**Temporary File System based Lightweight Container Migration:**

This solution is also named temporary file system (tmpfs) migration. The tmpfs migration solution starts by copying the container’s memory iter-
Atively from the source MEC host to the destination MEC host. In this step, the CRIU utility will be used for iteratively dumping the container's memory - while it is running - into a tmpfs-mounted directory at the source MEC host. In this case, at the source MEC host, there is one reading operation from the memory and one writing operation to the tmpfs-mounted directory. In fact, there is one reading and one writing at the source MEC host. Each iteration, i.e., Pre-Dump for non-blocking iterations and Dump for blocking iteration, is then copied to the destination host via the network into the tmpfs-mounted directory at the destination MEC host, which results in a writing operation at the destination MEC host. Finally, read actions are used to restore the container at the destination MEC host. The control operation shown in Step 4 in Figure 3.2 was devised to determine the number of iterations. Step 4 involves the following:

- It uses a well-defined page number to guarantee the shortest downtime possible;
- Alternatively, it uses a fixed number of iterations to avoid infinite looping where the number of pages will never go below a pre-defined threshold of dirty pages. This happens if the rate of page modifications is greater than the link speed.

**Disk-less Lightweight Container Migration:**

While the tmpfs-based migration approach requires two readings and two writings of the memory images, the disk-less migration solution overcomes this limitation by eliminating the step of copying the images to the local tmpfs directory, i.e., source MEC host. This results in a further reduction of the migration downtime and total time. The disk-less solution starts a page server at the destination MEC host indicating the images directory and the port used by the source MEC host to copy the files over the network. Then, on the source MEC host, using CRIU, we adopt a new strategy by combining our iterative approach of live migration, i.e., tmpfs-based migration, with the page server implementation. We start by dumping the memory pages directly into the target cloud using two extra parameters (the page server's address and port) while keeping the iterative concept working. Finally, we copy the rest of the images to the destination MEC host, and we restore our container immediately afterward. This approach uses the same control operation shown in Figure 3.2 for deciding on the final iteration, i.e., Dump or last iteration.
3.1.3 Stateful Service Migration Based on Undefined Mobility Pattern

In most real-world applications, the service provider (cloud service provider) does not know the users’ mobility patterns. Therefore, we propose a more generic solution that considers users’ movement paths to be unknown a priori. Under this condition, the file system and memory copy from the source MEC host to the destination MEC host could be a challenging process. For this case, we have devised a solution named “lightweight container migration with a shared file system”, which offers an alternative, fast and efficient migration method. This method is shown in Figure 3.2 as Step 0, where we consider an unknown mobility pattern in addition to non-latency-sensitive services. First, we eliminate the need to copy files over the network during the migration phase, storing the container’s file system and the system images in a shared storage pool. This allows us to focus on the page memory migration process to iteratively unload the container’s memory using CRIU on the source node and then immediately restore it in a container on the target node. This approach uses more network resources while reducing the total migration time for LXC benefiting latency-sensitive applications. We use the same logic as for the pre-defined mobility patterns after handling the memory copy phase.

3.1.4 Experimental Evaluation

We evaluate our proposed container migration solution, using virtualized nodes, each node running Ubuntu 16.04 Long-Term Support (LTS) with the 4.4.0-64-generic kernel, a 16 cores CPU, and 32 GB (Gigabyte) of main memory. The interconnection among the nodes is set to 1 Gbps (Gigabit). We configure two testbeds using LXC 2.8 and CRIU 3.11:

- The first testbed consists of two VM hosts, as shown in Figure 3.3(a), each representing a different MEC host, i.e., an independent IaaS provider. Our container host is running on top of the first VM. We also deployed a third host representing the management plane previously discussed;

- The second testbed consists of three VMs, i.e., Figure 3.3(b). The first VM is the source MEC host, whereas the second one is the destination MEC host. Meanwhile, the third VM is the Network File System (NFS) server used to store the containers’ file-system. Moreover, we ensure that the three VMs can communicate among
themselves to enable container migration in the testbed. We use the communication between the MEC hosts and the NFS server for disk migration, while the direct communication between MEC hosts is used to migrate the memory content.

![Diagram](image1)

(a) Prototype 1.

![Diagram](image2)

(b) Prototype 2.

**Figure 3.3.** Proposed prototypes for both known and unknown path solutions.

For every container migration, we evaluate the total migration time and the container downtime. The latter directly corresponds to the application responsiveness/availability during the migration process. Two sets of experiments were conducted, each with ten repetitions. The first one was a blank, i.e., there was no application running, Linux container migration, and the size of the file system was equal to 350 MB. We paid close attention to the reachability of the container's network throughout the migration process to observe the impact caused by adding persistent data. The second one was the migration of a video streaming server, NGINX, running on a container, whereby the file-system size was 590 MB. We began with the downtime comparison to determine the suitable approach for memory page migration. Then we continued with the total migration time analysis. Three migrations strategies were evaluated: full migration, partial migration, memory migration for the predefined mobil-
ity pattern case, and the shared migration as part of the unknown path situation.

**Migration induced downtime:**

(a) Downtime comparison in case of an iterative migration process.  
(b) Downtime comparison in case of a disk-less migration.

**Figure 3.4.** Downtime in case of different migration approaches.

The downtime, Standard Deviation (STD), 95% Confidence Interval (CI), and Coefficient of Variation (CV) results are shown in Table 3.1 for both the blank and the video-streaming containers considering tmpfs and disk-less migration approaches as part of finding an appropriate solution for an optimized memory pages migration. As expected, video streaming container downtimes are larger than blank container downtimes in both migration strategies. The difference in these results is due to the additional copies of the network connections status and the NGINX internal control data to the target cloud. We also noticed that the addition of the NGINX HTTP server introduced more variability in our experiments; nevertheless, this represented an increase in the CVs of less than 15.686% and 10.606% for the tmpfs and the disk-less approaches, respectively.

**Table 3.1.** Downtime comparison in case of tmpfs and disk-less migration approaches.

<table>
<thead>
<tr>
<th>Migration types</th>
<th>Strategies</th>
<th>Mean Time (s)</th>
<th>STD</th>
<th>95% CI</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmpfs migration</td>
<td>blank migration</td>
<td>1.043</td>
<td>0.053</td>
<td>0.040</td>
<td>0.051</td>
</tr>
<tr>
<td></td>
<td>video migration</td>
<td>1.282</td>
<td>0.076</td>
<td>0.087</td>
<td>0.059</td>
</tr>
<tr>
<td>disk-less migration</td>
<td>blank migration</td>
<td>1.152</td>
<td>0.076</td>
<td>0.088</td>
<td>0.066</td>
</tr>
<tr>
<td></td>
<td>video migration</td>
<td>1.465</td>
<td>0.107</td>
<td>0.081</td>
<td>0.073</td>
</tr>
</tbody>
</table>

Figure 3.4 presents a breakdown of the time for the three phases for both the tmpfs migration, i.e., Figure 3.4(a), and the disk-less migration, i.e., Figure 3.4(b), experiment scenarios. As previously described, the migration procedures adopted use two pre-copy phases – 1\textsuperscript{st} and 2\textsuperscript{nd} iterations in Figure 3.4 – before actually migrating the container to the destination compute node in the final iteration phase. In Figure 3.4, each iteration
can be viewed in the X-axis, while the Y-axis presents the time in milliseconds. For both approaches, we observe that in pre-copy phases, the video container took longer than the blank container, as the migration procedure needs to save and copy additional memory updates during each iteration, which ends up increasing the downtime in comparison with the blank container.

From Table 3.1, we can clearly observe that the disk-less migration procedure imposed a 10.499% and 14.277% increase in the downtime for the blank migration and the video-streaming containers in comparison with the tmpfs migration procedure. Furthermore, comparing the results presented in Figures 3.4(a) and 3.4(b), it can be easily noticed that for the blank container and in the pre-copy phases, i.e., 1st and 2nd iterations, the tmpfs migration took a longer time than the disk-less migration. However, the same iterations for the video containers in the page server implementation take a longer time than the normal iterations, i.e., tmpfs migration, which reinforces our suspicions about the page server implementation. Based on the obtained results, we adopt the tmpfs solution in what follows for the memory migration, the partial migration, the full migration, and the shared file system migration approaches.

**Total migration time evaluation:**

Our previous experimental results showed that our proposed approaches reduce the downtime caused by the migration procedure. However, to support ultra-short latency services, we need to address the total migration time. To evaluate this, the same experimental scenarios were used as previously described. The results are shown in Figure 3.5 for both the blank, (the slashed bar) and video-streaming (the standard bar) containers. In Figure 3.5, the Y-axis is in seconds. For each bar, we also plotted the 95% CI of the mean.

The mean total migration time, the STD, the 95% CI, and the CV for the blank and video-streaming containers are shown in Table 3.2. For the pre-defined mobility pattern solutions, the memory migration was the fastest, as there is no need for disk copy. Only the final rsync was leveraged to ensure the synchronization between the source and destination MEC hosts; this was an additional 7 s in the total migration time.

The results for empty containers show the impact of adding services on migration time. The results indicate that the long migration time was due to the file system copy for all the pre-defined mobility pattern migrations, while this was avoided in the shared file system migration scenario.
Note that the container size only increased the total migration time in the case with local storage, mainly due to the file system copy for the video streaming. Furthermore, for the shared file system scenario, the longer migration time of the video container, compared to the blank one, is due to the high number of memory pages copied. Moreover, the partial migration strategy also induces a longer total migration time than the full migration procedure. Thus, we can conclude that sending the whole disk, i.e., rootfs or file system, through the network is faster than cloning the base image, e.g., trusty for Ubuntu distribution, followed by the transfer of the application’s meta-data and data over the same network. However, partial migration is clearly a more promising solution because it limits the bandwidth consumed during the migration process since only the application’s data is transferred.

**Table 3.2.** Total migration time comparison in case of different migration approaches.

<table>
<thead>
<tr>
<th>Migration types</th>
<th>Strategies</th>
<th>Mean Time (s)</th>
<th>STD</th>
<th>95% CI</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-defined mobility pattern</td>
<td>B. Mem-Mig</td>
<td>8.990</td>
<td>0.271</td>
<td>0.204</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>V. Mem-Mig</td>
<td>9.855</td>
<td>1.617</td>
<td>1.219</td>
<td>0.164</td>
</tr>
<tr>
<td></td>
<td>B. Part-Mig</td>
<td>16.885</td>
<td>2.605</td>
<td>1.964</td>
<td>0.154</td>
</tr>
<tr>
<td></td>
<td>V. Part-Mig</td>
<td>38.918</td>
<td>2.670</td>
<td>2.013</td>
<td>0.069</td>
</tr>
<tr>
<td></td>
<td>B. Full-Mig</td>
<td>19.518</td>
<td>0.379</td>
<td>0.286</td>
<td>0.019</td>
</tr>
<tr>
<td></td>
<td>V. Full-Mig</td>
<td>31.609</td>
<td>1.394</td>
<td>1.051</td>
<td>0.044</td>
</tr>
<tr>
<td>Unknown mobility pattern</td>
<td>B. Sh f.sys Mig</td>
<td>2.831</td>
<td>0.269</td>
<td>0.203</td>
<td>0.095</td>
</tr>
<tr>
<td></td>
<td>V. Sh f.sys Mig</td>
<td>3.678</td>
<td>0.206</td>
<td>0.156</td>
<td>0.056</td>
</tr>
</tbody>
</table>

**Impact of the number of pages on the migration downtime:**

To avoid any bias from the underlying hardware technology on our experimental evaluation, the number of pages copied during the last iteration is listed in Table 3.3, along with its computed mean, the STD, and the 95% CI for tmpfs migration, disk-less migration, and the shared file sys-
tem migration procedures. An initial assessment of this table shows the disparity between the disk-less migration solution and the others. This corroborates the results presented in Figure 3.4(b). Both indicate that the larger downtime of this approach was due to the interaction of the page server and the CRIU migration code, which resulted in the number of copied pages to increase by a factor of almost 8.84 times in the worst-case scenario. Furthermore, it can be concluded that the behavior exhibited for the tmpfs and shared file system solutions is quite similar for both the blank container and our video streaming container, despite the disparate downtime performance observed. The higher downtime for the shared file system solution is caused by the multiple small writes over the network. Also, downtime using the tmpfs approach was significantly smaller because the number of copied pages in the last iteration is kept small.

We also qualitatively evaluated the impact of the network bandwidth on experimental results where our best experimental scenario was 500.7, i.e., 2,050,867 Bytes, and 706.6, i.e., 2,894,234 Bytes, pages transferred for the blank container and video container, respectively. As the amount of transferred data is unable to utilize a gigabit link fully, most of the downtime improvements result from our implementation and not the network itself. The network performance for small transfers is constrained by the slow start of the Transmission Control Protocol (TCP) and the packet header/trail overhead. Moreover, even a simplistic analysis that omits TCP issues, the data transfer would take around 164 ms for the blank container and 231 ms for the video container, assuming a Fast Ethernet connection (100 Mbps). In comparison, for a Gigabit Ethernet (1 Gbps), the time spent copying the pages over the network was of the order of 16.4 ms for the blank container and 23.1 ms for the video container.

Table 3.3. Summary of pages copied during the last iteration.

<table>
<thead>
<tr>
<th>Migration types &amp; cases</th>
<th>Mean N. Pages</th>
<th>STD</th>
<th>95% CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank-Tmpfs Migration</td>
<td>509.5</td>
<td>6.30</td>
<td>5.14</td>
</tr>
<tr>
<td>Blank-Disk-less Migration</td>
<td>1779.3</td>
<td>6.84</td>
<td>5.57</td>
</tr>
<tr>
<td>Blank-Shared file system Migration</td>
<td>517.8</td>
<td>17.56</td>
<td>14.30</td>
</tr>
<tr>
<td>Video-Tmpfs Migration</td>
<td>577.1</td>
<td>7.52</td>
<td>6.125</td>
</tr>
<tr>
<td>Video-Disk-less Migration</td>
<td>5100.2</td>
<td>12.52</td>
<td>10.29</td>
</tr>
<tr>
<td>Video-Shared file system Migration</td>
<td>551</td>
<td>16.80</td>
<td>13.68</td>
</tr>
</tbody>
</table>

Results Discussion:

We have observed that although the solutions based on predefined paths offer the best downtime results, the total migration time is the highest among the options considered. This is simply because the file system, user
files, and memory images are copied during the migration phase. This is not considered a problem because migration operations can be foreseen in advance. In contrast, the solution applied in unknown paths offers great overall migration time while sacrificing a few downtime milliseconds since the available bandwidth of the network limits the storage speed. However, this is the best solution for this scenario since the decision to trigger the migration, along with the migration process itself, is completed within a couple of seconds.

### 3.2 Benchmarking the ONOS Intent Interfaces to Support Network Slice Mobility Patterns

In conventional networks, enabling fast and reliable live service migrations, as detailed in Section 3.1, is sufficient to ensure the high availability of users' services. However, the introduction of the MEC paradigm helps NS to span across multiple technological and administrative domains. This introduces new networking challenges related to connection resumptions and path redirections after users' mobility and migration actions. It may be possible to use the programmability of software defined networks to manage the connections as the user moves. Indeed, an SDN is mainly a proactive approach for handling flow establishments. This approach can enable cross data-center migration and the development of complex SDN-enabled distributed applications considered as prerequisites to enable NSM.

Nevertheless, integrating an SDN approach directly through its southbound interface may add complexity to NSM. Thus, to meet complex user demands, support fast reconfiguration after service migrations, and NSM requirements, ONOS is providing a higher-level abstraction called Intents. These Intents are used to manage the network functionality through ONOS's northbound interfaces by hiding lower-level protocols. Therefore, we have done the following:

- Introduced a northbound benchmarking methodology relying on interface access method, type of Intent, and the number of installed Intents to measure agility and scalability;

- Experimentally evaluated and benchmarked the performance of ONOS RESTful and command line northbound interfaces for the three dif-
ferent basic Intent types;

- Determined best practices in developing complex behaviors and ONOS-aware distributed applications to support NSM and its different patterns.

### 3.2.1 Methodology

Evaluating SDN controllers based on simulated or emulated traffic sets provides a practical understanding of their strength, robustness, and scalability. However, such an approach is not suitable for benchmarking the Intent abstraction of ONOS. While southbound interface benchmarking can be done using Cbench [75] to generate “flow requests”, the northbound interface that implements Intent abstraction operates not on flows but policy-based directives containing descriptions of network resources, their constraints, and criteria to select traffic.

Therefore, our work focuses on how the SDN controllers behave and how their performance changes when using different Intents, and what is the impact of using different interfaces to communicate Intents to the ONOS controller. Moreover, we focus our evaluation on the basic Intents provided by the ONOS core architecture, as we study how ONOS behaves when an external application submits Intent installation requests at a high rate. The results presented in this benchmarking allow us to evaluate how an external application, i.e., NSM, can use ONOS to install its desired network configuration while limiting the communication cost with the SDN controller.

In our benchmarking of the Intent interface, we are interested in how ONOS behaves in terms of agility and scalability. We define “Agility” as the amount of time required to successfully install a set of Intents and “Scalability” as how many Intents a single ONOS instance can manage. These two metrics provide a clear understanding of the expected performance of an ONOS deployment to NSM use-cases. In our analysis, the focus was on three basic Intent types that can be used to create more complex Intents: “point-to-point”, “single-to-multi-point”, and “multi-to-single-point”.

The compilation and installation times of an Intent depend significantly on the used computational resources, like CPU cycles and the memory of the used servers. Therefore, to make our results independent from a deployed testbed configuration, we measured the amount of time required
CHAPTER 3. A FRAMEWORK FOR ENABLING NETWORK SLICE MOBILITY

to install a fixed set of Intents using both the CLI and the RESTful interfaces provided by ONOS. For the CLI, we augmented the ONOS code, introducing timekeeping mechanism inside the ONOS core for each CLI Java class, in our case: (1) “AddPointToPointIntentCommand.java”, (2) “AddSinglePointToMultiPointIntent.java”, and, (3) “AddMultiPointToSinglePointIntentCommand.java”. These are used to create “point-to-point”, “single-to-multi-point”, and “multi-to-single-point” Intents, respectively. We also measured the amount of time required to include the “add” method for each Intent we were benchmarking. For the RESTful interface tests, we measured the time required to push the benchmarked Intents through python scripts.

3.2.2 Experimental Evaluation

The basic Intent types over the Intent interfaces of ONOS were evaluated using two computer nodes, running Ubuntu 16.04 LTS with an Intel(R) Xeon E5-2640 v3 2.60GHz CPU and 8 GB of RAM. We use version 1.9.0 of ONOS, and our test network topology is built using OpenvSwitchs (OVS). The CLI evaluation was carried out directly on the server running our ONOS instance, and another computer was used to submit the RESTful requests through the network to ONOS. The auxiliary computer runs an Ubuntu Desktop 16.04 LTS with an Intel(R) Core (TM) i5-6300U @ 2.40GHz CPU and 8 GB of volatile memory.

For each type of benchmarked Intent type—“point-to-point”, “multi-to-single-point”, and “single-to-multi-point”, we ran 50 iterations to find out the mean installation times, STD, and 95% CI for each set of the Intents. We vary the number of Intents (workload) installed in each experiment from 1,000 to 20,000. This procedure was adopted to mitigate the noise that may be introduced by the processor cache, system daemons, Java interpreter, or the network jitter. For the results plotted henceforth, the Y-axis shows the total time to install the Intents in milliseconds, and on the X-axis, we show the number of installed Intents in the experiment. The figures also show the mean and 95% CI.

Multi-to-Single-Point Intent:
Figure 3.6 summarizes our results for applying “Multi-to-Single-Point” Intents through CLI and RESTful interfaces of ONOS varying workload from 1,000 to 20,000 Intents. The 95% CI and the mean are calculated us-
ing 50 executions of the same Intent. As the 95% CIs were tiny concerning the mean values for both the CLI and the RESTful interface, we magnify the CIs 50 times in the figure (instead of showing the original values).

As shown from the figure, the installation time through the ONOS RESTful interface grows 20 fold when we increase the number of Intents from 1,000 to 20,000. Still, in Figure 3.6, the CLI installation time also shows a linear increase as we increase the number of Intents, albeit with a lower computational cost.

The difference between computational costs of RESTful and CLI interfaces seems to stem from the amount of I/O code that the Intent requests have to pass through. The differences in installation times shown in Table 3.4 suggest this.

Table 3.4. Mean installation time and its 95% CI for Multi-to-Single Intent.

<table>
<thead>
<tr>
<th>Num. Intents</th>
<th>Mean RESTful (ms)</th>
<th>95% CI RESTful</th>
<th>Mean CLI (ms)</th>
<th>95% CI CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>637.990</td>
<td>2.816</td>
<td>35.771</td>
<td>4.770</td>
</tr>
<tr>
<td>2,000</td>
<td>1,251.510</td>
<td>4.718</td>
<td>142.857</td>
<td>9.793</td>
</tr>
<tr>
<td>3,000</td>
<td>1,859.501</td>
<td>7.295</td>
<td>238.800</td>
<td>19.257</td>
</tr>
<tr>
<td>4,000</td>
<td>2,506.155</td>
<td>7.769</td>
<td>285.971</td>
<td>17.619</td>
</tr>
<tr>
<td>5,000</td>
<td>3,122.163</td>
<td>7.831</td>
<td>372.629</td>
<td>21.819</td>
</tr>
<tr>
<td>10,000</td>
<td>6,155.655</td>
<td>16.996</td>
<td>658.857</td>
<td>51.652</td>
</tr>
<tr>
<td>15,000</td>
<td>9,215.017</td>
<td>28.424</td>
<td>794.800</td>
<td>50.064</td>
</tr>
<tr>
<td>20,000</td>
<td>12,330.511</td>
<td>34.455</td>
<td>1,054.343</td>
<td>73.421</td>
</tr>
</tbody>
</table>

**Single-to-Multi-Point Intent:**

A summary of the “Single-to-Multi-Point” Intent results is provided in Figure 3.7 and Table 3.5. In the previous experiment, the mean values for both the RESTful and CLI interfaces are shown. The RESTful interface presents a similar I/O overhead penalty as that in earlier experimen-
ments. Besides, Figure 3.7 presents the details of two workload points of our evaluation, mainly 10,000 and 20,000 Intents. Like the experiment in Figure 3.6, the growth in the ONOS RESTful interface was ten times higher than that of the ONOS CLI interface when the number of Intents increases. Further details, values, and results are shown in Table 3.5. Note we magnify 95% CIs for both the CLI and the RESTful interfaces due to their small values.

**Table 3.5.** Mean installation time and its 95% CI for Single-to-Multi-Point Intent.

<table>
<thead>
<tr>
<th>Num. Intents</th>
<th>Mean RESTful (ms)</th>
<th>95% CI RESTful</th>
<th>Mean CLI (ms)</th>
<th>95% CI CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>700.589</td>
<td>4.695</td>
<td>109.800</td>
<td>13.900</td>
</tr>
<tr>
<td>2,000</td>
<td>1,403.436</td>
<td>5.817</td>
<td>192.514</td>
<td>17.544</td>
</tr>
<tr>
<td>3,000</td>
<td>2,196.356</td>
<td>4.081</td>
<td>270.600</td>
<td>31.681</td>
</tr>
<tr>
<td>4,000</td>
<td>2,805.094</td>
<td>5.822</td>
<td>401.657</td>
<td>40.376</td>
</tr>
<tr>
<td>5,000</td>
<td>3,493.791</td>
<td>7.800</td>
<td>509.057</td>
<td>49.977</td>
</tr>
<tr>
<td>10,000</td>
<td>7,018.610</td>
<td>21.372</td>
<td>678.171</td>
<td>37.218</td>
</tr>
<tr>
<td>15,000</td>
<td>10,619.652</td>
<td>24.334</td>
<td>957.886</td>
<td>86.654</td>
</tr>
<tr>
<td>20,000</td>
<td>14,143.237</td>
<td>35.722</td>
<td>1,055.714</td>
<td>126.397</td>
</tr>
</tbody>
</table>

**Point-to-Point Intent:**

The two previous performance tests showed that the RESTful interface imposes a significant overhead on the Intents installation. To further evaluate this behavior, we created a simulated complex Intent that contains multiple Intents with just one RESTful call using the ONOS’s “Point-to-Point” Intent, which implements a simple forwarding from one OVS port to another. Figure 3.8 summarizes our results with a 50 times magnification of the 20,000 workload points for the RESTful and CLI interfaces. As shown in Figure 3.8, the simulated complex Intent’s behavior has a similar computational cost as the CLI “Point-to-Point”, albeit with
CHAPTER 3. A FRAMEWORK FOR ENABLING NETWORK SLICE MOBILITY

Figure 3.8. Point-to-Point Intent - benchmarking results.

Table 3.6 shows the precise mean values plotted in Figure 3.8. The results offer further support to our interpretation of the different overhead increase rates of the RESTful and CLI interfaces. This behavior is attributable to how our benchmark code interacts with ONOS. The CLI evaluation is based on an existing ONOS command that requests the ONOS core to execute an Intent installation multiple times. Therefore, our measurements consider only the time required by ONOS to install the Intent itself and not the time user I/O spends. However, for the RESTful interface, the installation of a workload was carried out through the ONOS default interface one by one per request.

Table 3.6. Mean Installation time and its 95% CI for Point-to-Point Intent.

<table>
<thead>
<tr>
<th>Num. Intents</th>
<th>Mean RESTful (ms)</th>
<th>95% CI RESTful</th>
<th>Mean CLI (ms)</th>
<th>95% CI CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>669.276</td>
<td>24.032</td>
<td>57.200</td>
<td>9.468</td>
</tr>
<tr>
<td>2,000</td>
<td>1,226.505</td>
<td>11.712</td>
<td>162.229</td>
<td>18.225</td>
</tr>
<tr>
<td>3,000</td>
<td>1,823.249</td>
<td>20.108</td>
<td>211.714</td>
<td>20.682</td>
</tr>
<tr>
<td>4,000</td>
<td>2,436.431</td>
<td>17.329</td>
<td>337.343</td>
<td>37.725</td>
</tr>
<tr>
<td>5,000</td>
<td>3,167.603</td>
<td>20.108</td>
<td>394.400</td>
<td>42.135</td>
</tr>
<tr>
<td>10,000</td>
<td>6,192.263</td>
<td>77.685</td>
<td>663.829</td>
<td>64.341</td>
</tr>
<tr>
<td>15,000</td>
<td>9,246.514</td>
<td>104.522</td>
<td>913.886</td>
<td>45.687</td>
</tr>
<tr>
<td>20,000</td>
<td>12,622.108</td>
<td>75.340</td>
<td>1198.086</td>
<td>66.572</td>
</tr>
</tbody>
</table>

Result Discussion:
The performance evaluation demonstrates that the ONOS CLI imposes a smaller overhead to the Intent northbound interface than the RESTful interface, and this behavior is consistent across all benchmarked Intent types. Our preliminary analysis indicates that the difference in perfor-
mance is caused by both the extra computational cost of the RESTful interface and the cost associated with the remote connection used in our testbed environment for the RESTful test application.

In addition to the performance results mentioned above, we have also computed the maximum number of Intents a single ONOS instance can execute in our test topology. We ran ten iterations of microbenchmarks that submitted Intents in an infinite loop. We waited until ONOS was unable to install new Intents for each iteration, then computed for each execution the time required to install all the Intents of the same type. The results are summarized in Table 3.7.

Table 3.7. Intents processing limitation for a single ONOS controller.

<table>
<thead>
<tr>
<th>Intent type</th>
<th>Maximum number</th>
<th>Required time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point-to-Point</td>
<td>486,363.9</td>
<td>311,046.2</td>
</tr>
<tr>
<td>Single-to-Multi-Point</td>
<td>364,249.7</td>
<td>299,648.5</td>
</tr>
<tr>
<td>Multi-to-Single-Point</td>
<td>335,267.6</td>
<td>334,450.5</td>
</tr>
</tbody>
</table>

Finally, our experimental results show that ONOS can deliver higher performance if existing Intent types are extended to create a more complex Intent. This conclusion holds for an ONOS-aware application that interacts with the SDN controller through the RESTful API, i.e., NSM in our case. Our results also show that, regardless of the Intent type and the northbound interface used, the computational cost increases linearly with the number of Intents. This is the case at least for these basic Intents.

3.3 MIRA!: An SDN-based Framework for Network Slicing Mobility and Ultra-Low Latency 5G Services

Given the continuously growing demand for inter-data-center services that 5G networks are bringing, live lightweight migration, presented in Section 3.1, has become a coveted and very challenging technology. Meanwhile, the emergence of SDN and NFV technologies has wholly transformed modern networks by offering more flexibility and, at the same time, more complexity. So far, investigations have been confined to integrating the live migration process with SDN/NFV paradigms to enable NSM while satisfying users’ QoE, as introduced in Section 3.2. However, simple integration is insufficient to handle such unexpected cases as resource unavailability, networking issues, and system control. In this
context, a complete framework that would combine the results obtained from the previous works is needed to provide an autonomic migration and handle unexpected cases, like networking misconfigurations, identical nomenclature in various MEC nodes, and resource shortage in the target MEC hosts, which can destroy the entire system instead of regularizing it. Consequently, our aims are as follows:

- Introduce the framework design and the main components of the envisaged system;
- Detail their respective working principals and their limitations;
- Perform a set of experimental evaluations under different configurations and compare the obtained results to the prior work presented in Section 3.1.

### 3.3.1 Framework Design

Our proposal is a novel framework to manage live migration across multidomain clouds based on LXC containers [25]. The MIRA! framework provides functions to control container-based applications for many use cases, including live streaming, enabling NSM with various mobility patterns via parallel migrations while ensuring the network scalability and guaranteeing the desired QoE. The proposed system allows management and control operations in an SDN-aware environment through RESTful or web interfaces. Furthermore, it enables service migration over different edge clouds by ensuring service connectivity through integration with SDN networking. Figure 3.9 depicts the main components of the envisaged system.

**Host Node:**

The computational resource used to host a MEC app or an NF in MIRA! can be a physical or a virtual node. Each Host Node (HN) must include the virtualization infrastructure, i.e., the LXC container engine and OVS. Service connectivity is guaranteed using OVS switches appropriately installed in the HNs and interconnected through Virtual eXtensible Local Area Network (VXLAN) and Generic Routing Encapsulation (GRE) tunnels to enable the overlay network, decoupling the virtual network from the physical infrastructure.
CHAPTER 3. A FRAMEWORK FOR ENABLING NETWORK SLICE MOBILITY

Figure 3.9. MIRA! framework architecture.

The SDN Controller:
The SDN controller may run on top of a physical machine: this can be a dedicated server or another VM with more powerful resources in any cloud provider domain. Based on the benchmarking done in Section 3.2, we use ONOS [76] to manage the OVS switches and install the flows required for steering the traffic directed to the services.

MIRA! Global Orchestrator:

The MIRA! Global Orchestrator (MGO) is responsible for managing service migration, which is composed of several steps. First, a client can trigger the service migration through either a web interface or RESTful calls by sending a request to MGO, i.e., step 1 in Figure 3.9. After that, our framework enables a checking process, monitoring behavior, flows composition through the SDN controller, and enables a parallel logic to allow a seamless migration as represented by steps 2, 3, and 4 in Figure 3.9. Finally, MIRA! migrates services and performs path redirection using the SDN controller through steps 5 and 6 of the architecture. All these features are provided by the MGO that is composed of different submodules:

Checking Module: The checking module verifies the resources on both the source and target HNs before starting a live migration of services over multiple cloud domains. The collection of information on the host includes: i) the size of the disk, ii) the available memory, and finally, iii) the CPU. This analysis is necessary to check if there are enough resources in the destination HN to host the migrated services while ensuring the
desired performance. Different strategies were developed according to the resource estimation:

- **Conservative Approach:** This considers the total utilization limits of the main memory and CPU on the destination HN of all services, both interrupted and running. These values are then computed to verify that the HN has sufficient resources to receive the services to be migrated. This approach ensures that services cannot interfere with each other's resources and guarantees the desired QoE to all users, i.e., even when a user starts his service off on the HN, it will still have its resources reserved. However, it is possible that this solution will under-utilize the system if the services turned off are not reactivated.

- **Semi-Conservative Approach:** This only considers the limits of the consumption of memory and CPU in the destination HN from its running services, and based on these values, it determines whether this HN has sufficient resources to receive the to-be-migrated services. However, some issues in terms of resource shortfall can be experienced in cases where stopped services, i.e., running on top of containers, are reactivated while different migration procedures are performed.

- **Live Consumption Approach:** This considers the current consumption of memory and CPU in the destination HN of all running services. These values are then subtracted from the total available memory and the numbers of available CPU cores available to the applications to determine whether the HN has sufficient resources to host the to-be-migrated services. This approach aims to maximize the usage of resources in the destination HN. However, potential resource shortage can be experienced, requiring appropriate countermeasures to balance the system's load.

We developed these three approaches to guarantee both a trade-off between the execution time and the consistency of the resources in the MEC environment and allow a diversity of choice in the implementation phase.

**Monitoring Module:** Enables a live resource control on the source HN and triggers a live migration of services across multiple cloud domains. Several thresholds, e.g., CPU, Memory, and Disk, can be fixed to enable automatic migration. The automatic migrations process can be used to
attain the high availability needed for the different 5G and NSM use-cases.

**SDN Module:** MGO integrates an SDN module to interact with the SDN controller component defined earlier. This module provides SDN-based networking for standard communication and handling the traffic redirection, i.e., path redirection, after the live migration. As discussed earlier in Section 3.2, this module leverages the Intent-based logic of ONOS to enable proactive path redirection [77].

**Concurrent Access & Parallel Logic Module:** MGO guarantees concurrent access to computational resources. The concurrency control must guarantee the correct execution of conflicting requests, e.g., one user migrates service\(_X\); meanwhile, another tries to delete service\(_X\). Furthermore, if sufficient computational resources are available, the MGO can enable the parallel live migrations of several services.

**Partial Migration Module:** The partial migration module was designed to optimize the disk migration and was introduced earlier in Section 3.1. This module drastically reduces the total migration time while conserving a low downtime. Indeed, this module reduces both network and system overheads considerably.

**Retry Module for the efficiency of the live migration:**

The retry module is developed due to the need for high efficiency of the live migration. This module was designed to solve failure cases in the last iteration of the live iterative migration in the second phase, i.e., memory pages migration. As a solution, we elaborated a mechanism to detect the failure of the last step and trigger an automatic retry to ensure a highly efficient migration that meets the requirements of the 5G network.

### 3.3.2 Experimental Evaluation

We experimentally evaluated our proposed framework using one physical server. The server has 48 cores with VT-X support enabled, 256 GB of memory, 1 Gbps interconnection, and Ubuntu 16.04 LTS with the 4.4.0-77-generic kernel and QEMU-KVM installed. The server contains three VMs: VM 1 is the source HN from where the migration starts, VM 2 represents the target HN, and VM 3 hosts the SDN controller to handle the flow resumption. Each VM has 16 CPU cores, 16 GB of main memory, the same OS as the host node, and the container environment based on
stable versions of LXC and CRIU. It shall be noted that our current experiments aim to show the behavior of the service migration under MIRA! supervision.

The benchmark used in our evaluation is the streaming of a video through an NGINX HTTP server. Moreover, we compared the performance of MIRA! against the Basic Solution developed, in Section 3.1, based on an iterative migration using LXC and CRIU. Note that the Basic Solution does not support disk optimization methods introduced in sub-section 3.1.1. Figure 3.10 presents an overall comparison of the total migration time for both the Basic Solution and MIRA!. In this figure, we plot the Y-axis in seconds, and the X-axis shows each solution evaluated. Besides, the red bar represents the time it took to migrate the container/service between two HNs, and the blue bar shows the control overhead imposed by MIRA!.

![Figure 3.10. Total migration time comparison for the Basic Solution and MIRA!.

The mean total migration time of 10 executions for the Basic Solution was 28.788 s, with an STD of 0.873 s, a 95% CI of 0.658 s, and a CV of 0.030. Meanwhile, the total mean migration time of MIRA! was 22.627 s, which is composed of 20.374 s for the migration process itself and 2.254 s of MIRA! control time overhead. The STDs for both total migration time and control time are 2.108 s and 0.456 s, with 95% CI of 1.590 s and 0.633 s. It is important to note that with a CV of 0.202, the control time shows more significant variability during the experiments than the 0.103 of CV generated by the total migration time of MIRA!. The proposed framework reduced the total migration time by about 21.401%, although with a higher CV, mainly due to its control component, which shows that our proposal is a promising solution. The reduction in the total migration time was largely due to the developed partial migration module explained...
earlier in sub-section 3.1.1.

![Figure 3.11](image)

**Figure 3.11.** Impact of CPU variations on total migration time for parallel migrations.

Figure 3.11 shows how the number of parallel migrations impacts the total migration time when we vary the number of CPU cores. The Y-axis in seconds is the migration time, while the X-axis is the number of parallel migrations. In Figure 3.11, the blue curve shows the behavior of sequential migrations, the green curve presents the parallel migrations over two 4-core servers, and the red curve is the evaluation of MIRA!’s parallel migrations over a 16-core hardware testbed. As anticipated, the sequential migrations show linear behavior as we increase the number of migrations from one to four, which allows us to use this scenario as an upper bound on the total migration time. The 4-core testbed experiment manifested distinctive behavior since the total migration time had a slower increase rate for two migrations than for three and four. Moreover, the 16-core testbed results exhibit similar behavior to that seen in the 4-core case, albeit with the derivative changing after three parallel migrations. This behavior was the result of computational resource sharing between the containers and the MIRA! management system, which limits the number of parallel migrations the server can support.

### 3.4 Summary

This chapter introduced an NFV compliant framework architecture that handles both the known and unknown mobility patterns while optimizing
both disk copy and the memory page synchronization parts of the service migration. The proposed approach ensures the support of latency-sensitive applications through live migration operations. We evaluated the proposed solution using real testbed experiments. Our results showed that while the shared file system approach delivered the shortest migration time, it also imposed the highest downtime. Meanwhile, we obtained a more considerable migration time in the approaches without a shared file system because of the file system copy, which we did while the container was still running. We also showed that our tmpfs iterative migration approach achieved an average downtime of 1.111 s, an improvement of 61.039 percent compared to that reported by Yang [78], and 50 percent better than that reported by Machen et al. [79].

This chapter also presented a benchmark comparison between the ONOS CLI and the ONOS RESTful API from a northbound interface perspective. The conducted benchmark measured agility and scalability based on interface access method, type of Intent, and the number of installed Intents to determine best practices for assisting NSM. It was shown that while the RESTful interface is more flexible since it allows applications and services to request ONOS to update the network configuration directly, it has a greater overhead in comparison to the ONOS CLI, and consequently with any application directly linked with the ONOS core. Nonetheless, our experimental results showed that the ONOS RESTful interface delivers higher performance when creating more complex Intents.

We finally introduced a framework that combines both service migrations and SDN-enabled networks to enable live migration across datacenters belonging to multiple administrative domains. The proposed framework integrates mechanisms to handle resource unavailability, network addressing issues, and system control to guarantee short downtime and service continuity. We evaluated the framework on a practical testbed environment under different SDN configurations. The obtained results showed the efficiency of the proposed framework compared to our prior work.

This chapter answered RQ2 and the initial part of RQ3. Within the context of this dissertation, identifying and evaluating suitable technological requirements is a critical factor for creating a framework able to support NSM in 5G networks.
4. AI-based Network Slice Mobility Patterns in Next Generation Mobile Systems

In this chapter, which covers aspects more extensively described in Publications V and VII, a series of use cases related to emerging mobility patterns beyond the current NS definition are presented. Based on those use cases, different NSM patterns with their corresponding grouping methods and relevant mobility triggers are introduced in Section 4.1. Another goal of this chapter, through Section 4.2, is to provide an RL-based agent in charge of selecting proper system triggers for NSM patterns. The proposed approaches are also evaluated, compared, and validated in this section. Finally, the achievements are highlighted in Section 4.3, and conclusions are drawn.

4.1 Network Slice Mobility Patterns and Enabling Triggers

The NS paradigm unquestionably offers a powerful apparatus to support verticals’ services. According to this paradigm, a group of users is associated with dedicated computing, storage, and network resources tailored for a given vertical use case. The use of specialized resources and customized NFs in a slice implies that such resources are not available everywhere in the network but require careful resource allocation policies and control. Therefore, there is a strong need to extend the notion of mobility in network slices, which is traditionally limited to user devices or services without much concern for the availability of the combined resources needed for NS. Nevertheless, the introduction of mobility support in NS comes in a number of different variants or mobility patterns depending on how the slice is to be modified due to the changes in its service consumption. This will spawn new emerging use-cases that require many triggers characterizing the slice dynamics and a set of different grouping
attributes depending on services and network specifications. Considering these points, we:

- Present three different scenarios in which new mobility patterns are observed;
- Introduce various NSM patterns to manage and use NSs with their allocated resources optimally;
- Define several key enabling NSM triggers, and UEs grouping attributes to enable NSM use cases;
- Examine service migration capabilities for enabling NSM patterns.

### 4.1.1 Network Slice Mobility Use Cases

Network slices with mobility support can be customized for numerous use cases, such as to support autonomously moving equipment with mobility patterns and latency requirements that differ from a regular mobile phone. Examples include drones, robotic vehicles, or a fast-moving train carrying mobile users enjoying infotainment services. In this sub-section, we discuss use cases that clearly show the need for the proposed NSM paradigm, its triggers, and grouping methods that 5G and beyond networks should adopt. The first two use cases form a new mobility pattern not perceived before in the previous generations. They also express the perfect need for a mobile slice to follow the users, i.e., cars and UAVs, to ensure specific resource availability and continuous delivery during the mobility time. The last use case is proposed to illustrate another aspect of the NSM patterns related to partial mobility. This will be detailed in sub-section 4.1.4.

**Drone Traffic Control:**

UAVs define the perfect example for predictable paths and high mobility, showcasing the NSM use case [80]. We assume a UAV use case, in which the objective is to perform a self-swarming control test over two university campuses. We have ordered a customized slice for the drone experiment that offers connectivity to the drones and hosts virtual flight controllers for all the involved drones. The virtual flight controller is a software application that controls a corresponding drone and receives location information from the drone and radio usage information from the network [81–83]. The virtual flight controller application is instantiated
in a container. Each drone has its own instance of the controller, and the controllers need to cooperate to group the drones into swarms in an orderly manner, so they move as a group in the same direction [84]. In these experiments, we noticed two possibilities: the first one is that the swarms of drones can have a synchronized mobility pattern where the swarm moves like a tight group outside the current service area or a second one where the drones move in smaller groups or one by one out from the initial service area depending on how tight the swarm is. In both situations, the containerized flight controller must follow its drone to provide the required low latency and accurate control of the drones. This use case leads to the notion of migrating the slice as a whole, i.e., a full NSM pattern, or the slice gets split over two service areas.

**Autonomous Vehicles Support:**
One of the key verticals expected to benefit from the upcoming 5G systems is autonomous vehicles. Autonomous vehicles are expected to be served by mobile operators or car manufacturer-operated network slices. Passenger safety and the safety of pedestrians and other vehicles on the road demand high reliability and low-latency connectivity to be an integral part of a slice for autonomous vehicles. In addition to the support of low latency and high reliability, the slice must also have redundancy that can be implemented at the VNF- and connection-levels inside a slice or by a backup slice. The latter approach would lead to simpler overall service orchestration and configuration. When such a slice is relocated, the backup slice also needs to be modified or even moved. The NSM management mechanism must take into account the availability of backup connections and the redundancy of the VNFs of the slice when migrating the slice and its services. The backup slice should not be relocated at the same time as the primary slice to ensure the availability of the service.

**Rapidly Changing Video Streaming Need:**
In this use case, a city hall meeting is taking place in the form of a webinar. All the residents of the city have been invited due to the importance of the decisions to be made. The community has arranged its own slice for this mass meeting. The webinar is also attended by an audience on a train departing from the railway station of the city. The train leaves according to its schedule and soon enters a neighboring area, or a tunnel, that has much less capacity to offer. The city hall video streaming slice needs to be complemented with a new and temporal slice in the new service area.
the train is passing through. This new slice solves the bottleneck by accommodating a special Content Delivery Network (CDN) capacity with a video streamer that aggregates multiple separate unicast video streams into a limited number of shared streams that are distributed to the traveling audience of the webinar [85–87]. Once the train enters an area with better infrastructure, the purpose-built video CDN slice can be released, and the original slice can continue to serve the audience. This use case shows the need for slice splitting and slice merging, leading to the concept of slice breathing. Slice breathing will discussed in sub-section 4.1.4.

4.1.2 Key Enabling Network Slice Mobility Triggers

This step will identify, discuss, and present several key triggers to enable different NSM patterns. Triggers broadly relate to the users’ mobility, the availability of physical resources and network resources at the hosting edge cloud or federated cloud, resource efficiency utilization, service reliability, and security.

**Group Mobility Trigger:**
The Group Mobility Trigger (GMT) can be considered the main catalyst for NSM in a real-life environment. It could be applied to all use cases cited above: drones, autonomous cars, and video streaming. In this trigger, a group of users simultaneously moves from one location to another one. An example is passengers on board a metro or train moving simultaneously from one location to another. This is a video streaming use case of a highly moving entity, which requires a full NSM pattern. The signal strength can be measured by the users and reported back to the access points. These measurements can be correlated and acted upon already on the access points or deeper in the network, such as at the Mobility Management Entity (MME) in the Evolved Packet System (EPS), Access Management Function (AMF) in the 5G core, or at the life cycle manager controlling the slice. The entity in charge of correlating these measurement reports will pull the GMT that will, in turn, launch the process of NSM to follow the mobility of that particular group of users, using the same slice and its services that were reported by the measurements.

**Resource Availability Trigger:**
Edge clouds tend to have fewer resources than the centralized cloud. This includes the network, processing, and storage capacity. Due to the limited
resources at the edge, system-level resource consumption must be monitored carefully. Once the upper limit of allowed total used capacity is reached, a Resource Availability Trigger (RAT) will be generated with a parameter indicating that the highest allowed resource consumption level of a certain type of system resources is reached. This trigger is sent to the slice life-cycle management to initiate the migration of services from the highly-loaded edge cloud(s) towards the centralized cloud while keeping at the edge only delay-sensitive services. Alternatively, the highly-loaded edge cloud services could be migrated towards neighboring edge clouds if they have the necessary system-level resources available. Once the resource consumption level decreases below a given parametrized threshold, the edge cloud should send a RAT signal to indicate that there is room to accommodate more users and services for that particular type of resource.

Let us consider that the users are static, and the network resources are exhausted by the requested data from a given group of users. In this situation, a scale-out operation needs to be considered. However, the slice is limited by the available physical resources of the edge cloud and the number of different slice migration strategies that need to be considered with potential impacts on the service QoE. One possible trade-off is migrating some users or services to other edges or to the distant centralized cloud, even if that reduces their QoE.

**Reliability Trigger:**
The Reliability Trigger (RT) would be typically generated by the operation and maintenance protocols and supporting Operations, Administration, and Maintenance (OAM) systems of the access point or the edge cloud that monitor the robustness of the connectivity. In case of a major disaster, there would be an abrupt interruption of connectivity towards the users or neighboring edge nodes or even to the centralized cloud. Therefore, the services should be simultaneously evacuated from one location to another as long as there is still some capacity left for that. All less important actions should be deferred until the system snapshot has been replicated. This trigger will cause a full NSM pattern where all slices served by this access point or edge cloud node need to be migrated elsewhere to ensure all valuable data integrity.
Security Trigger:
Security is of vital importance for the networks since a compromised entity or service may result in considerable damage to the whole infrastructure. For instance, if a Denial of Service (DoS) attack occurs in a specific location “A”, the services should be then shifted from that location to a more secure one. An Intrusion Detection System (IDS) could send a Security Trigger (ST) to start an NSM for the compromised slice. When the IDS detects an anomaly in a slice, this trigger is sent to the orchestrator that initiates a migration process of that slice.

Request Overload Trigger:
Request Overload Trigger (ROT) is based on the number of simultaneous service requests stemming from groups of users requesting the services available in an existing slice. As a given number of requests is likely to overload the requested service and eventually be a trigger for the limited availability of resources, this overload trigger is sent before any hard resource limitations, effectively initiating smooth overload control and the potential redistribution of the service across neighboring nodes.

Service Consumption Trigger:
The resource consumption of individual services needs to be monitored. Depending on the type of service and its resource consumption, a trigger is generated if the service consumption is under or above predetermined levels. Note that the previously-discussed RAT deals with the aggregate system-level resources, whereas the Service Consumption Trigger (SCT) copes with the performance of a single service.

4.1.3 Grouping Attributes

The key mobility triggers for shifting network slices alone are not sufficient to efficiently manage NSM. In addition to triggers, we need means to group various relevant objects, like services, users, and NFs, so that the mobility of the slices serving these groups can be managed separately and efficiently. For this purpose, various grouping attributes to support NSM were investigated.

Grouping By User Subscription Type:
Grouping users by using their distinctive subscription types is one of the most straightforward grouping attributes. In 5G, this identifier is called a Subscription Permanent Identifier (SUPI), and it is globally unique
throughout the 3GPP system [88]. Another useful identifier in 5G is the Generic Public Subscription Identifier (GPSI): this identifies a 3GPP subscription for different data networks. With these subscription identifiers, it is possible to separate, for example, IoT users of a given service provider from mobile broadband users of the same or different providers or prepaid users from enterprise users. Because this grouping is very coarse, it will be performed in combination with other types of groupings.

**Grouping By Access Type:**
Users can be grouped by the type of access they use. Access types could be 3GPP, e.g., 4G, 5G, 5G small cell, or non-3GPP, e.g., WiFi and Wi-Max access multi-access. This grouping attribute is coarse and often needs to be complemented with other grouping attributes.

**Grouping By Network Slice Type:**
3GPP defines three standard types of slices:

- eMBB slice,
- URLLC slice,
- massive Internet of Things (mIoT) or mMTC slice.

A UE can be connected at the same time to multiple slices. A slice is associated with a slice identifier, called Network Slice Selection Assistance Information (NSSAI), containing information about the slice type [88]. When a UE connects to the 5G network, it will use this identifier to express which slice it wants to join. NSSAI is also used in binding services and 5G NFs to a particular slice. It is critical for an NSM pattern to identify that particular slice that will be modified or moved as it impacts multiple bindings between multiple entities. Unfortunately, NSSAI is unique only within one operator domain, which introduces additional complexity when a slice is moving across operator boundaries.

**Grouping By Service Area:**
Grouping users based on consumed resources only is insufficient due to service area and service availability limitations. A trivial example would be the case of a user “A” that is consuming an ultra-low latency service. If user “A” is a static user or his mobility pattern is restricted to a single service area, this user cannot be grouped with other users allowed to use the low latency service in other service areas.
Grouping By Access Characteristics:
Grouping users by their experienced access characteristics constitutes one of the essential groupings attributes to identify an NSM pattern. Access characteristics take into account the radio metrics of the access, throughput, and frequency of handovers. For example, in our third use case where passengers are traveling in a high-speed train and receive video service over a network slice, one useful grouping of the users would be based on users experiencing the same radio characteristics, same radio frequencies, same handover frequency, and using same access points in addition to receiving the same service.

Grouping By Geographical Location:
Geographical location is an obvious grouping attribute that applies to all the above-mentioned use cases. Grouping based on geographical location is often combined with the other grouping methods, particularly with the mobility pattern grouping. In geographical location-based grouping, users, services, or slices are classified based on their current location.

4.1.4 Network Slice Mobility Patterns

We classify NSM patterns into the following categories: i) full NSM; ii) partial NSM, which includes slice breathing, slice splitting, and slice merging; and iii) NSM optimizer, which contains a slice shrinking pattern. In the remainder of this section, each NSM pattern is described, highlighting its respective triggers. As discussed earlier in sub-section 4.1.2, NSM events are generally triggered by the mobility of a group of end-users, the availability of the needed resources in the cloud, or by the security aspects of the requested services.

Full Network Slice Mobility:
In this use case, we consider a group of users moving to a different location, e.g., a swarm of UAVs moving from the service area of an edge cloud 1 to the service area of another edge cloud 3, i.e., see Figure 4.1. This group mobility will trigger a service migration process of all services and associated resources used by this group. This leads to the notion of a full NSM pattern as all resources and the services of a slice are impacted by the mobility and are migrated from the original resource location to a new one. Several triggers may be used to indicate the need for this type of mobility. The most important ones are GMT, RAT, and RT.
CHAPTER 4. AI-BASED NETWORK SLICE MOBILITY PATTERNS IN NEXT GENERATION MOBILE SYSTEMS

Figure 4.1. Full Network Slice Mobility pattern.

Partial Network Slice Mobility:
The full NSM pattern is an expensive operation, but some mobility patterns can be supported with less tedious operations. In partial mobility, only some identified resources of a slice are to be migrated. Thus, there are two possible scenarios: either the network slice will be extended to allow more coverage, i.e., Slice Breathing, or a different network slice will be considered in the destination. For the latter, we consider two cases: i) either existing slices, and ii) newly-created slices. Both of these divergent cases yield two types of NSM: Slice Splitting and Slice Merging.

Slice Breathing: The use case depicted in Figure 4.2 refers to the case when the group of users “C” attached to the existing slice, i.e., slice 1, causes skewed resource consumption, and over-consumes a subset of the offered resources of slice 1. A slice breathing operation will be triggered, causing replication of the content of highly-loaded micro-services, e.g., containers, followed immediately by user redirection to this newly-created (sub-)slice to guarantee the seamless continuity of the services.

The slice breathing operation can be based on several triggers. Clearly, the GMT, RAT, ROT, and SCT can be used to initiate a slice breathing mobility pattern. To summarize, in the slice breathing operation, a slice is temporarily expanded by combining service replication [27–29] and service migration processes with the redirection of users to the expanded slice to match a sudden need for scaling a slice.

In the case of the infeasibility of slice breathing operations, slice splitting and slice merging are adopted depending on the situation.
Slice Splitting: In the slice splitting case, we opt to create a new slice for the upcoming group of users, as shown in Figure 4.3. This action will be permitted by performing inter-slice service mobility to ensure the availability of the service in the newly-added slice by exploiting the ability of inter-data-center multiple migrations.

The GMT, RAT, ST, ROT, and SCT will be part of the envisaged triggers for the slice splitting use case. However, for these NSM patterns, the system should consider the service consumption behavior of the end-users as a trigger to choose between slice splitting or slice merging. Effectively, if the requested service is not a delay-sensitive service, the system may opt for a slice splitting action, wait until creating the new slice, and start the mobility process by performing migration and replication actions.
**Slice Merging:** On the other hand, considering the availability of interoperability between two slice providers, i.e., two IaaS, the system may optimize the distribution of new containers by forming a set of slices and allowing inter-slice connectivity. This kind of NSM pattern could be beneficial in case of no network coverage from the initial slice provider. A multiple migrations process added to a replication process should be envisaged to enable this use case.

![Figure 4.4. Partial Network Slice Mobility pattern (Slice Merging).](image)

The slice merging mechanism presented in Figure 4.4 is a similar mechanism to slice splitting except for the creation phase of the slice because in this case ultra short-latency services will be considered, which means that we cannot tolerate the extra time for the slice creation. Due to this constraint, a slice merging mechanism is employed. In the case of different slice providers, an interoperability contract is intuitively assumed to be established between these different providers.

**Network Slice Mobility Optimizer:**

The NSM Optimizer mechanisms aim to avoid the waste of unused resources due to users’ mobility, reduced usage of services, and/or the end of the service usage.

**Slice Shrinking:** In Figure 4.5, we illustrate the case of a group of users that moved and are served by a different slice, i.e., either an existing one or a newly created one, in their new location, i.e., Edge Cloud 3 & 4. The remaining users are served by Edge Cloud 2, and resources used initially in Edge Cloud 1 will be released, resulting in the shrinkage of the original slice. Slice shrinking is a cleaning mechanism that should release all previously used resources that became obsolete during the migration.
After deciding which NSM pattern to execute, the system should run a control algorithm that determines the optimal number of virtualization instances and sets the slice shrinking rules. For instance, in Figure 4.5, we have two containers serving the groups of users “A” and “B”. The system can use the number of requests that container 1 or 2 can handle. If either of them can provide the required service with respect to the SLA terms negotiated before, it can take over the services provided by the other one, which will be ultimately turned off.

![Figure 4.5. Network Slice Mobility Optimizer (Slice Shrinking).](image)

Table 4.1 summarizes the relationships between the NSM patterns and the different mobility triggers. The slice shrinking pattern is not included in the table as it is supposed to be an automatic process activated after each NSM pattern to free resources that become unused after resource migration.

<table>
<thead>
<tr>
<th>Triggers / Network Slice Mobility patterns</th>
<th>Full NSM</th>
<th>Slice Splitting</th>
<th>Slice Merging</th>
<th>Slice Breathing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group Mobility</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Resource Availability</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Reliability</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Security</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Service Overload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Service Consumption</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 4.1.5 Evaluation Setup & Experimental Results

In the performance evaluation, parallel migrations were considered for a better understanding of the limits of the envisioned system. Our focus
on parallel migration arises from our perceived need to support multiple simultaneous migrations during a slice reconfiguration caused by user mobility across domains.

**Figure 4.6.** Network Slice Mobility testbed setup.

Figure 4.6 portrays the testbed environment envisioned to emulate the NSM patterns, targeting the third use case in sub-section 4.1.1. In this use case, a slice includes a set of streaming functions used for streaming video services. The testbed consists of two physical servers, in the form of two administrative IaaS domains, as depicted in Figure 4.6. Both servers run Ubuntu, i.e., server version, and have KVM as a hypervisor. Server 1 hosts two VMs; one acts as the source cloud for the NS, while the other VM acts as an SDN controller, i.e., ONOS, that manages the communications between the different servers and their clients. The second physical server, server 2, hosts only one VM that plays the role of the destination cloud. Both servers and all VMs are running Ubuntu 16.04 LTS with the 4.4.0-64-generic kernel with 4 CPU cores and 4 GB of memory. The connection between the servers is 1 Gbps. An additional host is used for accessing the testbed from an external network. In our environment setup, we use the container technology LXC as a container engine and CRIU as a migration utility. We created three containers; each of them was running an NGINX server to stream videos to three different clients.

The full NSM pattern was used to evaluate the performance of the containers because it is a computation-heavy scenario. The migration process triggers the migration of running instances in parallel from the source
cloud to the destination cloud. To automate the orchestration process, we built a framework to handle the parallel container migrations. We consider the RAT as a trigger to enable parallel migrations. We use a mix between the grouping by service area method and the grouping by geographical location method since our clients are static and are served from their respective video streaming servers. By leveraging the SDN paradigm, we created an overlay network on top of the physical network. Such an action allowed us to design an isolated network topology over multiple physical data centers and carry out inter-data-center or inter-IaaS parallel migrations.

In order to enable the full NSM pattern, the test is performed using the iterative migration approach introduced in Section 3.1. We adopt the pre-copy logic to perform live migration operations. Then, the SDN controller handles the path redirection phase, and finally, containers are restored to that target node, Section 3.3.

![Figure 4.7. Sequential migrations vs parallel migrations in Network Slice Mobility.](image)

Figure 4.7 shows the duration of 2, 3, and 4 sequential migrations for the same number of parallel migrations. The test is performed with identical container sizes to carry out fair tests while comparing the parallel migrations to the sequential one. In Figure 4.7, the X-axis represents the number of LXC migrated containers, while the Y-axis represents the migration time in seconds. For each bar, i.e., parallel and sequential migration, we plotted the 95% CI of the mean.

The mean total migration time for two sequential migrations was 40.712 s, and the 95% CI for this experiment was 2.001 s. For the parallel migrations, we obtained a mean total migration time of 24.302 s and 1.976 s as its 95% CI. Regarding the migration of three instances, we achieved a
mean total migration time of 60.174 s and 27.399 s, respectively, for both sequential and parallel migrations. Finally, on the four instances migrations scenario, the mean total migration time is 81.9131 s for the sequential migrations with 95% CI of 2.631 s, while these values are 43.427 s and 9.087 s, respectively, for the parallel migrations. The results obtained through this evaluation reveal that the parallel migrations for enabling a prompt NSM pattern are more efficient than the sequential migration strategy.

### 4.2 AI-based Triggers selection for Network Slice Mobility

Recent 5G trials have demonstrated the usefulness of the NS concept that delivers customizable services to new and under-serviced industry sectors. NSs are provisioned with dedicated resources for a certain purpose to meet the required SLAs [20]. However, the impact of user mobility on the optimal resource allocation within and between slices deserves more attention. Slices and their dedicated resources should be offered where the services are to be consumed to minimize network latency and associated other overheads and costs. To this end, sub-section 4.1.4 presented different NSM patterns that consider user mobility and service migration and combine them with the mobility of network resources of a slice. Precisely, sub-section 4.1.4 categorized NSM patterns based on end-users’ mobility, resource availability, service consumption, and security concerns. Hence, NSM patterns are classified into three principal categories: i) full slice mobility; ii) partial slice mobility, which includes slice breathing, slice splitting, and slice merging; and iii) slice mobility optimizer, which contains the slice shrinking pattern.

The dynamism of these networks establishes the triggers which serve as the main catalyst for NSM scenarios in real-life environments. Thus, the six tightly coupled mobility triggers are defined in sub-section 4.1.2 as enablers that facilitate smooth NSM patterns. The six triggers defined in sub-section 4.1.2 broadly relate to users’ mobility, the availability of physical and network resources, resource efficiency utilization, service reliability, and security. Nevertheless, since triggers are non-orthogonal and can overlap, the mobility action selection process can be complex and ambiguous. Meanwhile, AI techniques have been introduced to optimize the use of system resources, e.g., latency, bandwidth, RAM, processor, disk,
and I/O in [89–92] based on given policies. The increase in the availability of affordable hardware resources [93] for data processing paves the way for extensive use of ML techniques in both clouds and MEC [94–96]. Besides, standardization communities, e.g., ETSI, and 3GPP, expect that AI will be the inherent path of smart and responsive next-generation networks [88,97]. Hence, to automate the selection of triggers responsible for service and resource migration and NSM action selection, we:

- Introduce an NFV-MEC architecture that incorporates the proposed AI-based triggers selection agent and its modules to implement service and slice migration;
- Introduce a DRL-based agent allowing a fine-grained selection of mobility triggers that may instantiate slice and resource mobility actions;
- Illustrate the agent’s operational mechanisms and the design and implementation details of the two DRL-based algorithms constituting the proposed agent;
- Evaluate the efficiency of the proposed methods in a simulated environment and compare them in terms of training stability, learning time, and scalability for permitting a fine-grained trigger selection pattern within the 5G network.

### 4.2.1 Envisioned Architecture

Figure 4.8 depicts an overview of the envisioned architecture, incorporating an agent capable of autonomously selecting triggers and actions for allowing various NSM patterns. The envisioned architecture, introduced in Figure 4.8, is divided into two separate layers to efficiently enable the use cases defined in sub-section 4.1.1 and upcoming ones, mainly the Orchestration and MEC layers. This layering model helps manage applications by casting MEC in NFV paradigms and therefore complying with ETSI’s MEC and NFV standards [98]. Considering the MEC-NFV standards, both the Mobile Edge Platform (MEP) and MEC applications (MEC app) are VNFs. Therefore, the NFV domain elements hosted in the MEC layer, i.e., the VIM, NFVI, and VNFM, manage their life-cycle. The Mobile Edge Platform Management (MEPM) acts as Element Management (EM) in the NFV architecture, thus providing application management.
features to the MEP. The NFVO and the Mobile Edge Application Orches-
trator (MEAO), in the Orchestration layer, share service application in-
formation and the network service information in the MEC-NFV domain
to provide a reliable orchestration system. It should be noted that we
omitted the reference point details between MEC and NFV components
for simplicity.

The Slice Mobility Decision Maker (SMDM) agent is an additional plu-
gin to MEAO [99]. The main components of SMDM are the Request
Manager (RM), the Learning and Exploration (LE) module, the Trigger
Selector and Exploitation (TSE) module, and the DRL Algorithms Com-
parator (DAC) module. The SMDM agent interacts with the MEC layer
(i.e., the environment) through the RM module, retrieves states, selects
decisions such as scaling up/down diverse types of resources (e.g., RAMs,
CPUs, and disks, or migrating MEC apps) and receives rewards for its
decisions. In sub-section 4.2.3, we further explain the properties of these
states, actions, and rewards. Following ETSI-MEC-NFV directives,
the SMDM agent communicates with the Operation/Business Support Sys-
tems (OSS/BSS) in order to execute administrative and billing instruc-
tions. It also leverages the NFVO to command all migrations and scaling
operations between MEC hosts/nodes. In our proposed architecture, the
information between the SMDM agent and the environment, i.e., MEC
hosts, transits via the standardized interfaces of the MEAO and MEPM
elements.
To ensure a reliable system, we designed both a testing environment within the LE module and a real-world framework used for production through the TSE module. The testing environment simulates the impact of actions on the real production environment, implementing shared features to represent both states and actions. We deployed our testing environment following the principle and the conventions of OpenAI Gym [100]. In our setup, a set of simulated MEC hosts was created, then for each MEC host, a random number of simulated MEC apps was instantiated. Meanwhile, the production environment followed a master/slave architecture and depended on real working implementations using hypervisors and containers: LXC for virtualization, and ONOS as an SDN networking component for connectivity purposes [41]. The slaves, MEC hosts in the MEC layer, collect the local data using a MEC service dubbed Metric Collector (MC) in the MEP. Each MC service sends the collected data to its respective MEPM-V, which in turn shares its content with the MEAO. Finally, the master through the SMDM agent retrieves and processes the received data from the MEAO. Alternatively stated, as long as the SMDM agent is not adept, the RM module will be forwarding the received requests to the testing environment in the LE module. With this, we can avoid network disasters and user dissatisfaction resulting from suboptimal action selection during the learning process. This part will be further elaborated in sub-section 4.2.4. In what follows, detailed information on the collected data and features is provided.

### 4.2.2 Environment Description and Type of Collected Data

We ensure that the type of data and features collected from the real and simulated environment are comparable, thus allowing us to use the testing environment results for the production environment. On the other hand, based on Section 4.1, we know that the triggers are non-orthogonal and overlapping, thus complicating the decision process. Even so, related triggers can be grouped. RAT, SCT, and ROT triggers, introduced in the section above, can be used as components of an aggregated trigger dubbed “Service Load Triggers (SLT)” since they are all related to system resources. By refining and combining the triggers, the proposed agent, i.e., the SMDM, can form optimized policies regarding trigger selection for NSM. Therefore, as SLT is a set or a combination of tightly related triggers, i.e., RAT, SCT, and ROT, data needs to be collected from the un-
derlying MEC system to the SLT trigger based on its sub-components, as listed below:

Resource Availability Trigger (RAT): The RAT trigger deals with the aggregate system-level resources related to the under-lying nodes, i.e., MEC hosts, hosting virtualization instances, which were container-based MEC apps in our case. Thus, each node, i.e., MEC host, is sending information of CPU, Memory, and Disk information contained locally to the SMDM agent for processing. The details provided by the RAT trigger are the CPU, RAM, and DISK capacities of the MEC host as well as their current consumption. Upon receiving the information from all the MEC hosts constituting the environment, the SMDM agent in the orchestration layer computes the percentage of the used resources in each MEC host. Moreover, all computation operations are done within the orchestration layer to free the MEC hosts of any additional overhead.

Service Consumption Trigger (SCT) & Request Overload Trigger (ROT): Compared to the RAT trigger, the SCT and ROT triggers cope with the performance of a single service, i.e., internal resource consumptions of the MEC app. The main idea behind developing these triggers is to monitor the services themselves instead of watching only the MEC hosts, allowing better flexibility and exploring a more comprehensive range of new actions such as scale-up/down operations. The details of the triggers, i.e., SCT and ROT, are the CPU and RAM of each container-based MEC app and their current consumption. Moreover, while we can expand these details to cover different parameters, including the number of requests/MEC app, the disk details are missing because container-based virtualization uses the notion of file-system instead of volumes. As was the case for the previous trigger, each MEC host sends those local data to the orchestration layer for computation to reduce the computation overhead as much as possible.

Based on the updates and preliminary exploitable models conceived leveraging the LE module, the TSE module will enable fine-grained trigger selection depending on the encountered situation for NSM. To corroborate the obtained model, both the LE and TSE modules utilize the DAC module instructed to apply and select formal DRL algorithms. This module can select which DRL algorithm is suitable for accurate decisions and trigger selection through a verification and comparison method based on the generations of scenarios. In sub-sections 4.2.4 and 4.2.5, the proposed verification method of the SMDM agent is detailed and the results of the
method are presented.

4.2.3 System Model

The states, actions, and rewards are the main characteristics allowing an agent to solve complex tasks without having dedicated programs. Thus, this sub-section defines the state space, the action space, and the adequate reward function to establish a proper RL system.

**State Space:**
Following the reception of information from all MEC hosts and their respective MEC apps, i.e., container-based MEC app [101], from the MEAO, the SMDM agent aggregates the data to form the input state to be fed into a function approximator [102, 103], i.e., a DNN in our case.

Our environment is a widely distributed system made up of a set of $N$ MEC hosts. Each MEC host $n \in N$ is hosting a given number of MEC apps, which are containers, $c \in C$. The following is a definition of the variable to introduce the MEC app to MEC host mapping:

$$
\forall c \in C, \forall n \in N:
$$

$$ \mathcal{X}_{c,n} = \begin{cases} 
1 & \text{if a container-based MEC app } c \text{ is running on} \\
\text{top of the MEC host } n. & \\
0 & \text{Otherwise}
\end{cases} $$

To represent the state space, the system resources of each MEC host are taken into account. Each MEC host consists of an ordered list, with each element in that list describing the percentage of used resources on that node, such as the CPU, Memory, and Disk for the RAT trigger, and also details the Memory and CPU percentage usage of the container-based MEC apps to cover SCT and ROT triggers.

Using the percentage-based representation, we get a hundred levels of variations for each selected feature. Therefore, we represent each MEC host by $100^{\mathcal{F}_n}$, with $\mathcal{F}_n$ being the number of selected features. The number of features can be derived as follows:

$$
\forall n \in N, \mathcal{F}_n = f_n + \sum_{c \in C} f_c \times \mathcal{X}_{c,n} \quad (4.1)
$$

Where $f_n$ and $f_c$ denote the number of features in MEC hosts and their container-based MEC apps, respectively.
By aggregating MEC host entries, the number of states obtained is equal to:

$$S = \prod_{\forall n \in N} 100^{F_n}$$  \hspace{1cm} (4.2)

The detailed state-space formation is shown in Figure 4.9, and represented along with its relationship to triggers. We omitted both the integrated ETSI-MEC-NFV Model and the SMDM agent presented in Figure 4.8 to concentrate on the formation of the state. Our example consists of two MEC hosts: the MEC Host 1 in red and MEC Host 2 in blue. Each one of them is hosting a container-based MEC app, i.e., MEC app (VNF), in green, delivering services to end-users. Assuming that $f_n$ and $f_c$ features are CPU and RAM, both $f_n$ and $f_c$ are equal to two. Knowing that each MEC host has one MEC app, the $F_n$ for each MEC host is equal to four, producing an ordered list of length four to represent the SLT trigger.

Our state handling passes through four distinct steps to implement Equation 4.2. The first two steps are executed in the MEC layer, within MEC hosts. The remaining steps, i.e., steps 3 and 4, are done at the orchestration layer level. In step 1, each MC in a MEC host collects the data related to the MEC host and the MEC app. Step 2 splits and organizes the collected data into two separate ordered lists following the RAT and SCT/ROT definitions. It also shares the data with the SMDM agent,
as explained in detail in sub-section 4.2.2 in the environment description part. Being executed at the orchestration level, step 3 computes the percentage of the used resources in each MEC host then forms an ordered list based on the obtained $F_n$, which was four in this example. Finally, in the last step, i.e., step 4, our SMDM agent compiles the state to be used in the DAC module by aggregating each SLT component together. Note that the RAT components are always at first positions as the SLT length, four in this case, may vary due to the increasing/decreasing number of container-based MEC apps or migration operations. By doing so, we can guarantee that our representation of state-space will not be dependent on variations of MEC apps.

**Action Space:**

The state-space defined above allows us to obtain a state at each time-step. We need to define an action space to be able to transit from one state to another. In our model, the action space is represented by:

- no-action, which conserves the current resource distribution.
- migrate from a given source MEC host to a given target MEC host.
- scale up/down various resource types such as CPU and RAM.

The SMDM agent explores various potential states and their respective rewards using these actions. The number of available actions within the proposed model can be expressed as follows:

$$A = \sum_{\forall n \in N} \sum_{\forall c \in C} X_{c,n} \times (N + \Phi(c)R_c)$$

(4.3)

In which $\Phi(c)$ is a function that returns the number of authorized operations for a container-based MEC app $c \in C$ except for migration actions. The number of migration actions and no-actions is equal to the number of available MEC hosts “$N$”: that is, when the destination MEC host is equal to the source MEC host, it is a no-action and the remaining are all migration actions. For instance, $\Phi(c)$ may return the value “two” to express scale up and scale down operations. $R_c$, a scalar value, denotes the number of resource types used for each authorized operation returned by the function $\Phi(c)$. For instance, scaling up operations may be applied to CPU and RAM; thus, $R_c$ will be equal to two.
**Reward Function:**

The reward function combines information related to system resources and operation time. First, the operation time, which is the time for completing an action (like migration or scaling up/down RAM) is measured from the agent. In the simulation environment, the times for scaling-up and scaling-down operations are static. The scaling-down operation is slower as this action in practical implementation requires a waiting time before execution. The migration time in the simulation environment depends on the current disk size. Meanwhile, these values can be directly measured in a production environment. We then reverse it to obtain a progressive reward. The reward is inversely proportional to the operation time; the longer the operation time is, the lower the reward will be.

Regarding the system resources, we know that when the percentage of exploitation increases, the performances decrease. Thus, we follow the same logic as the operation time, and we invert the sum of all system resources, like the CPU and RAM of MEC hosts. Based on this explanation, the reward function $R$ will be:

$$
R = \frac{1}{T_\eta} + \sum_{\forall n \in N} \sum_{\forall j \in K} \frac{1}{g(n, j)} + \sum_{\forall c \in C} \sum_{\forall j \in K} \frac{1}{g(c, j)}
$$

(4.4)

Where $T_\eta$ represents the time of operation $\eta$ and $g(x, j)$ is a function that returns the percentage of a given resource $j$ in the set of resources $K$ belonging to MEC hosts or MEC apps, i.e., $x \in N$ or $x \in C$.

Finally, coefficients were added to time and resource usage to make one parameter more influential than the other. In our case, time was considered the main parameter.

$$
R = \delta \times (1/T_\eta) + \beta \times \left( \sum_{\forall n \in N} \sum_{\forall j \in K} \frac{1}{g(n, j)} + \sum_{\forall c \in C} \sum_{\forall j \in K} \frac{1}{g(c, j)} \right)
$$

(4.5)

**4.2.4 Design of the Slice Mobility Decision Maker Agent**

**Operational Mechanisms:**

In the following, the internal operational mechanisms of the SMDM agent and the design of its adopted RL algorithms are presented.
Equation 4.2 demonstrates that the large number of states generated by our problem formulation makes it intractable. Moreover, it has been shown by Mnih et al. [56] that RL methods struggle to find an optimal policy in a reasonable time when state space is considerably large. Therefore, in the SMDM agent, the DAC module integrates and uses a DNN to approximate the state-space in the case of value-based approaches. To be precise, our developed DAC module employs DQN, a DRL value-based technique, to obtain an optimal policy regarding trigger selections for NSM patterns [57].

However, Equation 4.3 shows that the number of actions grows linearly with the number of MEC hosts and MEC apps, therefore producing a large action space in large scale networks [64]. Consequently, the hybrid DRL method called A2C, which combines both value-based and policy-based approaches, is adopted by our DAC module [67]. In the A2C based approach, we took advantage of the DRL value-based method to measure the quality of the action while optimizing the behavior of the agent, leveraging a DRL policy-based method [68], i.e., determining the policy function $\pi$ without worrying about a value function. While it is acknowledged that hybrid methods can solve problems that value-based methods cannot, they usually converge on a local maximum rather than on the global optimum [66]. Therefore, we introduce a verification function that makes a selection between DQN and A2C in the DAC module. The main reason for comparing and introducing a value-based method (DQN) and a hybrid method (A2C) is to decide whether a value-based or hybrid algorithm family is most suitable for trigger selection in NSM. Both types of algorithms present two distinctive and divergent objectives. Once establishing which is more appropriate, we can extend the work to compare and benchmark other algorithms within the category.

Before describing both the DQN and the A2C algorithms and their hyperparameters, the pseudo-code detailing the SMDM agent’s functionalities in Algorithm 1 is provided. The proposed agent implements three main principles:

- A request-based control interface: The implemented algorithm depends on requests to be either in the training phase or in the exploitation phase of the models;
- The training phase: Begins with neural network initialization then allows our agent to learn triggers and actions for the optimal policy through time and requests;
• The exploitation phase: The agent selects actions and triggers for the optimal policy to be deployed.

Algorithm 1, called Core-SMDM (C-SMDM), serves to describe in detail the aforementioned principles. Initially, C-SMDM is reactive to requests, which means that once a request is received, the C-SMDM algorithm is executed. Each request contains the “input state, i.e., \( S \), and the “request number, i.e., \( req_n \). While the first parameter is fed into both the DQN and A2C algorithms or just one of them, depending on the situation, the second parameter is used to track request numbers and for initialization purposes. In the case of \( req_n \), the first-ever request marks the beginning of the training phase through the initialization of three variables “iteration”, “drl_type”, and “trained” in lines 1 to 5 respectively in C-SMDM, in Algorithm 1.

\[
\text{Algorithm 1: Core-Slice Mobility Decision Maker (C-SMDM)}
\]

\[
\begin{aligned}
\text{Input : } & \quad S: \text{Input states/features.} \\
& \quad req_n: \text{request number.} \\
\text{Output: } & \quad \text{decision: action to carry out} \\
& \quad \text{(migrate, scale up/down CPU/RAM/Disk).} \\
1 \quad \text{if } req_n == 1 \text{ then} \\
2 \quad \quad \text{iteration } \leftarrow 0; \\
3 \quad \quad \text{drl_type } \leftarrow \text{None}; \\
4 \quad \quad \text{trained } \leftarrow \text{False}; \\
5 \quad \text{end} \\
6 \quad \text{if trained } == \text{False} \text{ then} \\
7 \quad \quad \text{RM.route(LE);} \\
8 \quad \quad \text{LE.input}(S, \text{DQN, A2C}); \\
9 \quad \quad \text{DAC.train();} \\
10 \quad \quad \text{iteration } \leftarrow \text{iteration } + 1; \\
11 \quad \quad \text{if iteration } \geq M \text{ then} \\
12 \quad \quad \quad \text{DAC.generate_scenarios();} \\
13 \quad \quad \quad \text{drl_type } \leftarrow \text{DAC.compare(DQN, A2C);} \\
14 \quad \quad \quad \text{trained } \leftarrow \text{True}; \\
15 \quad \quad \text{end} \\
16 \quad \text{end} \\
17 \quad \text{else} \\
18 \quad \quad \text{RM.route(TSE);} \\
19 \quad \quad \text{TSE.input}(S, \text{drl_type}); \\
20 \quad \quad \text{decision } \leftarrow \text{DAC.deliver();} \\
21 \quad \text{end}
\end{aligned}
\]

As long as the variable “trained” is equal to “False”, the input features, i.e., \( S, req_n \), will be fed to the training phase directly, to C-SMDM line 6. Each state, \( S \), is routed to the LE module through the RM module, i.e.,
C-SMDM line 7. Then, the LE module will input the state, \( S \), for both the DQN and A2C algorithms in the DAC module; this will ensure the training of both algorithms. Note that the initialization step of the two algorithms and their neural networks by the DAC module is omitted from the C-SMDM for the sake of clarity. After that, in line 10 of C-SMDM, the variable “iteration” is incrementally increased by one for each request. If the number of iterations/requests is bigger than “M”, the DAC module generates a set of scenarios and compares both the DQN and the A2C algorithms. The objective of this step is to determine the best algorithm choice for selecting the right triggers to the NSM patterns by updating the variable “drl_type” with the selected algorithm, in lines 12 and 13. A complete evaluation and validation process is presented in sub-section 4.2.5 to emphasize the criticality of the verification and comparison step used in the DAC module. It should be emphasized that “M” is the number of iterations used during the training phase and is fixed to 10000 in the following section. Besides, we set the variable “trained” to “True” to allow our SMDM agent to switch to the production phase, i.e., the exploitation phase, starting from the next request, see C-SMDM line 14.

Once the variable “trained” is equal to “True”, the SMDM agent will use the RM module to route the requests to the TSE module with the state, \( S \), and the “drl_type” as parameters. Finally, the TSE module contacts the DAC module by requesting only the winning algorithm. This last step will deliver accurate decisions regarding trigger selection in the context of the NSM, in lines 17 and 21 in C-SMDM. Furthermore, each function/method precedes with the name of module name that executes it to improve the understanding of the core features of the proposed SMDM agent.

The following descriptions detail the functioning of DQN and A2C algorithms used by the DAC module during both the training and the exploitation phases.

**Deep Q-Networks:** Mnih et al. [56] designed and introduced DQN, a value-based algorithm where the deep network takes a state \( s_t \) as an input while following policy \( \pi \) and produces the Q-values for every action in the action space. The proposed DQN uses experience replay to break the correlation between subsequent time-steps and allows a stable learning curve [59, 60]. At each batch size, DQN computes the Temporal Difference (TD) error by taking the difference between Q-targets [104], i.e., the maximum possible value from next states \( s_{t+1} \), shown in Equation 4.6 as \( \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \), and the predicted Q-values, which are repre-
presented by $Q^\pi(s_t, a_t)$ in Equation 4.6. This process results in a well-known regression problem which requires the total error of the training data to be minimised. Hence, the function approximator, which is a neural network, learns useful behavior by adjusting its parameters through forwarding/back propagation [102,103]. In short, the update of each Q-value represented as follows in Equation 4.6, i.e., in the case of Q-Learning, will be replaced by the update of weights in Equation 4.7. In both Equations 4.6 and 4.7, $\alpha$ is the learning rate used to set the error acceptance in which a higher value tolerates a larger error by adjusting aggressively while a smaller one adjusts conservatively. In the same Equations, $\gamma$ is the discount rate that promotes or reduces the impact of the next action according to the defined value.

\[
Q^\pi(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha (r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t)) \tag{4.6}
\]

\[
\Delta \omega = \alpha [(r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \omega) - Q^\pi(s_t, a_t, \omega))] \nabla Q^\pi(s_t, a_t, \omega) \tag{4.7}
\]

**Advantage Actor-Critic:** To cope with the constraint related to the growth of the action space mentioned above, the use of a continuous or a pseudo-continuous action space aware algorithm is proposed, namely A2C. The A2C has two main networks, mainly the Actor and the Critic networks. Using the current weights of the network, the Actor observes the environment “$E$”, then selects a given action by outputting a probability distribution across the action space. After that, the Critic evaluates the quality of the selected action regarding both the current state $s_t$ and the next state $s_{t+1}$. Finally, at each time-step, the A2C algorithm computes the loss of Critic and Actor and updates network parameters [67].

**Design of Neural Networks:**
We build our DNN with the following specifications to realize the proposed states to actions mapping:

**DQN Hyper-parameters:** We use two neural networks, mainly the Q-network and the target Q-network, to predict the current Q-values and the next Q-values, respectively. We adopt the “$c$-Greedy” policy to allow fair exploration/exploitation repartition. For both Q-Networks, we adopted Adam optimizer for adjusting the parameters of the network [105]. The learning
rate is identical for both Q-Networks, i.e., $2 \cdot 10^{-5}$, while the discount factor $\gamma$ is 0.99. We update the main Q-Network after every 32 iterations, i.e., the batch size is 32, while we alter target Q-Network weights every four episodes. We use two fully-connected hidden layers, each of which has 64 Rectified Linear Unit (ReLU) activation functions [106]. The ReLU activation function, denoted by Equation 4.8, is linear for all positive values and zero for all negative values. Therefore, it offers computational simplicity and better convergence features compared to other activation functions. The input and output layers depend on the number of used features and their generated actions, which were both introduced in Equations 4.2 and 4.3. Since we are using the DQN algorithm, the selection of actions in the output layer depends on finding the maximal action value.

$$\text{ReLU}(z) = \begin{cases} 
  z & \text{if } z > 0 \\
  0 & \text{if } z \leq 0 
\end{cases}$$  \hspace{1cm} (4.8)

**A2C Hyper-parameters:** A2C uses two neural networks: one network is used by the Actor and the Critic to determine the probability of selecting a given action and its evaluation, while the second network is used only by the Critic to evaluate the next action. For both networks, we adopt the Adam optimizer to adjust the networks’ parameters while we use the value $2 \cdot 10^{-5}$ as a learning rate, with a discount factor $\gamma$ of 0.98. We use a similar representation of hidden layers for both networks; that is, we use two fully-connected hidden layers in which the number of units, the activation functions, is 64 and 256, respectively. For the hidden layers, we adopt the ReLU activation functions, introduced in Equation 4.8. For the output layer, given the fact that the Actor is building an action classifier, we use Softmax as an activation function [107]; see Equation 4.9. In this case, the output layer has a length equal to the number of actions $\mathcal{A}$ defined in Equation 4.3. The highest value of $\sigma(a_i)$ is selected from the set of action values. It is worth noticing that Critic networks have a unique output used to evaluate actions at various states.

$$\sigma(a_i) = \frac{\exp(a_i)}{\sum_{j=1}^{\mathcal{A}} \exp(a_j)}$$  \hspace{1cm} (4.9)

### 4.2.5 Experimental Evaluation

We developed a simulator using Python to evaluate our proposed agent
for trigger selection in NSM patterns. We use Pytorch [108] to define our DNN models, weights computations, i.e., forward/back propagations, and optimization operations. The proposed approach is studied in active learning, where the SMDM agent directly interacts with the environment without prior explicit knowledge, and the availability of known public or private data-sets. Thus, in the evaluation, data is generated by the simulation setup composed of discrete events. Each event represents an iteration in the learning process where the SMDM agent observes the current states, executes actions on the environment, and receives a reward depending on the newly observed state. We generate MEC hosts and MEC apps, along with their system requirements, i.e., CPU, RAM, and DISK, through a random process. The requirements of each MEC app and MEC host in terms of different resources follow a discrete uniform distribution over a dynamic interval indicated in the implementation files. Table 4.2 details the environmental simulation specifications used in the training process. In the following evaluations, we compare these two DRL-methods and analyze their performance to validate our model. It shall be noted that in our current experiments, we consider an SMDM variant that only uses DQN and another SMDM variant that only uses A2C to properly and separately evaluate each algorithm.

Table 4.2. Environmental simulation specifications.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEC hosts</td>
<td>3 – 10</td>
</tr>
<tr>
<td>MEC applications</td>
<td>3 – 100</td>
</tr>
<tr>
<td>MEC hosts CPU</td>
<td>64 – 128 (cores)</td>
</tr>
<tr>
<td>MEC applications CPU</td>
<td>1 – 4 (cores)</td>
</tr>
<tr>
<td>MEC hosts RAM</td>
<td>64G – 128G</td>
</tr>
<tr>
<td>MEC applications RAM</td>
<td>1G – 4G</td>
</tr>
<tr>
<td>MEC hosts DISK</td>
<td>128G – 512G</td>
</tr>
<tr>
<td>MEC applications DISK</td>
<td>512M – 4G</td>
</tr>
</tbody>
</table>

We start with the evaluation of the SMDM agent using the DQN based algorithm. In this experiment, 10000 episodes are run, with the resources for MEC host and MEC apps randomly changed in the underlying layer. Figure 4.10 presents the SMDM DQN-based agent training leveraging on a non-linear Q-value approximation based on DNN to express the state space. The SMDM agent also uses the replay memory to store training steps and reduce correlation. In Figure 4.10, the Y-axis represents the rewards, while the X-axis portrays the number of episodes in the training process. The 100-episodes average is shown in the same figure in orange. Even though the agent can learn useful policy behavior, a rapid change in
the value of the rewards is observed during the training of our simulated environment.

Figure 4.10. Rewards history for the SMDM agent using the DQN algorithm.

Next, we evaluate the SMDM agent based on an A2C algorithm approach in our second experimental scenario. Figure 4.11 illustrates the episodic cumulative reward as well as the same metric averaged over 100 episode iterations, shown in orange. The graph was drawn by considering applying our DRL agent using Actor-Critic, A2C, based on the TD(0) principle, which does not require waiting until the end of an episode to perform updates, and updates TD(0) in every time-step. The same representations are kept for both the X-axis and the Y-axis, and the number of training episodes was the same, at 10000. As an initial reflection, the agent-based on A2C can achieve a higher cumulative reward while delivering a more stable learning curve compared to that delivered by the previous approach based on DQN.

Figure 4.11. Rewards history for the SMDM agent using the A2C algorithm.

To have an accurate comparison of algorithms, we decided to plot the re-
The SMDM agent based on the DQN network and the one based on the A2C network are featured in Figure 4.12 for comparison purposes. The SMDM-DQN agent is represented with the orange and the red colors for the rewards and the average rewards, respectively. The rewards and the average rewards of the SMDM-A2C agent are illustrated in blue and green. As Figure 4.12 combines the results of both previous experiments, the axis representations are the same. It should be noted that the X-axis output was reduced to 4000 episodes to show the main differences in their respective behavior. The variations of the graphs after these 4000 episodes were almost identical for both algorithms. The results in Figure 4.12 show the efficiency of the A2C-based agent compared to the DQN-based agent in terms of average/cumulative rewards and learning stability.

In earlier experimental results, it was shown that our proposed approach based on the A2C agent achieves better performance than the DQN based solution. The comparison was further extended for more clarity. In Figure 4.13, the accuracy of DQN and A2C based agents is compared for different MEC hosts, MEC app deployments, and configurations: in fact, 3 MEC hosts and 30 MEC apps, 10 MEC hosts and 10 MEC apps, 10 MEC hosts and 30 MEC apps. Since the term “accuracy” is not used in the RL context, our DAC module, in its verification method, starts by generating customized scenarios, in lines 12 and 13 in C-SMDM. Then, it evaluates and compares both the DQN and A2C algorithms while trying to ensure “L” consecutive actions without failing. That is, it over-utilized or under-used MEC hosts/MEC app, a total of “B” times, consecutively. The scaling up/down and migration operations, represented by “L”, were
set to equal to “30” and “B” equal to “10”. In Figure 4.13, the agent using the DQN algorithm is indicated by a slashed red bar and the standard blue bar represents the agent exploiting the A2C algorithm. The X-axis represents the types of DRL algorithms used and the employed characteristics, while the Y-axis represents the percentage accuracy. For each agent, we also plotted the 95% CI of the mean. We find a mean accuracy of 98.8%, with a 95% CI of 0.07% for the 3MEC-30MECapps configuration concerning the DQN agent case. For the A2C based agent, we obtained a mean accuracy of 99.7% with 0.03% as a 95% CI. Regarding the configuration 10MEC-10MECapps, we achieved an accuracy of 86.53% and 100%, respectively, for both the DQN and A2C agents. Finally, with regard to the 10MEC-30MECapps scenario, the mean accuracy was 71.05% for the DQN-based agent with a 95% CI of 0.28%, while these values are 100% and 0%, respectively, for the agent leveraging A2C.

![Accuracy Analysis](image)

**Figure 4.13.** Accuracies comparison in case of different RL algorithms.

After selecting the A2C-based algorithm for the SMDM agent, the additional hyper-parameters susceptible to improve the policy of the agent were explored and evaluated. Figure 4.14 portrays the effect of different learning rates, i.e., hyper-parameters, on the convergence performance of the proposed agent to select adequate triggers within NSM patterns scope. In Figure 4.14, the Y-axis represents the average reward values with different types of learning rates, while the X-axis represents the same units as shown in the graph in Figure 4.12. The impact of six different values of learning rates in the Adam optimizer was considered. Three distinct categories/classes were established. By convention, the first class was labeled “global-optimal”, and includes the learning rates $2 \cdot 10^{-5}$ and $3 \cdot 10^{-5}$, shown in blue and orange, respectively. The second category is
the “near-optimal” one; it contains the learning rates $1 \cdot 10^{-5}$ in green and $1.25 \cdot 10^{-5}$ in red. Finally, the last class is composed of learning rates $1 \cdot 10^{-4}$ and $5 \cdot 10^{-5}$, in purple and brown, respectively, and is dubbed the “local-optimum”. According to these results, a learning rate that is either too large or too small may cause local convergence with the designed algorithm, which results in inefficient learning. It should be noted that although this graph is presented after Figures 4.10, 4.11, and 4.12, the value of the learning rate was set to $2 \cdot 10^{-5}$ based on this experimentation to ensure suitable hyper-parameter training features.

![Learning rate variations comparison for the SMDM-A2C agent.](image)

**Figure 4.14.** Learning rate variations comparison for the SMDM-A2C agent.

To further explore the effect of learning rates in convergence characteristics, we plot the training loss of the A2C network, i.e., DNN. In Figure 4.15, the blue curve denotes the loss variation when the learning rate is equal to $2 \cdot 10^{-5}$. The loss changes shown in orange represent the learning rate of $1 \cdot 10^{-5}$, while the loss of $1 \cdot 10^{-4}$ is illustrated in green. In other words, one curve was selected from each class, as defined in Figure 4.14, to ensure a fair comparison. The Y-axis portrays the variation of the non-normalized computation rate while the X-axis represents the number of episodes, as in Figures 4.12 and 4.14. Based on the obtained results in Figure 4.15, it was determined that when the learning rate is equal to $2 \cdot 10^{-5}$, i.e., the blue curve, the loss gradually decreases and stabilizes at a small value. Besides, the obtained loss is smaller than the remaining ones as it was shadowed from the final training episodes. However, as expected, the loss of the big learning rate, i.e., $1 \cdot 10^{-4}$, oscillated and did not decrease to learn an optimal behavior. The loss changes of the learning rate equal to $1 \cdot 10^{-5}$ were considered acceptable with some variations and
biases. This enhanced experiment confirmed that the selection of a suitable learning rate helps accelerate the training process: the blue curve is faster than the orange curve and counters the local optimum optimization trap [66].

**Figure 4.15.** Training Loss variations comparison for the SMDM-A2C agent.

**Result Discussion:**

It was clearly shown in Figures 4.10, 4.11, and 4.12 that the solution using A2C achieves better cumulative rewards while remaining stable during the training phase. Moreover, the DQN approach is unstable with a reduced average reward value, as indicated in Figure 4.10. Furthermore, the results in Figure 4.13 show that when we increase both the number of MEC hosts and MEC apps, the accuracy of DQN is compromised. Meanwhile, the A2C results indicate that this ML technique has a lower sensitivity to the number of MEC hosts and MEC apps. Thus, the A2C agent learns a useful policy while satisfying the users’ QoE and preventing SLA violations.

Figures 4.14 and 4.15 show the effect of tuning and selecting hyperparameters to the performance of an RL algorithm in particular and an ML approach in general. Besides, the learning rate selected for the evaluation is directly related to the convergence of the defined reward function, as given in Equation 4.5. Large and small learning rates represented by the class referred to as the “local optimum” may lead to either missing the global optimum or slowing the learning speed drastically.

Based on those observations and the obtained results, we can assert that the A2C algorithm outperforms the DQN approach when adapted to select efficient triggers for NSM patterns.
4.3 Summary

In this chapter, three different use cases illustrating new mobility patterns which are not supported by the current NS definition are presented. Then, NSM patterns with corresponding grouping methods and relevant mobility triggers were introduced. The proposed framework was validated using a realistic testbed. The obtained results demonstrated the dominance of the proposed parallel migrations when compared to the sequential migrations strategy. From the results, the presented key enabling triggers, and the grouping methods, it was concluded that a mechanism to select the right combination of techniques for efficiently enabling NSM is indispensable.

In this chapter, an agent for automating the trigger selection process for enabling NSM was introduced. To this end, a joint NFV MEC architecture able to host an AI-based agent and its modules was presented. Then, the design, modeling, and evaluation of the agent responsible for a fine-grained selection of system-based triggers for NSM patterns were provided. After highlighting the operational mechanisms and the hyperparameters details of the two DRL-based algorithms constituting the proposed agent, the proposed methods were validated by implementing a simulated testbed environment. Through this thorough analysis, the A2C-based approach was shown to be more efficient than the DQN-based solution in terms of training stability, learning time, and scalability.

By defining different NSM patterns, their grouping methods, and the associated triggers, RQs 4 and 5 were answered along with the remaining part of RQ3. In addition, the automation of the triggers selection process through the use of a DRL-based agent in NSM patterns answered RQ 7.
In this chapter, which is mainly based on Publications VI and VIII, various approaches are proposed to enable SFC migration and reduce network overhead engendered by different NSM patterns. Section 5.1 introduces and implements four distinct SFC migration schemes. This section practically assesses the proposed SFC migration strategies to determine which method is most suitable for supporting mobility in network slices. In Section 5.2, we provide an agent relying on two DRL algorithms to fine-tune bandwidth resource allocations while ensuring a fast and reliable NSM. Practical evaluations measure and compare both algorithms in terms of accuracy, stability, and user QoE. In Section 5.3, the main contributions are summarized and conclusions are drawn.

5.1 Service Function Chain Migration Strategies in 5G and Beyond Networks

Among the advantages of recent technological proposals, like NS, MEC, SDN, NFV, SFC, and Service Based Architecture (SBA), the configurability and the ability to deliver, ensure, or even guarantee network latency requirements are perhaps the most compelling. However, since users nowadays are everything except motionless, the users tend to move quite a distance from the original MEC node where their services started running. There is a serious lack of flexibility in these models. Moreover, the types of NF services, forming NS entities in the Data Network (DN), are expected to follow a micro-service architecture. Therefore, a single NS based on NFs on top of micro-services may expand over multiple MEC nodes, which introduces the subsequent new issues:

- The management of instances on different MECs instead of one com-
pared to legacy approaches;

- Service continuity challenges related to ensuring links between the instances forming distributed MEC applications, and also to links associated with end-users;

- The substantial network and system overhead generated by a highly distributed design.

Motivated by the deficiency of the current NS to support user mobility, and assuming that all these issues are under the umbrella of SFC migrations that constitute the underlying enabling technology for allowing NSM, we aim to achieve the following:

- Design four practical SFC migration strategies to support NSM in the DN part from the proposed combined architecture of 3GPP and ETSI;

- Spell out the implementation details and key technological prerequisites;

- Evaluate the proposed strategies while considering different criteria for validating the new suggested type of migrations.

### 5.1.1 Service Function Chain Migration Strategies

We initially present all the envisaged SFC migration scenarios. Then, we describe the synchronization processes of NFs migration. Finally, we introduce the advantages and constraints of different SFC strategies as a first step to identify the approach that meets the low latency requirements of the 5G network.

**Asynchronous State-full Service Function Chain Migration:**

In this type of SFC migration, we start unsupervised live migrations for each NF of the SFC, and as the last live migration ends, we finish the SFC migration. Then, we can re-establish the NFs’ network connectivity. We use this scenario as a worst-case upper bound to evaluate the computational resources (the CPU, RAM, and DISK) and communication network resources (the delay and bandwidth consumption) for the SFC migration.
**Synchronized State-full Service Function Chain Migration:**
The first approach is considered a trivial solution that may consume all types of available resources; thus, we introduce the synchronized SFC migration. The well-known live migration process usually takes four steps, disk copy, non-blocking memory copy (Pre-Dump actions in CRIU [109]), final blocking memory copy (Dump action in CRIU), and restore. While we can do the first two steps without stopping the virtualized instances, the third step must freeze containers until they are restored in the final step later. Thus, the aim of a synchronized SFC migration approach is to efficiently control each step separately, as this fine-grained control reduces the overall system resource consumption. While we acknowledge that different strategies can be employed to eliminate the system overhead caused by multiple coordinated and parallel migration processes, we selected two strategies to be presented. For both strategies, we consider an SFC with two NFs:

**Synchronized Wait-For-Me Strategy:** In this strategy, we allow the first and second steps of the migration process to run in parallel, and we have a barrier, i.e., point of synchronization, just before the final memory blocking action, the Dump. Then, both instances have to wait to continue their migration process. We can observe the benefits of this approach in scenarios with plenty of network resources. However, since the size of virtualization instances is rarely the same or even equivalent, the first instance reaching the memory blocking phase may have to wait an extended period, resulting in an increased downtime period as memory pages could be updated while waiting for other virtualization instances. Also, the CPU, in that case, can be exhausted since the actions are in parallel.

**Synchronized Round-Robin Strategy:** The aim of the round-robin strategy is to reduce the migration CPU load. We achieve this through grouping by phase the steps of an SFC migration and then executing them in order. This approach consumes fewer system resources caused by an SFC migration, albeit at a significantly higher total migration time, since all actions are performed sequentially. In this case as well, the SFC migration strategy uses all available network resources.

**Network-aware Service Function Chain Migration:**

5G networks are expected to support various URLLC services, which require strict delay constraints. However, none of the previous approaches
can guarantee these prerequisites because of the randomized way for handling SFC migration when it comes to network resources. Indeed, because of the massive number of applications capable of following and serving users, all network resources among MECs can be compromised by allowing various migrations at the same time. We propose the network-aware SFC migration to address these requirements, as its purpose is to refine the network usage, reduce overhead, and enable a better QoE for the user. Our network-aware SFC migration triggers low-consumption migration operations across the networks by controlling the network’s bandwidth. Initially, we gather all the available information on bandwidth and latency between each MEC node pair, thus obtaining global knowledge of the distributed infrastructure. Then, after each migration decision is taken, given network resources are reserved to allow the migration and ensure better usage of the global bandwidth. Finally, our network-aware solution releases the used resources as it completes SFC migrations. Note that we reserve bandwidth for each SFC migration based on the last iteration that stops the container. Thus, if the reserved bandwidth offers a downtime transfer similar to when having the full utilization of the bandwidth, then the reservation limit is set.

5.1.2 Implementation Details and Key Technological Prerequisites

As stated above, enabling the migration of multiple independent virtual instances can be relatively easy compared to migrating several instances with a close relationship, i.e., SFC. Initially, before enabling an SFC migration, creating and forming an SFC is the starting point of interest. As a testing SFC implementation is unavailable, we design a service chain for a video streaming application. In this SFC, each video passes through an intermediate traffic redirection node, built on top of an LXC container, and has two SDN-enabled network interfaces. This redirection instance acts as a turnaround node in the network where all the traffic should go through in both directions, i.e., from the video server to the client and vice versa, as depicted in Figure 5.1.

As stated before, our solution considers using the SDN paradigm; thus, OVS switches are configured using the ONOS SDN controller, where each node in the SFC has its own switch. The only requirement for enabling the proposed implementation is the availability of the virtualization software to deploy LXCs container engine and programmable OVS switches. We
use our previous proposed solution, presented in Section 3.3, to allow isolation between incoming/out-coming traffic. To implement the turnaround logic in the Dummy-LXC host, a Bridge is created inside the Dummy-LXC container, i.e., the red Bridge in Figure 5.1. Two outer network interfaces, veth0 and veth1, are plugged in the outside OVS, i.e., the blue one in the same figure, while inner network interfaces named eth0 and eth1 are attached to the red Bridge in their turn.

5.1.3 Experimental Evaluation

We experimentally evaluated our proposed SFC migration schemes using one physical server. The server has 48 cores with VT-X support enabled, 256 GB of memory, 1 Gbps interconnection, and Ubuntu 16.04 LTS with the 4.4.0-77-generic kernel and QEMU-KVM installed. Two virtualized computer nodes, i.e., VMs, were used to evaluate the proposed implementation, each one representing a different MEC node, i.e., a DN node in ETSI and 3GPP proposals. The first VM is acting as a source DN, and the latter VM represents the target DN. Each DN uses Ubuntu 16.04 LTS with the 4.4.0-64-generic kernel and has 16 virtual core CPUs and 32 GB of main memory. The container environment was set up using LXC 2.8 and CRIU 3.11. Note that two additional hosts were used for the management plane. The first host acts as an SDN controller, i.e., ONOS, that manages the communications between the different DNs. Simulta-
neously, the second host serves as a global orchestrator for handling the life-cycle of SFCs from the creation phase until the migration or the deletion stage.

We start by evaluating the asynchronous SFC migration strategy under diverse network bandwidth limitations to select the most appropriate bandwidth limit for reducing the overhead of SFC migration. In that evaluation, both the downtime and the total migration time will be analyzed and discussed. Finally, based on satisfactory bandwidth usage, a CPU consumption analysis is presented to compare all the approaches introduced and detailed earlier in sub-section 5.1.1.

For each SFC migration scheme, we conducted a set of experiments to evaluate the downtime and total time under various network configurations and CPUs’ load. Each experiment was repeated ten times. The evaluation of the SFC consists of a video server streamer offering videos on demand to clients passing through an intermediate node dubbed Dummy-LXC that forms our second virtualization instance to be migrated when the SFC migration is triggered. The Dummy-LXC and video server nodes sizes are equal to 470 MB and 573 MB, respectively.

**Downtime Analysis:**

![Downtime Analysis](image)

**Figure 5.2.** Downtime comparison for the asynchronous SFC migration strategy under diverse network configurations.

Figure 5.2 depicts the induced downtime under diverse network configurations. The main purpose of this experiment is to optimally exploit network resources and avoid breaking down the whole network infrastructure. This experiment outputs the mean downtime results and their STD, 95% CI, and CV for both elements constituting our developed service chain considering various bandwidth values as part of defining the
most suitable network configuration. Detailed values are presented in Table 5.1. It is noticed that we used the asynchronous SFC migration strategy to compute a reasonable bandwidth limit since this strategy represents the worst-case approach due to the absence of control in that SFC migration scheme. Due to the additional copies of the network connection status, the video container has a higher downtime than the Dummy-LXC container.

Table 5.1. Downtime comparison in case of different bandwidth values.

<table>
<thead>
<tr>
<th>Bandwidth (asynchronous SFC mig.)</th>
<th>Mean Time (s)</th>
<th>STD</th>
<th>95% CI</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy-LXC 0.3 MB</td>
<td>2.674</td>
<td>0.075</td>
<td>0.056</td>
<td>0.028</td>
</tr>
<tr>
<td>Video server 0.3 MB</td>
<td>4.397</td>
<td>0.076</td>
<td>0.057</td>
<td>0.017</td>
</tr>
<tr>
<td>Dummy-LXC 3 GB</td>
<td>1.189</td>
<td>0.049</td>
<td>0.037</td>
<td>0.041</td>
</tr>
<tr>
<td>Video server 3 GB</td>
<td>1.429</td>
<td>0.047</td>
<td>0.035</td>
<td>0.033</td>
</tr>
<tr>
<td>Dummy-LXC 2 MB</td>
<td>1.222</td>
<td>0.066</td>
<td>0.05</td>
<td>0.054</td>
</tr>
<tr>
<td>Video server 2 MB</td>
<td>1.571</td>
<td>0.056</td>
<td>0.042</td>
<td>0.036</td>
</tr>
</tbody>
</table>

Meanwhile, the full bandwidth, 3 GBps, represents the maximum available bandwidth between two DNs, as represented by the second error bar in Figure 5.2. The maximum bandwidth value was set using the IPerf tool [110], measures were taken ten times, and the collected values were averaged to obtain the mean bandwidth. This case should deliver the best results in terms of downtime and total migration time when fully exploited by the instances forming the migrated SFC. However, limiting the bandwidth to SFC migration processes in 5G networks will allow better exploitation of network resources in case of a massive number of migrations. Choosing the correct value is a challenging process, as a low bandwidth can increase both the total migration time and the downtime causing damage to the migration process. In contrast, an overestimated threshold will waste network resources in vain. In Figure 5.2, we selected three bandwidth values for testing downtime efficiency. The full bandwidth usage is used as a reference and, at the same time, it is considered an overestimated value since it is the bigger value and the one offering the best results in case of lack of overhead. In fact, 0.3 MBps, shown in the first error bar in Figure 5.2, is the underestimated value, and 2 MBps value is the satisfactory value, indicated by error bar number three in Figure 5.2. It should be pointed out that the 2 MBps value was obtained by trying many bandwidth values with one constraint in mind, which is having similar results to the full bandwidth utilization. Based on Figure 5.2 and Table 5.1, we can conclude that the downtime for the asynchronous SFC migration strategy is quite similar for both the full bandwidth and
the 2 MBps migration bandwidth. The obtained value represents a reduction of 99.93% from the initially provided bandwidth without significantly affecting downtime results. However, selecting the 0.3 MBps value will increase the downtime by a factor of three compared to the two previous values, 2 MBps and 3 BGps.

**Total Time Analysis:**

![Figure 5.3. Total migration time experienced for the asynchronous SFC migration strategy under diverse network configurations.](image)

To strengthen our assumptions related to limiting network resources so that more efficient SFC migration schemes will be admitted, we extended our evaluation to cover the total migration time under different bandwidth configurations. We addressed this evaluation using the same experimental scenarios of this section and plot the results in Figure 5.3 for the asynchronous SFC migration strategy. In Figure 5.3, the Y-axis is in seconds, and for each bar, we also plotted the 95% CI of the mean. The mean total migration time, the STD, the 95% CI, and the CV for SFCs under different network configurations are presented in Table 5.2. As expected, the full bandwidth, 3 GBps (represented by the second error bar in Figure 5.3) and the 2 MBps (the last error bar in Figure 5.3) scenarios were quite similar, while the migration time for the 0.3 MBps case was four times longer than the expected value.

**Table 5.2. Total time comparison in case of different bandwidth values.**

<table>
<thead>
<tr>
<th>Bandwidth (asynchronous SFC mig.)</th>
<th>Mean Time (s)</th>
<th>STD</th>
<th>95% CI</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy-LXC 0.3 MB</td>
<td>16.296</td>
<td>0.667</td>
<td>0.503</td>
<td>0.041</td>
</tr>
<tr>
<td>Video server 0.3 MB</td>
<td>42.658</td>
<td>0.742</td>
<td>0.56</td>
<td>0.017</td>
</tr>
<tr>
<td>Dummy-LXC 3 GB</td>
<td>11.833</td>
<td>1.891</td>
<td>1.426</td>
<td>0.16</td>
</tr>
<tr>
<td>Video server 3 GB</td>
<td>12.661</td>
<td>1.273</td>
<td>0.96</td>
<td>0.1</td>
</tr>
<tr>
<td>Dummy-LXC 2 MB</td>
<td>11.922</td>
<td>1.28</td>
<td>0.965</td>
<td>0.107</td>
</tr>
<tr>
<td>Video server 2 MB</td>
<td>13.842</td>
<td>0.346</td>
<td>0.26</td>
<td>0.025</td>
</tr>
</tbody>
</table>
From these results, we can observe that the video-streaming container takes longer than the Dummy-LXC one for all bandwidth configurations. This additional time is logical as initially, the video-server has a bigger size than the Dummy-LXC. Furthermore, for the video-server container, the longer migration time compared with the Dummy-LXC one is due to the greater number of memory pages being copied. Knowing that the SFC consists of the Video Server and Dummy-LXC instances, we conclude that the overall total migration time will increase. However, other SFC migration strategies are considered and investigated in terms of CPU load for a better approach in the next sub-section.

**CPU Consumption Analysis:**

A CPU consumption analysis was performed to provide a better understanding of all the proposed SFC migration strategies. We set the bandwidth to 2 MBps based on the two previous analyses. Figure 5.4 illustrates the variation of CPU loads following three types of SFC migration schemes, mainly asynchronous, synchronized (wait-for-me), and synchronized (round-robin) SFCs migrations. In Figure 5.4, the Y-axis represents the percentage of the CPU load in the source DN node, and the X-axis represents a 100 second sampling time during which the SFC migrations occur.

![Figure 5.4. CPU consumption analysis in case of different SFC migration strategies.](image)

For the asynchronous SFC migration strategy, the red line represents the CPU load variation during the migration process while the grey and blue lines represent synchronized (wait-for-me) and synchronized (round-robin) SFC strategies. In Figure 5.4, before 20 seconds and posterior to 75 seconds periods outline before starting the SFC migration and after achieving the SFC migration, respectively. In the range between 21
and 74 seconds, while the asynchronous SFC migration strategy has the fastest migration time (from 35 to 45 seconds), it induces the highest CPU load. The migration time can be verified leveraging the previous total time graph in Figure 5.3 and in Table 5.2. The synchronized (wait-for-me) SFC migration strategy is shown to be the symmetrical approach as it stresses the CPU considerably while not consuming a lot of time during the SFC migration, as indicated in grey in Figure 5.4. This strategy takes 2 to 3 extra seconds longer than the asynchronous approach. This additional time is due to the increase in the downtime as a consequence of waiting for the second container to reach the last iteration phase. It should be mentioned that this time can be even longer when increasing the number of SFC components, especially if they have different disk sizes. Finally, the round-robin synchronized approach consumes less CPU power than all other strategies. It is quite similar to the sequential migration scheme, except that the round-robin policy is applied within the decomposed parts of the migration between various SFC components. However, this approach takes longer than previous strategies in terms of total migration time.

**Results Discussion:**

Based on the observations gathered from the previous sub-sections related to the downtime, total migration time, and CPU loads, it is not possible to decide on a clear winner in terms of performance. Thus, the SFC migration strategy must be selected based on the motion of the user, the requirements of the application, and the resources of the MEC nodes. Furthermore, the network-aware SFC migration strategy is selected to support the synchronized (wait-for-me), synchronized (round-robin), and the asynchronous SFC migration strategies considering the delicacy of 5G networks in terms of the number of users and available resources, that is network or system resources. For instance, a combination of the round-robin approach and the network-aware SFC strategies is adopted when the users' path is known, and we can proactively plan their trajectories. This will reduce the overhead of CPUs and minimize network resource wastage. While the synchronized wait-for-me strategy could be exploited with the network-aware SFC strategy to handle applications that do not require ultra-low latency as the wait-for-me approach does not guarantee the lowest downtime.
5.2 AI-based Network-aware Service Function Chain Migration in 5G and Beyond Networks

To enable different NSM patterns defined in Section 4.1, reduce system overhead, and allow low communication latency, we have proposed SFC migration approaches for 5G networks in Section 5.1. SFC migration, through the use of the SDN and NFV paradigms, steers the network traffic and flows across multiple physical and logical infrastructures while ensuring low-latency communication by following end-users motions as in Sections 4.1 and 5.1. Live service migration processes are well-known to be problematic in inter-cloud settings, and the use of chained microservices, i.e., SFC, is presenting its own challenges due to the dependency of the chained micro-services. Moreover, in 5G and beyond networks, it is expected that the number of URLLC type services that require strict delay constraints will increase as 5G networks become more widely deployed [111]. This emphasizes the importance of careful management of the end-to-end service delivery. To summarize, end-user mobility events may trigger a large number of application migrations concurrently, and therefore exhausting the network resources shared between the distributed edge clouds and their servers.

Recently, both in the academic community and in industry, interest in ML methods has increased. ML methods are a subarea of AI [96, 112]. The effect of the use of ML methods on mobile networking has also been evaluated [94]. Furthermore, ML will automatically perform corrective re-configurations to the infrastructure, ensuring the availability of the network [91]. ML techniques also involve the use of efficient policies, leading to the optimization of the system resources. This enables the efficient use of critical network and service resources, like latency and bandwidth, and system resources, like RAM, CPU, DISK, and I/O, [89]. ML techniques will be part of the networking and communication area, as has been indicated in different research projects, proposals, and white papers [14, 15]. Besides, AI/ML will act as a support for enabling smarter and more responsive generation of networks while maintaining the currently proposed architectures by the standards community, e.g., ETSI and 3GPP [88, 97].

Motivated by the limitations mentioned above, and the complex challenges that the current 5G service delivery faces in fulfilling inter-cloud
bandwidth constraints, including the stochastic (non-deterministic) use of networking resources during mobility events, and by the recent advances that ML techniques bring to edge computing and next-generation networking, we intend to achieve the following:

- Introduce and define an architecture that hosts the proposed network-aware agent and its constituent elements;

- Model and design a DRL-based agent capable of handling bandwidth allocation, refine the network usage to reduce the overhead and allow better user QoE;

- Present the internal operational mechanisms of our network agent, neural network architectures used by the two different DRL-based algorithms constituting our network agent, and their hyper-parameter values;

- Evaluate the proposed agent under different configurations and in real-world deployments, and determine the most suitable DRL algorithm/approach to enable an optimized SFC migration pattern within the 5G network.

5.2.1 Proposed Architecture

To ensure a reliable networking system and the availability of critical network resources, we adopt a conventional three-layer architecture widely used for representing 5G and beyond network systems in the previous chapter. The proposed system in Figure 5.5 follows ETSI-NFV standards. The MEC layer, in the defined system, is controlled through the interaction between the components of the Orchestration layer and the elements constituting the NFV architecture [99]. The Orchestration layer is hosted separately and communicates with the NFV domain through NFVO to emit corrective decisions and actions. We explicitly omit several Orchestration layer components to focus on the Smart Network-Aware (SN-A) agent that is supposed to fine-tune the bandwidth allocation process.

The Request Handler (RH) module offers to the SN-A agent a technology-agnostic abstraction to access MEC-layer entities, i.e., public or private cloud platform. Therefore, the SN-A agent retrieves states, outputs decisions of bandwidth values accordingly, and receives rewards for each decision. In addition, the SN-A agent also receives administrative instructions from the OSS/BSS components as defined in the ETSI-NFV model.
Figure 5.5. Framework architecture for ML-based orchestration and allocation of bandwidth resources.

The RH module must ensure reliable communication and synchronization between the SN-A agent and the MEC layer. It can achieve this through a message broker functionality, like RabbitMQ, or a standardized API.

Our agent SN-A was designed under the assumption that any processes using file transfer and synchronization tools, like rsync, exploit all the available bandwidth. Once other system processes start their network transfer operations, either migrations or application data traffic transfers, the bandwidth is shared among them using the best-effort policy [113]. Thus, we note that as we increase the number of concurrent migrations, the time to complete any individual migration will increase due to resource contention. Indeed, if the number of concurrent transfers is big enough, the migration times will become too large since none of them can be completed within a reasonable time.

Following these assumptions and knowing that the transfers of the disk and memory pages are the main steps of any live migration process, i.e., Sections 2.1, 3.1, and 3.3, we conclude that searching for an acceptable network bandwidth limit has to consider the following:

- The heterogeneity of application size;
- The content of the virtualized instances, i.e., containers and VMs;
- The types of migration selected, i.e., SFC or single live migrations.

Note that we have shown in Sections 3.1 and 5.1 that SFC migration data differs from single or simple live migration data.

In other words, colossal action space is required. These conditions make
the action selection a non-trivial, non-deterministic, and exhaustive procedure. Consequently, we consider employing DRL techniques to bypass the brute force search method, as introduced earlier in Section 5.1, or an uncontrolled migration process. Both cases have a detrimental effect on the QoS.

Nevertheless, both DRL and RL techniques are based on trial-and-error processes and therefore cannot be directly integrated with production environments, as some tried actions may worsen the already achieved performance [92]. We address this issue by integrating a Training and Exploration (TE) module responsible for creating identical digital twin environments used for the training phase into the SN-A agent.

Initially, the TE module, through the RH module, collects the available bandwidth capacity shared between each MEC node pairs to acquire information about the distributed edge infrastructure. We use a client/server-based IPerf test integrated with the TE module in this scouting stage. This step is a reconnaissance phase that generates most of the network information we use as an upper-bound for selecting bandwidth actions [110]. Then, after each migration decision in the test environment, the TE module reserves the network resources to successfully complete the SFC migration operations while improving the global bandwidth utilization. Finally, we release the used resources whenever migrations are completed. Note that we use a practical implementation of the SFC migration schemes presented in Section 5.1, which, in addition to ensuring service migration, guarantees the predetermined order of SFC components and their respective network and system dependencies. The presented process allows the SN-A agent to learn how to attribute optimal/near-optimal bandwidth values over time through the TE module. It should be noted that we can replicate these offline trial and error achievements in other environments, like 5G networks, since the training and the testing phases share the same input features and output decisions.

Once obtaining preliminary results, the TE module shares its learned model with the Bandwidth Allocator and Exploitation (BAE) module to minimize network resource utilization. Therefore, we can validate the usability of the results by comparing them to their handcrafted counterpart, as defined in sub-section 5.1.1. The SN-A agent compares the learned policies against the handcrafted values: if both downtime and total migration time of the SFC migration increase, the TE module will continue the learning process without reporting its current findings to the BAE.
module. However, if the TE finished learning a fully working model, the
SN-A agent will use BAE to forward the accurate decisions to the MEC
layer. Finally, both TE and BAE use the “DRL Algorithms Trainer (DAT)”
module, which trains DRL algorithms based on the received inputs and
delivers adequate bandwidth values. Furthermore, in sub-sections 5.2.3
and 5.2.4, we detail and analyze the proposed comparison method of the
SN-A agent.

5.2.2 System Model

Before going deep into the implementation of the SN-A agent, we first
define the used state as the input problem and the action as the output
problem. The reward function guiding the agent’s decisions is also ex-
plained.

State Space:
Because a bandwidth selection problem always occurs between two MEC
nodes, only the source and destination nodes will be considered when
defining state space. We can model this problem using the size of the
last iteration of a given SFC migration process together with the number
of memory pages written in that iteration as a state. The dump size is the
memory size of the last iteration in an iterative live migration process.
The memory pages are the number of written pages by a live migration
process. These two parameters are crucial since the number of memory
pages, the dump size, and the available bandwidth are directly correlated
with the downtime duration of each instance, which is the key factor in
the users’ QoE and satisfaction, i.e., see Sections 5.1 and 3.1.

\[ S = (d_s, p_r) \] (5.1)

where \( S \) represents the state space, \( d_s \) denotes the dump size, and \( p_r \)
denotes the number of memory pages.

Action Space:
The action space is represented as allowed bandwidth allocations for mi-
grations. The DRL agent selects a given bandwidth value at each time-
step, offering by the same time the possibility to test actions as much as
possible.

\[ A = \{bw_1, bw_2, bw_3, ..., bw_n\} \] (5.2)
where $\mathcal{A}$ represents the set of all possible bandwidth values in the case of discretized values. However, most of the time, $\mathcal{A}$ tends to infinity. Therefore, we must consider both continuous and discrete action spaces in our problem.

**Reward Function:**

By using a reward function that covers the required metrics, an agent maximizes profits, thereby optimally performing and selecting the right actions within all defined states. As the live migration process uses both system and network resources, the adequate reward function must cover both categories of resources. Moreover, our modeling assumes a direct relation between the bandwidth used and the transmission delay, as well as the propagation delay, thus covering the network resources part. Regarding the system part, the processing delay can be considered to be the synchronization time. By measuring the required time for copying memory pages/file system, i.e., rootfs, in all live migration actions, these three-time delays are considered. Besides, by inverting the obtained time, we ensure that the longer the migration time is, the lower the reward $\mathcal{R}$ will be.

$$\mathcal{R} = 1/T$$  \hfill (5.3)

where $T = T_{\text{transmission}} + T_{\text{propagation}} + T_{\text{processing}} + \Delta T$

$\Delta T$ represents a constant related to the queuing delay as well as to the Kernel/Userspace transitions.

However, as we have several possible bandwidth values, this reward function becomes inefficient. For instance, if we consider a bandwidth equal to 3 GBps and a second one equal to 2 GBps, both provide similar times for the SFC migration metrics, which prevents the agent from being able to determine which action is best to select. Thus, to increase the accuracy of our reward function, the addition of the numerical value of the selected bandwidth is mandatory. The new reward function $\mathcal{R}$ is then expressed as:

$$\mathcal{R} = (1/T) + (1/bw_i)$$  \hfill (5.4)

where $bw_i$ denotes the current selected bandwidth value, $bw_i \in \mathcal{A}$.

To demonstrate the importance of introducing the selected bandwidth’s value into the reward function, we present a detailed example in Figure 5.5. Let us assume that a group of users is moving to a different

119
location, like the connected cars in Figure 5.5 moving from the service area of MEC 1 to the service area of MEC2. Those connected cars are consuming a video streaming service hosted initially in MEC1. Therefore, intuitively, we must ensure a minimum bandwidth allocation to support the migration of virtualized instances, an SFC, composing the video application, and serving the connected cars while ensuring users’ QoE. We represent the globally available bandwidth between the two MEC nodes in Figure 5.5 using a cylinder to indicate the capacity.

Meanwhile, we assume that the SN-A agent is in the training phase, where the TE module measures the total available bandwidth $bw_c$ in the reconnaissance phase presented earlier. The maximum bandwidth value measured is equal to 3 GBps, which is shown in blue in the cylinder between MEC1 and MEC2 in Figure 5.5. The TE module will be constituting the state “$s_t$”, $(d_s = 1.2 \text{ MB}, p_r = 596 \text{ memory pages})$, based on the information gathered from the MEC source via the periodical sampling of the RH module.

The TE module then selects an action, i.e., bandwidth value, based on a given policy, either learned, e.g., PG, or followed, e.g., $\epsilon$-Greedy. In the presented example, the first selected bandwidth value is equal to 2.5 GBps, as illustrated in red inside the cylinder representing the available bandwidth, while the downtime was equal to $1 \approx 1.2 \text{ (s)}$. Once done, the SN-A agent sends the selected action to the environment, i.e., the MEC layer, for execution. Then, the environment returns a reward and a new state to the SN-A agent. After that, the SN-A agent evaluates its choices and selects another action equal to 2 MBps, like that illustrated in green inside the cylinder representing the available bandwidth, in which the downtime was equal to $1 \approx 1.2 \text{ (s)}$. We must highlight that this is only a simplified example of two stages that differ from the training used by the SN-A agent, which keeps repeating this process until convergence. This allows us to conclude that defining a reward function only based on time is not helpful in this situation. In other words, selecting action $bw_1 = 2.5 \text{ GBps}$ and action $bw_2 = 2 \text{ MBps}$ is identical for the SN-A agent since $R_1 \approx 0.83$ and $R_2 \approx 0.83$. In contrast, adding the inverse of the chosen action will undoubtedly result in our SN-A agent selecting lower bandwidth values, like $R_1 \approx 0.834$ and $R_2 \approx 1.33$, and therefore selecting the second, i.e., 2 MBps, bandwidth value in the example, and ensuring optimal bandwidth allocation. It must be noted that the performances are not affected since the first term of the reward function, i.e., $1/T$, prevents this through time.
Finally, we added coefficients to both the time and bandwidth values to make one parameter more influential than the other depending on the considered objective of the network provider.

\[
R = \theta \times \left( \frac{1}{T} \right) + \vartheta \times \left( \frac{1}{bw_i} \right)
\]

(5.5)

where \( \theta \geq \vartheta \) as we emphasize the importance of the downtime.

### 5.2.3 Design of the DRL Algorithms Trainer

**Operational Mechanisms:**

The DAT module is introduced in Figure 5.6 to highlight the different components of the algorithms which constitute it and their working mechanisms. The DAT module aims to achieve two distinctive and divergent objectives through two particular types of DRL algorithms, DQN and DDPG. The DQN algorithm focuses on accelerating the delivery of decisions and the learning process along with the loss of precision, while DDPG is more constrained by the learning time, but outputs effective decisions [114, 115].

![Figure 5.6. Design and principles of the DRL Algorithms Trainer (DAT) module.](image)

Our objective is to develop a hybrid approach capable of coping with the stringent demands of 5G and beyond networks. Leveraging the OSS/BSS components of the NFV architecture, the SN-A agent can obtain information about the type of service, i.e., URLLC, mMTC, and eMBB. Besides, we know via MEC APIs [116, 117] the number of MEC services and applica-
tions susceptible to request an SFC migration pattern. We note a trade-off between how quickly and how accurately decisions can be made, depending on the underlying DRL algorithm. In the case of a large number of users requiring low-latency communications, a massive number of mMTC services, or enhanced mobile broadband resources, the DAT module will deliver actions, i.e., bandwidth values, based on the DDPG algorithm to either the BAE module in case of exploitation or to the TE module during training. However, if the resource requirements and the number of end-users applications are reasonable, the DAT module will deliver results following the DQN algorithm as those services do not consume or require strict bandwidth values.

Before describing both the DQN and DDPG algorithms and their hyperparameters, we provide the pseudo-code detailing the functionalities of the SN-A agent in Algorithm 2. The proposed agent is divided into two distinctive steps:

- The training phase: This begins with neural network initialization. It then allows our agent, through the TE and DAT modules, to learn the optimal policy by selecting appropriate actions in terms of bandwidth values;

- The exploitation phase: By following the optimal policy, the agent delivers actions and optimized bandwidth values leveraging the BAE and the DAT modules.

Algorithm 2, called Kernel SN-A (KSN-A), serves to describe in detail the two steps constituting the SN-A agent. Initially, KSN-A initializes both “base_bw” and “base_downtime” variables with “baseline_bw” and “baseline_downtime” values, respectively. The variable “baseline_bw” is the baseline bandwidth value, and the variable baseline_downtime is the optimal downtime achieved using the “baseline_bw” value, i.e., both defined in Section 5.1. The variable “trained” is set to “False” to allow KSN-A to start the training phase. The initialization procedure is shown in lines 1 to 3 in KSN-A, i.e., Algorithm 2.

As long as the variable “trained” is equal to “False” and the variable “iteration” is smaller than “M”, KSN-A will continue learning, and states will be fed into the training phase directly, i.e., lines 4 to 18 in KSN-A. It should be noted that “M” was introduced for reducing complexity and efficiency purposes. KSN-A obtains states, i.e., “S”, through a blocking function, i.e., get_state(), using the RH module that interacts with the
Algorithm 2: Kernel-Smart Network-Aware (KSN-A).

1. \( base_{bw} \leftarrow baseline_{bw}; \)
2. \( base_{downtime} \leftarrow baseline_{downtime}; \)
3. \( trained \leftarrow False; \)
4. while \( trained == False \) do
   5. \( iteration \leftarrow 0; \)
   6. while \( iteration < M \) do
      7. \( S \leftarrow RH.get_state(); \)
      8. \( RH.route(TE); \)
      9. \( type_{of\_service} = NFV.service_type(); \)
      10. if \( type_{of\_service} == Critical \) then
            11. \( TE.input(S, DDPG); \)
      12. else
            13. \( TE.input(S, DQN); \)
      14. end
      15. \( bw_{value} \leftarrow DAT.train(); \)
      16. \( iteration \leftarrow iteration + 1; \)
   17. end
   18. if \( bw_{value} < base_{bw} \) and \( RH.downtime() < base_{downtime} \) then
      19. \( base_{bw} \leftarrow bw_{value}; \)
      20. \( base_{downtime} \leftarrow RH.downtime(); \)
      21. \( trained \leftarrow True; \)
   22. end
   23. end
24. \( S \leftarrow RH.get_state(); \)
25. \( RH.route(BAE); \)
26. \( type_{of\_service} = NFV.service_type(); \)
27. if \( type_{of\_service} == Critical \) then
      28. \( BAE.input(S, DDPG); \)
29. else
      30. \( BAE.input(S, DQN); \)
   31. end
32. \( bw_{value} \leftarrow DAT.deliver(); \)
MEC layer, i.e., KSN-A line 7. Each state “S” is routed to the TE module through the RH module, i.e., KSN-A line 8. Meanwhile, we obtain the type of service requesting an SFC migration operation from the ETSI-integrated NFV domain, i.e., line 9. The TE module will then input the state “S” for either the DQN algorithm or the DDPG algorithm in the DAT module, depending on the criticality of the service requesting service migration, i.e., KSN-A, line 10 to 15. Note that the criticality of the service was deeply explained in the introduction in sub-section 5.2.3. Also, the initialization of both algorithms, i.e., DQN and DDPG, is omitted in KSN-A for the sake of simplicity. After that, in line 10 of KSN-A, the DAT module trains the selected algorithm, and the variable “iteration” is incremented by one for each new state, i.e., lines 16 and 17. Whenever the variable “iteration” is bigger than “M”, we compare the learned bandwidth values and downtime to the “base_bw” and “base_downtime” of the baseline solution. This step ensures that the learned values are better than the current baseline values. If the learned values are worse than the baseline values, KSN-A sets new baseline values and updates the variable “trained” to “True”, i.e., lines 19 to 23; thus, switching to the exploitation phase starting from the next input states.

Once the variable “trained” is equal to “True”, the SN-A agent, through its RH module and the ETSI-NFV integrated domain, gathers new states and routes the requests to the BAE module with the state “S” and the adequate algorithm depending on the type of service. Finally, the BAE module contacts the DAT module, which will deliver accurate bandwidth values in the context of SFC migration operations, i.e., lines 28 and 34 in KSN-A. Note that we precede each function/method with the name of the module that executes it to improve the understanding of the core features of the proposed SN-A agent.

To enhance the understanding of the role of each used algorithm, a brief introduction is provided which explains their necessity and complimentary usage in different situations.

**Deep Q-Network:**

To remedy the bandwidth selection problem, we kept the same principles associated with the DQN algorithm, as presented in sub-section 4.2.4 (Operational Mechanisms, the DQN part). Although DQN has its advantages when solving problems with a small discrete action space, it fails to compute $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \omega)$ of the target Q-value term in Equation 4.7, i.e., presented in sub-section 4.2.4, in case of a continuous or
pseudo-continuous action space. We can solve this issue by discretizing
the action space. For example, if our maximum available bandwidth is 3
GBps, we can use all values starting from 1 MBps to 3000 MBps, i.e., 3
GBps, to generate 3000 actions. This will reduce the number of actions,
but it will also neglect some bandwidth values, reducing bandwidth allo-
cation precision. However, due to computational limitations when using
ML frameworks such as Pytorch or Tensorflow, the discretization has to be
more refined and smaller. That is, the computation involved in a simple
task discretized into 3000 actions is huge \[108\]. By leveraging a hand-
crafted bandwidth value equal to 2 MBps, the downtime is equivalent to
using the whole available bandwidth for both video streaming applica-
tions and blank containers. Consequently, we discretize our action space
via the Discretization Module (DM), shown in Figure 5.6, to twenty dif-
ferent actions while centralizing the range around 2 MBps, which gives
twenty actions between 1 MBps and 3 MBps with a step of 0.1 MBps. This
proposed algorithm is fast and ensures convergence to the near-optimal
value while lacking precision in the bandwidth value selection, at the cost
of less precise bandwidth reservations.

**Deep Deterministic Policy Gradient:** DQN ensures fast training and de-
livery of predictions \[114\]. However, 5G and beyond networks are ex-
pected to involve numerous services with different characteristics and
types \[90\]. Consequently, a lack of precision in bandwidth action se-
lection may increase bandwidth capacity usage without being fully ex-
ploited. This poor action selection may result in an implicit reduction
of available network resources, thus impacting sensitive services such as
URLLC and eMBB ones. To cope with the previously cited constraint
and the continuous/pseudo-continuous action space limitation, we pro-
spose using the DDPG algorithm, which is considered a policy-based RL
algorithm \[69\]. As illustrated in Figure 5.6, DDPG is quite similar to
A2C and A3C principles, with a difference in the Actor’s operations \[68\].
Note that A3C is an A2C variant that implements parallel training where
multiple workers in parallel environments independently update a global
value function, hence the addition of “asynchronous”. The DDPG Actor
maps the states to actions instead of outputting the probability distribu-
tion across action space like in A3C and A2C. The Actor will start by ob-
serving the state \( s_t \) of the environment E. It will then select a given action
with the current weights of the approximator network, i.e., steps 1 and 2
in the DDPG part of Figure 5.6. Besides, the DDPG Actor adds noise \( \mathcal{N} \)
while selecting a given action \( a_t \) to encourage exploration. Upon performing the action \( a_t + N \), the environment returns a reward \( r_t \) and the next state \( s_{t+1} \) and considering that we are using experience replay principals in all our proposals, a tuple containing \( (s_t, a_t, r_t, s_{t+1}) \) will be collected and stored in the experience pool, i.e., step 4, DDPG, Figure 5.6. Once reaching the defined batch size, the DDPG Actor will start retrieving the next states and predicting the actions to be selected. Meanwhile, the Critic network uses the same selected batch of next states and the predicted actions from the Actor network to compute and evaluate the target Q-values, i.e., step 5, DDPG, Figure 5.6. Finally, the loss function will be updated to learn how to make more accurate evaluations in case of a Critic network and to increase the probability of choosing the right actions in case of an Actor network, i.e., steps 7 and 8, DDPG, in Figure 5.6. It should be noted that both Actor and Critic networks use target networks to prevent the optimization, i.e., the prediction of the actions of the next states and their evaluations, from encountering tight correlation problems.

**Design of Neural Networks:**

To realize an accurate mapping from states to actions, we build our neural networks following a pseudo-grid-searching mechanism, in which we take the hyper-parameters values utilized in original papers. We then vary those hyper-parameters to obtain optimal configurations and parameters related to selecting bandwidth value in SFC migration schemes. This method is similar to grid searching, which was developed in [118]. We selected the following specifications.

**DQN Hyper-parameters:** Regarding the DQN approach, we use two neural networks, the main Q-Network and a target Q-Network as a replica of the main network. While the main Q-Network is used to predict the current state’s Q-value, the target Q-Network is employed to predict next-state Q-values. We utilize an “\( \epsilon \)-Greedy” based on the “\( \epsilon \)” decay policy to allow a trade-off between the exploration/exploitation dilemma. For both Q-Networks, we adopt an Adam optimizer to adjust the network’s parameters [105]. The learning rates, i.e., \( \alpha \), and the discount factors, i.e., \( \gamma \), parameters were equal to \( 5 \cdot 10^{-2} \) and 0.95, respectively, for both Q-Networks. The target parameters of the Q-Network are updated every four episodes. The batch size used for updating the main Q-Network weights is 32. We also consider two fully-connected hidden layers in which the number of units, i.e., activation functions, is the mean between input
and output features. For both hidden layers and the output layer, we select ReLU as an activation function, represented by Equation 5.6. The number of output decisions is obtained via the DM, as introduced in subsection 5.2.3 and shown in step 1, DQN part, in Figure 5.6.

\[
\text{ReLU}(z) = \begin{cases} 
z & \text{if } z > 0 \\ 
0 & \text{if } z \leq 0
\end{cases}
\]  
(5.6)

**DDPG Hyper-parameters:** DDPG uses four neural networks. A Q-Network and a target Q-network are the Critic networks for evaluating the selected actions. In addition, a deterministic policy network and a target policy network as Actor networks play the role of action prediction. For both Critic and Actor networks, we adopt the Adam optimizer to adjust the parameters of the networks. The learning rates of the Actor and Critic networks were \(25 \cdot 10^{-5}\) and \(25 \cdot 10^{-4}\), respectively. The discount factor \(\gamma\) was 0.99. The parameter of the target Actor and Critic networks are updated with a coefficient \(\tau\) equal to \(1 \cdot 10^{-2}\). The batch size used for updating the deterministic policy network weights is 8. We employ a similar representation of hidden layers for all Actor/Critic networks; mainly, we use two fully-connected hidden layers in which the number of units, i.e., the activation functions, is 400 and 300, respectively. Each activation function uses the ReLU function for weights computation introduced in Equation 5.6. For the output layers of Actor networks, we utilize a Tanh activation function, expressed by Equation 5.7. It is worth noticing that Critic networks have a unique output used to compute the value of taking a given action at a given state. By using Tanh as an output activation function for Actor networks, the selected bandwidth values are confined between the range of \([-1, 1]\), as indicated in Equation 5.7.

\[
\tanh(z) = \frac{1 - e^{\exp(-2z)}}{1 + e^{\exp(-2z)}},
\]  
(5.7)

However, limiting bandwidth values to this small range is not realistic. Therefore, by leveraging the Action Refinement (AR) module, as illustrated in Figure 5.6, i.e., step 3 in the DDPG section, and as given in Equation 5.8, we multiply the obtained values by a given number dubbed “X”. Then, we add “X” to the obtained number. The main objective of this process is to centralize the output around “X”. For instance, our Actor network output 0.5, the AR module will output 750 KBps if the “X” = 500 KBps.
Note that \( "X" \) is also considered a critical hyper-parameter. It can therefore be varied to obtain optimal configurations and parameters related to selecting bandwidth value in SFC migration schemes. Initially, we exploit the results in Section 5.1, where we achieved the best results using a handcrafted bandwidth value equal to 2 MBps. With this initial indicator, knowing that the Tanh function varies from -1 to +1, and using the proposed formula developed in Equation 5.8, we initially set \( "X" \) equal to 1 MBps. This value is half that of the handcrafted value and allows us to visit the range from 0 MBps to 2 MBps. Then, we vary the hyper-parameter \( "X" \) while observing the different learning curves. This method is similar to that of grid searching, which was developed in [118]. Note that we selected the value 800 for the hyper-parameter \( "X" \). Finally, we add the Ornstein-Uhlenbeck Noise to the obtained action to encourage exploration [119].

5.2.4 Experimental Evaluation

This section presents our preliminary training and assessments of the two DRL-based algorithms for enabling a fine-grained selection of network bandwidth values for service migration. Our focus on the networking part of migration processes arises from our perceived need to support multiple simultaneous migrations, i.e., SFC migrations, caused by user mobility across domains or resource shortages.

Figure 5.7 describes the testbed environment used for the training phase, thus allowing our SN-A agent to refine bandwidth values selection. The testbed consists of three edge servers, Intel Fog Reference Design (FRD)\(^1\), as depicted in Figure 5.7. Note that we adopted FRDs servers to match the computations standards of MECs. Each FRD has 8 cores: an Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz, with VT-X support enabled, 32 GB of memory, and Ubuntu 16.04 LTS with the 4.4.0- 77-generic kernel installed. FRD1 and FRD2 act as the host source and destination, respectively. Both of them use LXC 2.8 as a container engine to enable container-level virtualization and CRIU 3.11 to allow service migration. We use the synchronized wait-for-me SFC migration strategy developed

\[ bw_{ddpg}(X, z) = X \times (1 + \tanh(z)) \] (5.8)

\(^1\)The Fog reference design is not a product sold by Intel and is rather a reference design offered to certain industry leaders to allow rapid development of Edge products.

128
in Section 5.1. This SFC migration strategy allows us to run all migration steps for each instance of the SFC in parallel except the final memory blocking action, i.e., Dump, where all instances should wait for each other. In a basic handcrafted controlled network, the aim of this approach is to efficiently control each step separately, thus allowing a fine-grained control and reducing the overall system and network resource consumption. Besides, we use ONOS as an SDN controller to configure OVS switches and steer network traffic between the SFC constituents. We consider an SFC for a video streaming application with three virtualized instances, i.e., length three. Each SFC contains a client, a video streaming server, and a turnaround node that analyzes and routes the integrality of the traffic in both directions. Note that FRD3 consists of the SN-A agent, the life-cycle orchestrator, which manages NF operations (i.e., creation, deletion, migration, and scaling), and the message broker server, which is RabbitMQ in our case.

Our proposed agent is first evaluated by showing the training phases for both DQN and DDPG while considering different input feature to highlight features selection, training speed, and stability, as shown in Figures 5.8 and 5.9. A detailed comparison is provided in Figure 5.10, which includes the action selection (bandwidth values) for DDPG against DQN and handcrafted values (baseline solution). Finally, the efficiency and the capacity of the proposed agent, i.e., SN-A, to reduce the network consumption is compared to the basic solution via a downtime comparison.
as shown in Figure 5.11. Note that we use an SN-A variant that only uses DQN and another SN-A variant that only uses DDPG to efficiently separate and evaluate each algorithm in the performance evaluation. Note also that we overcome the issues of migration failures in the training of both DDPG and DQN based algorithms, i.e., Figures 5.8 and 5.9, through the use of the retry module developed in Section 3.3. We designed a mechanism able to detect the failure of the last step and trigger an automatic retry to ensure a highly efficient migration that meets the requirements of the 5G networks. Thus, the correlation between migration requests is respected. In the case of unknown issues or errors, the whole scenario is deleted or not used for the training to keep the results concrete.

The initial experiment is related to the SN-A agent based on the DQN algorithm. We run 800 migration operations while randomly modifying the resources of the virtualization instances, i.e., CPU, RAM. Figure 5.8 shows the training comparison while considering the agent based on SN-A DQN for two input features (dump size and memory pages) and a single input feature (dump size), respectively. In Figure 5.8(a), the Y-axis represents the rewards collected over time-steps while the X-axis shows the number of iterations in the training process. In Figure 5.8(a), the rewards for two input features are represented in orange while those of the single inputs are illustrated in blue. Figure 5.8(b) conserves an identical representation, except for the Y-axis, in which we show the average rewards for every 32 iterations. As an initial reflection, we can state that DQN using both the dump size and the memory pages features outperforms DQN using one input feature, i.e., dump size.

Next, we evaluate the SN-A agent based on a DDPG algorithm approach in our second experimental scenario. We trained the model for thousands of migrations while randomly selecting application types and modifying...
the resources, i.e., CPU, RAM, of the virtualization instances. Figure 5.9 provides a comparison of the dump size, shown in blue, as a unique input feature and both the dump size and the memory pages, shown in orange, when considering the DDPG-based algorithm. In Figure 5.9(a), the Y-axis represents the obtained rewards over all iterations, while the X-axis represents the number of iterations or migrations we did during training in the proposed architecture. Figure 5.9(b) keeps the same depiction for the X-axis, while the cumulative reward for every 8 iterations is used for the Y-axis. Unlike the results in Figure 5.8, the findings in Figure 5.9 show that our DDPG-based algorithm is immune to the number of input features.

In Figure 5.10, we compare the bandwidth action selection for SN-A based on the DQN algorithm, the SN-A with the DDPG algorithm, and a baseline solution. Note that the baseline solution that uses handcrafted bandwidth values is an existing approach developed in Section 5.1 and serves as an upper boundary for the comparison. In Figure 5.10, for all sub-figures, the left Y-axis represents the bandwidth values in “KBps” while the X-axis portrays the number of iterations done in the training phase. Note that for sub-figures 5.10(a), 5.10(b), and 5.10(c), the baseline solution is represented in green. The single input feature-based solution and two inputs features are shown in red and blue, respectively, in sub-figures 5.10(a) and 5.10(b). While in sub-figure 5.10(c), the red and blue portray actions based on DQN and DDPG algorithms, both for two features, respectively. Figure 5.10(a) highlights the comparison between bandwidth action selection for SN-A agent based on DQN when considering one and two features, respectively. Based on the results, the agent using the DQN algorithm with one input feature, i.e., dump/memory size, failed to outperform the baseline solution, i.e., handcrafted, while the SN-
CHAPTER 5. AI-ASSISTED SERVICE FUNCTION CHAIN MIGRATIONS FOR NETWORK SLICE MOBILITY

(a) Comparing using different input features in the DQN-based agent.

(b) Comparing using different input features in the DDPG-based agent.

(c) Comparing DQN to DDPG when input features = 2.

Figure 5.10. Comparing bandwidth action selection for DQN against DDPG and handcrafted values.

A agent using the DQN algorithm with two features surpasses clearly both the baseline solution and the one feature DQN algorithm. Thus, we conclude the necessity of considering memory pages as a second input feature. We also notice the high variation in action selection for both approaches; that is, the DQN with one input feature or two features. Unlike DQN, the DDPG-based agent, shown in Figure 5.10(b), outputs stable and consistent training for both considerations, i.e., one/two input features. Besides, both DDPGs approaches outperform by far the baseline solution showed in green; the selected bandwidth actions were near to 1,600 KBps.

To compare DDPG and DQN agents, we plotted Figure 5.10(c) while considering two input features. The preliminary results demonstrate that both DQN and DDPG achieved better results compared to the baseline solution. From Figure 5.10(c), we confirm that DDPG is more stable than DQN and explores a broader range of actions during the training phase. However, Figure 5.10(c) indicates that in convergence, DQN is selecting lower bandwidth values than the DDPG-based agent, i.e., 1,400 KBps. Based only on action selection, we cannot determine the best approach in terms of resource efficiency. Thus, we extend our evaluation to cover downtime comparison.

Figure 5.11 depicts the induced downtime under distinct network config-
urations while using a DQN-based agent, DDPG-based agent, and hand-crafted bandwidth, respectively. The purpose of this experiment is to compare the proposed DRL algorithms, DQN and DDPG, with each other as well as against the limited handcrafted bandwidth used in Section 5.1 in terms of induced downtime. In Figure 5.11, the DQN-based agent, the DDPG-based agent, and the baseline solution downtimes can be viewed in the X-axis, while the Y-axis presents the time in seconds. To determine the most suitable DRL algorithm, Figure 5.11 portrays the mean downtime results and their STD, 95% CI, and CV when using the learned policy in case of DQN and DDPG algorithms and the static handcrafted value of 2 MBps. Table 5.3 summarizes those values in detail. For the three approaches, the downtime related to video-streaming containers is larger than blank, i.e., turnaround or dummy, containers. The main reason behind this additional time is the additional copies of the network connections status. From Table 5.3, we can observe that the agent based on the DDPG algorithm outperforms the remaining proposed approaches in terms of downtime. Indeed, the DDPG-based agent is the only DRL algorithm which reached less than one-second downtime when migrating blank, i.e., turnaround, containers.

Table 5.3. Downtime for different DRL approaches.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Strategies</th>
<th>Mean Time (s)</th>
<th>STD</th>
<th>95% CI</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN-based agent</td>
<td>Blank Container</td>
<td>1.026</td>
<td>0.208</td>
<td>0.0181</td>
<td>0.203</td>
</tr>
<tr>
<td></td>
<td>Video Container</td>
<td>1.224</td>
<td>0.324</td>
<td>0.0286</td>
<td>0.265</td>
</tr>
<tr>
<td>DDPG-based agent</td>
<td>Blank Container</td>
<td>0.957</td>
<td>0.098</td>
<td>0.008</td>
<td>0.102</td>
</tr>
<tr>
<td></td>
<td>Video Container</td>
<td>1.198</td>
<td>0.138</td>
<td>0.012</td>
<td>0.115</td>
</tr>
<tr>
<td>Handcrafted</td>
<td>Blank Container</td>
<td>1.222</td>
<td>0.066</td>
<td>0.05</td>
<td>0.054</td>
</tr>
<tr>
<td></td>
<td>Video Container</td>
<td>1.371</td>
<td>0.056</td>
<td>0.042</td>
<td>0.036</td>
</tr>
</tbody>
</table>

Figure 5.11. Downtime comparison in case of different DRL algorithms.
Results Discussion:
Based on the bandwidth action selection shown in Figure 5.10(c) and the downtime comparison displayed in Figure 5.11, we conclude that the minimal downtime is unbacked by selecting lower bandwidth values actions. This was confirmed by the downtime results in Table 5.3. Moreover, increasing the downtime may affect end-users’ QoE and break the SLA defining the requested QoS. Therefore, DDPG is preferred over DQN for its training stability, lower downtime achievements, and user satisfaction.

5.3 Summary

In this chapter, the design and implementation of four SFC strategies to support NSM are presented. The results of an evaluation of the four SFC strategies are also presented. Since our results showed no clear winner among the introduced strategies, trade-off or hybrid combinations are the recommended proposals. Additionally, we have shown that a network-aware SFC strategy should support the other proposed approaches.

Another significant aspect of this chapter is the design, modeling, and evaluation of a DRL-based agent with two different algorithms for allowing a fine-grained selection and allocation of bandwidth resources. We evaluated the proposed agent under different configurations and real-world deployments to enable an optimized NSM within the 5G network. Our results showed that the DDPG-based agent outperforms the DQN-based agent in terms of accuracy, stability, and users’ QoE.

The main goal of this chapter was to answer RQs 6 and 8. In the context of the studies carried out in this thesis, determining which SFC strategy is suitable for supporting 5G network slices and reducing network overhead while performing different NSM patterns is a key factor towards enabling smooth NSM in 5G and beyond networks.
6. Conclusion and Future Research

This chapter summarizes the challenges, the main contributions, and the key principles behind enabling NSM, its patterns, and triggers while satisfying network and system resources requirements. The various outstanding questions are outlined, and future challenges and research directions are suggested.

6.1 Conclusion

The aim of the research presented in this dissertation is to advance the current state of knowledge about mobility in network slices and autonomic SFC management and control in 5G and beyond networks. To achieve this objective, the investigation was focused on the following: 1) suitable enabling technologies for NSM (RQ2), 2) the definition and the introduction of various mobility patterns to the current network slice paradigm (RQ3), 3) the means to group users for optimizing resource consumption and allowing smooth mobility patterns (RQ4), 4) the definition of triggers for enabling NSM patterns (RQ5), 5) the selection of adequate SFC migration strategies to support mobility of network slices (RQ6), 6) the selection of triggers for optimizing the system resources of slices (RQ7), and 7) refining and reducing network resource overheads for NSM use-cases (RQ8). Responding to those diverse research issues served to strengthen the possibility of expanding the ongoing network slice definition to support upcoming use-cases involving new mobility patterns while fulfilling network and system resource requirements (RQ1).

To respond adequately to the main research question, RQ1, we have addressed seven distinct RQs, RQ2 to RQ8, in this dissertation. We first introduced, in Chapter 2, the main enabling technologies and frameworks behind the mobility of network slices. Chapter 2 also showcased the main
definitions, nomenclatures assumed in the dissertation, and ML-related frameworks as well as their features. The definition of the basic principles and technological requirements, in Chapter 2, rely principally on Publications I, II, III, VII, and VIII.

In Chapter 3, an NFV compliant architecture was presented to support NSM. Experimental results revealed that the proposed approach, through live migration operations, is latency-sensitive. Also in this chapter, a new northbound benchmarking methodology related to intent-based networking to measure agility and scalability in SDN-enabled environments was presented. Finally, in Chapter 3, a framework that combines both service migration and SDN-enabled networks was introduced to manage and assist NSM operations. The results obtained in Chapter 3, from Publications I, II, III, and IV, achieved the key design objectives and answered the second research question, RQ2, and part of RQ3.

To answer the fourth and fifth research questions, RQs 4 and 5, as well as the remaining part of the third research question, RQ3, we have defined, in Chapter 4, various NSM patterns, their grouping methods, and their key enabling triggers. The experimental findings, based on the simultaneous migrations of multiple containers constituting network slices, demonstrate the effectiveness of the proposed approaches. Additionally, in Chapter 4, through the design, modeling, and evaluation of two DRL-based algorithms, a fine-grained selection of system-based triggers regarding the NSM patterns was presented. This answered the seventh research question, RQ7. This chapter was entirely based on the insights features in Publications V and VII.

To answer the sixth research question, RQ6, four practical SFC migration strategies were presented in Chapter 5 to support NSM. We then concluded, through a practical evaluation, that a compromise between the proposed approaches is suitable for supporting mobility in network slices. To reduce network overhead while performing different NSM patterns, i.e., addressing RQ8, we introduced our envisioned architecture hosting the proposed network-aware agent. Next, we designed, modeled, and evaluated the proposed agent with its features. The performance evaluation results showed in Publication VIII validated that the proposed network-aware agent is capable of selecting accurate bandwidth values while ensuring fast and reliable service migration to address a large number of emerging use-cases requiring strict requirements and improving users’ QoE.
6.2 Future research

As a central emerging technology for next-generation networking, the NS paradigm offers several advantages, particularly the reduction of network latency and the ability to deliver highly customizable services to new industry sectors through the use of MEC-enabled networks. However, the impact of user mobility on the optimal resource allocation within and between slices deserves more attention. Slices and their dedicated resources should be offered where the services are to be consumed to minimize network latency and associated other overheads and costs. Motivated by the deficiency of current network slices, a set of theoretical and practical implementations and frameworks are proposed in this dissertation to enable smooth mobility of network slices in 5G and beyond networks, while reducing service consumption and network overhead. Nevertheless, the following challenges for future research should be highlighted.

The MIRAI framework, described in Chapter 3, allows fast service migration between multiple administrative and technological domains. Leveraging MEC-enabled networks, SDN paradigms, and the FMC concept, MIRAI acts as the low-level enabler for NSM, thus improving the end-user experience. An assessment showed a reduction of approximately 50 percent in downtime, which demonstrates the efficiency of the proposed solution compared to prior works using similar underlying technology, like LXC or Docker. Although the shortest downtime was provided by the iterative migration approach in Chapter 3, it is essential to note that future improvements related to proposing new migration models or reducing downtime should be considered. A prime example is using AI techniques to detect the right iteration susceptible to reduce downtime without breaking users’ QoE. Besides, investigating highly optimized ways to create executable images might help reduce downtime significantly. Another consideration would be to recognize that use case-specific experiments are needed in order to determine what kind of downtime would be acceptable or, alternatively, to find out whether live migration avoidance strategies can be created for the use case.

The SMDM agent developed in Chapter 4, through its two distinct DRL-based algorithms, i.e., A2C and DQN, is an effort toward adding intelligence to NSM triggers while saving their original definition and implementations. Even though numerical results, established by the simulated environment, showed the efficiency of the A2C based approach compared
to the DQN solution in terms of training stability, learning time, and scalability, the A2C-based agent presented limitations caused by the increase in the size of the action space. Thus, for the author and the research community, a future research direction would be to investigate extending the model to cover the DDPG based on Wolpertinger architecture [120]. This new policy architecture can efficiently learn and act in large discrete action spaces. Besides, the author plans to expand the deployment and evaluation in real production environments as well as cover the remaining triggers related to user mobility.

The findings in Chapter 5 demonstrated that DDPG outperforms DQN in terms of accuracy, stability, and users’ QoE. Nonetheless, the proposed SN-A agent with its two DRL-based algorithms, DDPG and DQN, is limited to only detecting the required bandwidth for a given workflow. A significant future challenge relates to the extension of the proposed ML-based solution presented in this dissertation to cover the scheduling of workflows after detecting their requirements in terms of bandwidth.
References


