André Luiz Beltrami Rocha

Slice Elasticity Architecture (SEA): Definition, Implementation and Integration

São Carlos

2022

André Luiz Beltrami Rocha

Slice Elasticity Architecture (SEA): Definition, Implementation and Integration

Thesis presented to the Graduate Program in Computer Science at the Federal University of São Carlos, as part of the requirements for obtaining the title of Ph.D. in Computer Science.

Universidade Federal de São Carlos - UFSCar

Centro de Ciências Exatas e de Tecnologia - CCET

Programa de Pós-Graduação em Ciência da Computação - PPGCC

Supervisor Fábio Luciano Verdi

São Carlos

2022

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato André Luiz Beltrami Rocha, realizada em 14/02/2022.

Comissão Julgadora:

Prof. Dr. Fabio Luciano Verdi (UFSCar)

Prof. Dr. Paulo Matias (UFSCar)

Prof. Dr. Luiz Fernando Bittencourt (UNICAMP)

Prof. Dr. Cristiano Bonato Both (UNISINOS)

Prof. Dr. Rodolfo da Silva Villaça (UFES)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

To my family and God. I am nothing without them.

Acknowledgements

I want to thank God for blessing me and always remembering the important things giving me the strength to conclude this chapter in my life.
I want to thank my father and mother, Diogenes and Marcia, for believing and supporting my decisions. Without them, I will not be here.
I want to thank my family, Ana Laura, Ana Paula, Ana Claudia, Mario, Larissa, Amanda, Duda, Ricardo, Riccardo and Fabiano, for always being present.
I want to thank my girlfriend, Bianca, for her unconditional love and support. Also, her family, who have always been present.
I want to thank my supervisor Fábio L. Verdi. It has been over eight years together since my graduation, master's and Ph.D. I learned a lot from him as a professional, a researcher, and a man. May this bond last for a long time.
I want to thank my friend Paulo Ditarso for all our time together, his scientific contributions, and his help in my life. These five years of Ph.D. would not be the same without you. Thank you very much.
I want to thank my laboratory colleagues Leandro, Gustavo, Felipe, Thiago, Guilherme, Felipe, William, Victor, Sergi, Pedro.
I want to thank all the researchers and students participating in the NECOS project. In special to Prof. Christian (UNICAMP), Prof. Rafael (UFU), Prof. Sand (UFMG), Prof. Javier (Cinvestav), Prof. Joan (UPC), Celso (UNICAMP), Francisco (Cinvestav) and Francesco (UCL).
I want to thank all my friends Caio, Matheus, Guilherme, Igor, Caique, Alexandre, Helder, Bruno, Fernando, Douglas, Lucio, Felipe, Carlos, Caique and others for all the funny moments playing or enjoying together.
I want to thank all my working colleagues for participating in my work day during this period.
I want to thank everyone who touched my life in good and bad moments.

"Resilience: it's the ability that all human beings have to face the displeasures of life with their heads held high."

Abstract

Network slicing initiatives (e.g., Cloud Network Slicing, 5G Network Slicing) are being widely studied by academia and industry as enablers for future networks like 5G and 6G, as well as verticals such as V2X, critical communications, augmented reality, and others. Several challenges arise with this emerging concept, which needs to be solved to further groundbreaking technologies. A significant challenge is slice elasticity, defined as the capacity to grow or shrink slice resources (e.g., computing, networking, or storage). One alternative to solve this challenge is providing a *closed control loop* (CCL) involving slice monitoring, management, and orchestration. In addition, this CCL may be easily adapted and coupled with most of the slicing initiatives of standardization bodies. In that sense, this thesis proposes, implements, and integrates the Slice Elasticity Architecture (SEA), an architecture to support a new business model called *Slicing Elasticity as a Service* (SlEaaS). This new business model enables slice resource orchestration across a heterogeneous end-to-end infrastructure that spans multiple administrative and technological domains in a facilitated and automated way for tenants and slice providers. The SEA monitors slice resources periodically, offers the policies/SLAs administration, and performs slice elasticity operations autonomously when the predefined policy is violated to ensure the QoS of the instantiated service. The SEA design is scalable and generic enough to support several technologies transparently. The results demonstrate the SEA functionalities based on scenarios of slice elasticity operations for computing and networking resources.

Keywords: Cloud network slicing, Slice elasticity, Slice monitoring, Slice orchestration, Slice elasticity as a service, CCL.

List of Figures

List of Tables

Listings

List of abbreviations and acronyms

3GPP	3rd Generation Partnership Project
5GPPP	European 5G Infrastructure Public Private Partnership
API	Application Programming Interface
AR	Augmented Reality
CCL	Closed Control Loop
CDN	Content Distributed Network
CNS	Cloud-Network Slice
CSP	Communication Service Providers
DB	Database
DC	Datacenter
DSP	Datacenter Slice Part
E2E	End-to-End
ETSI	European Telecommunications Standards Institute
EU	European Union
HE	Horizontal Elasticity
IaaS	Infrastructure as a Service
IETF	Internet Engineering Task Force
IMA	Infrastructure and Monitoring Abstraction
ITU-T	ITU Telecommunication Standardization Sector
KPI	Key Performance Indicator
LSDC	Lightweight Slice-Defined Cloud
MA	Management Adapter
MANO	Management and Orchestration

ME	Monitoring Entity
MPLS	Multi-Protocol Label Switching
NE	Network Element
NECOS	Novel Enablers for Cloud Slicing
NFV	Network Function Virtualization
NGMN	Next Generation Mobile Networks
ONF	Open Networking Foundation
OSM	Open Source MANO
P4	Programming Protocol-independent Packet Processors
PaaS	Platform as a Service
QoS	Quality of Service
RNP	Brazilian National Network for Education and Research
RQ	Research Question
SaaS	Software as a Service
SDN	Software-Defined Networks
SEA	Slice Elasticity Architecture
SLA	Service Level Agreement
SlEaaS	Slice Elasticity as a Service
SP	Slice Part
SRO	Slice Resource Orchestrator
TA	Telemetry Adapter
V2X	Vehicular to X
VE	Vertical Elasticity
VIM	Virtual Infrastructure Manager
VLSP	Very Lightweight Network & Service Platform
VR	Virtual Reality

WIM	WAN Infrastructure Manager
WSP	WAN Slice Part
YAML	Yet Another Markup Language

Contents

1 Introduction

The network slicing concept dates from 2009 (Galis et al., 2009), but only recently it has been widely studied due to the specification of future mobile networks. Several slicing concepts with similar definitions can be found in the literature and standard bodies (e.g., Network Slicing, Cloud Network Slicing, and 5G Network Slicing). Therefore, it is necessary to define the term used throughout this thesis. We have adopted the Cloud Network Slicing (CNS) definition, which is *an end-to-end (E2E) infrastructure responsible for providing services across multiple domains, including different types of infrastructure, for example, cloud computing, networking, and storage* (Clayman et al., 2021a; Rocha et al., 2022).

With the rise of slicing, new verticals are introduced, which demand hard constraints coupled with the 5G/6G requirements to ensure the quality of service (QoS). Critical communications, enhanced mobile broadband, massive IoT applications, vehicular communication (V2X), and Augmented Reality and Virtual Reality (AR/VR) are only a few examples of verticals that could be instantiated in a slice infrastructure comprising many benefits and challenges (Contreras, 2018; Galis, 2018). Such benefits are possible mainly due to the following intrinsic characteristics of the slice concept (Clayman et al., 2021a; Luong; Outtagarts; Ghamri-Doudane, 2019; Sanabria-Russo; Verikoukis, 2021): (i) the abstraction of E2E communication through different infrastructures allocated on multiple administrative and technological domains; (ii) the segmentation of infrastructure allowing the deployment of the service in shared or isolated, physical or virtual infrastructures; (iii) the orchestration of slice resources to continuously guarantee the QoS, optimizing the resources usage or the service performance through the growing or shrinking of the slice on demand, known as slicing elasticity; (iv) the support of well-established technologies used in the provision and management of infrastructure, such as virtual infrastructure managers (VIMs), software-defined networking (SDN), network functions virtualization (NFV).

Although all the efforts to create slice standards and definitions by entities such as the 3rd Generation Partnership Project (3GPP) (Foy; Rahman, 2017), European Telecommunications Standards Institute (ETSI) (Zarrar Yousaf et al., 2018), and Next Generation Mobile Networks (NGMN) (Thalanany, 2017), there are many open questions in the context of CNS and slicing in general. Under current approaches, most efforts are related to defining an architecture to support slice deployment and management. A new actor that has come up with the definition of slicing is the Slice Provider responsible for providing the E2E slice. The such provider (e.g., NECOS (Contreras, 2018), 5G-ZORRO (Gavras et al., 2021), OSM (Marsico, 2022)) is an essential player that can

abstract the slice infrastructure deployment by interacting with different infrastructure providers (e.g., cloud, edge, WAN) to connect the requested E2E slice, providing the service instantiation and ensuring the established SLAs.

Activities related to managing, monitoring, and orchestrating the slice resources can also be the responsibility of a slice provider. Therefore, they are essential to ensure the QoS according to the SLAs/SLOs defined by a tenant. One of the most important examples of orchestration is *slice elasticity*, which means dynamically increasing or decreasing the slice resources. Recent works (Almeida; Maciel; Verdi, 2020; Gutierrez-Estevez et al., 2018) show a significant increase in the number of articles that explore the terms related to orchestration, monitoring, management, and elasticity of resources in the context of slices, from 2017 onward. Among them, slice elasticity plays a vital role in future network verticals, mainly due to resource heterogeneity in infrastructures comprised of computing, storage, and networking resources, which can be physical or virtual. Moreover, elasticity operations can benefit countless use cases to improve service performance.

To exemplify the importance of elasticity, we present a use case (AR/VR) provided on top of a CNS infrastructure and illustrated in Figure 1. The AR/VR verticals are being widely explored for several applications (Morín; Pérez; Armada, 2022) and can benefit from the slice elasticity operations. For example, suppose a tenant wants to provide an AR/VR game connecting users from different countries, e.g., Brazil and Spain. Additionally, the tenant required that the used infrastructure for this service should be instantiated between both countries. Therefore, she chooses to instantiate the service in a slice, passing to a slice provider the requirements, i.e., the number of servers, network characteristics, and SLAs that must be respected (e.g., latency cannot be higher than 100 ms). As this game becomes more popular, it is expected an increase in the number of simultaneous users from both locations, may decrease the quality of this service. However, based on the constant monitoring of the infrastructure and the SLA, an elasticity operation is triggered as soon as the service latency exceeds the threshold. To improve latency and consequently the QoS of the AR/VR service, an elasticity operation is performed, adding more computational resources in this slice for both locations (illustrated in purple in the figure).

Analogously, for any other use cases and verticals (e.g., V2X (Alalewi; Dayoub; Cherkaoui, 2021), critical communications (Contreras, 2018), enhanced mobile broadband (Popovski et al., 2018)), the ability to grow or shrink the infrastructure of a slice is essential, so that the defined SLAs are respected based on the service Key Performance Indicators (KPIs). This dynamic optimization on slice resources must occur regardless of the slice resource type (computing, networking, and storage). In the context of elasticity operations in computing resources, we can highlight the following possibilities: increase or decrease the number of VMs, servers, CPUs, container replicas, amount of memory, disk (storage), etc (Rocha et al., 2022). The elasticity operations on networking resources

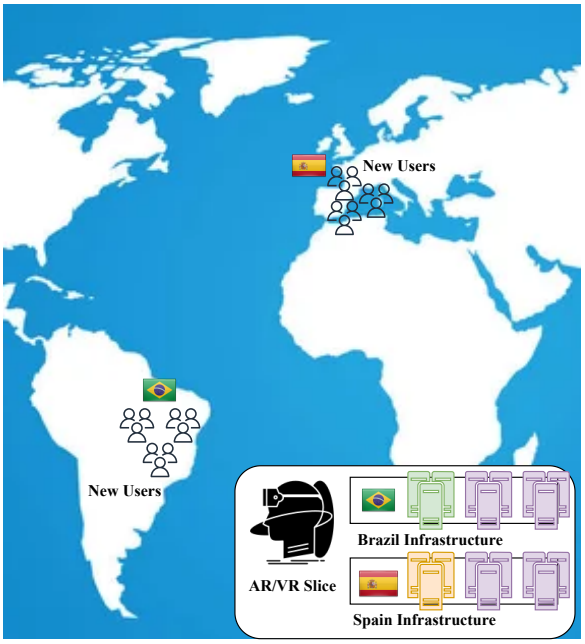


Figure 1 – Slice elasticity AR/VR use case.

are similar but focused on growing or shrinking the network elements (NEs) (Turkovic; Nijhuis; Kuipers, 2021; Breitgand et al., 2021). Moreover, the NEs softwarization in the control and data plane rises and enables novelties in this field (e.g., OpenFlow (McKeown et al., 2008) and programmable data plane (Bosshart et al., 2014)). To exemplify those operations, we can cite the network element's bandwidth adjustments, the creation of priority queues, and the establishment of MPLS tunnels.

One of the first initiatives that began to define concepts related to CNS (e.g., slice provider, slice elasticity, slice marketplace) was the international project called NECOS (Clayman et al., 2021b) (Novel Enablers for Cloud Slicing[1]). The project contemplated the participation of eleven universities and two companies from Brazil and the European Union (EU). Furthermore, from the beginning of the project, I was one of the Ph.D. NECOS students. Therefore, this work originated and evolved from the efforts we initiated at NECOS. In summary, the objective of NECOS was to define, implement and integrate an architecture capable of providing *Slicing as a Service* (SlaaS) in a simplified way for tenants. The features of the NECOS architecture were numerous and included negotiating with a slice marketplace, instantiating the slice infrastructure, instantiating the service, managing the infrastructure and service, monitoring and orchestrating resources, and the slice decommissions.

In the context of NECOS, we researched and defined an architecture capable of monitoring and managing the resources of a slice considering multiple administrative and technological domains. Monitoring consisted of periodically collecting metrics from the

[1] http://www.h2020-necos.eu/.

infrastructure per slice, and supplying the needed information to an orchestrator so that elasticity operations could be triggered. Finally, the management aspect provided the necessary abstraction to interact with different technologies to instantiate the services in the slice infrastructure.

Elasticity operations in the context of slices play a very important role in making this technology resilient and widely used to provide services. The NECOS project did not cover such operations, as we focused on defining and providing entities that would eventually support them. However, after an extensive literature review, we identified that elasticity operations are still a research challenge and constitute the core of this thesis proposal. Slice elasticity is not a trivial task and requires high control and management of resources that must be jointly orchestrated to meet the E2E service requirements. Therefore, we can cite the following challenges of performing the slice elasticity:

- The increasing or decreasing of slice resources, at the same time or individually, leads to the need to redeploy or reconfigure the service given the new infrastructure provided after the elasticity;
- The components responsible for monitoring and managing the slice must be able to continue to gather metrics from the newly introduced slice features. As soon as the change in slice topology occurs as a result of a slice elasticity operation;
- The heterogeneity of slice features adds to the slice elasticity operation's complexity. Therefore, there is a need for components to abstract technologies transparently to tenants or slice providers.

Similarly to NECOS, which presented the role of the Slice Provider and a new business model called Slicing as Service, the proposal of this thesis emerged. The slice elasticity may be provided following the *as a service* model, in which both slice tenants and providers can delegate such elasticity operations to an entity capable of providing the closed control loop (CCL), which comprises the monitoring, management, and orchestration of slices in an automated and simplified way.

This thesis then proposes, implements, integrates and evaluates the **Slice Elasticity Architecture (SEA)**, an architecture capable of providing a new business model called **Slicing Elasticity as a Service (SlEaaS)**. Unlike the Slicing as a Service business model, our proposal does not include the slice infrastructure instantiation phase but rather the post-slice instantiation phase. The idea is that new players that want to offer SlEaaS implement their platforms based on the SEA components, just as we have providers that provide other well-known business models, for example, Platform as a Service (PaaS), IaaS (Infrastructure as a Service), SaaS (Software as a Service), SlaaS (Slicing as a Service), among others. Although a slice provider may offer slice elasticity agnostically on its

platform, there are the following benefits of providing such a service separated from the instantiation and negotiation of a slice's infrastructure:

- The disassociation of SlEaaS from SlaaS guarantees that this platform can be used transparently by tenants who have slices instantiated without depending on the slice provider. In addition, different slice providers can offer SlEaaS from third-party entities implementing the proposed architecture;
- The SlEaaS enables numerous possibilities related to the intrinsic characteristics of the slices, and there may be differences in the prices of the services offered depending on several aspects, for example, the location of slice infrastructure, the complexity of elasticity operations offered as a service, the SLAs defined, etc;
- The isolation of such services from the slice provider facilitates the management of the components responsible for providing exclusively the SlEaaS. Furthermore, since the SlEaaS provider specializes only in the CCL of the slices, such a service may be offered with even more quality, as tenants and slice providers use it transparently and automatedly to ensure the required SLAs.

The SEA supports a CCL of slice orchestration, which means that the tenant or slice provider can delegate the monitoring, management, and orchestration of slice resources to a SlEaaS provider that implements the SEA. In addition, the SEA provides the policies CRUD and the capacity to trigger the elasticity automatedly or not. This approach attenuates the complexity of orchestrating slice resources by delegating this task to a specialized, customized, and scalable third-party entity (the SlEaaS provider). Therefore, the SEA can abstract the resources' orchestration, periodically check the violation of the defined policies, and perform slice elasticity operations to ensure tenants' QoS. In other words, the SEA monitors the heterogeneous slice resources (e.g., computing, networking, and storage resources) and performs elasticity operations based on the tenant's policies. Such operations are performed by increasing or decreasing slice resources through abstractions (implemented as adapters) for different technologies, e.g., Virtual Infrastructure Managers (VIMs), WAN Infrastructure Managers (WIMs), or Monitoring Entities (MEs).

We implement and evaluate the SEA components to validate the architecture design and feasibility. The evaluation intends to cover the main aspects of the slice concept and the demonstration of the CCL slice orchestration on computing and networking resources. Most scenarios were evaluated in real slices instantiated in different cities or even countries. Furthermore, we explored the different types of slice elasticity defined in the context of this work to prove the CCL and improve the QoS of the instantiated service. Finally, we also present results that validate the SEA scalability, as the number of slices being orchestrated

increases; and that it is generic enough to support different types of infrastructure and technologies to manage or monitor the slice.

1.1 Main contributions

This thesis has the following main contributions:

- We designed, implemented, integrated, and evaluated the SEA to support a new business model proposed in this thesis called SlEaaS. Furthermore, the architecture design is scalable, generic, expandable, and easily integrated with any technology or actors.
- We define elasticity operations in the context of CNS resources, comprising resources of computing, networking, and a new entity called Slice Part. New elasticity operations can be defined for each of them, considering the nature of the resource and its technological alternatives. This work presents 10 different types of slice elasticity divided into computing and networking resources and operations that increase or decrease the resources. Finally, we address the heterogeneity of slice resources, showing operations on physical servers, virtual machines, containers, switches, and routers.
- We presented a new business model focused on providing slice elasticity operations in an automated way for the tenant or slice provider. The SlEaaS follows well-known business models: Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Software as a Service (SaaS).
- We provided well-defined APIs, information models, and component descriptions that form the base for the SEA components implementation.
- We performed the SEA evaluation on different scenarios, including slices instantiated over diverse cities and countries, exercising most of the slice elasticity operations defined in this research.

1.2 Objective

This thesis aims to propose, implement, integrate and evaluate SEA providing slice elasticity operations as a service in a transparent and automated way for tenants and slice providers.

1.3 Research questions

Below, we presented the research questions (RQs) that drove the entire evaluation of the SEA:

- ***RQ 1:*** Does SEA scale in response to several metrics from multiple slices monitored simultaneously?
- ***RQ 2:*** Does SEA support the CCL of computing resources instantiated in multiple administrative and technological domains?
- ***RQ 3:*** Does SEA handle CCL of networking resources, including policy creation?
- ***RQ 4:*** Is SEA generic enough to handle the elasticity operations, no matter which technologies are present in the slice?

1.4 Thesis structure

Following, we present the description of the remaining chapters:

- Chapter 2 (**Theoretical Fundamentals**) aims to present the main concepts used in this work. This chapter has been divided into the following themes. First, we start discussing the concept and definitions surrounding CNSs. We then detail the NECOS project, making clear the major evolutions of our proposal during and after the end of the project. Next, we propose the slice elasticity operations and the new business model SlEaaS. Finally, we present the concepts of Closed Control Loop (CCL), Virtual Infrastructure Managers (VIMs), WAN Infrastructure Managers (WIMs), and Monitoring Entities (MEs), which will be widely used in the text and are essential for understanding the SEA.
- Chapter 3 (**Related Work**) presents the literature review for this thesis. The study was divided into three main categories since this work has several areas: the projects and initiatives of standardization bodies, the papers related to computing elasticity efforts, and the review of networking elasticity literature.
- Chapter 4 (**SEA**) describes the SEA design, detailing the APIs and components' responsibilities. The SEA comprises three main layers: the APIs, the control, and the per-slice components.
- Chapter 5 (**SEA Implementation**) details the implementation aspects of the SEA divided into three sections: the *SEA APIs*, the *Controllers*, and the *Per-slice components*. The first section describes all the APIs defined, explaining their

responsibilities and input parameters. Such APIs are of paramount importance to emerging other possible SEA implementations. The Controllers section discusses the control layer, where the logic to provide the SlEaaS for multiple slices is presented. Two controllers will be described: the SEA Controller and the Policy Controller, to provide the CCL of the slice resources. Finally, the Per-slice components section details how the SEA can abstract the heterogeneous technologies on a slice, through the implementation of adapters. Two adapters will be presented: Management Adapters (MAs) and Telemetry Adapters (TAs).

- Chapter 6 (**Evaluation and Scenarios**) answers the RQs described above, presenting the evaluation of the SEA. In addition, the proposed elasticity operations are exercised in real scenarios to improve the QoS of different services.

- Chapter 7 (**Conclusion**) remarks on challenges, lessons learned, open research questions, future works, and all dissemination activities originated from this thesis.

2 Theoretical Fundamentals

2.1 Cloud Networking Slices

The term commonly found in the literature is *Network Slicing*, from which several other concepts have been derived (e.g., Cloud Network Slicing, 5G Network Slicing). Due to the divergences in the literature about *Network Slicing*, those derived terms should be clarified. We found two main definitions of the same concept of network slicing: the first means that network slices are composed of only networking resources, and the second is the exact definition that we have for CNS (NECOS, 2019; Galis, 2018). In the context of the NECOS project and this work, we defined the CNS concept *as an E2E infrastructure to provide a service composed of heterogeneous infrastructures cross-domain, which may contain cloud computing, networking, storage, and VNFs, among other types of resources.*

This variety of resources must be instantiated transparently to the tenant responsible for managing and orchestrating the services that will be instantiated across the infrastructure. As an E2E concept, slices include components belonging to different administrative domains and demand a higher abstraction level for resource management and monitoring. In addition, the slice tenant-provider relationship introduces new business models such as SlaaS and SlEaaS.

Figure 2 aims to illustrate the concept of CNS and define important terms used

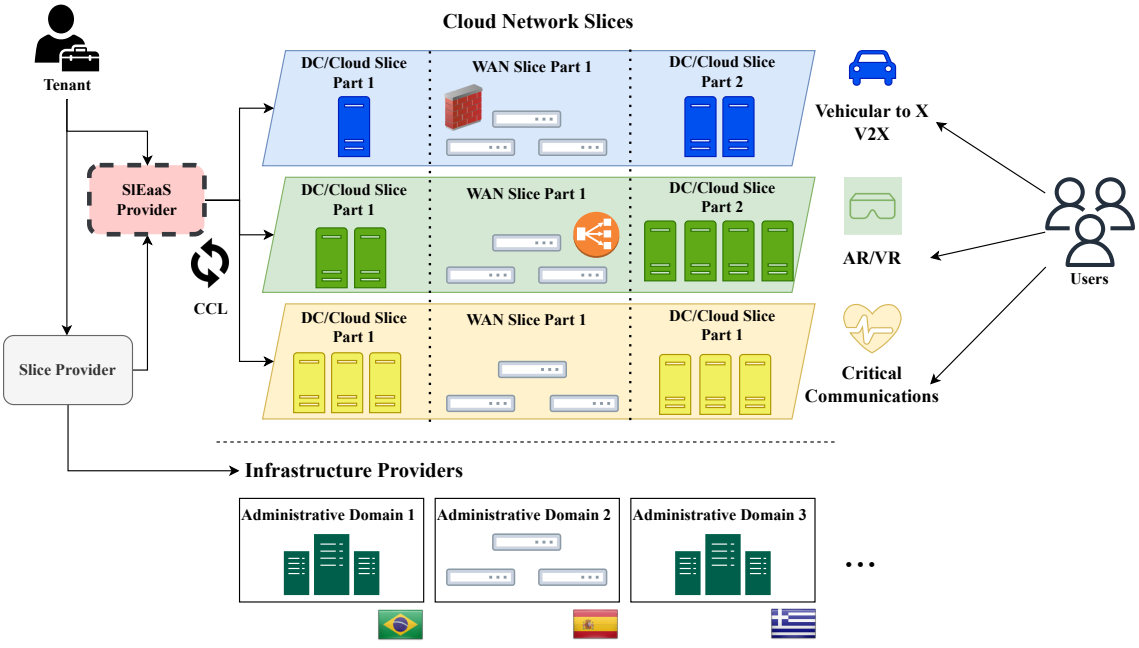


Figure 2 – Cloud Network Slicing (CNS).

throughout this work. The figure shows three services from different verticals: a V2X service in blue, an AR/VR service in green, and a critical communications service in yellow. The slice infrastructure could be composed of infrastructures located elsewhere, even in infrastructure providers on distinct countries or continents. For example, Slice 2 provides an AR/VR service that connects resources from Brazil, Spain, and Greece based on the geolocation of the infrastructure providers. The following definitions will be constantly used in this work:

- **Tenant:** hires cloud network slices from slice providers and offers services to final users over the slice hired. The tenant can delegate the slice orchestration to the SlEaaS provider defining SLAs/policies to ensure the QoS;
- **Users:** are the final consumers/users of the service instantiated over the slice;
- **Slice Provider:** offers the SlaaS business model, instantiating and negotiating the infrastructure requested by the tenant with different infrastructure providers. The following operations are part of the slice provider: slice creation, decommission, management, infrastructure negotiation, and others. The NECOS Project (Clayman et al., 2021b), OSM (Marsico, 2022), and 5G-ZORRO (Gavras et al., 2021) are examples of slice providers;
- **SlEaaS Provider:** introduces the alternative to provide the SlEaaS business model to tenants or slice providers. This new entity handles all three main aspects related to the CCL of slice elasticity: monitoring of the slice resources, policies or SLAs definition, and orchestration of the slice resources. The isolation of those operations from the SlaaS brings many benefits already presented in the Introduction.
- **Infrastructure Provider:** owns the physical or virtualized infrastructure offered to compose a slice. The two types of providers most known are: the *Data Center/Cloud (DC) Provider* providing computing resources (e.g., virtual machines, servers, containers, clusters) and the *WAN Provider* offering network resources (e.g., switches, virtual switches, routers, virtual routers, VNFs) capable of connecting the E2E slice. The infrastructure providers are not limited to the ones mentioned before. Depending on the purpose of the infrastructure, others can be offered, for example, RAN Providers, NFV Providers, and others (Sattar; Matrawy, 2020; Baba et al., 2022);
- **Administrative and Technological Domain:** maintains the resources under the same administration and may or may not provide different technologies for managing the slice infrastructure;
- **Slice Part (SP):** is the slice division per administrative domain, which means that one slice is composed of slice parts (one or more), and each slice part typically

corresponds to one infrastructure provider (DC, WAN, or other). Throughout the text, we named SPs for data centers as *DSPs* and *WSPs* for WAN Providers. Those concepts are essential because the architecture design relies on this division to deliver the SlEaaS properly.

2.2 Novel Enablers for Cloud Slicing (NECOS)

The NECOS project (Silva et al., 2018; Clayman et al., 2021b), was envisioned during the Fourth EU-BR Collaborative Call by the European Commission and the Brazilian National Network for Education and Research (RNP)[1]. The consortium was devoted to studying, proposing, and implementing a platform to enable the concept of CNS. Therefore, the objective of NECOS is to provide CNSs for tenants with features such as the automatic configuration of the infrastructure across multiple federated domains, service-independent operation, and slice adaptation to service needs. Additionally, it has to provide an autonomous platform for managing, monitoring, and orchestrating the slice resources and the internal components without depending on any action from the tenant.

The lightweight slice-defined cloud (LSDC) architecture was defined in NECOS and showcased the concept of CNS under a SlaaS business model that uses available cloud platform features and functions. In addition, LSDC introduces a new way to automate the cloud configuration process by being able to deploy CNSs split into slice parts across all the providers as a set of federated data centers or WAN, providing uniform management of computing, networking, and storage resources.

Our first responsibility at NECOS was to design and implement an Infrastructure Monitoring Abstraction (IMA) subsystem responsible for monitoring and managing services and infrastructure (physical and virtual). We also designed and implemented a minimalist version of a Slicing Resource Orchestrator (SRO) responsible for orchestration operations, such as verifying the monitored metrics and testing them against tenant policies to trigger elasticity operations, if necessary. We idealized those components from scratch and were the initial or first SEA design.

In this thesis, we constantly evolve the design of the SEA, starting by decoupling the SEA from the bond with the Slice Provider developed in NECOS. We then allow SEA to offer CCL to tenants who already own their slices, regardless of the Slice Provider. Furthermore, within the scope of SEA, we included orchestration features previously coupled to NECOS and not well explored, such as: the policy administration, the definition of external APIs to be consumed by tenants or slice providers, and the triggering and performing the slice elasticity. Another main difference from the work we started on NECOS is the type of feature supported. The NECOS project did not thoroughly explore

[1] http://www.h2020-necos.eu/.

the networking resource, focusing mainly on the computing infrastructure. However, for the SEA design, we also define network elasticity operations, proving the feasibility and generality of the architecture considering the slice resource heterogeneity.

2.3 Slice elasticity operations

2.3.1 Computing resources elasticities

One of the most critical operations in cloud network slices is slice elasticity. Moreover, the elasticity must consider that the slice is provisioned in a multi-domain and multi-technological environment, capable of dynamically *growing* or *shrinking* slice resources. Therefore, infrastructure changes at run-time require an architecture capable of monitoring and managing the slice resources or services to dynamically detect and provide these alterations based on predefined policies or SLAs.

In line with the resource virtualization (NECOS, 2019), we identify different elasticity operations related to how the slice resources (computing, networking, or storage) can grow or shrink in CNS. This subsection will define the elasticity operations in the context of CNS computing resources (e.g., computers, virtual machines, containers). We identify and propose the following four types of elasticity in this context: the Vertical Elasticity (VE) Scale Up, the VE Scale Down, the Horizontal Elasticity (HE) Scale Out, and the HE Scale In.

Vertical elasticity is the ability to dynamically resize slice parts as needed to adapt the slice infrastructure based on workload changes. This property expresses the ability to increase (*VE scale up*) the number of resources available inside a particular slice part when the demand for the services supported by the slice increases. The resources in the context of computing can be physical machines, virtual machines, containers, number of CPUs, memory, disk, and others. On the other hand, the *VE scale down* occurs when idle resources are identified and thus removed from a particular slice part.

Horizontal elasticity is the ability to create (*HE scale out*) or remove (*HE scale in*) slice parts dynamically, using resources from other providers, following the need to adapt to workload changes. For example, as the service workload increases and available slice part resources are not enough to cope with the demand, it may be possible to scale out resources by requesting the creation of new slice parts to additional infrastructure providers and then connecting them appropriately. The horizontal elasticity definition is the same for computing and networking resources. However, the VE operation is defined quite differently, as seen in the following subsection.

Figure 3 illustrates the previously defined types of elasticity, i.e., VE and HE. The former can be seen on the left side of the figure, where SP 1 resources are removed/added

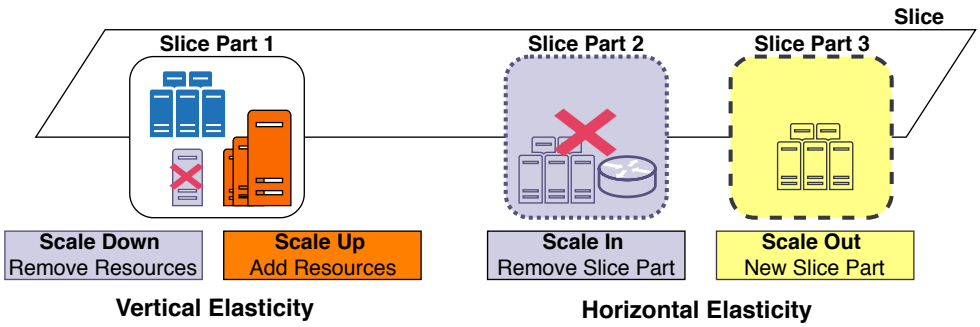


Figure 3 – Slice elasticity models in CNSs.

simultaneously (in purple and orange, respectively). The latter can be seen in SPs 2 and 3, where SP 2 was removed due to a HE scale-in operation (in purple), and SP 3 was added due to a HE scale out operation (in yellow).

Elasticity is more than just increasing (increasing/decreasing) or decreasing (decreasing/increasing) the CNS. In other words, SEA components (such as adapters) must be dynamically reconfigured according to the new slice components resulting from a VE or HE. In general, elasticity can be considered as an event that affects three different dimensions: (*i*) resource elasticity, (*ii*) service elasticity, and (*iii*) elasticity of SEA components. Therefore, to handle changes in service workload, you will need to scale up or down in these three dimensions. In this work, we highlight the essential role of SEA in reflecting the required changes during the CNS lifecycle management to support elasticity operations and real-time policy-making.

2.3.2 Networking resources elasticities

Nowadays, network elements (NEs) can be physical or virtual resources. Both have several open possibilities to be explored in the context of slice elasticity operations. For example, suppose that WSP 1 consists of virtual switches and routers on top of the network's physical infrastructure, which a WAN Slice Provider manages. Furthermore, the technologies to handle those elements can be proprietary, an open-source solution, and an OpenFlow controller. Therefore, the definition of the slice elasticity operations on networking is crucial considering this heterogeneity of administrative and technological domains native to the slicing concept.

We classify the elasticity operations in network resources equal to computing resources as *VE* or *HE*. The former adds/removes resources (e.g., switches, memory, routers, links, queues) in an existing slice part, while the latter instantiates/decommissions a whole slice part and all related resources. The slice elasticity operations of networking resources are described in Fig. 4. The figure shows the VEs and HEs possibilities on a slice that exemplifies a simple CDN service to broadcast videos and images.

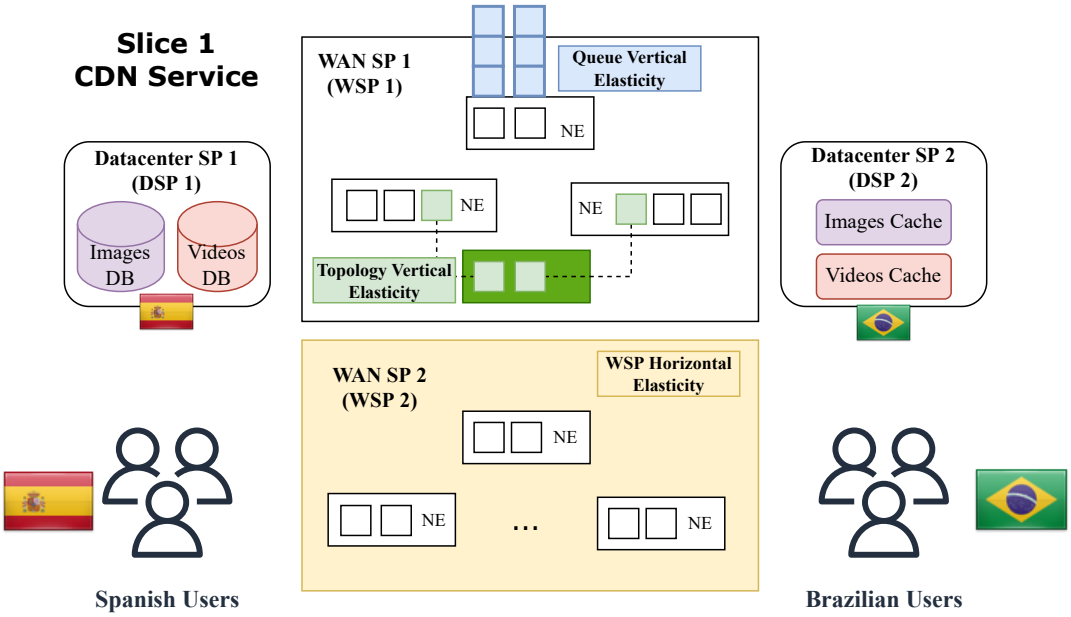


Figure 4 – Examples of slice elasticity operations of network resources.

The service consists of an end-to-end slice connecting databases and caches, initially instantiated over two DSPs across the ocean and one WSP. The purpose of this illustration is to exemplify a scenario where the number of Brazilian users increased significantly, requiring an improvement in the network performance to retrieve faster the videos and images from the storage located in Spain and cache the data on the DSP 2 situated in Brazil. Based on Fig. 4, we illustrate the following elasticity operations to improve the Brazilian users' experience:

1. **Queue vertical elasticity (highlighted in blue)**: creates (scales up) or removes (scales down) a queue on ports of switches to dynamically prioritize the video flows since they are commonly bigger than images;

2. **Topology vertical elasticity (highlighted in green)**: increases (scales up) or decreases (scales down) the network topology by adding or removing virtual NEs inside a WSP to improve the bandwidth or decrease the network latency of downloading video flows;

3. **WSP horizontal elasticity (highlighted in orange)**: adds (scales out) or removes (scales in) an entire network slice part. In case of an addition, the new WSP could have different requirements, such as new virtual elements with more bandwidth available or finding a WSP closer to the DSP to reduce the bottlenecks.

In the HE scale out, the tenant should negotiate with a WAN Slice Provider. The horizontal operations are more complex than the vertical ones since the new SP may have different requirements defined by the tenant in terms of bandwidth, location,

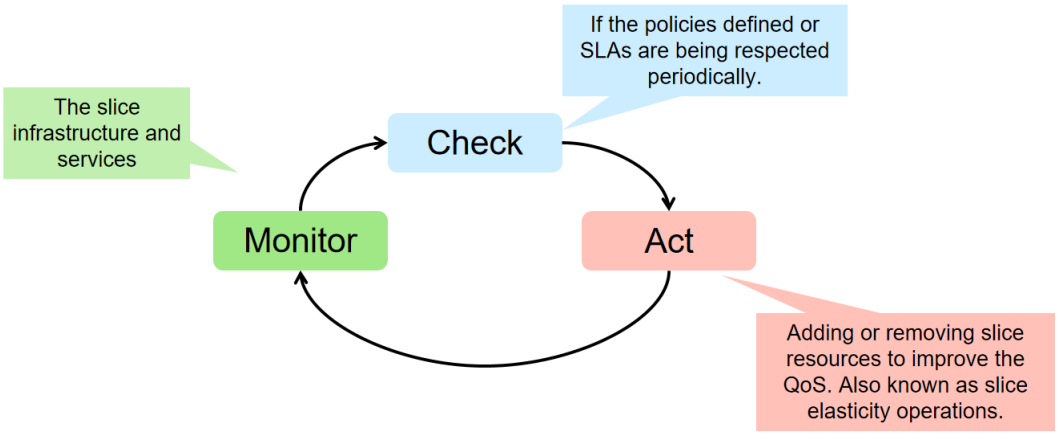


Figure 5 – Closed control loop.

latency, and others. After a WSP horizontal elasticity scaling out, the DSPs should be able to send traffic through this new slice part. The DSPs and WSPs connection is beyond the scope and responsibility of slice providers such as (Clayman et al., 2021a). We present only a few examples of slice elasticity in the context of networking resources. However, more possibilities could be defined considering other networking technologies (e.g., programmable data planes). From the SEA perspective, new slice elasticity operations must be transparently supported due to the MAs abstraction.

2.4 Closed control loop (CCL)

The CCL in this work consists of constantly monitoring and verifying the slice infrastructure metrics being monitored so that the SLAs defined by the tenant continue to be respected. A representation of the CCL can be seen in Figure 5 composed of 3 steps: Monitor (in green), Check (in blue) and Act (in pink).

The Monitor in the context of this work is related to constantly gathering the slice infrastructure metrics, as well as, the service metrics persisting that information. The second step in the CCL is responsible for checking periodically the metrics stored based on the SLAs and policies defined by the tenant. Soon after a policy is violated, an action must be taken, such action in response to a violation is the responsibility of the third part of the closed control loop, called Act. In this work, the main action to be performed in order to improve QoS is to increase or decrease the slice resources, by performing an elasticity operation.

2.5 Slicing Elasticity as a Service (SlEaaS)

The SlEaaS concept is a new business model defined in this thesis and one of our contributions. The SlEaaS follows other well-known business models such as PaaS, SaaS, and IaaS. The idea is to have entities capable of offering the slicing resources CCL as a service to tenants or slice providers, regardless of who instantiates those resources.

The SlEaaS starts after the slice instantiation phase and comprises the deployment of the components needed to monitor and manage the slice resources, the creation of policies, and the possibility of triggering and performing slice elasticity operations manually or automatically based on the policies defined. The tenant can delegate those operations to SlEaaS and use the monitored metrics and the management functionalities as she wants.

This new business model could be the future for orchestrating CNS resources in a transparent, automated, personalized, and scalable way. In addition, this abstraction makes life easier for the tenant and slice provider. We explore a crucial aspect of delegating and automating such operations.

2.6 Technologies enablers

This section details the following three essential enablers of the slicing concept: Virtual Infrastructure Managers, WAN Infrastructure Managers, and Monitoring Entities. Those core technologies make the proposed solution possible because they are crucial for managing and monitoring the slice infrastructure considering resource heterogeneity.

2.6.1 Virtual Infrastructure Managers (VIMs) and WAN Infrastructure Managers (WIMs)

VIMs and WIMs have been used for a long time to manage cloud computing infrastructures and WAN networks, respectively. Therefore, due to the overlapping nature of the slice infrastructures, it is also expected that such technologies are fundamental to enable the slices with different administrative and technological domains. Furthermore, many technologies can be considered VIMs or WIMs in different segments. Such technologies are essential to manage and orchestrate the slice infrastructure. Based on what was defined in the NECOS project, we assume that during the slice instantiation phase (which is not in the scope of this work), the tenant must request the VIMs and WIMs that she wants to manage his slice. This concept is called VIM/WIM on demand and originated in the NECOS (NECOS, 2019).

It is important to note that the ratio of VIMs and WIMs is: each WAN slice part must have 1 WIM, and each DC/Cloud slice part must have 1 VIM. So, for example, assuming a tenant requested a slice with three slice parts being 1 WSP and 2 DSPs. During

the slice instantiation phase, she must inform the VIMs or WIMs technologies on each SP. So, for example, the OpenFlow Ryu controller is the WIM of the WSP, the OpenStack is the VIM of the first DSP, and the Kubernetes is the VIM of the second DSP. This same behavior happens with the MEs (one ME for each slice part) that will be detailed in the following subsection.

The architecture design is based on adapters that abstract technologies of VIMs, WIMs, and MEs. The adapters are crucial to support such heterogeneity of slice resources and technologies. This subject will be detailed during the presentation of the proposal. Therefore, we present only a summary of these important components in our proposal. Specific adapters for particular VIMs and WIMs are required to support multiple technologies, usually providing different levels of abstraction and several features.

We will name the adapters that abstract the implementations of the different VIMs or WIMs as Management Adapters (MAs), they provide a primary wrapping-agnostic service platform to standard SlEaaS functions. In practice, the adapters translate generic calls into specific commands or methods in the context of specific VIMs and WIMs. Typical VIM and WIM technologies include:

- **Cloud VIMs:** Openstack[2], Heat[3], Kubernetes[4], VLSP[5], Docker Swarm[6], Openshift[7];
- **VNF VIMs:** OpenMANO[8], Open Baton[9], OPNFV[10];
- **WIMs:** Programmable data plane solutions, SDN controllers, such as Opendaylight[11], Floodlight[12], Ryu[13].

2.6.2 Monitoring Entities (MEs)

We can see the MEs as monitoring tools well-known or not used to monitor resources. Again, the resources can be of any type, including the MEs. From the SEA perspective, collecting the slice infrastructure metrics must be abstracted regardless of the ME chosen by the tenant during the slice instantiation. The only premise the SEA need is: we must reach the MEs to collect the metrics monitored periodically based on the time interval predefined.

[2] https://www.openstack.org/.
[3] https://docs.openstack.org/heat/latest/.
[4] https://kubernetes.io/.
[5] https://clayfour.ee.ucl.ac.uk/downloads/usr_guide_book.pdf.
[6] https://docs.docker.com/engine/swarm/.
[7] https://www.redhat.com/pt-br/technologies/cloud-computing/openshift.
[8] https://osm.etsi.org/.
[9] https://openbaton-docs.readthedocs.io/en/3.0.0/.
[10] https://www.opnfv.org/.
[11] https://www.opendaylight.org/.
[12] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview.
[13] https://ryu-sdn.org/.

Therefore, the MEs are responsible for monitoring each slice part infrastructure. They must be instantiated during the slice instantiation phase because we assume that the SEA may not interact with the infrastructure providers to instantiate the VIMs, WIMs, and MEs inside the provider. The idea of SEA is to abstract those technologies interacting with them rather than deploying them in each infrastructure provider to provide the CCL. This abstraction, as already explained, is made by the adapters implementations, which must be implemented in the SEA scope to interact with different MEs. In this thesis, the adapters that abstract the MEs are called Telemetry Adapters (TAs).

To materialize the MEs, some examples are presented below:

- **Cloud MEs:** OpenStack Ceilometer[14], OpenStack Gnocchi[15], Prometheus[16], NetData[17], Datadog[18], Graphite[19] and others;
- **Network MEs:** sFlow[20], SNMP[21], NetFlow[22], Nagios[23], OpenFlow protocol, in-band telemetry using programmable data plane, and others.

[14] https://docs.openstack.org/ceilometer/latest/.
[15] https://wiki.openstack.org/wiki/Gnocchi.
[16] https://prometheus.io/.
[17] https://www.netdata.cloud/.
[18] https://www.datadoghq.com/.
[19] https://graphiteapp.org/.
[20] https://sflow.org/.
[21] https://www.techtarget.com/searchnetworking/definition/SNMP.
[22] https://www.kentik.com/kentipedia/what-is-netflow-overview/.
[23] https://www.nagios.org/.

3 Related Work

3.1 Projects and initiatives

This section summarizes a set of relevant network slicing standardization efforts and research projects. First, we present a qualitative comparison of selected projects considering the following (non-exhaustive) list of six key characteristics:

1. **E2E:** Does the project or initiative consider the end-to-end connectivity of the slice resources to provide services?
2. **Multi-domain:** Does the project or initiative support the multiple administrative and/or technological domains?
3. **Slice lifecycle management:** Does the project or initiative realize the lifecycle management of slice resources?
4. **Tenant slice management:** Does the project or initiative enable the slice resources management by tenants?
5. **SlEaaS:** Does the project or initiative provide slice elasticity as a service?
6. **Virtual Infrastructure Manager (VIM)/WAN Infrastructure Manager (WIM) on demand:**: Does the project or initiative offer the possibility of requesting specific VIMs or WIMs on demand for each slice part?

Table 1 relates the aforementioned six characteristics to six slicing standardization initiatives (blue), seven European 5G Infrastructure Public Private Partnership (5GPPP) projects (yellow), and the proposed SEA (red). The green ticks indicate that the corresponding work considers the applicable slice characteristic in its scope.

The initiatives considered in this analysis were as follows: (i) ITU Telecommunication Standardization Sector (ITU-T) (ITU-T, 2012; Group, 2017), (ii) NGMN (NGMN, 2015; Hedman; Team, 2016), (iii) Internet Engineering Task Force (IETF) (Galis et al., 2017; Geng et al., 2018a; Farrel et al., 2021; Homma et al., 2019; Galis, 2017; Geng et al., 2018b; Makhijani et al., 2017; Qiang et al., 2018b; Qiang et al., 2018a; Galis et al., 2018), (iv) 3GPP (3GPP, 2015; 3GPP, 2016b; 3GPP, 2016a; 3GPP, 2016c; 3GPP, 2016d; 3GPP, 2017b; 3GPP, 2017a), (v) ETSI (ETSI, 2017a; ETSI, 2017b; ETSI, 2018), and (vi) Open Networking Foundation (ONF) (Foundation, 2016). The EU 5GPPP projects studied

were as follows: (i) 5GEX[1], (ii) 5G-SONATA[2], (iii) 5G-NORMA[3], (iv) 5G-Transformer[4], (v) 5G-PAGODA[5], (vi) 5G-Slicenet[6], and (vii) the SEA. Our main goal was to compare the proposed architecture with relevant slicing initiatives and similar projects. More details of the architectural survey on network slicing are available in (Galis et al., 2020).

Most projects and initiatives use the *network slicing* concept. However, in the literature, we have different definitions for this term. Therefore, in the context of this proposal, we adopt the term cloud-network slicing as a set of networking, computing, and storage resources. Currently and still in definition, IETF is the only standardization body that defines the term network slicing comprising networking, computing, and storage resources (Farrel et al., 2021).

The closest related project is the 5G-EX, which considers three key characteristics presented before. The analysis presented in Table 1 conveys a more flexible conceptualization of slicing, which goes beyond most of the presented initiatives' peer-to-peer orchestration and management interactions. The proposed architecture, SEA, is based on this conceptualization, introducing and designing components that allow for a slicing elasticity approach that covers the highlighted characteristics. Our proposal is the first in the literature that defines and details the concept of SlEaaS. The characteristics mentioned above are essential for the architecture design since they are requisites of the slice concept. Therefore, reinforcing the contributions and innovations of this thesis to the community and slicing initiatives.

Initiatives/ EU Projects	E2E	Multi-Domain	Slice Lifecycle Management	Tenant Slice Management	SlEaaS	VIM/WIM on demand
ITU-T	×	×	×	×	×	×
NGMN	×	×	×	×	×	×
IETF	√	√	√	√	×	×
3GPP	×	×	√	√	×	×
ETSI	√	×	×	√	×	×
ONF	×	×	×	×	×	×
5G-EX	×	√	√	√	×	×
5G-SONATA	×	×	√	√	×	×
5G-NORMA	×	×	√	×	×	×
5G-Transformer	×	×	√	√	×	×
5G-PAGODA	×	×	√	√	×	×
5G-SliceNet	×	×	√	×	×	√
SEA	√	√	√	√	√	√

Table 1 – Summary of key characteristics for slicing initiatives, European projects, and the NECOS project.

[1] http://www.5gex.eu/.
[2] http://sonata-nfv.eu/.
[3] http://www.it.uc3m.es/wnl/5gnorma/.
[4] http://5g-transformer.eu/.
[5] https://5g-pagoda.aalto.fi/.
[6] https://slicenet.eu/.

3.2 Slice monitoring

Although topics related to the monitoring and management of cloud and network infrastructures are widely studied in the literature, we could not find any work in the context of cloud network slices and all novelties raised by this new entity. Therefore, the papers presented in this section focus on proposals that are somehow related to ours, as they involve resource monitoring and management or even different slice concepts.

The survey in (Fatema et al., 2014) presents an overview of the term monitoring with an emphasis on cloud environments and data centers. This work aims to qualitatively compare some of the most used monitoring tools, considering some monitoring pillars, such as scalability, portability, multi-tenant environment, and the ability to adapt according to the tenant. These characteristics, among others mentioned in the article, should be considered to meet tenants' needs better and incorporated into the context of CNS.

DASMO (Kuklinski; Tomaszewski, 2018) proposes an extension of the MANO architecture to support network slicing in the context of the management and orchestration of virtual network functions. The paper discusses possible changes to the MANO architecture to encompass the concept of network slicing and issues regarding the monitoring aspects (e.g., scalability). The authors describe in detail the proposed components and their responsibilities. In addition, qualitative aspects of the presented proposal are discussed. However, neither a quantitative analysis nor a proof of concept implementation of the architecture is presented.

The work presented in (de Carvalho et al., 2013) proposes a framework for cloud monitoring. The idea is to provide a self-configured monitoring system for cloud computing, particularly running OpenStack and different monitoring tools, for example, native cloud monitoring solutions (i.e., OpenStack Ceilometer) and non-cloud monitoring solutions (i.e., Nagios and MRTG). Furthermore, this framework can automatically create monitoring slices corresponding to the cloud infrastructure allocated in the OpenStack environment. However, the slicing concept in this work is restricted to single clouds, leaving out of scope key aspects such as multi-domain and heterogeneity of resources (computing, network, storage).

The slice monitoring approach presented in (Tusa; Clayman; Galis, 2019) provides an architecture for monitoring end-to-end slices and a related software implementation specifically focused on the DC slice parts. However, while the paper discusses the need to ensure slice monitoring abstractions and the dynamic instantiation of monitoring adapters, it needs to address the research problems arising from slice elasticity operations as already mentioned in this text.

3.3 Slice management

The focus of this section is to show the related works and initiatives for the management and orchestration of slices resources. In the 5G-SONATA project (Bonnet, 2016), the authors describe a prototype version of the service programming and orchestration for virtualized software networks, an orchestrator that allows programmability for 5G networks. This proposed orchestrator uses an open-source monitoring model that presents several desirable features, such as flexibility to operate on different microservice platforms, scalability, and customization. However, it does not consider a multi-domain environment and cannot label metrics or group them in ways required by business models.

Lattice[7] is an intelligent monitoring system presented by Tusa et al. (Tusa; Clayman; Galis, 2018). This tool proposes a monitoring framework for virtualized resources, services, and network elements, geared toward cloud computing systems. The framework monitors a cloud computing environment by collecting metrics and detecting increases or decreases in traffic, with configurable setup parameters such as the interval at which metrics are collected and sent to an orchestrator. However, this framework does not support all the roles of a monitoring solution in the context of CNSs. For example, the multiple technological domains and the ability to communicate with several monitoring tools.

The authors in (Tusa et al., 2018) unify components developed within the 5GEX and 5G-SONATA projects to orchestrate the deployment and management of Virtual Network Functions (VNF) services using the Very Lightweight Network & Service Platform (Mamatas; Clayman; Galis, 2015) (VLSP) as a VIM, allowing multi-domain orchestration of slices. Despite this being a proposal in the context of slices, we can identify the following differences between our work and the one described. (i) The architecture presented by the authors encompasses only the VLSP as a VIM; it does not deal with heterogeneity in terms of different VIMs or WIMs in the same end-to-end infrastructure. (ii) The SEA considers the management of the infrastructure and services and presents the monitoring of both physical and virtual resources per slice.

Cloud computing management has been widely explored in the literature. However, it is necessary to map these studies to the concept of CNSs, which demands the analysis of several topics, such as cloud computing, network, and storage. Two surveys found in the literature helped develop the proposed architecture. The first one presented in (Manvi; Shyam, 2014) focuses on the broad study area of cloud management and its main challenges. The authors classify eight functional areas in the field: global scheduling of virtualized resources, resource demand profiling, resource utilization estimation, resource pricing and profit maximization, local scheduling of cloud resources, application scaling and provisioning, workload management, and cloud management systems. In the second

[7] http://clayfour.ee.ucl.ac.uk/lattice/.

survey (Jennings; Stadler, 2014), the authors also classify different functional areas in the field, identifying several works in the literature for each of them and raising open research questions. The theoretical basis of these surveys was used by adapting some of the characteristics explicitly presented for cloud computing to the context of the cloud-network slicing concept.

We want to highlight the following three papers. The first work (Montero et al., 2020) presents how to provision and maintain 5G services over network slices with a focus on network elements and Network Function Virtualization (NFV). The second paper (Chiha et al., 2020) focuses on a techno-economic analysis to provide a cost allocation model to network slices. Finally, the authors in (Valera-Muros et al., 2020) propose an architecture responsible for creating network slices for wireless networks. The work presented in this thesis goes deeply into the CNS topic and presents the SEA design with contributions such as slice elasticity, the multiple administrative and technological domains, and the PoCs in real environments.

3.4 Slice elasticity

In literature, we seek works that discuss the orchestration of slicing resources and present the capacity to grow/shrink resources dynamically. However, to the best of our knowledge, we did not find works that advance this area in the aspects presented. Therefore, this section will present a qualitative comparison of the related works considering the five main aspects presented in Table 2, which are:

- **Slicing:** Does the work considers the slicing concept?
- **Cross-Domain:** Does the work relies on a cross-domain infrastructure?
- **CCL:** Does the work present a closed control loop for slice orchestration?
- **Elasticity:** Does the work discusses slice elasticity operations?
- **SlEaaS or similar:** Does the work propose an architecture to support SlEaaS or similar business models?

	Slicing	*Cross-domain*	*CCL*	*Elasticity*	*SlEaaS or similar*
D. Luong et al. (Luong; Outtagarts; Ghamri-Doudane, 2019)	√	×	×	√	×
B. Turkovic et al. (Turkovic; Nijhuis; Kuipers, 2021)	√	×	×	√	×
L. Sanabria-Russo et al. (Sanabria-Russo; Verikoukis, 2021)	√	√	×	×	√
D. Breitgand et al. (Breitgand et al., 2021)	√	√	√	√	×
K. Slawomir et al. (Kukliński et al., 2021)	√	√	√	√	√
ETSI OSM[8]	√	×	√	√	√
SEA	√	√	√	√	√

Table 2 – Slice elasticity qualitative comparison.

D. Luong et al. (Luong; Outtagarts; Ghamri-Doudane, 2019) proposes multilevel scheduling for slicing in the context of 5G. The author's focus is to schedule the placement of slice resources considering three phases, slice creation, service scaling, and slice removal considering a single domain. The elasticity operation performed is simple, only increasing the number of service instances, and quite different if compared with this proposal. Our proposal focuses right after the slicing instantiation phase in a cross-domain environment with multiple Infrastructure providers, providing elasticity as a service capable of orchestrating the slicing resources that can be shared between the tenants.

As mentioned, programmability and softwarization of network resources enable the slicing concepts. The authors at (Turkovic; Nijhuis; Kuipers, 2021) propose elasticity operations for slicing, considering programmable switches as network functions (supporting P4). The slicing concept for the authors is a set of network functions instantiated over a physical network. The elasticity operations proposed include increasing/decreasing (scale-up and scale-down) the bandwidth between the network functions instantiated and duplicating (scale-out for the authors) a network function over other physical elements adding new virtual links between the network functions already instantiated. The definitions used in the paper are similar to the ones presented in Chapter 2. However, we support any technology combination for each infrastructure provider (P4, open vSwitch, OpenFlow), increasing the possibilities, complexity, and novelty of horizontal elasticities covering cross-domain slices.

D. Breitgand et al. (Breitgand et al., 2021) discuss the challenges of cross-domain slice scaling in the context of 5G network slices. The paper is part of the 5GZORRO, a European project to provide an architecture and platform focused on cross-domain interactions between Communication Service Providers (CSPs). The main difference between this paper with ours is related to which resources will be scaled to improve the service quality. The authors are scaling the slice similarly to the horizontal elasticity presented in Section 2.3. However, we propose intermediate levels of elasticity considering the vertical elasticities on already deployed slice parts, and as the last option, renegotiate a new slice part with new infrastructure requirements defined by the tenant and orchestrated for our service. In addition, the following differences should be highlighted: the SEA provides elasticity operations based on policies defined by the tenants, the SEA design supports multiple slices from different tenants at the same time, and all the architecture components can easily be deployed, terminated, or updated because they are designed following a microservices approach.

Sanabria-Russo L. and Verikoukis C. in (Sanabria-Russo; Verikoukis, 2021) propose a monitoring framework for 5G network slices on top of a Kubernetes infrastructure to monitor mainly instances of NFVs. The proposed framework can be used to provide the metrics of a CCL similar we described in this thesis. However, some important aspects are

not covered in this proposal, for example, the slice infrastructure resources heterogeneity, and the slice elasticity operations are not part of the scope of this paper. Therefore, we presented an architecture more generic and technologically independent to provide not only the monitoring of CNS but also the management and orchestration through the slice elasticity operations.

The authors on (Kukliński et al., 2021) present an architecture focusing on 6G Network Slices based on the 5G network standardized by 3GPP. As in this work, right from the start, they point out that slice management and orchestration are crucial open challenges. The proposed architecture was designed modularly, like the one in this work, and comprises essential characteristics inherent to slices. There are several similarities in both architectures; for example, what they call a sub-network slice, we call slice parts. In addition, the authors presented a concept called Management as a Service (MaaS) which has similarities with the SlEaaS. They also discuss other possible elasticity operations in the context of slicing. The idea of the architecture and concepts presented are well aligned with those presented in this work. However, we went further at SEA as we not only define but implemented, and evaluated the SEA on real slices considering the slice resources heterogeneity, as well as proposing the native support of this novelty business model called SlEaaS. In other words, the SEA is completely decoupled from any initiative and project enabling the SlEaaS for any tenant or slice provider.

One of the most related works is the platform ETSI OSM (Marsico, 2022) (Open Source MANO), which can act as a Network Service Orchestrator controlling the lifecycle of Network Services and Network Slices offered on demand as a service northbound. The OSM provides several functionalities and a well-defined user interface to handle the resources based on interfaces that communicate with Virtual Infrastructure and WAN Infrastructure managers. The elasticity operations on OSM are limited to increasing/decreasing the VNFs, but in our proposal, we define different types of elasticity based on the resource and technologies. Tenants can use our well-defined interfaces to have a complete closed loop executed to ensure the SLAs required by that slice, customizing the policies and the monitoring parameters according to their needs. This proposal can work on top of infrastructures being instantiated/managed by the OSM, providing all the contributions already mentioned.

4 SEA

4.1 SEA design

This section presents the architecture components to provide the *Slicing Elasticity as a Service*. The SEA is responsible for managing the CCL to trigger the elasticity operations and ensure the SLAs defined by the tenant. In addition, the SEA provides a standard policy API, including the CRUD (create, read, update and delete) operations. Two clients can benefit from using the SEA: a tenant who created a slice to provide a specific service for end users and wants to delegate the slice resource orchestration; and slice providers that handle the slice creation and management on behalf of tenants. However, the nuances of slice providers regarding slice infrastructure instantiation are out of the scope of this work.

Figure 6 illustrates the SEA, including the interactions among main components, actors, and APIs needed to support the SlEaaS. The SEA comprises three layers, the *API Layer* (in orange), the *Control Layer* (in red), and the *Per-Slice Components Layer* (colored by slices). The integration with the SEA assumes two premises related to what must have on SPs: (*i*) a Virtual Infrastructure Manager (VIM) or WAN Infrastructure Manager (WIM); and (*ii*) a Monitoring Entity (ME). We also assume that the tenant can choose which solutions for VIM/WIM and ME, and the Slice Provider must provide them during the slice instantiation. For the SEA, the important is the APIs for whichever solutions were chosen. This loosely-coupled approach is typically enforced to orchestrate the elasticity, keeping the separation between providers and the SEA.

The architecture contains fixed components that are shared among clients (APIs, *SEA Controller*, and *Policy Controller*), as well as dynamic ones instantiated for each slice. The white circles in Fig. 6 represent the dynamic components, namely *Management Adapter* (MA), *Telemetry Adapter* (TA), and *Database* (DB). The MAs are responsible for abstracting the interaction with the different VIMs/WIMs to manage physical or virtual resources provisioned for the corresponding SPs. Similarly, the TAs collect metrics from the MEs and store them in the slice database DB.

The main idea is to have one adapter for each technology used over the SPs. For instance, the DSP 1 of Slice 1 (blue) may use OpenStack as the VIM and Prometheus as the ME, while the WSP 2 of the same slice may use an OpenFlow controller and SNMP as WIM and ME, respectively. On the other hand, the (green) Slice 2 and the (yellow) Slice 3 can use completely different sets of solutions once heterogeneity is guaranteed by using adapters.

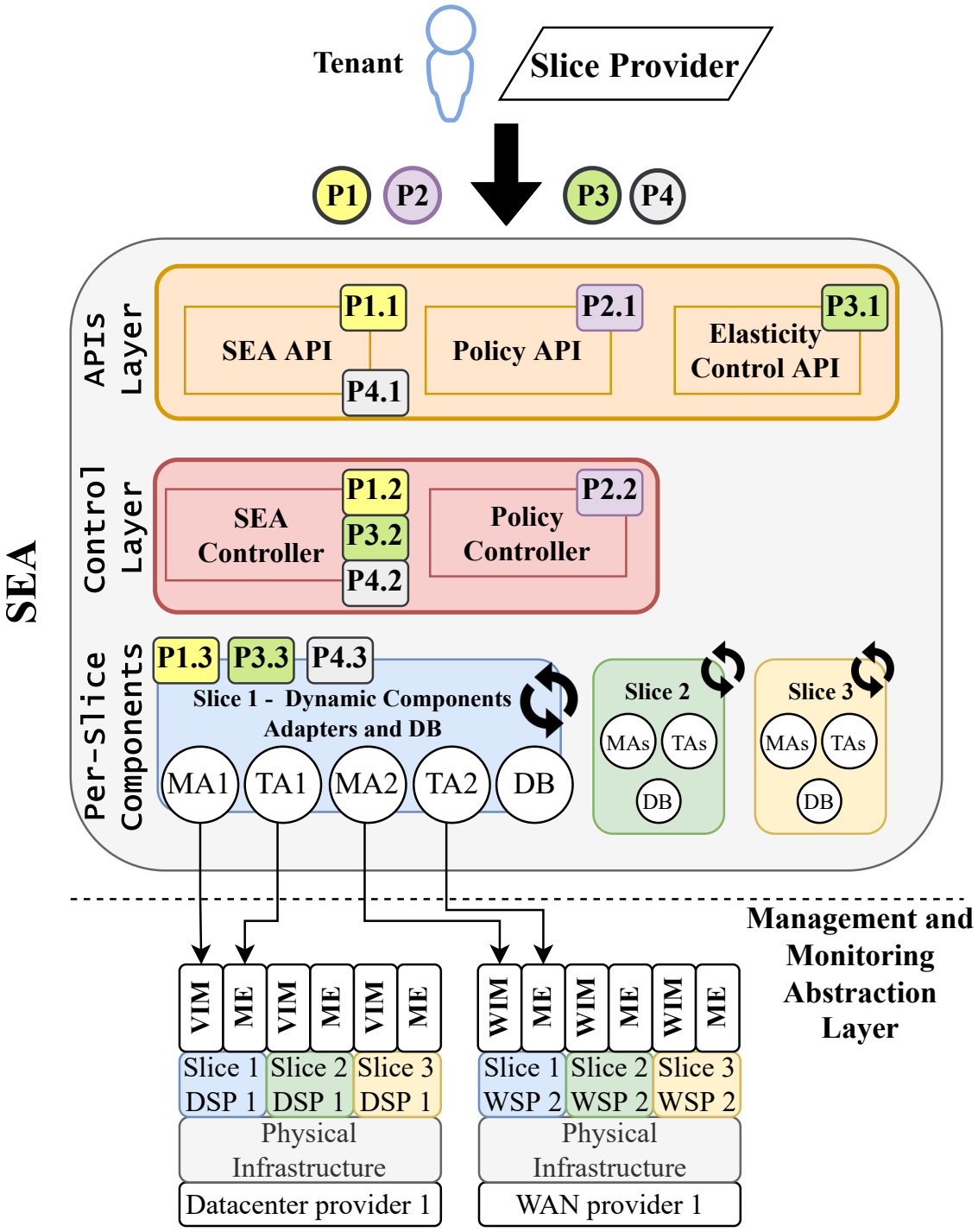


Figure 6 – SEA main components and workflows.

Figure 6 also illustrates the SEA workflow, which is described in phases and encompasses the CCL responsible for providing the elasticity operations. The high-level explanations of each phase are as follows, starting right after slice instantiation.

- **Phase 1 (P1) - SEA components instantiation**
  The client (tenant or Slice Provider) calls the SEA API to request the creation of the per-slice components through a well-defined API (P1.1). The SEA Controller

receives the request (P1.2) and creates the resources/adapters to orchestrate that slice (P1.3) isolated from other slices. In practical terms, each one of the dynamic per-slice components (MAs, TAs, and DB) is created and instantiated inside an individual container.

- **Phase 2 (P2) - Policy/SLA creation**
  Regarding the CCL orchestration, the clients must follow predefined methods and parameters of the Policy API to request the CRUD operations (P2.1). In addition, the Policy Controller component creates the checking mechanism of SLAs (P2.2), which alerts may trigger (if available) created directly on the database or by a pooling scheme to periodically check the database and verify whether the metric has been violated.

  Six groups of parameters are essential for the policy creation process: (1) the slice id plus slice part id; (2) the metric to be monitored; (3) the threshold used to trigger an operation; (4) the policy violation check interval; (5) the elasticity operation that should be enforced after a trigger; (6) tenant-provided extra information about the operation to be executed (e.g., characteristics of new queues, topology extensions, slice part attributes).

- **Phase 3 (P3) - Policy/SLA elasticity action**
  Any time after P1, the client can explicitly call an elasticity operation through the Elasticity Control API without any automation (P3.1). The next step occurs differently depending on the elasticity operation requested. The SEA Controller performs the elasticity operation in a non-automated manner (P3.2) or automatically triggered (P3.2) due to a policy violation, which was identified by the checking mechanism of SLAs. Either way, the SEA Controller will use the MA if vertical elasticity is needed (P3.3). Otherwise, if a horizontal elasticity is needed, the Slice Provider is invoked to instantiate a new SP (scale out) or to decommission an existing SP (scale in).

- **Phase 4 (P4) - SEA dynamic components adaptation**
  After a horizontal elasticity operation, the Slice Provider uses the SEA API to adjust the adapters to reflect the infrastructure changes (the addition or removal of a new SP) since such an operation directly impacts the SEA per-slice components. Therefore, the SEA API (P4.1) should be called to instantiate new MA/TA components (scale out) or to remove the adapters (scale in). The communication between the SEA API and the SEA Controller (P4.2) is responsible for instantiating the new MA and TA adapters (P4.3). The horizontal elasticity is the only operation that requires those adaptations. When a vertical elasticity occurs, the adapters already instantiated are enough to monitor and manage an existing SP's resources (e.g., VMs, NEs, containers).

4.2 SEA components responsibilities

This section aims to present the responsibilities and details of the SEA components. The idea is to point to the reader how generic the proposed architecture is, enabling its use for any tenant or slice provider as long as the premises and API parameters are respected during implementation. The idea of SEA is that SlEaaS providers make the APIs and components described available to tenants or slice providers who want to have the elasticity of their slices as a service in a simplified and transparent way. The SlEaaS provider is responsible for implementing details, technologies used, and which methods are available, as well as proposing changes and improvements. Although in the subsequent implementation chapters, we will detail the internal and external APIs parameters and the development and integration of the components shown below, the implementation is only an example of the described characteristics. The technologies used can change for each SlEaaS provider and work similarly.

4.2.1 APIs layer

Nowadays, to guarantee the generality of architectures, it is crucial that communication interfaces, such as northbound or southbound interfaces, are very well defined. Such a definition must include the responsibilities, parameters, and methods for external or internal use. One of the significant contributions of this work is the definition of interfaces or APIs. Based on what we researched in the works and initiatives in the context of slices and what we identified as essential to providing slice elasticity as a service, we present the three external APIs: SEA API, Policy API and Elasticity Control API.

Starting with the required SEA API responsibilities, it is vital for managing the per-slice components. This API provides interfaces so the tenant can interact with the VIMs and WIMs from the instantiated adapters. Furthermore, adjusting the adapters as the slice infrastructure increases or decreases after a horizontal elasticity operation is essential. The Policy API handles the policy CRUD as well as the creation of the policy check mechanism. Finally, the Elasticity Control API allows the tenant or slice provider to manually trigger defined elasticity operations without having a pre-established policy. The APIs description presents a high-level view of activities. Further, the implementation description will reflect those as methods or endpoints.

Figure 7 shows examples of functionalities of each API described above. In the figure, the *other* means that the API can support any other functionality. It is the responsibility of each SlEaaS provider to decide which features they will provide to the tenants. Following, we describe the main responsibilities of the SEA components.

- **SEA API: Components administration**

  - *start SEA (*):* responsible for instantiating all the per-slice components needed to be able of monitoring and managing the slice requested.
  - *decommission SEA (*):* responsible for decommissioning all the per-slice components of the slice requested.
  - *scale SEA (*):* responsible for adjusting the per-slice components adding or removing them based on a horizontal elasticity operation.

- **SEA API: Per-slice components management**
  - *update MAs/TAs/DB:* responsible for updating any parameters related to the MAs, TAs, or the DB previously chosen by the tenant. The update is specific for each technology. Therefore, the adapters or database must be redeployed based on the technology and the requested change. For example, updating the metrics monitored should be fine without the TA re-deploy. However, changing the ME in that SP should require the TA to be redeployed.
  - *list per-slice components:* responsible for listing all the per-slice components attributes instantiated for that slice.
  - *query DB:* responsible for providing the ability to query all the data being monitored for the requested slice. This functionality should abstract the database technology and return the data agnostically.
  - *other:* represents any functionality that can help the management of the per-slice components. Example: disable the adapters without removing them and enable the adapters back.

- **SEA API: VIM/WIM Abstraction**
  - *deploy service:* responsible for instantiating the service inside the slice requested infrastructure. The tenant can use the SEA MAs to abstract the calls for that and deploy the service interacting directly with the VIMs/WIMs without using the SEA API provided.
  - *create VM/container:* responsible for creating VMs or containers for the requested slice or slice part.
  - *re-route flow:* responsible for creating control plane rules to re-route or prioritize a specific application or flow.
  - *other:* represents infinite possibilities that are very coupled with the VIM/WIM chosen. Examples: delete VM/container, decommission service, list topology, list control plane rules, update control plane rules, etc.

- **Policy API**

  - *create policy (*):* responsible for creating the policies for the specific slice.
  - *delete policy (*):* responsible for deleting the policies for the specific slice.
  - *update policy (*):* responsible for updating policies attributes for the specific slice.
  - *list policy:* responsible for listing the policies for the specific slice.
  - *disable policy:* responsible for disabling the policies for the specific slice without excluding them.
  - *enable policy:* responsible for enabling the policies previously disabled by the management of the per-slice component.
  - *unlock policy:* responsible for unlocking the policies for the specific slice. The policy can be locked by the SEA controller when triggered consecutively. The tenant can unlock the policy manually or define an expiration time to unlock the policy automatically.

- **Elasticity Control API**
  - *VE queue scale up (*):* responsible for manually triggering the VE queue scale up.
  - *VE queue scale down (*):* responsible for manually triggering the VE queue scale down.
  - *VE topology scale up (*):* responsible for manually triggering the VE topology scale up.
  - *VE topology scale down (*):* responsible for manually triggering the VE topology scale down.
  - *VE scale up (*):* responsible for manually triggering the VE scale up.
  - *VE scale down (*):* responsible for manually triggering the VE scale down.

It is essential to highlight that the Elasticity Control API only provides the VE operations to be triggered manually by the tenant. The Slice Providers may be responsible for performing the HE and connecting/disconnecting the WSP from the DSPs. The tenant will use the SEA API to adjust the per-slice components right after the HE completion by the slice provider. The idea is to discuss the possibilities of the SEA design and all the APIs that can be expanded, improved, or changed by the SlEaaS providers.

4.2.2 Control layer

The Control Layer is the brain of the SEA design. In this layer, all the logic and complex tasks are made basically by two components: the SEA Controller and Policy

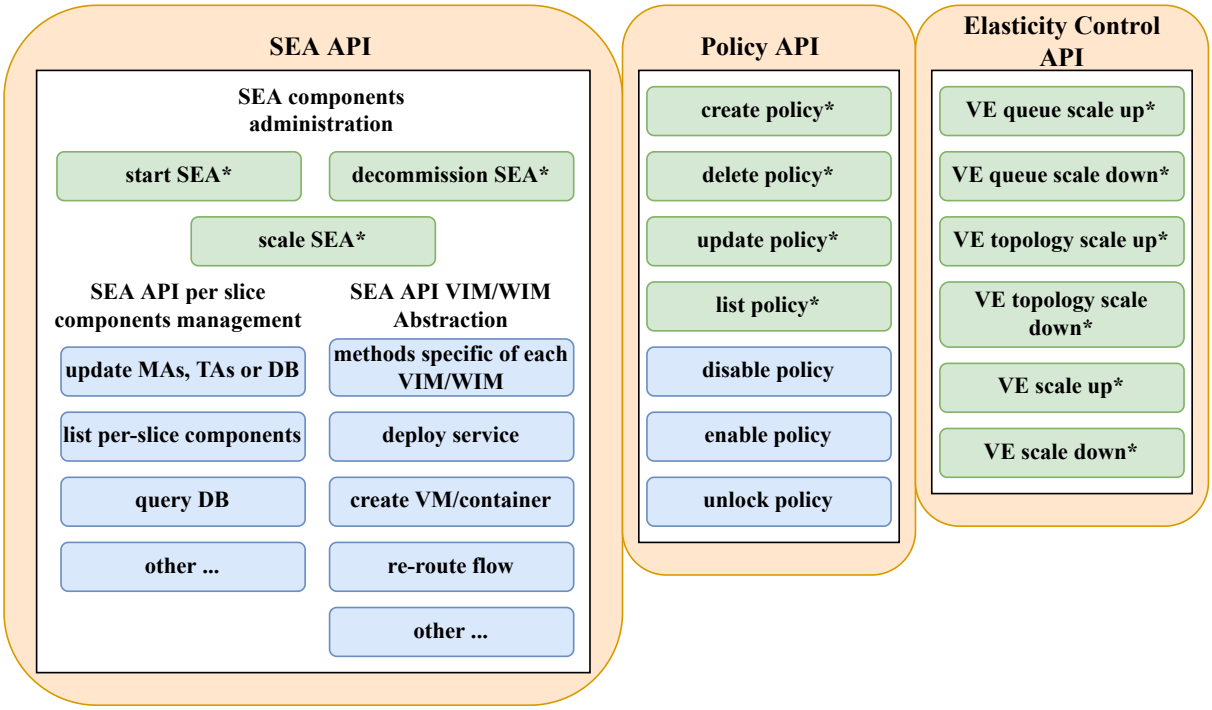


Figure 7 – SEA APIs high-level description.

Controller. In addition, the Control Layer abstracts the calls to the per-slice components. The components of this layer handle all the per-slice components and policies for all slices managed by the SlEaaS provider.

The SEA Controller is called not only by the APIs but also by the Policy Controller to trigger a VE in response to a policy violation. The SEA Controller abstracts all the SEA APIs and Elasticity Control APIs. This means that the logic of those functionalities is located at the Control Layer. In addition, the SEA Controller is the main component to handle multiple slices and SPs simultaneously, managing several sets of TAs, MAs, and DBs for different technologies. The details of the implementation will be explored in the next chapter.

Similarly, the Policy Controller handles the policies of multiple slices simultaneously. The most important detail of the Policy Controller is the instantiation of the policy checker mechanism. We defined two different types of check mechanisms[1]: the *pooling check mechanism* and the *alert check mechanism*. The former mechanism is the most simple and technologically agnostic. The idea is to provide a pooling query for each policy to check whether the metrics defined in the policy creation were violated. In the positive case, the SEA Controller will be called for VE actions, or the slice provider will be called for HE actions. One great benefit of this mechanism is the ability to work well with several database technologies (e.g., SQL, non-SQL, time series).

The cons of this mechanism are the pooling for each policy and slice, and probably

[1] More mechanisms can be defined.

this mechanism will not scale very well. For example, with 100 slices, each one having three policies to be managed by the SEA, 300 pooling checker mechanisms would be necessary to handle the policies inside the Control Layer. Furthermore, in most cases, those pooling checker mechanisms will be reflected on database queries with different parameters. Therefore, in terms of implementation, those queries should be optimized for each slice, ensuring the feasibility of this mechanism.

The latter mechanism, *alert check*, scales better than the former because it is coupled with the database already used to store the monitored metrics. Most recent databases provide alert mechanisms to inform users when something happens. In the context of this proposal, we defined the alert check mechanism using those technologies of databases. Therefore, the Policy Controller abstracts the alert creation based on the parameters sent during the policy creation and the database technology. To support this mechanism, the SEA should provide an implementation for each database technology that will dynamically create the *alert check mechanism* during the policy creation. The *alert check* mechanism can be seen as an adapter to abstract the alert CRUD for a specific database technology (e.g., Prometheus, InfluxDB, MongoDB). All the SEA policy operations will be reflected in database calls.

4.2.3 Per-Slice components layer

This layer comprises all components (MAs, TAs, and DB) instantiated for each slice managed by the SEA. For each slice orchestrated, the SEA deploys one DB to abstract a database technology defined by the tenant. In addition, a MA and a TA for each SP must be instantiated. The former is responsible for abstracting the VIM/WIM, and the latter for interacting with the ME. This process is invisible to the tenant, which should only provide the needed parameters to the SEA API.

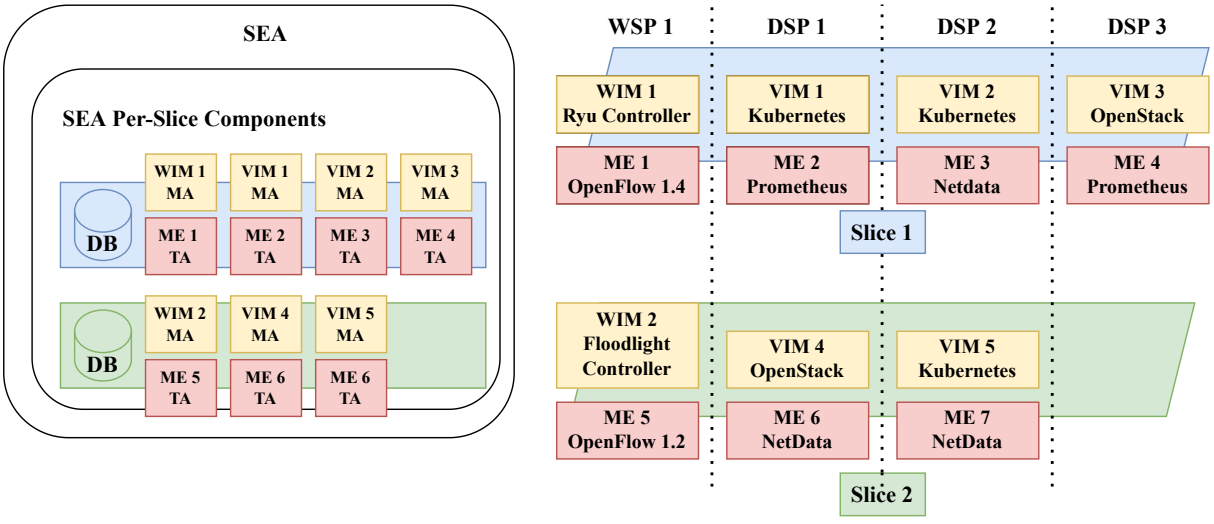


Figure 8 – SEA per-slice components example.

To help the reader materialize those components, consider the following example. Based on Figure 8, suppose that there is a SEA implementation orchestrating only two slices (Slice 1 in blue and Slice 2 in green). The infrastructure of Slice 1 is formed by 1 WSP and 3 DSPs, where each of them uses different technologies combinations of VIM/WIMs and MEs based on the tenant's requirements. On the other hand, Slice 2 in green comprises 1 WSP and 2 DSPs. Each SP has a pair of VIM/WIM and ME technologies, as shown on the right side of Figure 8. We want to highlight that this proportion of per-slice components must always be respected. In other words, the number of them will follow the formula below (where *SPs* means the number of slice parts of the slice). The *2* in the formula is related to the SP pair of MA and TA, and the plus *1* represents the database component.

$$2 * SPs + 1$$

The database component is crucial in the SEA design. In the first version of the architecture proposal, this component was the only one to store data for all slices managed by SEA. However, we identified this design as a limitation of the solution, and then we changed to have one database for each slice running in the SlEaaS provider. This design decision improved the following aspects of the SEA:

- The tenant manageability and the customization freedom of which technologies will be used to store the data (following the same approach of VIMs/WIMs);
- The database isolation between different slices, ensuring that metrics from different slices will not be stored in the same place;
- The proposal of a generic information model to store data from different MEs since we have a huge technology heterogeneity in the slice infrastructure. Therefore, all the data independently of the ME used and slice will be stored equally, differing only in the metrics stored.

The technology used as the database is a requirement defined by the tenant during the P1. Therefore, the choice must comprise the tenant's needs. The database can support or not the alert mechanism described on P2, so the tenant should consider the pros and cons of each technology to select them with wisdom. If the database supports alert creation, the Policy Controller on P2 will be responsible for interacting with the specific slice database to create the alert with the threshold and configurations given during P2 by the tenant. The proposed information model and the implementation of the alert mechanism will be shown in Chapter 5.3.

The TAs and MAs are very specific to each technology the tenant chose as VIMs and WIMs. First, the TAs must interact with MEs to collect the metrics requested by the

tenant. Then, each TA is responsible for pooling the metrics and communicating with the database to store the data from time to time. Depending on the ME chosen, the SlEaaS provider can provide configuration methods in addition to the monitoring. For example, suppose that the ME can support the update of the interval between the polling. In that case, the TA can offer this abstraction to the tenant extending the SEA API to support these new configurations at run-time.

Finally, the MAs present several possibilities of functionalities depending on the VIMs/WIMs technologies required by the tenant. In summary, the MA abstracts the interaction with VIMs and WIMs. In other words, all the tasks/features of those technologies can be called by the tenant using the SEA API or calling directly the VIMs/WIMs. The scope of those adapters is extensive, and their primary responsibility in the SEA design is performing the VE operations abstracting the resource type and the technology used. Other responsibilities include tasks related to the service, such as deployment, deletion, updating, and the configuration/management of the virtual resources created exclusively for that slice.

5 SEA Implementation

5.1 SEA APIs

This chapter focuses on the APIs' description, presenting their methods and inputs. They are one of the main contributions of SEA since it is a novelty in the literature, as already explored in Chapter 3. The APIs should be offered by a SlEaaS provider who implements the SEA to provide the SlEaaS. Then, tenants and slice providers can call those APIs. The responsibilities described in the previous chapter in Figure 7 are reflected on the following APIs, and they will have the same name as the tasks described to help the comprehension. All the SEA components were implemented in Python 3, and the internal and external APIs are REST APIs.

To exemplify the details provided in this chapter, we present a CDN slice called a touristic slice. Figure 9 illustrates the touristic-slice infrastructure and technologies that will be used to describe the SEA components input of internal and external APIs. In other words, all the input parameters, descriptor files, and examples presented in this chapter will reference the touristic slice exclusively for this chapter's explanation. At final of this chapter, the readers will have a detailed explanation of the APIs defined in the SEA architecture with materialized examples of almost all of them.

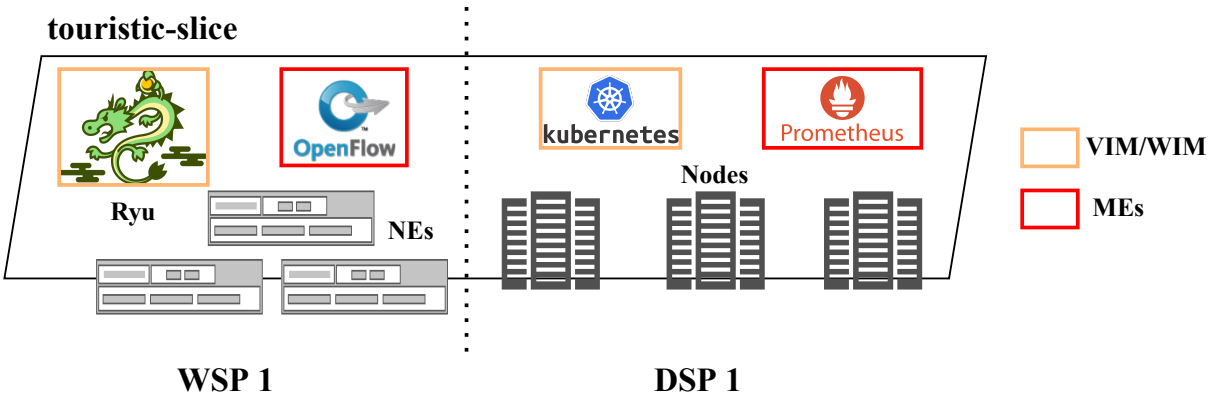


Figure 9 – Touristic-slice example to help the description of the components.

5.1.1 SEA APIs

Starting with the SEA APIs, we will focus firstly on the SEA component administration APIs. After that, the idea is to choose some not required methods and detail how we implemented them or how they can be implemented. We can only cover some of them since they are very technologically dependent in most cases. In that sense, when we discuss the MAs and TAs later, more details and examples will be presented to the reader.

Table 3 summarizes the implementation details of the APIs related to SEA APIs. The same columns and interpretations will be found in the following tables describing the other APIs. The first column is the API name (same described in Figure 7). The second column (*Input*) is the descriptor file in YAML format used as the API parameter. The third column (*Who calls?*) represents who is calling that API. The fourth column (*Will call?*) illustrates the next component that will be called. Finally, the last column (*Phase(s)*) shows which phase described in Section 4.1 that API is related to. For the sake of understanding, we will focus on the mandatory or highly recommended APIs, detailing the parameters. The non-mandatory APIs will be covered in fewer details, but all of them will have an input file descriptor example as a footnote to help the readers.

API	Input	Who calls?	Will call?	Phase(s)
start SEA*	start-sea.yaml 5.1[1]	Tenant or Slice Provider	SEA Controller	P1.1
decommission SEA*	stop-sea.yaml 5.2[2]	Tenant or Slice Provider	SEA Controller	NA
scale SEA*	he-sea-scale-in.yaml he-sea-scale-out.yaml 5.3[3]	Tenant or Slice Provider	SEA Controller	P4.1
update MAs, TAs, DB	start-sea-update.yaml[4]	Tenant or Slice Provider	SEA Controller	NA
list per-slice components	list-per-slice.yaml[5]	Tenant or Slice Provider	SEA Controller	NA
deploy service	deploy-service.yaml 5.4[6]	Tenant or Slice Provider	SEA Controller	NA
create VM/container	create-virtual-host.yaml[7]	Tenant or Slice Provider	SEA Controller	NA
re-route flow	re-route-flow.yaml[8]	Tenant or Slice Provider	SEA Controller	NA

Table 3 – SEA APIs implementation details.

Listing 5.1 is one of the most important examples of an input descriptor file. This example represents the start SEA per-slice components input called *start-sea.yaml*. The listing is self-contained, explaining the fields defined by that API inside the listing. First, we will explain some of them to help the reader materialize the concepts with the information needed to start providing SlEaaS.

The *start-sea.yaml* descriptor file starts with the slice *id* and *name*. The field called *db-parameters* contains all specifications related to the database chosen to store the data, for example, the database technology (*db*), the credentials (*user* and *password*), the IP address of the database (*db-ip*) and database technological specific parameters that varies depending on the database (e.g., *org-name*, *bucket-name* and *token*). The *slice-parts* field can also be seen in other input APIs, presenting a list of SPs information fulfilled by the tenant or the slice provider. The main attributes of the SPs are: the SP infrastructure type (*type*), the unique SP identifier (*sp-id*), the attributes related to the VIM/WIM used in that SP (*management-parameters*), and all information of the ME instantiated to monitor that SP (*monitoring-parameters*).

[1] https://github.com/dcomp-leris/SEA/blob/main/yamls/start-sea.yaml.
[2] https://github.com/dcomp-leris/SEA/blob/main/yamls/stop-sea.yaml.
[3] https://github.com/dcomp-leris/SEA/blob/main/yamls/he-sea-scale-in.yaml, https://github.com/dcomp-leris/SEA/blob/main/yamls/he-sea-scale-out.yaml.
[4] Similar of the start-sea.yaml descriptive file.
[5] https://github.com/dcomp-leris/SEA/blob/main/yamls/list-per-slice.yaml.
[6] https://github.com/dcomp-leris/SEA/blob/main/yamls/deploy-service.yaml.
[7] https://github.com/dcomp-leris/SEA/blob/main/yamls/create-virtual-host.yaml.
[8] https://github.com/dcomp-leris/SEA/blob/main/yamls/re-route-flow.yaml.

Both management and monitoring parameters can be extended depending on the technologies chosen. Therefore, we defined the basic ones on the *start-sea.yaml* that can fit in almost all technologies. Each *management-parameters* is composed of the VIM or WIM technology (*vim* or *wim*), the access information (*ip*), *port* and *ssh-port*, the basic credentials (*user* and *password*). On the other hand, each *monitoring-parameters* presents the ME technology used in that SP (*tool*), the access information (*tool-ip* and *tool-port*), the basic credentials (*user* and *password*), the interval between the TAs collect the metrics from the ME (*granularity-secs*), the list of metrics to be collected from the ME (*metrics*) and any other ME specific parameters (e.g., *mon-port-enabled*). The SEA Controller will use all the described information to create the MAs, TAs, and DB components to orchestrate the touristic slice.

```
---
slice:
    id: slice1
    name: touristic-slice
    db-parameters:
      db: influxdb:2.0 # Database name or container image
      user: admin # User created to administrate the db
      # Database IP address to be used by the TAs and Policy mechanisms
      db-ip: 10.10.10.10
      password: adminpwd
      # Parameters below are specific to the database influxdb
      org-name: touristic-slice1
      bucket-name: touristic-slice1
      token: secrettoken
    # List of n slice-parts that compose this slice infrastructure
    slice-parts:
       - slice-part:
         name: wan-sp-1 # The slice-part name
         # The slice-part type, can be: [Datacenter, WAN]
         type: WAN
         sp-id: WSP1 # The slice-part ID
         management-parameters:
            wim: ryucontroller # WIM used in the WSP 1
            ip: x.x.x.x # Ryu Controller IP
            port: 8080 # Ryu Controller REST API port
            # Port to be used for SSH when needed
            ssh-port: 22
            # Basic credentials to access the WIM
            user: admin
            password: ****
         monitoring-parameters:
            # ME used to monitor the slice part WSP 1
            tool: openflowv1.4
            tool-ip: x.x.x.x # The ME IP address
            # The port used to communicate with this ME
            tool-port: 8080
            # Basic credentials to access the ME
            user: admin
            password: ****
            # Specific for this ME, enables or not the port monitoring
            mon-port-enabled: true
            # Time interval to collecting the metrics
            granularity-secs: 1
```

```
44              # List of metrics to be monitored in this SP
45              metrics:
46                  - PACKETS_RX
47                  - PACKETS_TX
48                  - BYTES_RX
49                  - BYTES_TX
50                  - DROPS_TX
51                  - DROPS_RX
52                  - ERRORS_RX
53                  - ERRORS_TX
54                  - FRAME_ERRORS_RX
55                  - OVERRUN_ERRORS_RX
56                  - CRC_ERRORS_RX
57                  - COLLISIONS
58          - slice-part:
59              name: dc-sp-1 # The slice-part name
60              type: Datacenter # The slice-part type, can be: [Datacenter, WAN]
61              sp-id: DSP1 # The slice-part ID
62              management-parameters:
63                vim: Kubernetes # The VIM used in this SP
64                ip: y.y.y.y # The IP where the VIM is running
65                port: 6443 # The port used to communicate with this VIM
66                # Basic credentials
67                user: admin
68                password: adminpwd
69              monitoring-parameters:
70                tool: Prometheus # The ME used in this SP
71                tool-ip: y.y.y.y # The IP where the ME is running on this SP
72                tool-port: 19999 # The port used to communicate with this ME
73                # Basic credentials to access the ME
74                user: admin
75                password: ****
76                granularity-secs: 1 # Time interval to collecting the metrics
77                metrics: # List of metrics to be monitored in this SP
78                  - MEMORY_UTILIZATION_PHYSICAL
79                  - CPU_UTILIZATION_PHYSICAL
80                  - RATE_BYTES_READ_PHYSICAL
81                  - RATE_BYTES_WRITTEN_PHYSICAL
82                  - RATE_BYTES_RX_PHYSICAL
83                  - RATE_BYTES_TX_PHYSICAL
84                  - RATE_PACKETS_TX_PHYSICAL
85                  - RATE_PACKETS_RX_DROP_PHYSICAL
86                  - RATE_PACKETS_TX_DROP_PHYSICAL
```

Listing 5.1 – SEA API: start-sea.yaml example.

The second SEA API is highly recommended to be available in the *decommission SEA*, which receives only two parameters: the slice id (*id*) and the slice name (*name*), as can be seen in the Listing 5.2. The SEA Controller is responsible for looking at the per-slice components allocated to the requested slice and deleting them.

```
1 ---
2 slice:
3     id: slice1
4     name: touristic-slice
```

Listing 5.2 – SEA API: stop-sea.yaml example.

The third SEA API to be explored is the *scale SEA*, the start point for Phase 4, which occurs right after a HE operation is executed to adapt the per-slice components needed. The input of this API is very similar to *start-sea.yaml* presented on Listing 5.1. The only difference is that since they did not change in this case, the *he-sea.yaml* does not contain the *db-parameters* fields. Therefore, the *he-sea.yaml* contains the slice id, slice name, and a list of new SPs with the attributes already described on *start-sea.yaml* when it is a HE scale out adaptation. On the other hand, if the adaptation is a response to a HE scale in operation, the *he-sea.yaml* should contain only a list of the SPs' ids and names that were removed. An example of the *he-sea.yaml* of HE scale out operation can be seen here[9] and the descriptor file for *he-sea* HE scale in is presented below on Listing 5.3. In this example, we suppose that the DSP 1 was removed in response to a HE scale in, so the MA and TA of DSP 1 should be decommissioned as well.

```
1  ---
2  slice:
3     id: slice1
4     name: touristic-slice
5     # List of slice-parts that are removed after HE scale in
6     slice-parts:
7       - slice-part:
8         name: dc-sp-1
9         type: Datacenter
10        sp-id: DSP1
```

Listing 5.3 – SEA API: he-sea-scale-in.yaml HE scale in example.

The SEA per-slice components APIs are simple. The *update MAs, TAs, and DB* API receives a descriptor file very similar to the Listing 5.1 with the updated parameters. The *list per-slice components* API starts with the slice name and id as a parameter and returns a list of MAs, TAs, and DB information. The SEA VIM/WIM API Abstraction comprises the APIs interacting with those technologies. Listing 5.4 shows an example of the *deploy service* API reflecting on container creation using Kubernetes as VIM. The presented deploy-service.yaml demonstrates a very simple deployment of the NGINX service with two replicas. The tenant should specify which SPs the service will be instantiated, but this can be improved further.

```
1  ---
2  slice:
3     id: slice1
4     name: touristic-slice
5     slice-parts:
6       - slice-part:
7         name: dc-sp-1
8         type: Datacenter
9         sp-id: DSP1
10        # Kubernetes specific parameters
11        deploy_type: "Deployment"
12        deploy_name: "nginx-deploy"
```

[9] https://github.com/dcomp-leris/SEA/blob/main/yamls/deploy_wan_sp.yaml.

```
13          app_name: "my-nginx"
14          container_name: "nginx"
15          deploy_image: "nginx:latest"
16          replicas: 2
```

Listing 5.4 – SEA API: deploy-service.yaml example.

5.1.2 Policy APIs

The Policy APIs are crucial for the SEA design to have the automated CCL of the slice elasticity operations. Therefore, this section details the defined APIs to handle the policies presenting the input parameters of almost all APIs. It is important to highlight that those parameters are defined by us and could be easily extended or modified. Table 4 summarizes the available Policy APIs. The columns' interpretation is the same as in Table 3 in the previous section.

API	Input	Who calls?	Will call?	Phase(s)
create	create-pol.yaml 5.5[10]	Tenant	Policy Controller	P2.1
delete	delete-pol.yaml 5.6[11]	Tenant	Policy Controller	NA
update	update-pol.yaml[12]	Tenant	Policy Controller	NA
list	list-pol.yaml[13]	Tenant	Policy Controller	NA
enable	enable-pol.yaml[14]	Tenant	Policy Controller	NA
disable	disable-pol.yaml[15]	Tenant	Policy Controller	NA
unlock	unlock-pol.yaml[16]	Tenant	Policy Controller	NA

Table 4 – Policy APIs implementation details.

In the context of slicing, the tenant should create the policy per SPs due to the multiple administrative domains. Therefore, the tenant must pay attention when defining the policy metrics ensuring they are defined to be collected by the SPs he chose in the policy creation. Listing 5.5 exemplifies the *create-pol.yaml* descriptor YAML file that can be used as input of the create policy API. The tenant can define the policies per SP as shown in the listing; inside the slice-part attribute, the tenant must pass the SP *name* and *sp-id*, the list of policies (*policies-elasticity*) and the *database-parameters*. The *database-parameters* attribute presents the needed information to create the policy checker mechanism explained in the next chapter.

[10] https://github.com/dcomp-leris/SEA/blob/main/yamls/create-pol.yaml.
[11] https://github.com/dcomp-leris/SEA/blob/main/yamls/delete-pol.yaml.
[12] Similar of create-pol.yaml descriptive file.
[13] https://github.com/dcomp-leris/SEA/blob/main/yamls/list-pol.yaml.
[14] Equal to delete-pol.yaml descriptive file.
[15] Equal to delete-pol.yaml descriptive file.
[16] Equal to delete-pol.yaml descriptive file.

It is important to highlight the *policies-elasticity* fields. Listing 5.5 also has a brief description of the fields. The policy is represented by an identifier (*p-id*), the elasticity operation that should be performed in case of violation (*elasticity-operation*), and by the algorithm attributes[17]. In the scope of this thesis, we defined in more detail the most simple algorithm type, the threshold algorithm. However, other types can be explored and defined later, such as learning-based algorithms, reinforcement learning-based algorithms, etc.

A list of triggers defines the algorithm parameters for the threshold type. Each trigger is composed of the metric name to be periodically checked (*metric-name*), the interval to verify if the threshold is violated (*check-interval*), the threshold value and unit (*threshold* and *unit*), the comparison operator (*operator*) can be one of the following: >, <, >=, <=, =. In addition, we defined an attribute called *statistic* to improve this simple policy based on thresholds. This attribute defines heuristics for the threshold check. Examples of heuristics are: trigger the elasticity if the metric violation occurs three times in a row or if the metric violation lasts for 30 seconds[18].

For each policy defined, the tenant must inform the *elasticity action* that should be performed if the policy was violated. In this field, the elasticity type and the actions that should occur to perform the requested elasticity operation are important. In the example presented below, we have a very simple VE scale up action increasing the number of replicas of the service running on DSP 1. Therefore, the fields of elasticity action change according to the technology and the elasticity operation type.

```
1
2  ---
3  slice: # List of slices to be configured
4    id: slice1 # The slice ID
5    name: touristic-slice # The slice name
6    slice-parts: # List of slice-parts in this slice
7      - slice-part:
8        name: dc-sp-1
9        type: Datacenter
10       sp-id: DSP1
11       nodes:
12       - node:
13         name: slice1-sp2-master
14         node-id: slice1-sp2-master
15         VIM: Kubernetes
16       policies-elasticity: # List of policies for this slice-part
17       - policy:
18         p-id: p1-touristic-slice
19         name: policy1 # Name of the policy
20         elasticity-operation: scale-up # The type of elasticity triggered in this policy
21         algorithm: # Information about the algorithm used in this policy
22           # The type of the algorithm
23           type: Threshold
```

[17] The attributes may change based on the algorithm type.

[18] Other heuristics can be defined and will be interpreted by the SEA if there is an implementation for that.

```
          # List of algorithm parameters specific for each type of algorithm
          algorithm-parameters:
            # For the threshold algorithm we defined the parameters in terms of triggers
            triggers:
            - trigger:
              # Name of one of the metrics specified in the start-sea
              metric-name: MEMORY_UTILIZATION_PHYSICAL
              # Which is the push/pull interval to check the metric in the database
              check-interval: 5s
              threshold: 8 # Which is the threshold for this metric in this policy
              unit: Gb # The unit of the threshold
              operator: ">=" # The operator for this metric
              statistic:
                # The type of logic behind the threshold checking.
                # This stats type can be: [window, exact-match, and others]
                stats-type: window
                # In the window type you can define a threshold window, this field means
                # how many consecutive times the threshold need to be trespassed to
                # trigger the operation
                threshold-window: 3
                # How many seconds hitting the threshold is the minimum to trigger
                # the elasticity operation for this metric
                threshold-seconds: 10s
        elasticity-action:
          - action:
            elasticity-type: scale-up # The elasticity type in the actions needed
            description: Increase replicas
            replicas: 4
            token: xxxxx
  database-parameters:
    type: influxdb
    url: http://xxxx:port
    token: secrettoken
    org-name: touristic-slice1
    bucket-name: touristic-slice1
    elasticity-url: http://xxxx:1014/elasticity
```

Listing 5.5 – Policy API: create-pol.yaml example.

The delete policy API receives as input the *slice id*, *name*, and a list of *policy identifiers*, as seen in Listing 5.6. The *list policies* method has the slice *id* and *name* as parameters and returns all the policies created per SP. The updated policy API input is very similar to the create-pol.yaml, but only changing the parameters that should be updated in the policy description. The Policy Controller identifies whether the policy checker mechanism should be recreated. The APIs *enable policy*, *disable policy*, and *unlock policy* receive the slice id and a list containing the policies that will be enabled, disabled, or unlocked.

```
---
slice:
  id: slice1
  name: touristic-slice
  # List of policies that should be removed
  policies:
    - p1
    - p2
```

```
9       - p3
```

Listing 5.6 – Policy API: delete-pol.yaml example.

5.1.3 Elasticity control APIs

The Elasticity Control APIs are triggered by the tenant or slice provider to perform the specified VE operation. Therefore, we provide one method for each possible VE (scale up, scale down, queue scale up, queue scale down, topology scale up, topology scale down). The input parameter for them is very similar. It is the *elasticity-action* attribute described in the Listing 5.5 plus the slice identifier and SP identifier. For the sake of simplicity, we will show only one example of this file for the same VE scale up shown in *create-pol.yaml* file but now triggering the elasticity manually without the policy creation. These APIs represent Phase 3, described in Chapter 4.

Listing 5.7 illustrates the example mentioned above, showing how to trigger the same VE scale up presented on *create-pol.yaml* manually calling the Elasticity Control API. After this call, the SEA Controller performs the requested VE action communicating with the MAs. As already explained, the HE is the slice provider's responsibility to perform those operations. Then after the completion of the HE, the SEA scale API should be called to adjust the per-slice components based on the SPs added or removed as a result of the HE.

```
1
2  ---
3  slice: # List of slices to be configured
4    id: slice1 # The slice ID
5    name: touristic-slice # The slice name
6    slice-parts: # List of slice-parts in this slice
7      - slice-part:
8        name: dc-sp-1
9        type: Datacenter
10       sp-id: DSP1
11       nodes:
12       - node:
13         name: slice1-sp2-master
14         node-id: slice1-sp2-master
15         VIM: Kubernetes
16       elasticity-action:
17         - action:
18           elasticity-type: scale-up # The elasticity type in the actions needed
19           description: Increase replicas
20           replicas: 4
21           token: xxxxx
```

Listing 5.7 – Elasticity Control API: ve-scale-up.yaml example.

5.2 Controllers

This chapter presents the implementation details of the SEA and Policy Controller. They were implemented in Python and the source code can be found on github[19]. To facilitate comprehension, we present the controllers as workflows describing in detail the steps done for them after they receive an API call. As mentioned in Chapter 4, the controllers are the core of the SEA design, orchestrating the per-slice components and policies for all slices.

5.2.1 SEA controller

The workflow for the SEA Controller will be based on the phases and the mandatory APIs described in Chapter 4. A significant differential of the SEA design is how the per-slice components are deployed and managed. We decided to follow a microservices approach in those components to easily maintain them and isolate the traffic between the components of different slices in the SlEaaS provider. Therefore, each per-slice component will be instantiated over a pre-built container image containing the adapter implementation, which will be detailed in the next chapter. In this section, the focus is on how the phases are implemented on the SEA Controller.

Figure 10 illustrates the workflow of Phase 1, where the tenant requests the SEA to deploy the components needed to monitor, manage and orchestrate the informed slice. On P1.1, the tenant calls the SEA API already described passing the *start-sea.yaml* as input. After that, the SEA Controller parses the information to discover which per-slice components should be instantiated (P1.2). To start the instantiation of the component, the SEA Controller creates one docker network for each slice using docker SDKs. This network will be used internally for the per-slice components containers. Phase P1.3 comprises all the processes of creating the docker containers based on the VIMs/WIMs and MEs technologies. The pre-built images are selected and configured with the parameters informed by the tenant on *start-sea.yaml*. The container image of the database may not be pre-built on the SlEaaS infrastructure. Once the tenant chooses the database technology, the container image will be downloaded from docker repositories if not presented locally.

Before finishing the P1, the SEA Controller checks the adapters' instantiated calling them. The TAs were configured with the database info (IP and port) because they would communicate directly with the container database to store the data, following an information model described later. To complete Phase 1, the information of the containers for all slices is stored in memory and persisted on a file in the SlEaaS provider. This persistence is very important to retrieve the containers running if anything goes wrong in the SEA components. One example of this file can be found in Listing 5.8. Moreover,

[19] https://github.com/dcomp-leris/SEA.

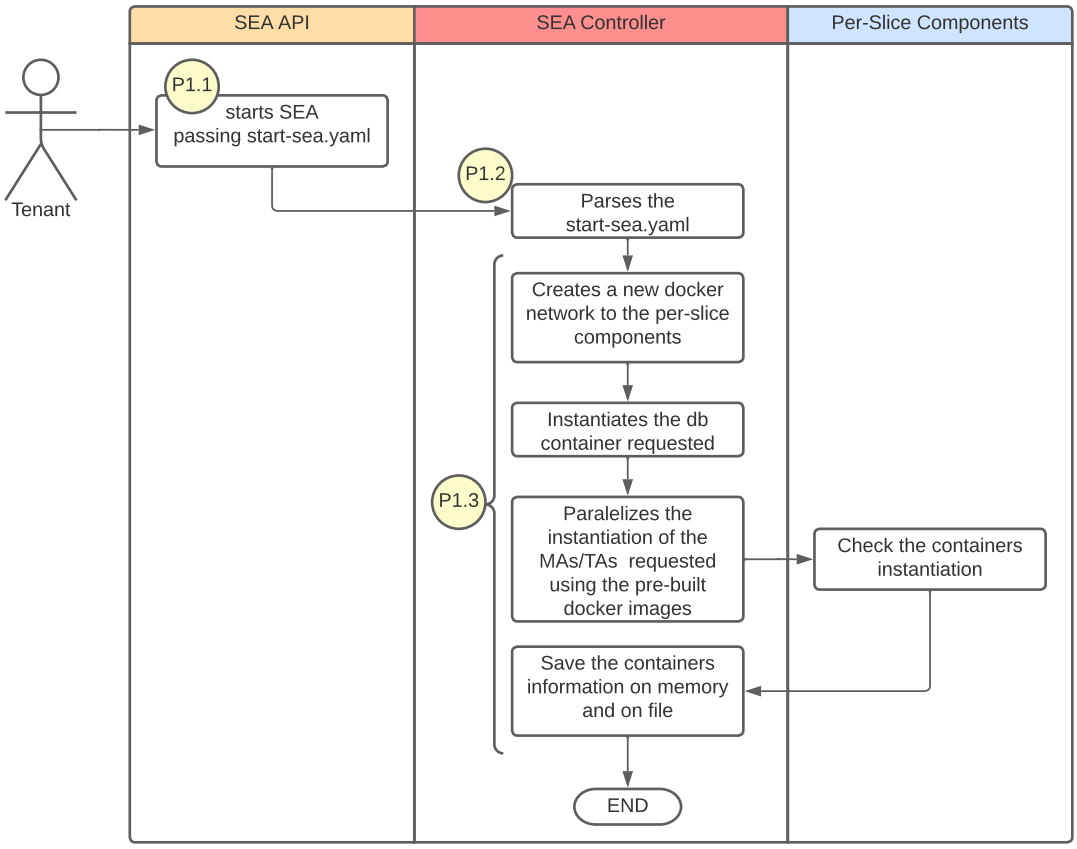


Figure 10 – SEA controller workflow (*start SEA*).

the decommission of per-slice components is also performed by deleting the respective containers.

```
1
2  {
3      "slice1":[
4          {
5              "container_name":"slice1_database",
6              "container_id":"1234",
7              "container_port":34791,
8              "type":"database",
9              "db_ip":"x.x.x.x",
10             "username":"admin",
11             "password":"adminpwd",
12             "org-name":"touristic-slice1",
13             "bucket-name":"touristic-slice1",
14             "token":"secrettoken"
15         },
16         {
17             "container_name":"dsp1_k8s",
18             "container_id":"1235",
19             "container_port":42297,
20             "type":"man_adapter",
21             "vim":"kubernetes",
22             "sp_id":"dsp1",
23             "vim_ip":"localhost",
24             "vim_port":6443,
```

```
25            "username":"admin",
26            "password":"adminpwd"
27        },
28        {
29            "container_name":"dsp1_prometheus",
30            "container_id":"1236",
31            "container_port":40003,
32            "type":"mon_adapter",
33            "tool":"prometheus",
34            "sp_id":"dsp1",
35            "tool_ip":"x.x.x.x",
36            "tool_port":9092,
37            "granularity":3,
38            "metrics":[
39                "MEMORY_UTILIZATION_PHYSICAL"
40                "CPU_UTILIZATION_PHYSICAL"
41                "RATE_BYTES_READ_PHYSICAL"
42                "RATE_BYTES_WRITTEN_PHYSICAL"
43                "RATE_BYTES_RX_PHYSICAL"
44                "RATE_BYTES_TX_PHYSICAL"
45                "RATE_PACKETS_TX_PHYSICAL"
46                "RATE_PACKETS_RX_DROP_PHYSICAL"
47                "RATE_PACKETS_TX_DROP_PHYSICAL"
48            ]
49        }
50    ]
51 }
```

Listing 5.8 – Per-slice components file persistence example.

The SEA Controller methods related to performing the VEs can be triggered by the tenant or slice provider on P3.1 or by the Policy Controller when a policy is violated. Both cases will reflect the same SEA Controller method internally. This method looks at the *elasticity-actions* mentioned in Listing 5.5 and calls the MAs for each SP that must be scaling up or scaling down and performs the slice elasticity operation through the VIMs/WIMs APIs transparently to the tenant. For example, in Listing 5.5, the action is effortless, increasing the number of replicas. The SEA Controller will parse the new number of replicas and then the MA of DSP 1 is called to increase the number of replicas on Kubernetes of the DSP 1. The other APIs described before work similarly and will reflect on changes in the per-slice components containers or calls for the VIMs/WIMs via the MAs or MEs via the TAs.

5.2.2 Policy controller

The Policy Controller manages the policy life-cycle and creates mechanisms to trigger the slice elasticity operations. The workflow presented in Figure 11 shows Phases 2 and 3 in sequence and the tasks done by the SEA components. First, the tenant requests the policy creation (P2.1), then the Policy Controller is called to create the requested policy checker mechanism (P2.2). Two possibilities are defined for this mechanism. The first one is based on database technology that uses database alerts to trigger an action

when the threshold is violated (P3.2). The second is more straightforward and involves the creation of a pooling mechanism that will query the database metrics to verify if the policies are being respected. In both cases, if there are being violated, the internal API of the SEA controller is triggered.

This internal API contains all the elasticity operations alternatives based on the elasticity-actions parameter, and the SPs communicate with the MAs to perform the elasticity operation triggered. Finally, the MAs interact with the VIMs/WIMs executing the elasticity operation informed. The tenant can manually request an elasticity operation to be performed for that he should use the Elasticity Control API (P3.1), and the workflow follows the same process described before through the SEA Controller on P3.2 and P3.3.

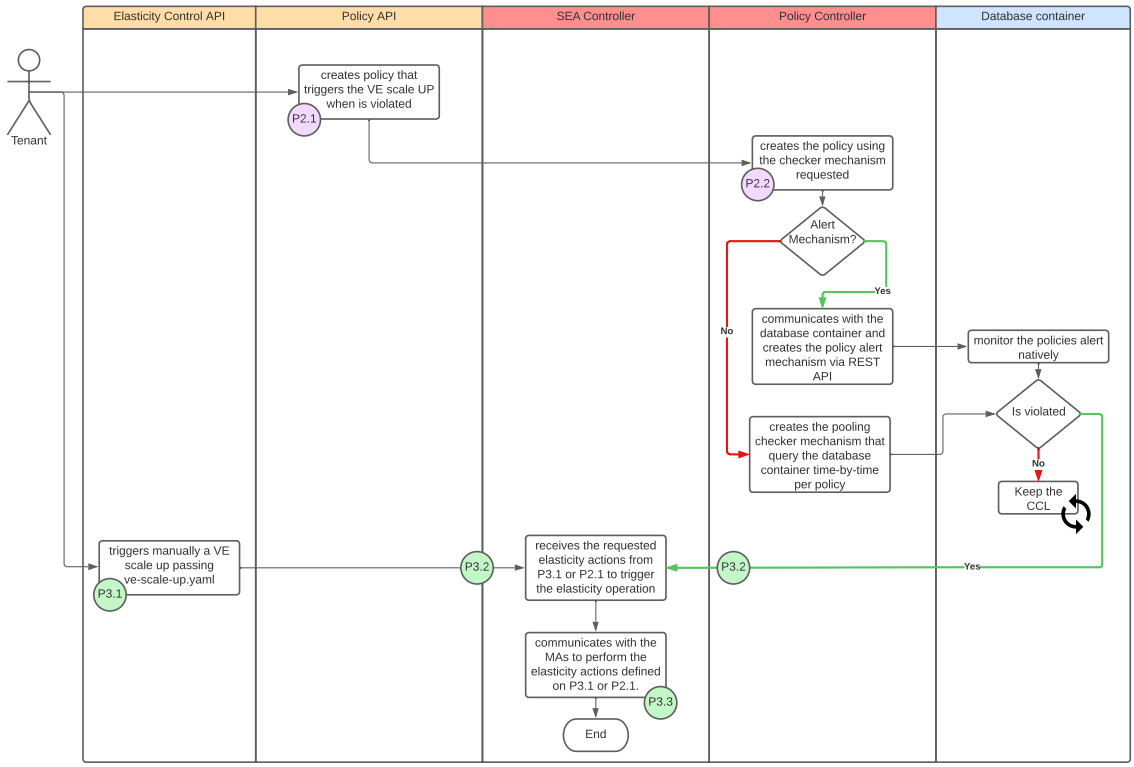


Figure 11 – Policy controller workflow phases 2 and 3.

This is the idea of the Policy Controller workflow, but we still need to give more details about the alert mechanism and how this can be materialized and done in this work's context. We will explain this mechanism using the InfluxDB database as an example. Still, other database technologies can offer the same ability to create alerts on the database (e.g., Prometheus, MongoDB, Amazon Timestream and Cloud Watch).

Regarding implementing the InfluxDB alert mechanism, we have three tasks that must be implemented to have the InfluxDB alert mechanism working and coupled with the SEA design: the checks, the notification rules and the notification endpoints. The checks

are created based on the query language of the database technology (Influx[20] in this case). The query is built based on the triggers, metrics and thresholds specified by the tenant during the policy creation. The notification endpoint is what should be called if the policy is violated. We configured the notification endpoint with the SEA Controller internal API responsible for performing the elasticity operation requested. The policy id is sent to the SEA Controller because it retrieves the policy data based on the policy id. Finally, the notification rule is created with the notification endpoint and the time interval of checking whether or not the thresholds are being violated. The source code example is developed in python using the InfluxDB client library and can be seen at[21].

5.3 Per-slice components

The adapters are crucial in the design to provide the needed abstraction for the multiple administrative and technological domains native to the slicing concept. Furthermore, with those adapters, the SEA is generic enough to support any technological choice. Therefore, below we define the components' responsibilities considering the nature of the slicing infrastructure and the support of the SlEaaS, which we believe is one of the major contributions of the SEA.

5.4 MAs and TAs

The MAs are the abstraction of different VIMs and WIMs. To support a new VIM or WIM, the adapter of this technology should be implemented in any language. The communication between the SEA Controller and MAs/TAs is based on REST APIs. We implement the methods to abstract the VIMs/WIMs or MEs in each adapter that will mainly perform an HTTP request to them. This code runs in containers, which should have the image pre-built on the SlEaaS provider infrastructure. The adapters' prerequisites and libraries should be pre-installed in the container image, and the adapter code should be downloaded and executed in each container.

In summary, for each technology, we will have a pre-built docker image (e.g., Kubernetes, Docker Swarm, ssh, Prometheus, Netdata). It is important to highlight that each SlEaaS provider can provide the VIMs, WIMs, and MEs that are supported, which means that the implementation of the adapter is the responsibility of those who implements the SEA.

In this thesis, we develop some MAs and TAs to validate the SEA and perform the test scenarios. Before describing each of them, we would like to present an example of a

[20] https://www.influxdata.com/.
[21] https://github.com/dcomp-leris/SEA/blob/main/controllers/policy/policy_checker_influxdb.py.

Dockerfile used to pre-built an adapter image; On our github[22] contains all Dockerfiles. The Listing 5.9 shows the Dockerfile of the TA OpenFlow. We can observe that the base image is ubuntu focal, and several python packages are installed. The adapter code is obtained from the SEA github[23], and then the code is initialized in the ENTRYPOINT.

```
1  FROM ubuntu:focal
2  RUN \
3      apt -y update && \
4      apt -y install git python3-pip curl && \
5      pip3 install requests && \
6      pip3 install pika==0.13.1 && \
7      pip3 install influxdb --default-timeout=100 && \
8      pip3 install flask --default-timeout=100 && \
9      pip3 install flask_request_params --default-timeout=100 && \
10     pip3 install pyyaml --default-timeout=100 && \
11     pip3 install paramiko --default-timeout=100 && \
12     pip3 install docker --default-timeout=100 && \
13     pip3 install https://github.com/kubernetes-client/python.git && \
14     git clone https://github.com/dcomp-leris/SEA.git && \
15     cd SEA
16 ENTRYPOINT python3 SEA/adapters/monitoring/adapter_openflow.py >> adapter.log
```

Listing 5.9 – MA and TA Dockerfile example.

The MAs implementation is very specific for each VIM or WIM, but the more important task that must be implemented is the capability of interacting with the VIMs and WIMs to perform the slice elasticity operations. For example, if we have Kubernetes as VIM, the MA must be able to increase the number of replicas. On the other hand, if we have an OpenFlow controller as WIM, the MA must be able to re-route the flows of the specific application. The slice elasticity operations defined in this thesis are not unique, and more operations can be defined, and the SEA should support them due to the architecture design of abstracting the different technologies.

For testing purposes, we developed 4 MAs: the Kubernetes MA, the docker swarm MA, the ssh MA, and the Ryu OpenFlow controller MA. The source code of them can be found here[24]. The SSH adapter is developed based on the SSH protocol, and the idea is to provide an adapter capable of executing Linux commands inside VMs or containers via the SSH protocol. To help readers understand, we elaborate a python skeleton MA pseudocode represented on Listing 5.10. This pseudocode considers a VIM capable of creating or deleting VMs as a VE operation. First, we have the VIM/WIM attributes as IP, port, credentials, and so on (lines 5-8). Those attributes came from the SEA Controller passed as a parameter on P1.1.

In summary, the MA code is a REST API that offers endpoints to calling the VIMs/WIMs methods/endpoints. In the example below, we have three functions: the *ve-*

[22] https://github.com/dcomp-leris/SEA/blob/main/DockerFiles/adapter_openflow.Dockerfile.
[23] https://github.com/dcomp-leris/SEA/tree/main/adapters.
[24] https://github.com/dcomp-leris/SEA.

elasticity-scale-up() (line 11), the *ve-elasticity-scale-down()* (line 15), and the list-topology (line 19). The first two functions perform the VE operations, while the third illustrates an abstraction of the VIM/WIM usage, listing the virtual topology in that case. For example, in the *ve-elasticity-scale-up()*, this POST method will reflect on the *create_vm()* method provided by the VIM. On the other hand, the *ve-elasticity-scale-down()* is an abstraction of the *delete_vm()* method in the VIM REST API. The same understanding for the MA method *list-topology()* with the *get_topology()* from the VIM.

As already mentioned, any method can be implemented on the adapter side and always be reflected on a VIM/WIM REST API. These premises are very important for the SEA design to provide the SlEaaS. With this well-defined, the SEA administrator can easily support any technology. The SEA provides those per-slice components following the microservices approaches, which also corroborates our work's contributions. Facilitating the inclusion of new technologies and infrastructure types that transparently support the slice elasticity operations for tenants.

```
1
2  # MA python skeleton
3
4  # VIM/WIM attributes
5  IP = x.x.x.x
6  port = 5000
7  user = admin
8  password = admin
9
10 # [POST] method on MA:port/ve-elasticity-scale-up
11 def ve_elasticity_scale_up()
12     call VIM/WIM create_vm()
13
14 # [POST] method on MA:port/ve-elasticity-scale-down
15 def ve_elasticity_scale_down()
16     call VIM/WIM delete_vm()
17
18 # [GET] method on MA:port/get_topology
19 def list_topology()
20     call VIM/WIM get_topology()
21
22 # Init the MA REST API
23 def init()
```

Listing 5.10 – MA high-level python code.

The TAs design is very similar to the MAs, and we also have to pre-build the docker images. However, the main objectives of the TAs are retrieving the monitoring data of the MEs instantiated on the SPs infrastructure and adding them into the database container. In summary, the TAs also calls the MEs REST APIs to retrieve those data periodically based on the time interval defined by the tenant (*granularity-secs* on Listing 5.1 previously presented). We provide a TA python skeleton pseudocode to illustrate the most important responsibilities of the TAs, which can be seen on Listing 5.11. The TAs should have the ME and DB attributes to interact with them to collect and store the metrics (represented

on lines 4-17). Those attributes are fulfilled when the adapter is started from the tenant's input (*start-sea.yaml*, input file for the external API). The SEA Controller fulfills the database IP and port because the database is a container and the SEA Controller creates the docker network during Phase 1.

TAs are mainly REST APIs that must provide some methods of paramount importance to provide the SlEaaS. The first to be described is the *start_monitoring()*, a POST method that initializes the monitoring process (lines 21-26) mainly it is an infinite loop that performs three tasks: (a) call the ME REST API to retrieve the metrics passed as a parameter, again we have the technology abstraction but now for the MEs (line 23, supposing that the method on ME to perform the collecting is *collect_metrics()*); (b) the metrics collected should be formatted based on our information model defined (line 24), it is the same independently of the ME technology and resource type; (c) last, the TA should call the DB to write the data collected already well formatted (line 25). After completing those three steps, the TA waits for the time interval defined by the tenant to collect the next round of data (line 26). In addition, the TAs can be updated at run time, and we added an example on lines 38-39 where the TA provides a POST method to update the time interval parameter (called *granularity-secs*). Since it is a tenant operation, this REST method should also be implemented on the external SEA API and internally on the SEA Controller.

```
1
2  # TA python skeleton
3
4  # ME attributes
5  me_ip = x.x.x.x
6  me_port = 4000
7  me_user = admin
8  me_password = admin
9  metrics = [CPU, MEMORY, PACKETS_IN, PACKETS_OUT]
10 granularity_secs = 1
11 ...
12
13 # DB attributes
14 db_ip = y.y.y.y
15 db_port = 3333
16 db_user = db_admin
17 db_password = db_admin
18 ...
19
20 # [POST] method on TA:port/start-monitoring
21 def start_monitoring()
22     while (true)
23         result = call ME collect_metrics(metrics)
24         result_formatted = information_model(result)
25         write_data(IP, port, credentials, result_formatted)
26         wait granularity-secs
27
28 def information_model(result)
29     for data in result
30         # format the data following the information model
```

```
31      return result_formatted
32
33  def write_data(IP, port, credentials, result_formatted)
34      for data in result_formatted
35          # call DB write(data)
36
37  # [POST] method on TA:port/update_granularity_secs
38  def update_granularity_secs()
39      granularity-secs = new granularity-secs
40
41  ...
42
43  # Init TA REST API
44  def init()
```

Listing 5.11 – TA high-level python code.

Three TAs are implemented to validate the SEA: two for DSPs and one for WSPs. For the DSPs, we implemented adapters for the Prometheus and Netdata MEs. The WSP TA implemented is based on the OpenFlow protocol and is retrieved from the Ryu controller. In addition, those MEs provide a REST API to collect the metrics periodically, as requested by the TA design. The source code of the TAs can be found in more detail here[25].

The technologies chosen to be implemented as MAs and TAs are very suitable with the slicing concept and already used in cloud computing or WAN context. For example, the Prometheus and Netdata are under the Cloud Native Computing Foundation[26] initiative and are being used widely in the community and cloud projects, such as shown in the works presented in (Beltrami et al., 2020; Sukhija; Bautista, 2019).

In the next chapter, we will present all the results and evaluation scenarios using those MAs and TAs described here. Some of them naturally evolve from our initial results, which is expected from a Ph.D. perspective. The implementation decision of having the per-slice components following the microservices approach gives the SEA many alternatives for abstraction, manageability, and a significant impact on the scalability of those components, as will be seen later.

5.5 Database

This section explains the database implementation and the information model defined to store the slice metrics generically. In summary, the SEA may provide the database technologies supported, which should be implemented for each technology. The tenant can specify as a database any technology available since it has a container image to facilitate the database deployment and configuration provided by the SEA. Based on the

[25] https://github.com/dcomp-leris/SEA.
[26] https://www.cncf.io.

tenant's technological requirements, the database is configured using the *db-parameters* field. The SEA does not support on-demand database management without running inside containers once this is part of the SEA scope and how it is designed.

The type of database (e.g., NoSQL, SQL, time series) influences the code that should be implemented on SEA to support each of them. In the scope of this work, we implemented only the support for the InfluxDB, a time series database that is very used and well-known. However, the SEA should support any other database type with minor modifications in the actual code. The InfluxDB was chosen because the alerting mechanism improves the feasibility of providing the SlEaaS. In addition, the fast policy violation detection and easy integration with the SEA Controller.

Therefore, we used the native InfluxDB container image on the docker hub to store the monitored data. For each slice being managed and orchestrated by the SEA, a new database container is created, ensuring slice data isolation and the easy deployment of a customized database with the configurations requested by the tenant. The database is the component that most changed in this thesis. Initially, during the NECOS project, we idealized that only one database should exist to store the data for all slices. At that time, the TAs did not communicate directly with the database. Instead, TAs add the measurements already formatted to a message queue system (e.g., RabbitMQ). Then, we had a database collector that consumed this queue and stored the data in the slice provider infrastructure. However, the database was re-designed when we decoupled the SEA (aka CNS-AOM) from the NECOS Project (Rocha et al., 2022).

To make the architecture more service-oriented and focused on providing slice elasticity, we removed the queuing mechanism and added the database in containers configured based on the tenant's request. This evolution brought greater flexibility to the tenant and made the proposed architecture even more generic. Therefore, the results and evaluation have shown before and after this re-design. We will emphasize this in the next chapter.

5.5.1 Information model

An information model is crucial when we are storing data from different technologies. We developed a minimalist information model for the monitored data independently of the technology or the resource (e.g., VM, container, switches). The information model can be seen on Listing 5.12. The Measurement is specific for time series databases but can be easily changed to tables or documents. The data contained in the information model is: (a) the timestamp that data was collected (*timestamp*); (b) the slice identifier (*slice_id*); (c) the SP identifier (*slice_part_id*); (d) the resource identifier that can uniquely identify the resource (*resource_id*); (e) the unit of the value (*unit*) saved on the information model (e.g., percentage, absolute, Mb, Gb, Mbps); (f) the value collected of the specific metric

(*value*). The Listing 5.13 materializes the information model giving a real example of how it can be fulfilled.

```
Measurement A:
    timestamp: <Integer>,
    slice_id: <Text>,
    slice_part_id: <Text>,
    resource_id: <Integer>,
    unit: <Text>.
    value: <Float>
```

Listing 5.12 – Information model to store all the metrics.

```
Measurement CPU_UTILIZATION:
    timestamp: 1599921848,
    slice_id: slice1 ,
    slice_part_id: DSP1,
    resource_id: 5,
    unit: percentage ,
    value: 49.5
```

Listing 5.13 – Example of information model.

The SlEaaS can change the information model quickly, and with few adjustments, the TAs and database should be capable of monitoring the slice resources. The time series databases are indicated to store this kind of data since they are optimized to store time series information. The query simplification and the features, such as the alerting mechanism and the aggregation functions, combine a lot with monitoring systems, including the SEA.

6 Evaluation and Scenarios

This chapter aims to present the main results and artifacts that prove the feasibility of SEA. Each section will discuss the results presented in published or submitted scientific articles. We organized the sections chronologically to present the evolution of the architecture. First, in Section 6.1, our initial focus was to define an architecture to only monitor slice resources (as part of the NECOS project). This is our initial SEA design and has only the TAs (already running inside containers) being instantiated to monitor the slices. Therefore, the analysis was the starting point to verify the architecture's feasibility in terms of scalability and functionality. In this first section, the SEA was responsible for supplying the metrics of the slice resources with multiple administrative and technological domains.

Section 6.2 introduces the MAs to the SEA. In that time, we are providing not only a scalable architecture for monitoring slices but also managing the slices with the abstraction of VIMs, WIMs, and MEs. This evaluation focused on the CCL of computing resources, which means that the operations increase or decrease the VMs, containers, etc. At this moment, we decided to give more emphasis to the elasticity operations due to their importance. With the abstractions of MAs and TAs, the SEA could perform the VEs through the MAs. However, the idea of providing the SlEaaS and the policies administration are not part of our scope during this evaluation of the SEA.

With the research evolution and focus on slice elasticity, we defined the external APIs and how the components will interact and integrate to provide the SlEaaS, resulting in the last evaluation of the SEA proposal. This evaluation shows the CCL of networking resources performing the elasticity operations based on policies defined through the SEA. These aspects of SEA were evaluated in Section 6.3, providing several new characteristics that were not incorporated before in the previous evaluations, for example: (i) the policy administration by the SEA; (ii) the architecture support and evaluation of networking elements, before we evaluated only considering the computing resources; (iii) the database being chosen by the tenant during the start SEA and unique for each slice being monitored. Below we remember the research questions that will be answered in the following sections:

- ***RQ 1:*** Does SEA scale in response to several metrics from multiple slices monitored simultaneously?
- ***RQ 2:*** Does SEA support the CCL of computing resources instantiated in multiple administrative and technological domains?
- ***RQ 3:*** Does SEA handle CCL of networking resources, including policy creation?

- ***RQ 4:*** Is SEA generic enough to handle the elasticity operations, no matter which technologies are present in the slice?

6.1 SEA monitoring subsystem: scalability and feasibility

This section discusses our first results evaluating the monitoring subsystem and the idea of having the TAs running inside containers. The results are presented on (Beltrami et al., 2020). In this first publication, the SEA was not service oriented yet. Therefore, we evaluated the TAs performance to answer RQ1. As explained in Chapter 5.3, the initial proposal of SEA does not have the database running on containers, which means that the results in this section focused only on the TAs (the MAs and database containers are not introduced in the architecture yet). Nevertheless, it is very important to evaluate the TA since the TAs play an essential role in monitoring the slice resources, and it is the most consuming container in terms of activities, tasks and usage of computing resources.

The TAs consume more resources than the MAs and the database containers because of the capacity to increase the number of metrics being monitored or decrease the time interval to collect the data faster, generating more data. For example, suppose that the SEA monitors 100 slices with 1 SP on each. Considering the TA, we will have 1 TA per slice, monitoring n metrics from time to time. The SlEaaS provider running SEA will have at least 100 containers (TAs) running and collecting metrics. What is the performance of this? How long does it take to instantiate and stops this number of containers? Is the number of metrics collected for each TA influenced when the SlEaaS provider is overloaded? Those questions are part of RQ 1 and will be clarified below.

We evaluate the performance of the SEA when instantiating and removing TAs as the slice provider requests the creation, decommission and scaling after a HE scale out/in operations of the SEA components (only the TAs in this evaluation). We analyze the following aspects:

- The time required to scale out/in the monitoring components when increasing/decreasing the number of slice parts;
- The impact of instantiating/decommissioning one slice by varying the number of metrics being monitored in an environment comprised of several slices being monitored at the same time;
- The time between adding the metrics in RabbitMQ[1] (the old mechanism used to queue all the metrics to the unique database for all slices being monitored) until they are inserted into the database, varying the number of slices and monitored metrics.

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

6.1.1 Testbed

The evaluation setup comprises two servers (Intel® Xeon® CPU D-1518 2.20GHz, 64 GB RAM, 2 TB HD). We instantiated a set of VMs for representing the end-to-end slices. The servers act as the infrastructure providers and have one slice part with two VMs each. The MEs used in each slice part are Prometheus and Netdata, respectively. To represent the SlEaaS provider, we deployed the monitoring components in one HP desktop workstation, assembling a second-generation Intel i7 processor 3.40GHz, 32 GB of RAM, and 1 Terabyte of storage. This workstation was responsible for instantiating and managing the monitoring components for all slices, for example: the SEA Controller, the database, the TAs, and the distribution mechanism (RabbitMQ). All tests were performed 30 times, and the metrics polling interval was set to 10 seconds.

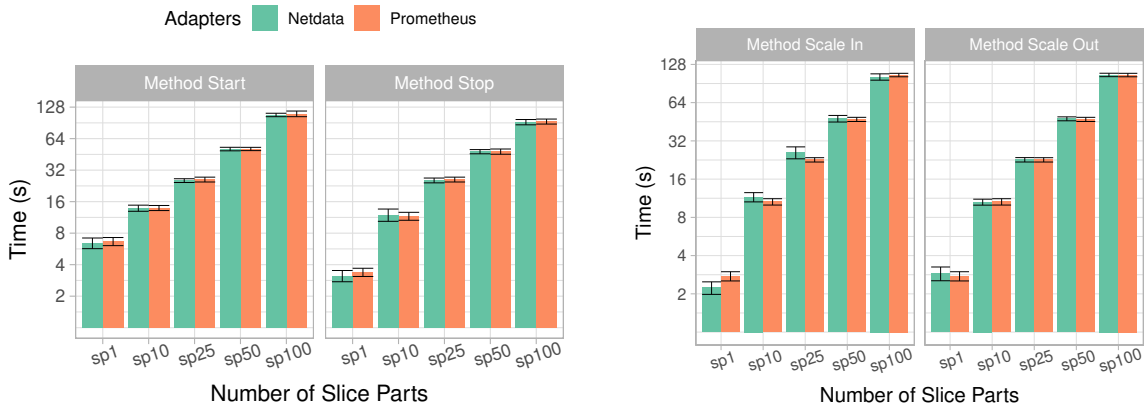
6.1.2 Impact of the number of SPs

Below we present the first analysis showing the performance of the SEA APIs: start, decommission (stop), and the scale SEA (for both scale out/in). In addition, we evaluate the time to create or delete the TAs containers when we call those actions varying the number of slice parts being monitored. This analysis considers the first two TAs implemented in this thesis, the Prometheus and NetData adapter. The TAs were configured with granularity-secs of 10 seconds to monitor the same 16 metrics per SP.

Figure 12a shows the average time to start and stop the monitoring components required for an end-to-end slice when its total number of slice parts grows from 1 to 100 on steps of 1, 10, 25, 50, and 100 SPs. Fig. 12b illustrates the average time to perform the scale SEA methods (for HE scale-in and scale-out) by also varying the number of slice parts being added or deleted (1 to 100 slice parts). In other words, the two graphs represent the time related to instantiating/removing only the requested TAs in response to the start, stop or scale SEA methods. In the x-axis, the reader will see labels such as *sp50*, which means the creation of one slice with 50 slice parts.

We can observe that the type of adapter (Prometheus in Green or Netdata in Orange) does not impact the execution time of the described methods since the obtained values were very close in all graphs. Regarding the architecture's performance, we noticed that the average time to instantiate the TAs components is slightly higher in the start method than the stop method, as seen in Figure 12a. On the other hand, the scale SEA methods had almost the same performance in terms of time to complete the TAs instantiation or removal. In quantitative terms, in the worst scenarios (labeled as sp100 on both graphs) where we are creating/deleting the 100 TAs to monitor one slice, the SEA took $\approx$ 105 seconds to finish the operation, which results in $\approx$ 1 second per slice part.

As mentioned in Section 5.2, we chose to instantiate the components of this ar-



(a) Average time to start and stop the SEA components.

(b) Average time to scale in and scale out SPs by the SEA.

Figure 12 – Impact of increasing the number of SPs on the start, stop, scale in and scale out operations.

chitecture in Docker containers, bringing all the benefits presented earlier (e.g., isolation, easy deployment/management). Consequently, we could observe that the time to instantiate/stop containers is about 95% of the total time presented in our analysis. If we chose another technology, such as processes, the time would be lower. However, we would lose all the benefits containers bring to the solution related to slicing.

Thus, taking $\approx$ 1 second to instantiate each component of a slice with many components is plausible, given the gains that containers bring to the solution. We also believe that the number of SPs that will make up a slice should not be high, which makes the solution even more viable. Although this performance is only possible because the SEA was developed to create and delete the containers in parallel, with a sequential approach, the results are unfeasible.

6.1.3 Impact of the number of slices

The purpose of this analysis is: (*i*) to show the impact of instantiating a new slice with just one slice part in environments that are already monitoring a set of slices; (*ii*) to analyze whether the number of metrics being monitored in new slices influences the time to start or stop the TAs.

Figure 13 illustrates the execution time for the start/stop methods. In these graphs, we are evaluating the impact of starting/stopping a new slice, with just one slice part, in scenarios where the number of slices already being monitored by the SlEaaS provider increases according to the following values: 1, 5, 10, 25, 50, and 100 slices. Each of these already deployed slices has two slice parts monitored by Prometheus. To carry out the second analysis (*ii*), we changed the number of metrics being monitored by the new instantiated or removed slice according to the following values: 10, 100, 1000, and 10000,

(labeled respectively m10, m100, m1000 and m10000).

The number of metrics being monitored in the new instantiated/removed slice did not impact the obtained results, i.e., the time to create/delete components in the proposed architecture does not change, even in scenarios where the number of metrics being monitored increases considerably. However, as the number of slices being monitored increases, the time to instantiate/remove the TAs can also increase considerably, depending on how overloaded the environment is. For example, the time to start a single-slice-part slice in an environment with only one slice already deployed (label *s1* in red) was close to 5 seconds, regardless of the number of monitored metrics. On the other hand, to create the same slice in an environment that is already monitoring 100 slices (label *s100* in pink), the average time has increased to about 15 seconds in all cases shown in Fig. 13. The stop method has a similar behavior.

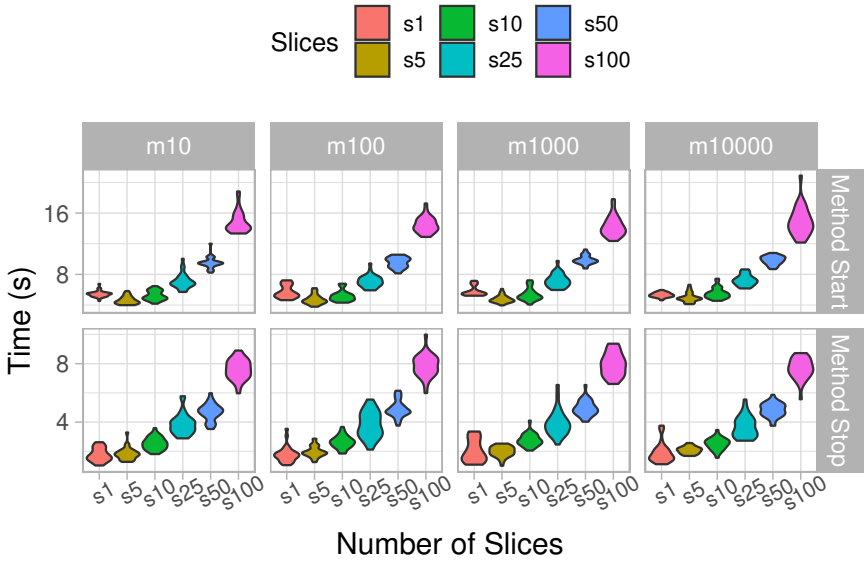


Figure 13 – Average time to start and stop SEA components in scenarios overloaded.

In this analysis, we raise the environment load, increasing the number of slices monitored by the SEA. We would like to see if the number of metrics being monitored and stored in the data negatively impacts the time to instantiate or remove the components to monitor a slice. The results corroborate the answer to RQ 1. First, the SEA is sensible with the SlEaaS provider load, which can give us an aspect to be improved in future work, considering scaling the SEA running on a master/worker strategy or distributed system. This sensibility can be seen in Figure 13. The second concerns the number of metrics being monitored, not impacting the time to create or delete the TAs.

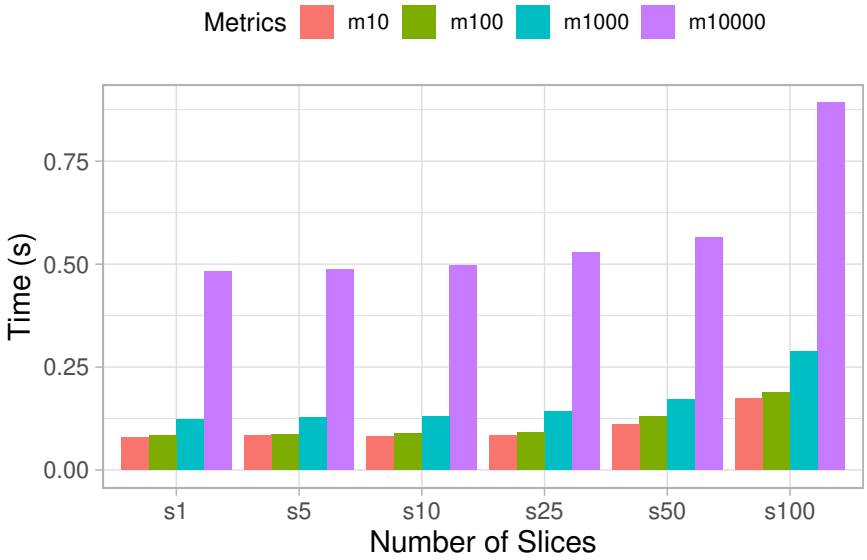


Figure 14 – Impact of several metrics in overloaded scenarios.

6.1.4 Impact of the number of metrics

The analysis carried out here focused on the performance of the monitoring components when the number of metrics being monitored increases. The measurement includes the time required by the TA to format the monitored metrics (according to the proposed information model), to publish them to the message queue RabbitMQ, and finally for a database agent (not present in the last SEA) to consume and add them to the database. In this analysis, as the previous one, we analyze the impact of this time on overloaded environments by varying the number of slices already being monitored in the environment and the number of monitored metrics in the new instantiated slice.

Figure 14 shows the time taken to process metrics until they are inserted into the database, following the same information model described in Section 4.1. The time increases as the number of metrics being monitored also grows. Therefore, to process and store 10000 metrics (label *m10000*) in a single-slice environment (label *s1*), it takes 0.48s on average, while the time to process and store the same amount of metrics in a 100-slice environment is 0.88s on average.

Therefore, the number of metrics does not influence the time to start or stop monitoring components of a slice, even in overloaded environments. On the other hand, the time to process metrics in overloaded environments rises as we increase the number of metrics being processed. With the three analyses presented, we can answer RQ 1 and start answering RQ 4. We can see that number of metrics does not significantly impact the SEA performance. The time to start, decommission and scale SEA components is promising if compared with the time to instantiate a new slice over the sea. Which took almost 20 minutes (Clayman et al., 2021b). We identify a decrease in the performance when the SEA

monitors 100 slices, considering the computer used in the testbed.

6.2 SEA (aka CNS-AOM): use cases on real slices

This section presents the SEA design published in the journal (Rocha et al., 2022). Therefore, we explore the architecture functionalities with use cases instantiated over real slices with multiple administrative and technological domains. The use cases comprise two slicing services instantiated over infrastructure in different cities and countries. In addition, we also performed the VE and HE elasticity operations on computing resources.

In these results, we have not yet introduced the policies and concept of SlEaaS and the SEA was still coupled with NECOS. The triggers used in this evaluation are the responsibility of a NECOS component called the Slicing Resource Orchestrator (SRO), which at that time was not part of the scope of the SEA. However, when we decoupled our components from the NECOS project providing the SlEaaS, some of the SRO responsibilities took place on the SEA design, for example: the policy administration and all the external APIs defined. At final of this section, the reader will see the SEA providing the CCL of real slicing services and the defined slice elasticity operations being performed on the computing resources improving the service quality. We intend to answer RQ 2 in this section and continuing answer RQ 4 about the SEA support of multiple administrative and technological domains.

We describe two services instantiated over distinct CNSs, aiming to validate the proposal through the PoC implementations. The SEA monitors, manages, and orchestrates the two different slices. The first slice allocates a content distribution network (CDN) service deployed among three cities in the state of São Paulo, Brazil. The second slice instantiates the IoT platform called *dojot* (CPqD, 2020), comprising an architecture of microservices in which the development and offering of IoT services are supported. The CNSs presented in this evaluation are deployed across different cities in the case of the CDN and different countries (Brazil and Spain) in the case of the IoT platform which is an important characteristic of the slicing concept. With those, we highlight essential assumptions of CNSs, such as the support of multiple administrative and technological domains and the elasticity operations.

The purpose of the PoCs discussed in this section is to verify whether the proposed architecture can provide monitoring, management, and orchestration of CNSs. First, the CDN PoC will demonstrate the service functionalities, deployment of SEA components, the service deployment through MAs and multiple VIMs, and the service metrics being monitored through TAs from multiple MEs. Second, the *dojot* PoC will evaluate the slice elasticity operations by using a tenant-defined policy (via the SRO since the Policy API is not introduced yet) and periodically monitoring the CNS through the SRO component.

If the monitored metrics exceed a predefined threshold, the SRO triggers an elasticity operation and adapts the SEA components when necessary (in the HE case). In the following two subsections, each of the services will be presented in detail, specifically regarding the configuration setup and analysis of the obtained results.

6.2.1 CDN service description

This service provides multimedia content (photos, videos, etc.) of tourist areas to end-users. Based on the user's location, the CDN can provide content from the closest server and minimize the delay in downloading the content. The CDN comprises a core cloud, responsible for processing the end-users requests, and edge clouds instantiated on demand, responsible for delivering the content close to the requester. The reason behind this service is that users (tourists) located in high-profile tourist areas are more likely to request content about local sites and consume videos on limited-resource devices (e.g., tablets and smartphones with battery and bandwidth limitations). Therefore, we use SEA via the MAs to promptly instantiate the service on local edge clouds and deliver the content from a closer server, seeking to reduce the delay of downloading the consumed multimedia. The functionalities of the core and edge cloud components are as follows.

- The `core cloud` hosts a main service page and a video streaming content repository related to tourist attractions worldwide. A DNS lookup service is also deployed, which is capable of redirecting a user's request to the most appropriate server (edge or core) based on the requester's location;
- The `edge cloud` provides web and video servers hosting cached content geographically distributed in predetermined areas around the world. These servers are responsible for replying to redirected requests to where each edge cloud service is located. For example, assuming the edge cloud infrastructure is already deployed, the video service will be instantiated upon a request to the core cloud from a user closer to the edge servers.

6.2.1.1 Configuration setup

Based on the proposed CDN service, we consider essential aspects of CNSs, such as geolocation, multiple VIMs and MEs, and different administrative domains. This PoC aims to show that the SEA is capable of monitoring and managing the CDN slices and validating these important aspects through a real scenario following a set of steps, which are detailed next.

Figure 15 illustrates the steps performed in the evaluation of the CDN, as well as the slice deployment. Three slice parts are instantiated across different cities of São Paulo state, Brazil. The slice provider contains the SEA components and is deployed in Sorocaba

city. The core cloud CDN is also located in Sorocaba. The first edge cloud is deployed in the city of São Carlos, and the second edge cloud is located in Campinas. In addition, to represent the technological heterogeneity supported by the SEA, different VIMs and MEs are chosen for each slice part.

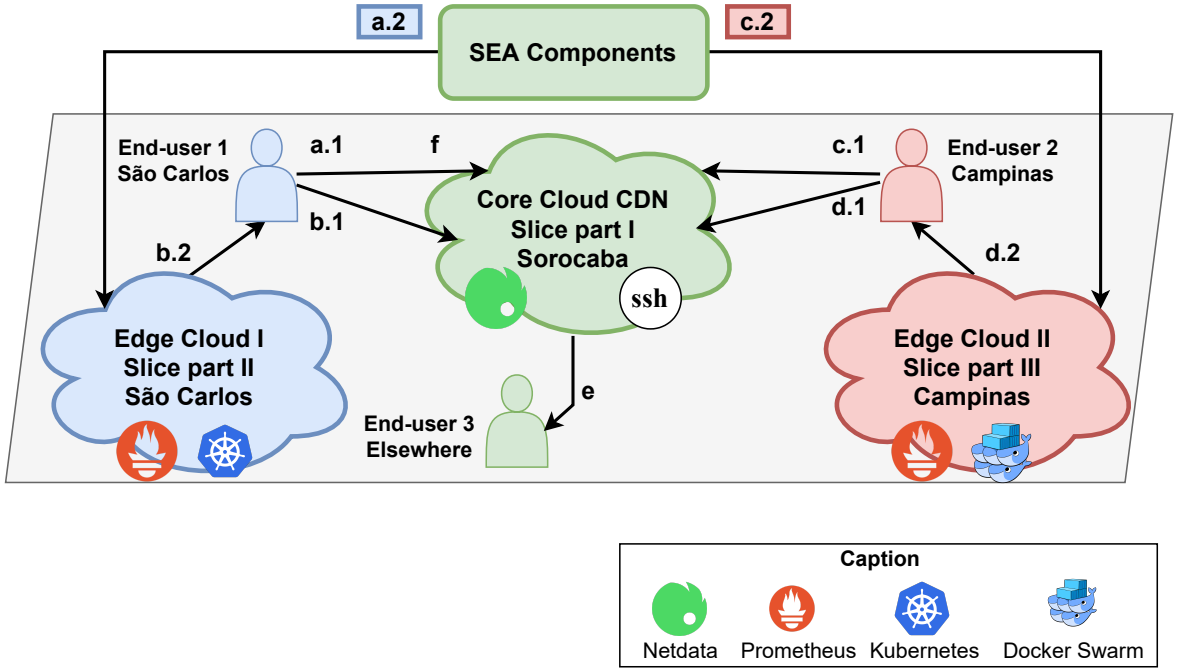


Figure 15 – CDN service across São Paulo cities.

The steps in Figure 15 demonstrate the monitoring and management operations performed. For the sake of simplicity, a single video (≈ 2.4 MB) is present in each slice part and is transmitted at 20 kBps to the three end-users requesting video content. In this PoC, requests do not coincide since it aims to validate the SEA management (including monitoring the infrastructure). To identify the end user's location, we define and implement different subnets for each city (São Carlos, Sorocaba, Campinas, and others). We also assume that there are enough resources to deploy and start the edge services closer to the user. The network connectivity between the slice parts is simplified using a VXLAN tunnel through the Internet. Finally, the following steps are executed to collect the metrics from the offered service:

a.1) `end-user 1`, located in `São Carlos`, requests a `video` from the `core cloud` service, which delivers the requested video;

a.2) after identifying one request from São Carlos, the `core cloud` service deploys `edge cloud 1` through the MAs;

b.1) `end-user 1` requests the `video` from the `core cloud` service again;

b.2) the video is delivered by `edge cloud 1`, since it is geographically closer to `end-user 1`;

c.1) `end-user 2`, located in `Campinas`, requests a `video` from the `core cloud` service, which delivers the requested content (likewise to step `a.1`);

c.2) after identifying one request from Campinas, `edge cloud 2` is deployed through the MAs;

d.1) `end-user 2` requests the `video` from the core cloud service again;

d.2) the video is delivered by `edge cloud 2`, as it is geographically closer to `end-user 2`;

e) `end-user 3`, located elsewhere, requests a `video` from the `core cloud`, which delivers the requested content;

obs: after a period of inactivity, both edge cloud services are decommissioned using the MAs to stop the service;

f) `end-user 1` requests the `video` for the third time, which is again delivered by the `core cloud` service as `edge cloud 1` was decommissioned.

For the evaluation, we assumed that the SEA had already deployed core cloud service and could distribute web and video content to end users. In the same way, the edge clouds are previously instantiated, but the edge services are activated only after a core cloud request, depending on the end user's location. Then, the SEA deploys the services and per-slice components. Next, the service checks the end user's location based on the IP address to redirect the video. If the closest edge service is not available, the infrastructure orchestration component starts the closest available edge service and gets the video from the core cloud. On the other hand, the end user gets the video from the edge service closer to it. The experimental setup used for this assessment comprises the following configuration:

- **Slice provider**
  - **Slice provider configuration:** Supermicro model X10SDV-TP8F, with OS Linux Ubuntu 18.04 LTS, Kernel 4.15.0-58-generic x86_64, CPU Quad core Intel Xeon D-1518 2.2GHz (-MT-MCP-) with 8 threads, RAM 64GB DDR4 and HD 2TB.
  - **Localization:** Sorocaba - São Paulo, Brazil.
- **Core Cloud - Slice Part 1**
  - **Resource provider configuration**: Supermicro model X10SDV-TP8F, with OS: Linux Ubuntu 18.04 LTS, Kernel: 4.15.0-58-generic x86_64, CPU Quad

core Intel Xeon D-1518 2.2GHz (-MT-MCP-) with 8 threads, RAM 64GB DDR4 and HD 2TB.

  - **Localization:** Sorocaba - São Paulo, Brazil.
  - **VIM:** We use an SSH adapter to interact/deploy the service.
  - **Monitoring Entity:** Netdata.

- **Edge Cloud - Slice Part 2**
  - **Resource provider configuration:** One virtual machine with 8gb de RAM, OS Linux Ubuntu server 18.04 LTS, 4 virtual CPUs and 100 GB of storage.
  - **Localization:** São Carlos - São Paulo, Brazil.
  - **VIM:** Kubernetes.
  - **Monitoring Entity:** Prometheus.

- **Edge Cloud - Slice Part 3**
  - **Resource provider configuration:** Supermicro model X8DT3-LN4F, OS: Linux Debian 9, CPU Intel Xeon E-5520 2.26GHz with 8 MB cache, RAM 8GB DDR3 1066, HD 1TB.
  - **Localization:** Campinas - São Paulo, Brazil.
  - **VIM:** Docker Swarm.
  - **Monitoring Entity:** Prometheus.

6.2.1.2 CDN service evaluation

Figure 16 presents the number of bytes transmitted by each service over time. In the case of the *core cloud*, the CDN service runs as an application on a physical host. The edge services run inside containers for both `edge cloud 1` and `edge cloud 2`. The labels **a**, **b**, **c**, **d**, **e**, and **f**) represent the steps described in the previous section. For instance, label **a** illustrates the transmitted bytes for the `core cloud` service considering steps **a.1** and **a.2**.

Based on the plot in Figure 16, we conclude the following important observations. First, the black lines represent the `core cloud` service located in Sorocaba and show the time between receiving the request and delivering the video to the end user. A short time interval between steps indicates that all operations executed by SEA are performed as expected. These operations include the start of the SEA components, the service monitoring, and the service deployment via the MAs.

The containers running `edge cloud` instances are deployed shortly before steps **b** and **d**; more precisely, during steps **a.2** and **c.2**, respectively. Right after, the TAs

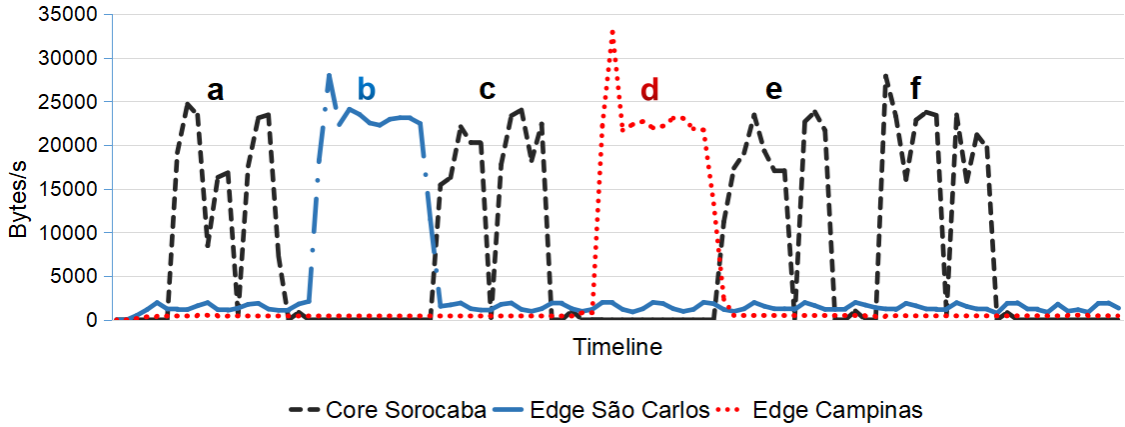


Figure 16 – Bytes/s transmitted by core cloud and edge cloud instances over time.

can collect the metrics from the service being monitored as soon as the edge SPs are instantiated. The indicated behavior is illustrated in the figure using blue and red lines, representing metrics collected from `edge cloud 1` and `edge cloud 2`, respectively.

Finally, step **e** shows that *end-user 3*, who is located elsewhere, receives the video from the *core cloud* service, which in turn is located geographically closer to it. After a period of inactivity, the edge services are decommissioned. The last request from *end-user 1* is delivered by the *core cloud* service (step **f**) because *edge cloud 1* is no longer in service. However, after the MAs' instantiate the edge services, the content is delivered to the requester from the closest server.

6.2.2 IoT service description

Figure 17 illustrates the IoT service "big picture", which consists of a real-time cargo monitoring and tracking solution. There are monitoring devices with multiple sensors (temperature, humidity, light, and GPS) communicating through wireless networks. Those sensors travel in cargo containers during their journey. Their purpose is to provide monitored data periodically from the sensors to a centralized system located in the core cloud, allowing customers to have up-to-date information about their cargo during transportation. The sensors send the retrieved information, such as door openings, extreme temperature shifts, localization, and humidity, to the core cloud instance.

The IoT service is instantiated in a real CNS setup, built across four cities Goiânia, Campinas, Sorocaba, and Madrid (Spain). The objective is to show the computing elasticity operations being performed by the SEA based on the monitored metrics from the CNS infrastructure.

An open-source IoT platform is used to implement the aforementioned service. The development of *dojot* is led by CPqD and comprises several services such as Kong, Redis, Kafka, Zookeeper, MongoDB, and PostgreSQL. All these services are configured

Figure 17 – IoT service big picture.

and deployed to provide IoT service validation. In addition, a load testing tool is developed for message queuing telemetry transport (MQTT) of IoT devices, which is used to increase the number of requests.

Figure 18 illustrates how the *dojot* platform services are allocated in two different slice parts, one representing the core cloud and another representing the edge cloud. The former includes all the services running (in yellow at Campinas, Brazil), whereas the latter (in red at Madrid, Spain) uses an MQTT IoT agent to monitor the devices and a Redis microservice to publish the monitored metrics to the core when deployed. The main responsibilities of the core cloud are managing the IoT device lifecycle, storing the device's measurements, and providing a REST interface to retrieve historical and real-time data about the devices. One MQTT IoT agent, responsible for interacting with the IoT devices and sending the collected metrics to the core cloud, is instantiated for each edge cloud.

6.2.2.1 Configuration setup

The *dojot* slice is composed of three slice parts, one core cloud instance, and two edge cloud instances. The objective is to show the slice being monitored, managed, and orchestrated by the SEA. As shown in Figure 19, the SEA is located in the city of Goiânia, representing the slice provider. The core cloud is located in Campinas and comprises all the *dojot* microservices. The edge cloud instances are deployed in the city of Madrid as

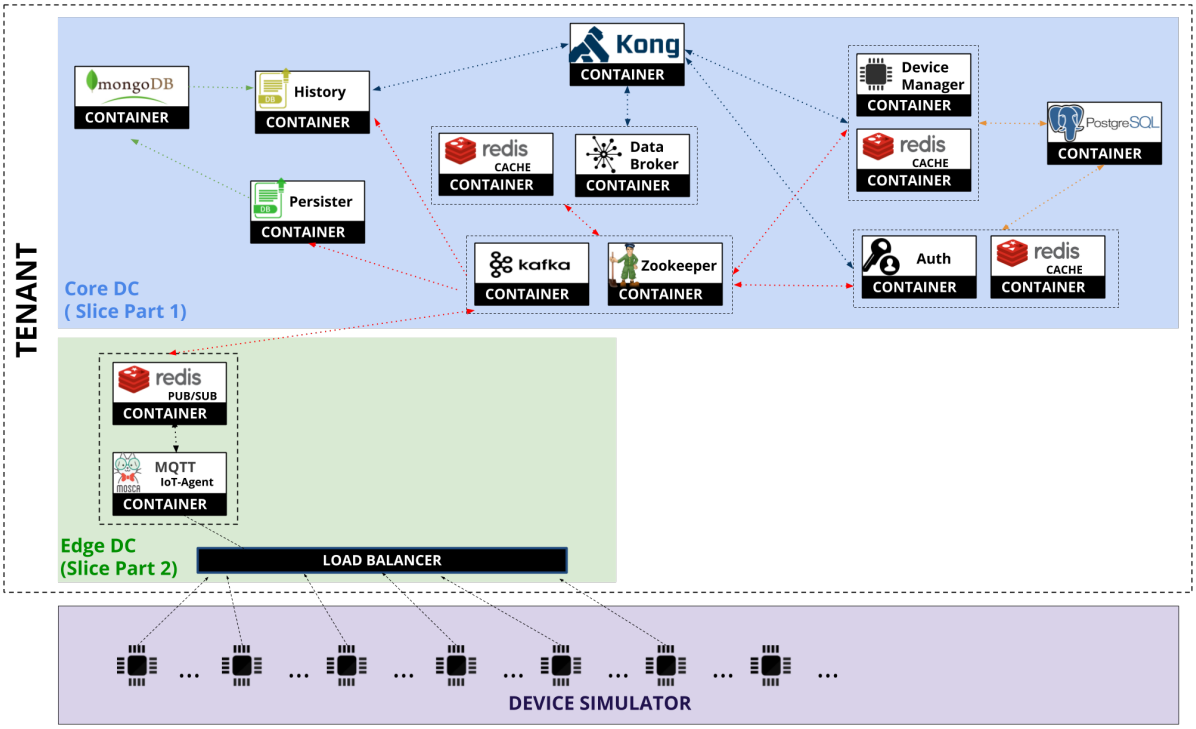


Figure 18 – Microservices of the *dojot* platform.

well as in the city of Sorocaba. VXLAN tunnels provide connectivity between the different slice parts to connect the cities between Brazil and Spain. The VXLAN tunnels are created during the slice creation phase.

The complete description of the configuration setup used in this experiment follows.

- **NECOS slice provider**
  - **Slice provider configuration**: Dell EMC PowerEdge R740 server, equipped with two Intel Xeon Silver 4114 processors, 128 GB (8x 16GB RDIMM, 2666MT/s, Dual Rank) of RAM, and 12 TB of HD.
  - **Localization:** Goiânia - Goiás, Brazil.
- **Core Cloud - Slice Part 1**
  - **Resource provider configuration**: Dell PowerEdge R740, with OS: Linux Ubuntu 18.04 LTS, Kernel: 4.15.0-51-generic x86_64, two Intel Xeon Silver 4114 CPU 2.20GHzD-1518 processors, RAM 64GB DDR4 and HD 2TB.
  - **Localization:** Campinas - São Paulo, Brazil.
  - **VIM:** Kubernetes.
  - **Monitoring Entity:** Netdata.
- **Edge Cloud - Slice Part 2**

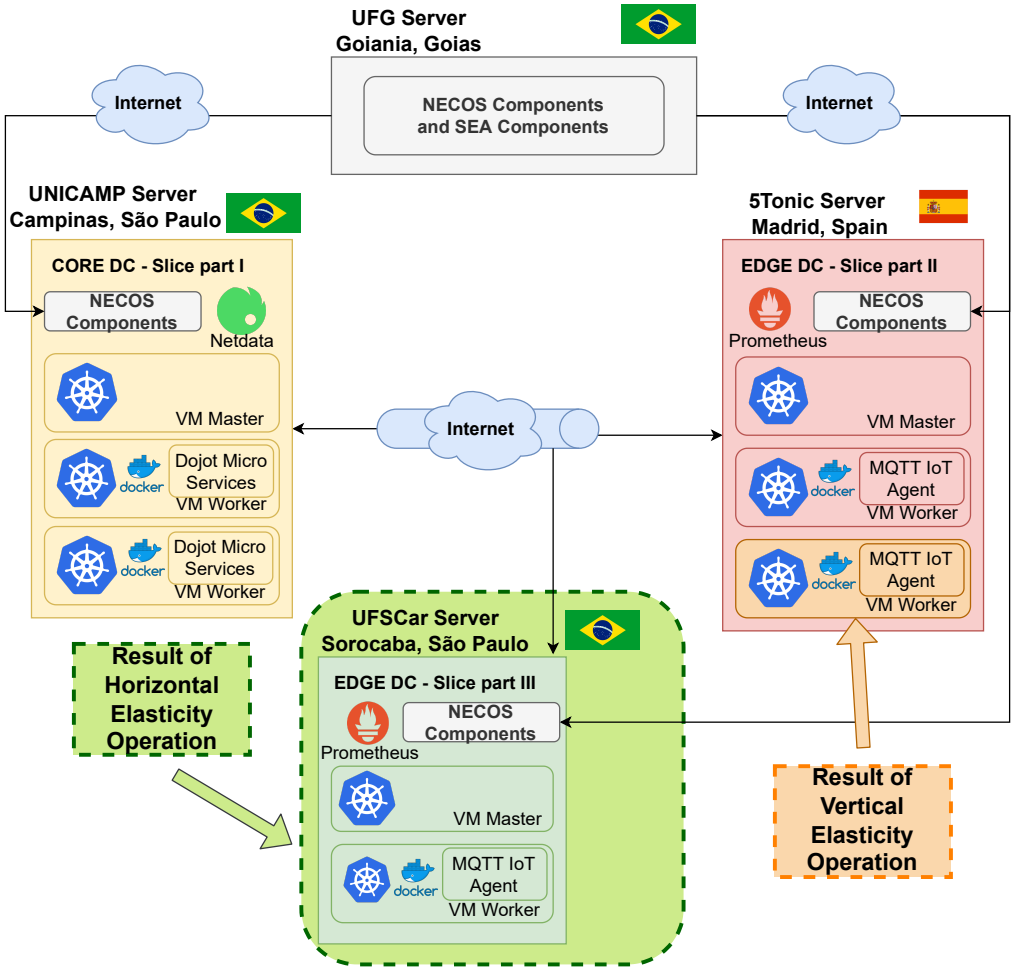


Figure 19 – *Dojot* slice PoC deployment.

- **Resource provider configuration:** Dell EMC PowerEdge R720 with 2x Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, 6 cores per socket (for a total of 12 cores or 24 vCPU), 128GB of RAM, and 2TB as HDD.
- **Localization:** Madrid, Spain.
- **VIM:** Kubernetes.
- **Monitoring Entity:** Prometheus.

- **Edge Cloud - Slice Part 3**
  - **Resource provider configuration:** Supermicro model X10SDV-TP8F, with OS: Linux Ubuntu 18.04 LTS, Kernel: 4.15.0-58-generic x86_64, CPU Quad-core Intel Xeon D-1518 2.2GHz (-MT-MCP-) with 8 threads, RAM 64GB DDR4 and HD 2TB.
  - **Localization:** Sorocaba - São Paulo, Brazil.
  - **VIM:** Kubernetes.
  - **Monitoring Entity:** Prometheus.

In this work, we consider two types of elasticity: vertical and horizontal. Vertical elasticity means the addition of a new resource (in this case, a new VM worker) in one slice part already instantiated. Horizontal elasticity means adding a new slice part with the service running inside it. For more detailed information about elasticity operations in the context of CNSs, we recommend the work presented in (Beltrami et al., 2020). This PoC experiment aims to demonstrate the SEA's capacity to execute both elasticity operations fully in a real environment. Seeking to show these operations, we start the *dojot* slice with only the core cloud and edge cloud 1, allowing for edge cloud 2 to be added after an elasticity operation.

The experiment attempts to increase the requests to IoT devices located at edge cloud 1 using a tool developed by the CPqD development team. Based on a policy in the infrastructure orchestration defined by the tenant, the CPU usage of the infrastructure needs to be monitored, and an elasticity operation has to be triggered if this metric exceeds an 80% threshold value three times consecutively. We use a time window of 5 seconds. After interpreting this information, the SRO (infrastructure orchestration) has to check the metrics the TAs monitor periodically. An elasticity operation is triggered upon detecting an increase in CPU usage due to many requests. Upon completion, two different elasticity scenarios are exercised. First is a VE operation, adding a new worker VM to edge cloud 1 (highlighted in orange on Figure 19). Second is a HE operation, adding a new slice part to edge cloud 2, as shown in green in Figure 19.

The vertical elasticity using Kubernetes as a VIM does not require the (re)deployment of the service, as it is handled automatically by the VIM. This behavior also happens with Prometheus and Netdata MEs. Therefore, after the occurrence of a VE operation in the SP already being monitored, these entities automatically recognize the new resource, and the TAs start to store the collected metrics. However, after executing a HE operation, the service needs to be instantiated in the new slice part, considering that a new VIM is also instantiated. The MAs should perform the service's (re)deployment and communicate with the VIMs to distribute the requests to the IoT devices. The following subsection details the elasticity operations and the obtained results of these use cases.

6.2.2.2 IoT service evaluation

We aim to show the results and describe the most important challenges addressed in this evaluation. More specifically, we present the results in terms of the time required to (i) instantiate the whole slice infrastructure and VIMs (on the scope of the slice provider); (ii) deploying the *dojot* service (on the scope of the MAs); and (iii) performing a HE operation. In addition, we discuss the timeline of the CNS being monitored before and after the elasticity operations are performed. Finally, we identify three important moments when analyzing the monitored metrics; first, when the infrastructure orchestration triggers

the elasticity operation; second, when the monitoring of the new resources (VM or slice part) starts; and third, when a decrease in CPU usage occurs after the addition of a new resource, as this load-balanced the requests.

Table 5 presents the instantiating times of the *dojot* slice service deployed in a real setup environment between Brazil and Europe. First, we observe that the slice infrastructure instantiation takes 1917.16 seconds (about 32 minutes) to be performed, from the time a request is made to the slice provider (NECOS platform) until all resources, management, and monitoring components are up and running. During this time, the task that takes more time to complete is the VIMs/WIMs deployment, completed in 1718.74 seconds (about 28 minutes). Second, the reservation time, represented by the time elapsed to choose the best resource allocation options, takes 9.58 seconds. After slice creation and component deployment, the MAs take 505.81 seconds (about 8 minutes) to deploy the *dojot* platform using Kubernetes as VIMs in the core cloud (Brazil) and the edge cloud (Spain). This time is taken from a tenant service deployment request until all the microservices run correctly. The total time elapsed to deploy the slice and to have the services operational is 2432.55 seconds (about 40 minutes).

Slice Instantiation Time (s)	1917,2
Reservation Time (s)	9,6
Dojot Service Deploy Time (s)	505,8
Total	**2432,6**

Table 5 – The *dojot* slice deployment times.

Next, we discuss the monitoring moments when the elasticity operations are triggered. In Figure 20, we can see the CPU usage for each VM inside the edge cloud in 5Tonic (Spain), and the effects of the VE implemented during the test in the first scenario. We highlight three important moments marked as **P1**, **P2**, and **P3**. Point **P1** depicts the time the infrastructure orchestration first detects an increase in CPU usage due to increased requests for IoT devices. At this moment, the SEA starts the VE operation, which aims to deploy a new worker VM in edge cloud 1. After a few minutes, the new VM is deployed and can be seen in the figure as **P2**. Finally, **P3** shows the moment when the requests are equally distributed between the two workers' VMs, decreasing the CPU usage of the first worker VM (lines in yellow and blue).

The horizontal elasticity operations follow a similar procedure. However, here we observe a new slice part deployment and distribution of the IoT device requests, as illustrated in Figure 21a. For the 5tonic CPU usage monitoring, **P1** highlights the moment when the infrastructure orchestration triggers the HE operation due to the high volume of device requests. Figure 21b shows that the Edge Cloud 2 deployment starts at **P2**, activating the monitoring of the new infrastructure. A few minutes later, monitoring the

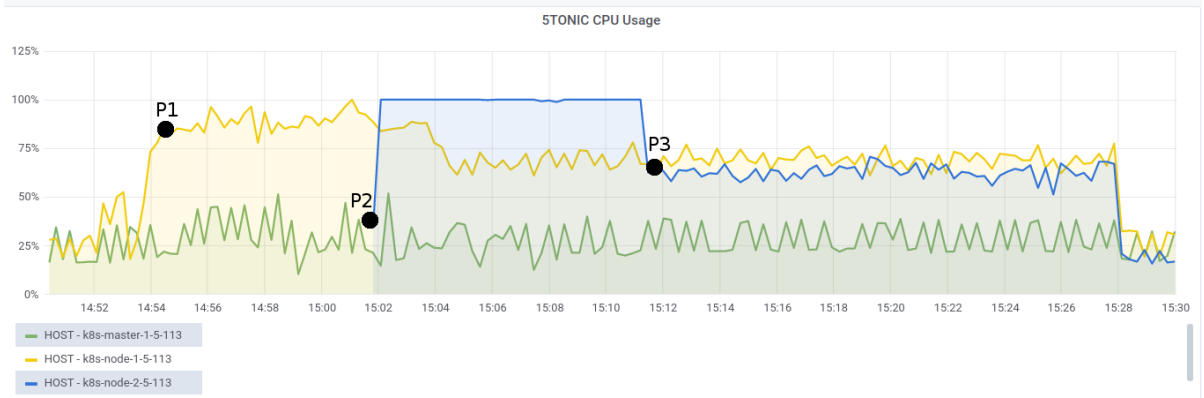
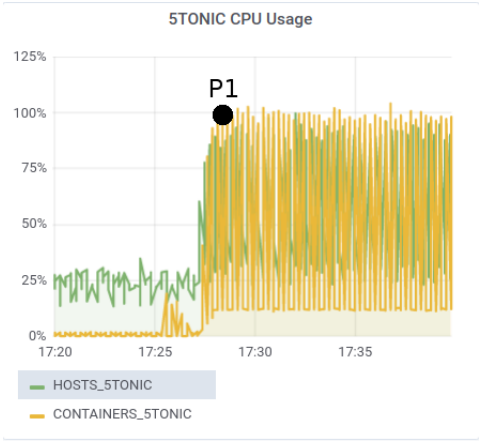
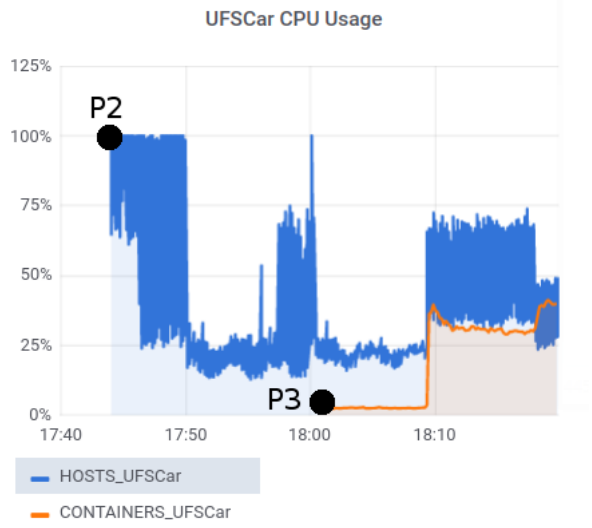


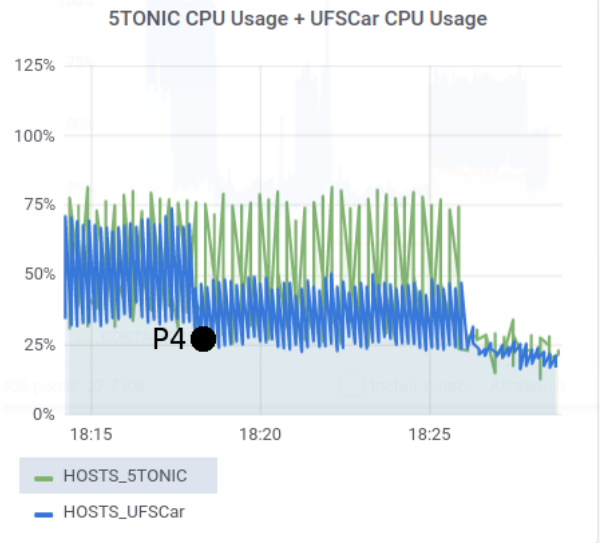
Figure 20 – Vertical elasticity operation taking place.



(a) Dashboard - 5tonic Slice Part.



(b) Dashboard - UFSCar Slice Part.



(c) Dashboard - 5tonic + UFSCar.

Figure 21 – Horizontal elasticity operations taking place.

dojot microservices by the new TA is activated, as indicated by **P3**. Finally, Figure 21c presents the splitting of requests for IoT devices between both edge clouds 1 and 2 (5tonic

and UFSCar), as highlighted by **P4**.

In this section, we answered RQ 2 by exercising the elasticity operations through the SEA. We demonstrated slice monitoring, management and orchestration with the objective of performing elasticity operations on computing resources to improve QoS. Therefore, with these PoCs and results, we validate the following important aspects of the proposed solution contributing to the answers of RQ 2 and RQ 4.

- The instantiation of a real IoT scenario overseas and CDN scenario over cities, ensuring the communication between the services;
- The execution of management and monitoring tasks for the IoT service by handling the computing elasticity operations;
- The presentation of part of the SEA functionalities, as well as the accuracy of monitored metrics reflecting the actual state of the infrastructure;
- The interoperability capacity in a multi-domain environment and the abstraction of working with different MEs and VIMs;
- The infrastructure and service metrics are collected from two different monitoring entities (Prometheus for the edges and Netdata for the core cloud).

6.3 SEA providing the CCL of networking resources

The last evaluation section comprises the SEA providing the SlEaaS, which includes decoupling from the NECOS project, defining external APIs, and including the policy administration and policy checker mechanisms. Also, this evaluation focuses only on the networking resources, which add a new MA and TA implementation to support the Ryu controller as WIM and the ME to collect the networking metrics. At final of this section, the reader will be able to verify if all the four RQs defined are well answered with the evaluations proposed in this thesis. The main objective of this section is to perform the networking elasticity operations defined in Sub-Section 2.3.2 to ensure that the service performance was improved as a result of the CCL on the scenarios specified below.

This section aims to demonstrate the three elasticity operations defined previously. In addition, experimental scenarios were elaborated, ranging from the instantiation of components necessary to orchestrate the slice, to the creation of policies and, finally, the execution of the elasticity operation. Fig. 22 illustrates the evaluation scenarios corresponding to the three elasticity operations defined in Sub-Section 2.3.2. At first, Fig. 22 (a) shows the Slice 1 infrastructure in its initial state, which consists of two DSPs connected via a WSP. Two applications run on VMs in each DSP, representing an E2E communication service. App 1 is running through VM2 (DSP 1) and VM3 (DSP 2), while

App 2 is running through VM1 (DSP 1) and VM4 (DSP 2). The WSP1 connects both DSPs via three switches (S1, S2, and S3), managed by an OpenFlow Controller as WIM.

In all scenarios, the objective is to improve the throughput of App 2 (highlighted with a yellow star) through the execution of elasticity operations. App 1 is used as background traffic to assist in the evaluation. Fig. 22 (b) represents Scenario I, where the *Vertical Elasticity Queue Scale Up* is executed. Scenario II is shown in Fig. 22 (c), corresponding to the *Vertical Elasticity Topology Scale Up*. Lastly, Fig. 22 (d) illustrates the *Horizontal Elasticity Scale Out* on Scenario III. All elements highlighted in purple inside the figure indicate the resources added due to the elasticity operations.

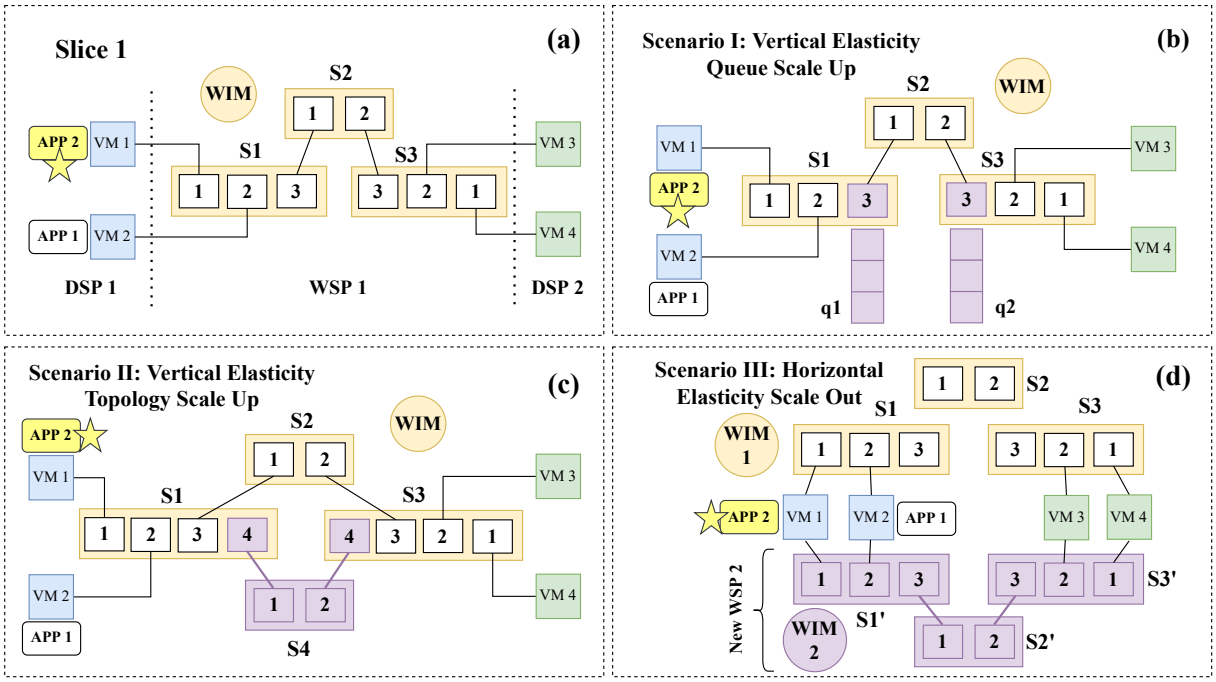


Figure 22 – SEA networking elasticity operation scenarios evaluated.

For the experimental setup, we used Mininet[2] for emulating open vSwitches with support to the protocol OpenFlow v1.4. Moreover, the OpenFlow Ryu Controller was chosen for the control plane of WSPs. We assumed a total bandwidth limited to 20 Mbps and used the Iperf3 tool[3] to generate traffic for both applications. Regarding the traffic composition, App 1 comprises 9 TCP flows and App 2 of 1 TCP flow, totaling 10 flows competing for the bandwidth capacity of 20 Mbps. No limitations were imposed for any TCP flow. The elasticity operations aim to improve the throughput of App 2 compared with the throughput before the elasticity operation is triggered.

The infrastructure used as a testbed is composed of VMs instantiated over the same physical computer with 32 GB of RAM DDR4, an Intel(R) Core(TM) i5-9600K CPU, running Windows 11 as the SO. The VMs were virtualized using the Hyper-V Manager

[2] http://mininet.org/.
[3] https://iperf.fr/

and were instantiated two VMs with 8 GB of RAM, 2 vCPUs, and running the Ubuntu 20.04 LTS. Each of those VMs represents one WSP with the Mininet installed inside. The SEA was running inside a third VM with the same hardware configuration but with the SEA installed and the APIs running correctly. It is important to highlight that the SEA instantiated the per-slice components in the same third VM.

During each experiment, the four-phase workflow (Section 2) is executed to accomplish the entire CCL. As we focus on the network slice part, only the actions on related elements are described next. Therefore, in P1, the dynamic per-slice components were created. In addition, the MA adapter interfaces the Ryu Controller inside the WSP, and the TA adapter also collects monitoring metrics from the Ryu Controller using the OpenFlow protocol. So, in this case, the Ryu Controller also acts as the ME. In P2, the configured policy defines that when a given switch port reaches 18 Mbps of received rate, an elasticity operation must be triggered[4]. The policy checking interval is 5 seconds. In P3, the elasticity operation is executed in each scenario. Finally, when needed (Scenario III), P4 is executed to instantiate the new MA and TA elements.

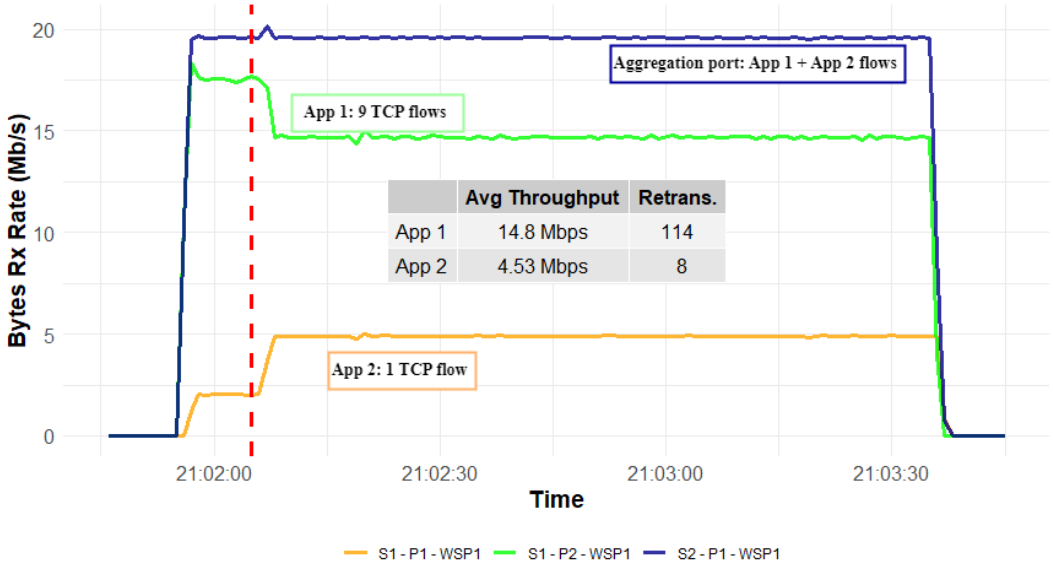


Figure 23 – Scenario I: VE queue scale up.

6.3.1 Scenario I: Vertical elasticity queue scale up

- **Policy Action:** After detecting policy violation, this elasticity operation triggers the creation of two queues with a ceiling throughput of 5 Mbps on port 3 of both S1 and S3 (purple queues on Fig. 22(b)). As defined by P2 in Section 4.1, the tenant provides the extra information necessary for the execution of this operation (group 6).

[4] Applications' metrics can also be used, such as frames per second, retransmissions, etc.

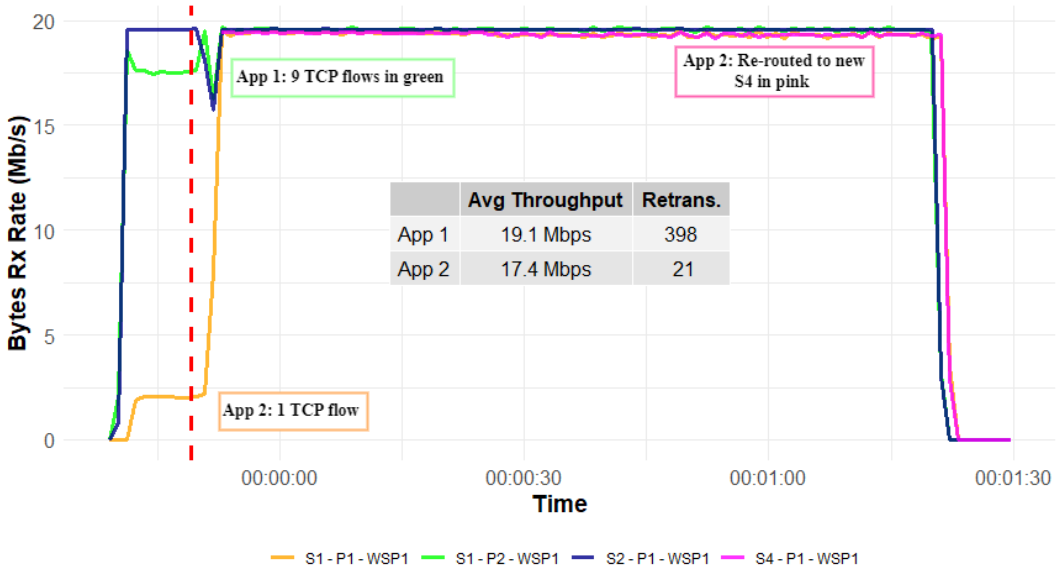


Figure 24 – Scenario II: VE topology scale up.

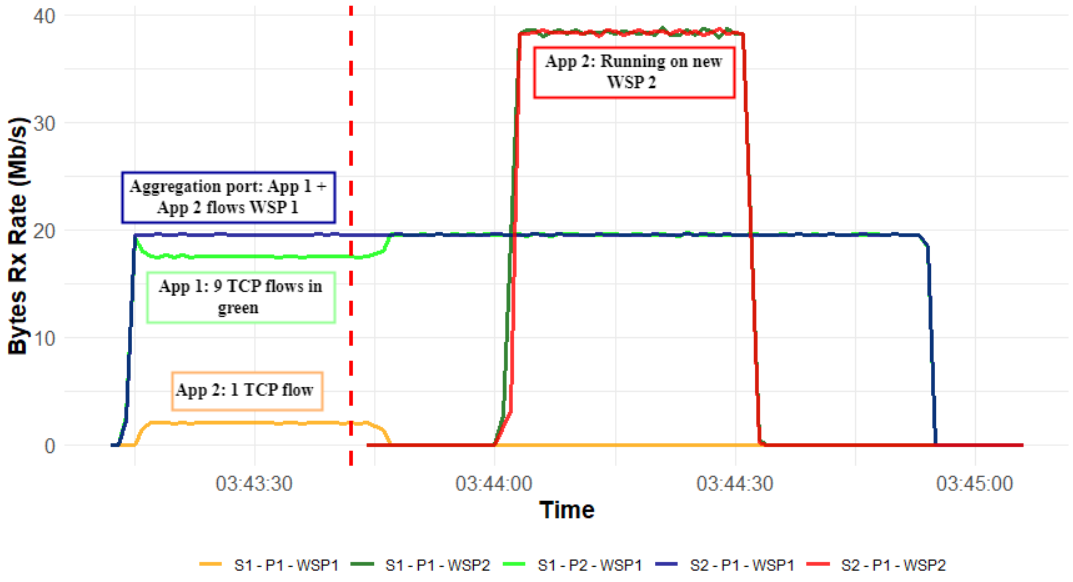


Figure 25 – Scenario III: HE scale out.

- **Results:** Fig. 23 shows the traffic (bytes received rate) being monitored on specific ports of the topology and the exact moment when the elasticity operation is triggered, marked as a dashed red line. The aggregated traffic (dark-blue line on top) comes from the flows of App 1 (green line) and App 2 (orange line) to port 2 of S1. As a result of the policy violation on that port, the created queues enforce the pretended improvement in App 2 throughput to an average of 4.53 Mbps (instead of the ≈2.5 Mbps before the elasticity), limiting App 1 flows to the remaining bandwidth. It is noteworthy that the applications' flows share the available bandwidth (20 Mbps) inside the slice, with App 1 flows being even more "greedy" before the elasticity operation.

- **Discussion:** The table inside Fig. 23 highlights the intended behavior by showing the applications' average throughput and the number of retransmissions during the test. After the elasticity operation, the App 2 transfer rate increases from ≈2.5 Mbps to ≈5 Mbps, consequently decreasing App 1 transfer rate. In addition, we can notice a continuous transmission during the experiment. Some packets were retransmitted (*114* on App 1 and *8* on App 2) due to the new OpenFlow rules that prioritized the queue to App 2 traffic. However, even with these retransmissions, there was no service interruption, packet loss, or negative impact on the application performance. Finally, we conclude that the tenant was able to improve the App 2 QoS without setting any strict policy on App 1, which is expected to have its throughput decreased in this scenario.

6.3.2 Scenario II: Vertical elasticity topology scale up

- **Policy Action:** After detecting a policy violation, this operation creates the new NEs and re-routes the App 2 flow. The extra information needed for this scenario consists of the new topology configuration, with a newly created switch S4 connecting S1 and S3 (purple elements on Fig. 22(c)). S4's ceiling transmission is restricted to 20 Mbps.

- **Results:** Fig. 24 shows the bytes received rate on specific ports of the topology and a dashed red line indicates the moment the elasticity operation is triggered. As soon as this operation is completed, the newly created port 1 in S4 receives the re-routed traffic of App 2 (in pink), bounded at ≈20 Mbps as expected.

- **Discussion:** The table in Fig. 24 indicates how the App 2 throughput improved from ≈2.2 Mbps to ≈20 Mbps after the elasticity operation is completed. The following steps were performed: the creation of topology elements (switch, ports, and links), re-routing of App 2 traffic (without interruption), and the monitoring of new NEs immediately after the SEA instantiates them. The elasticity operation creates the OpenFlow rules to ensure that App 1 traffic goes through the S2 and App 2 traffic through the new S4.

6.3.3 Scenario III: WAN Horizontal elasticity scale out

- **Foreword:** horizontal elasticity is the capability to add or remove an entire slice part. Regardless of how it is triggered, if manually by the tenant or automatically by the SEA, the Policy Controller delegates the elasticity to the Slice Provider so that it executes the operation as stated in P3. In case of a slice part addition, the Slice Provider must invoke the SEA API to instantiate the adapters according to

pre-defined parameters of the new WSP, as part of P4. Then, SEA reclaims control and assumes the next steps of the CCL.

- **Policy Action:** the Slice Provider must invoke SEA as defined in P4, with the following extra information needed to instantiate the adapters: IP address, port, and credentials of the WIM and ME[5]. The tenant pre-defined attributes to the new WSP includes: the WIM controller (Ryu Controller), the ME (OpenFlow protocol v1.4), the metric to be monitored (rate of received bytes), the polling interval (1s), the new topology requirement in terms of which network elements (clone of WSP 1), and the bandwidth limit of network elements (40 Mbps).

- **Results:** Fig. 25 also illustrates the bytes received rates of specific ports in this scenario, related to WSP 1 and WSP 2. We configure the App 2 flow as a 30-seconds stream before the elasticity operation and after that, we created a new App 2 flow configured to use WSP 2. The figure shows that, right after the first flow of App 2 finishes (in orange), App 1 flows reach the max bandwidth capacity of port 1 in S2 (20 Mbps). It is also noted that after the elasticity operation trigger indicated by the dashed red line, the WSP 2 is instantiated and the App 2 flow is started throughout it, eventually reaching its max bandwidth capacity of ≈40 Mbps, shown in the red line. We experimented this way to emphasize that it is not SEA's responsibility to re-route any application flow after a horizontal elasticity happens. After P3 and P4, the tenant can interact with the MA to manage the new WSP infrastructure and must reconfigure the applications to use it or not. For the sake of simplicity, we emulate this behavior by executing App 2 twice in different instants, before and after the elasticity.

- **Discussion:** Firstly, the App 1 and App 2 flows compete for the available bandwidth at WSP 1. After the elasticity operation was completed, App 1 continues using only the WSP 1 infrastructure. On the other hand, App 2 was manually configured to use only the newly instantiated WSP 2 (shown in the red line). As a result, both applications could reach the max throughput of corresponding WSPs (20 Mbps for WSP 1 and 40 Mbps for WSP 2). The WSPs elements are exclusively dedicated and instantiated to each slice, meaning isolated traffic inside each slice. Therefore, the WSP 2 does not have any initial traffic on it. In addition, no retransmissions occurred in this scenario since the applications were reconfigured at the service level. This evaluation aimed to answer RQ 3 by highlighting SEA's ability to provide SlEaaS, with a focus on the CCL of network elements. Also, showing the CCL for both networking and computing resources. Therefore, we can conclude that RQ 4 was answered incrementally in each of the analyzes presented as it was able to provide SlEaaS in a generic way for different types of technologies and infrastructures.

[5] More details about this operation can be found in (Rocha et al., 2022).

7 Conclusion

In this thesis, we present SEA, a novel architecture to support slicing elasticity as a service. The SEA offers the new SlEaaS business model defined by us, aiming at abstracting the heterogeneity of slice resources and infrastructure providers. To the best of our knowledge, this is the first architecture design, implementation, and integration found in the literature. Furthermore, we focus on the elasticity operations of networking and computing elements, which still need to be well-explored in other projects and are crucial for the slicing concept.

The objective of the evaluation of this work was to answer the four RQs presented in Section 6. As expected from a Ph.D. thesis, the proposal was improved during that time. In the evaluations, we would like to show all the SEA components integrated and improving the service quality, which is the main motivator of the slice elasticity operations. Also, proving how generic the SEA can be in supporting multiple technologies (VIMs, WIMs, MEs, and databases) since it is a vital requirement of the slicing concept. This thesis steps forwards in this research area with the definition of slice elasticity operations, but more elasticity options should appear, and we believe that they can be easily integrated with the SEA because of the abstractions defined.

Participation in the NECOS project contributed a lot to the path this thesis took and was very important to my growth as a student and researcher, working with excellent researchers. The following two sections focus on: the challenges and lessons learned during this period and the future work and open questions.

7.1 Challenges and lessons learned

This work presents SEA architecture and its implementation, integration, and evaluation. If on one hand, standardization efforts from organizations like 3GPP, ETSI, and IETF are crucial for the working plan in terms of slicing; on the other hand, implementing such standards is of paramount importance for the validation and integration of solutions. In this context, the main challenges we faced in this work include:

- The definition of the elasticity operations in the context of cloud network slices, which enable the growth and shrinking of a huge heterogeneity of resources and a new kind of resource called SP;
- The proposal of SlEaaS, which is a new business model that any player can offer;

- The proposal of an architecture (SEA) to support the SlEaaS, for the first time seen in the literature. However, this architecture should have to be generic enough to support the multiple administrative and technological domains with the abstractions provided by the adapters.
- The capability to deal with different VIM and WIM technologies from each Slice Provider, which requires the implementation of various adapters, such as for the standard solutions OpenStack, Kubernetes, OpenFlow, and Multiprotocol Label Switching (MPLS);
- The integration with several available Slice Providers, which requires well-defined APIs to provide the CCL properly;
- The integration with autoscaling mechanisms or well-known mechanisms in some VIMs and WIMs, e.g., Kubernetes Vertical/Horizontal Pod Autoscaler, might be used by SEA to perform the slice elasticity.

As the main lesson learned from this work, we highlight the usage of adapters as an abstraction layer needed to support different technologies. The focus of this thesis was not developing adapters but providing a sufficient number of them to validate the SEA, showing that they can be easily implemented if following the APIs and components description presented in this thesis. Therefore, to prove the feasibility of the architecture, different adapters (MAs and TAs) are implemented for mainly the two most used types of resources, computing, and networking. However, more technologies can appear for other types of infrastructure, and we believe they can be incorporated into the SEA and get all the benefits of having the SEA CCL.

7.2 Future work and open questions

One of the most exciting things about research is that it can never end. A gap can always be explored, and new ones can arise during this process. Therefore, this work is no different, and we will discuss such gaps and questions that can still be answered from this thesis. Firstly, we emphasize a scientific aspect that could be amazing and fits very well with all the CCL described in this thesis. The incorporation of machine learning algorithms to improve the capacity of taking decisions in the components. Such capacity can be translated as improving the prediction of the policy violation, the adaptation of the policy based on historical information, the build of policies using natural language instead of a descriptive file, more efficient triggers of policies, and others. Those are only a few examples of how machine learning algorithms can be used to improve the SEA to give tenants better and more automated elasticity operations.

Another aspect not well explored in this thesis is the multiple policies and cascade management that should be performed to ensure that one policy will not impact another and that several elasticity operations occur simultaneously. About the technologies and scenarios, we would like to idealize a 5G scenario introducing NFVs and exploring more network technologies such as the p4 that is widely studied in the literature nowadays. The open questions rely on the following:

- Which new elasticity operations can be defined? Will the sea support those new elasticities transparently? More elasticity operations can rise in the literature and it should be transparent from the SEA perspective, only requiring the adapters and DB abstractions if needed.
- Which functionalities or enhancements should be added to SEA to improve security, scalability, manageability, and other essential aspects?
- What is the correlation of computing and networking metrics with the QoS? In this thesis, we did not explore complex policies that combine different types of slice resources (computing and networking). Performing a VE at the same time on SPs of different types is a challenge that should be explored and tested.
- There is an improvement in combining policies of computing and networking resources?

7.3 Dissemination Activities

This chapter describes all the dissemination activities done during the Ph.D.

7.3.1 Papers and demos

- **A. L. B. Rocha**, P. D. Maciel, F. L. Verdi. SEA: Towards Slicing Elasticity as a Service. Submitted at IEEE Communications Magazine in October of 2022. **Paper under review**.
- **A. L. B. Rocha**, C. H. Cesila, P. D. Maciel, S. L. Correa, J. Rubio-Loyola, C. E. Rothenberg and F. L. Verdi. CNS-AOM: Design, implementation, and integration of an architecture for orchestration and management of cloud-network slices. Journal of Network and Systems Management, Plenum Press, USA, v. 30, n. 2, 04 2022. ISSN 1064-7570. Available on: <https://doi.org/10.1007/s10922-022-09641-z>.
- G. V. Luz, **A. L. B. Rocha**, L. C. de Almeida; VERDI, F. L. Verdi. InFaRR: Um algoritmo para reroteamento rápido em planos de dados programáveis. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS

(SBRC), 40. , 2022, Fortaleza. Porto Alegre: Sociedade Brasileira de Computação, 2022 . p. 154-167. ISSN 2177-9384. DOI: https://doi.org/10.5753/sbrc.2022.221980.

- **A. L. B. Rocha**, P. D. Maciel, F. L. Verdi et al., Design and Implementation of an Elastic Monitoring Architecture for Cloud Network Slices, NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-7, doi: 10.1109/NOMS47738.2020.9110415.

- **A. L. B. Rocha**, C. H. Cesila, P. D. Maciel, C. Rothenberg and F. L. Verdi, Elastic Monitoring Architecture for Cloud Network Slices, NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-2, doi: 10.1109/NOMS47738.
  2020.9110447.

- **A. L. B. Rocha**, M. Nadaleti, V. Furukawa, P. D. Maciel, F. L. Verdi. Uma Proposta de Arquitetura para o Monitoramento Multidomínio de Cloud Network Slices. In: Workshop de teoria, tecnologias e aplicações de slicing para infraestruturas softwarizadas (WSlice), First Edition, 2019, Gramado. Porto Alegre: Sociedade Brasileira de Computação, 2019 . p. 42-55. DOI: https://doi.org/10.5753/wslice.2019.7721.

- **A. L. B. Rocha**, F. Tusa, F. L. Verdi. Elastic monitoring for elastic cloud network slices. Poster and Demo video. ONF Connect 19, Santa Clara, CA, 2019.

- F. N. N. Farias, B. A. Pinheiro, A. J. G. Abelém, P. D. Maciel, **A. L. B. Rocha**, A. V. Neto, S. L. Correa, M. Nascimento, R. Pasquini, F. L. Verdi, C. E. Rothenberg. Projeto NECOS: Rumo ao Fatiamento Leve de Recursos em Infraestruturas de Nuvens Federadas. 10th Workshop de Pesquisa Experimental da Internet do Futuro (WPEIF), co-hosted at the XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) 2019, Gramado, Brazil, 6-10 May 2019.

7.3.2 Other activities

- I co-oriented three undergraduate students on scientific initiatives.

- I was a Ph.D. student of the NECOS Project for two years and a half. My responsibilities on the NECOS projects were designing, implementing, and researching the monitoring and management abstraction of slice resources. I participated in several meetings with great researchers and learned much during that time.

- I helped to propose and organize the first Workshop on Slicing Theory, Technologies, and applications (WSlice) co-located at the SBRC 2019.

References

3GPP. *3GPP SA2 - Study on Architecture for Next Generation System /Network slice related functionality (3GPP TR 23.799)*. 2015.

3GPP. *3GPP SA2 - Procedures for the 5G System: Procedures and flows of the architectural elements/Network slice related procedures (3GPP TS 23.502)*. 2016.

3GPP. *3GPP SA2 - System Architecture for the 5G System /Network slice related functionality (3GPP TS 23.501)*. 2016.

3GPP. *3GPP SA3 - Study on the security aspects of the next generation system/ Network slice related security (3GPP TR 33.899)*. 2016.

3GPP. *3GPP SA5 - Study on management and orchestration of network slicing/ Network slice management (3GPP TR 28.801)*. 2016.

3GPP. *3GPP SA5 - Management of network slicing in mobile networks - concepts, use cases and requirements (3GPP TS 28.530)*. 2017.

3GPP. *3GPP SA5 - Provisioning of network slicing for 5G networks and services: Detailed specification of network slice provisioning/ Network slice management (3GPP TS 28.531)* . 2017.

Alalewi, A.; Dayoub, I.; Cherkaoui, S. On 5g-v2x use cases and enabling technologies: A comprehensive survey. *IEEE Access*, v. 9, p. 107710–107737, 2021.

Almeida, L. C. de; Maciel, P. D.; Verdi, F. L. *Cloud Network Slicing: A systematic mapping study from scientific publications*. arXiv, 2020. Available on: <https://arxiv.org/abs/2004.13675>.

Baba, H. et al. End-to-end 5g network slice resource management and orchestration architecture. In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 2022. p. 269–271.

Beltrami, A. et al. Design and implementation of an elastic monitoring architecture for cloud network slices. In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 2020. p. 1–7.

Bonnet, J. *D4.2 Service Platform First Operational Release and Documentation*. 2016. <https://bscw.5g-ppp.eu/pub/bscw.cgi/d137267/SONATA%20D4.2%20Service%20platform%20first%20operational%20release%20and%20documentation.pdf>. Accessed 10 October 2020.

Bosshart, P. et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014. ISSN 0146-4833. Available on: <https://doi.org/10.1145/2656877.2656890>.

Breitgand, D. et al. Dynamic slice scaling mechanisms for 5g multi-domain environments. In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021. p. 56–62.

Chiha, A.; Wee, M. Van der; Colle, D.; Verbrugge, S. Network slicing cost allocation model. *Journal of Network and Systems Management*, v. 28, n. 3, p. 627–659, Jul 2020. ISSN 1573-7705. Available on: <https://doi.org/10.1007/s10922-020-09522-3>.

Clayman, S. et al. The necos approach to end-to-end cloud-network slicing as a service. *IEEE Communications Magazine*, v. 59, n. 3, p. 91–97, 2021.

Clayman, S. et al. The necos approach to end-to-end cloud-network slicing as a service. *IEEE Communications Magazine*, v. 59, n. 3, p. 91–97, 2021.

Contreras, L. D2.2: Consolidated definition of use cases, business models and requirements analysis. NECOS, 2018. Available on: <http://www.maps.upc.edu/public/d2.2_final_v2.0.pdf>.

CPqD. *Dojot documentation*. 2020. <https://dojotdocs.readthedocs.io/en/latest/>. Accessed 10 November 2020.

de Carvalho, M. B.; Esteves, R. P.; da Cunha Rodrigues, G.; Granville, L. Z.; Tarouco, L. M. R. A cloud monitoring framework for self-configured monitoring slices based on multiple tools. In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. 2013. p. 180–184. ISSN 2165-9605.

ETSI. *Network Functions Virtualisation - White Paper on NFV priorities for 5G*. 2017. <https://portal.etsi.org/NFV/NFV_White_Paper_5G.pdf>. Accessed 12 May 2021.

ETSI. *Network Functions Virtualisation (NFV) Release 3; Evolution and Ecosystem; Report on Network Slicing Support with ETSI NFV Architecture Framework. ETSI GR NFV-EVE 012 V3.1.1*. 2017. <https://www.etsi.org/deliver/etsi_gr/NFV-EVE/001_099/012/03.01.01_60/gr_NFV-EVE012v030101p.pdf>. Accessed 12 May 2021.

ETSI. *Next Generation Protocols (NGP); E2E Network Slicing Reference Framework and Information Model. ETSI GR NGP 011 V1.1.1*. 2018. <https://www.etsi.org/deliver/etsi_gr/NGP/001_099/011/01.01.01_60/gr_ngp011v010101p.pdf>. Accessed 12 May 2021.

Farrel, A. et al. *Framework for IETF Network Slices*. 2021. Work in Progress. Available on: <https://datatracker.ietf.org/doc/html/draft-ietf-teas-ietf-network-slices-04>.

Fatema, K.; Emeakaroha, V. C.; Healy, P. D.; Morrison, J. P.; Lynn, T. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, v. 74, n. 10, p. 2918 – 2933, 2014. ISSN 0743-7315. Available on: <http://www.sciencedirect.com/science/article/pii/S0743731514001099>.

Foundation, O. N. *TR-526 Applying SDN Architecture to 5G Slicing*. 2016. <https://opennetworking.org/wp-content/uploads/2014/10/Applying_SDN_Architecture_to_5G_Slicing_TR-526.pdf>. Accessed 12 May 2021.

Foy, X. de; Rahman, A. Internet-Draft, *Network Slicing - 3GPP Use Case*. 2017. <http://www.ietf.org/internet-drafts/draft-defoy-netslices-3gpp-network-slicing-02.txt>. Accessed 23 November 2020.

Galis, A. *Network Slicing - Revised Problem Statement draft-galis-netslices-revised-problem-statement-01*. 2017. <https://datatracker.ietf.org/doc/html/draft-galis-netslices-revised-problem-statement-01>. Accessed 14 May 2021.

Galis, A. *Perspectives on Network Slicing - Towards the New 'Bread and Butter' of Networking and Servicing*. 2018. <https://sdn.ieee.org/newsletter/january-2018/perspectives-on-network-slicing-towards-the-new-bread-and-butter-of-networking-and-servicing>. Accessed 23 November 2020.

Galis, A. et al. Management and service-aware networking architectures (mana) for future internet — position paper: System functions, capabilities and requirements. In: *2009 Fourth International Conference on Communications and Networking in China*. 2009. p. 1–13.

Galis, A.; Dong, K. M. J.; Bryant, S.; Boucadair, M.; Martinez-Julia, P. *Network Slicing - Introductory Document and Revised Problem Statement draft-gdmb-netslices-intro-and-ps-02*. 2017. <https://datatracker.ietf.org/doc/html/draft-gdmb-netslices-intro-and-ps-02>. Accessed 10 May 2021.

Galis, A.; Makhijani, K.; Yu, D.; Liu, B. *Autonomic Slice Networking draft-galis-anima-autonomic-slice-networking-05*. 2018. <https://datatracker.ietf.org/doc/html/draft-galis-anima-autonomic-slice-networking-05>. Accessed 10 May 2021.

Galis, A.; Tusa, F.; Clayman, S.; Rothenberg, C.; Serrat, J. Slicing 5G Networks: An Architectural Survey. In: *Wiley 5G Ref: The Essential 5G Reference Online*. John Wiley & Sons, Inc, 2020. p. 1–41. ISBN 9781119471509. Available on: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119471509.w5GRef095>.

Gavras, A. et al. 5g ppp architecture working group - view on 5g architecture, version 4.0. Zenodo, 2021. Available on: <https://zenodo.org/record/5155657>.

Geng, L. et al. *Network Slicing Architecture draft-geng-netslices-architecture-02*. 2018. <https://datatracker.ietf.org/doc/html/draft-geng-netslices-architecture-02>. Accessed 14 May 2021.

Geng, L. et al. *COMS Architecture draft-geng-coms-architecture-02*. 2018. <https://datatracker.ietf.org/doc/html/draft-geng-coms-architecture-02>. Accessed 14 May 2021.

Group, I.-T. F. *IMT-2020 Deliverables*. 2017. <https://www.itu.int/en/publications/Documents/tsb/2017-IMT2020-deliverables/mobile/index.html#p=1>. Accessed 12 May 2021.

Gutierrez-Estevez, D. M. et al. The path towards resource elasticity for 5g network architecture. In: *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. 2018. p. 214–219.

Hedman, P.; Team, N. P. W. E. A. *Description of Network Slicing Concept 160113*. 2016. <https://www.ngmn.org/wp-content/uploads/160113_NGMN_Network_Slicing_v1_0.pdf>. Accessed 12 May 2021.

Homma, S. et al. *Network Slice Provision Models draft-homma-slice-provision-models-02*. 2019. <https://datatracker.ietf.org/doc/html/draft-homma-slice-provision-models-02>. Accessed 14 May 2021.

ITU-T. *Y.3001 : Future networks: Objectives and design goals*. 2012. <https://www.itu.int/rec/T-REC-Y.3001-201105-I>. Accessed 12 May 2021.

Jennings, B.; Stadler, R. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, v. 23, 03 2014.

Kuklinski, S.; Tomaszewski, L. Dasmo: A scalable approach to network slices management and orchestration. *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, p. 1–6, 2018.

Kukliński, S.; Tomaszewski, L.; Kołakowski, R.; Chemouil, P. 6g-lego: A framework for 6g network slices. *Journal of Communications and Networks*, v. 23, n. 6, p. 442–453, 2021.

Luong, D.-H.; Outtagarts, A.; Ghamri-Doudane, Y. Multi-level resource scheduling for network slicing toward 5g. In: *2019 10th International Conference on Networks of the Future (NoF)*. 2019. p. 25–31.

Makhijani, K. et al. *Network Slicing Use Cases: Network Customization and Differentiated Services draft-netslices-usecases-02*. 2017. <https://datatracker.ietf.org/doc/html/draft-netslices-usecases-02>. Accessed 14 May 2021.

Mamatas, L.; Clayman, S.; Galis, A. A service-aware virtualized software-defined infrastructure. *IEEE Communications Magazine*, v. 53, n. 4, p. 166–174, 2015.

Manvi, S. S.; Shyam, G. K. Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey. *Journal of Network and Computer Applications*, Elsevier, v. 41, p. 424–440, 2014.

Marsico, A. Osm in action. OSM ETSI White Paper, 2022. Available on: <https://osm.etsi.org/images/OSM_EUAG_White_Paper_OSM_in_Action.pdf>.

McKeown, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar 2008. ISSN 0146-4833. Available on: <https://doi.org/10.1145/1355734.1355746>.

Montero, R.; Agraz, F.; Pagès, A.; Spadaro, S. Enabling multi-segment 5g service provisioning and maintenance through network slicing. *Journal of Network and Systems Management*, v. 28, n. 2, p. 340–366, Apr 2020. ISSN 1573-7705. Available on: <https://doi.org/10.1007/s10922-019-09509-9>.

Morín, D. G.; Pérez, P.; Armada, A. G. Toward the distributed implementation of immersive augmented reality architectures on 5g networks. *IEEE Communications Magazine*, v. 60, n. 2, p. 46–52, 2022.

NECOS. *D3.1: NECOS System Architecture and Platform Specification. V1 Deliverable*. 2019. <http://www.maps.upc.edu/public/NECOS%20D3.1%20final.pdf>. Accessed 10 October 2020.

NGMN. *NGMN 5G White Paper v1.0*. 2015. <https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0.pdf>. Accessed 12 May 2021.

Popovski, P.; Trillingsgaard, K. F.; Simeone, O.; Durisi, G. 5g wireless network slicing for embb, urllc, and mmtc: A communication-theoretic view. *IEEE Access*, v. 6, p. 55765–55779, 2018.

Qiang, L. et al. *Technology Independent Information Model for Network Slicing draft-qiang-coms-netslicing-information-model-02*. 2018. <https://datatracker.ietf.org/doc/html/draft-qiang-coms-netslicing-information-model-02>. Accessed 15 May 2021.

Qiang, L.; Geng, L.; Makhijani, K.; Foy, X. de; Galis, A. *The Use Cases of Common Operation and Management of Network Slicing draft-qiang-coms-use-cases-00*. 2018. <https://datatracker.ietf.org/doc/html/draft-qiang-coms-use-cases-00>. Accessed 14 May 2021.

Rocha, A. L. B. et al. Cns-aom: Design, implementation and integration of an architecture for orchestration and management of cloud-network slices. *J. Netw. Syst. Manage.*, Plenum Press, USA, v. 30, n. 2, 04 2022. ISSN 1064-7570. Available on: <https://doi.org/10.1007/s10922-022-09641-z>.

Sanabria-Russo, L.; Verikoukis, C. A cloud-native monitoring system enabling scalable and distributed management of 5g network slices. In: *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*. 2021. p. 42–46.

Sattar, D.; Matrawy, A. Proactive and dynamic slice allocation in sliced 5g core networks. In: *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. 2020. p. 1–8.

Silva, F. S. D. et al. Necos project: Towards lightweight slicing of cloud federated infrastructures. In: *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. 2018. p. 406–414.

Sukhija, N.; Bautista, E. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In: *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 2019. p. 257–262.

Thalanany, S. *5G end-to-end architecture framework v3.0.8*. 2017. 04–0ct p. <https://www.ngmn.org/wp-content/uploads/Publications/2019/190916-NGMN_E2EArchFramework_v3.0.8.pdf>. Accessed 22 November 2020.

Turkovic, B.; Nijhuis, S.; Kuipers, F. Elastic slicing in programmable networks. In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021. p. 115–123. ISSN 2693-9789.

Tusa, F.; Clayman, S.; Galis, A. Real-time management and control of monitoring elements in dynamic cloud network systems. In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. 2018. p. 1–7.

Tusa, F.; Clayman, S.; Galis, A. Dynamic Monitoring of Data Center Slices. In: IEEE. *5th IEEE International Conference on Network Softwarization (NetSoft 2019)*. 2019. p. 1–7.

Tusa, F.; Clayman, S.; Valocchi, D.; Galis, A. Multi-domain orchestration for the deployment and management of services on a slice enabled nfvi. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018. p. 1–5. ISSN null.

Valera-Muros, B.; Panizo, L.; Rios, A.; Merino-Gomez, P. An architecture for creating slices to experiment on wireless networks. *Journal of Network and Systems Management*, v. 29, n. 1, p. 1, Oct 2020. ISSN 1573-7705. Available on: <https://doi.org/10.1007/s10922-020-09571-8>.

Zarrar Yousaf et al. *GR NFV-IFA 022 - V3.1.1 - Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Report on Management and Connectivity for Multi-Site Services*. 2018. <https://www.etsi.org/deliver/etsi_gr/NFV-IFA/001_099/022/03.01.01_60/gr_NFV-IFA022v030101p.pdf>. Accessed 23 November 2020.