Orquestração de Cloud-Network Slices Orientada à Predição de Métricas de Serviço a Partir do Monitoramento da Infraestrutura

Aryadne Guardieiro Pereira Rezende

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

Aryadne Guardieiro Pereira Rezende

Orquestração de Cloud-Network Slices Orientada à Predição de Métricas de Serviço a Partir do Monitoramento da Infraestrutura

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Rafael Pasquini
Coorientadora: Raquel Fialho de Queiroz Lafetá

Uberlândia
2020

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

R467o
2020

Rezende, Aryadne Guardieiro Pereira, 1993-
Orquestração de Cloud-Network Slices orientada à predição de métricas de serviço a partir do monitoramento da infraestrutura [recurso eletrônico] / Aryadne Guardieiro Pereira Rezende. - 2020.

Orientador: Rafael Pasquini.
Coorientadora: Raquel Fialho de Queiroz Lafetá.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2020.3053
Inclui bibliografia.
Inclui ilustrações.

1. Computação.I. Pasquini, Rafael, 1981-, (Orient.). II. Lafetá, Raquel Fialho de Queiroz, 1983-, (Coorient.). III. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Rejâne Maria da Silva – CRB6/1925

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Mestrado Acadêmico, 26/2020, PPGCO				
Data:	27 de agosto de 2020	Hora de início:	**12h00min**	Hora de encerramento:	**14h30min**
Matrícula do Discente:	11812CCP008				
Nome do Discente	Aryadne Guardieiro Pereira Rezende				
Título do Trabalho:	Orquestração de Cloud-Network Slices Orientada à Predição de Métricas de Serviço a Partir do Monitoramento da Infraestrutura				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Rodrigo Sanches Miani - FACOM/UFU; Paulo Rodolfo da Silva Leite Coelho - FACOM/UFU; Erika Susana Rosas Olivos - Universidad Técnica Frederico Santa María - Santiago/Chile; Raquel Fialho de Queiroz Lafetá - XP Inc (coorientadora) e Rafael Pasquini - FACOM/UFU, orientador da candidata.

Os examinadores participaram desde as seguintes localidades: Erika Susana Rosas Olivos - Santiago/Chile; Paulo Rodolfo da Silva Leite Coelho, Rodrigo Sanches Miani, Raquel Fialho de Queiroz Lafetá e Rafael Pasquini - Uberlândia-MG. A discente participou da cidade de Uberlândia-MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Rafael Pasquini, apresentou a Comissão Examinadora e a candidata, agradeceu a presença do público, e concedeu à Discente a palavra para a exposição do seu trabalho. A duração da apresentação da Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir a candidata. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando a candidata:

Aprovada.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

Ressalta-se que a examinadora Erika Susana Rosas Olivos é estrangeira e não possui documento brasileiro, portanto não deverá assinar este documento.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Rodrigo Sanches Miani**, **Professor(a) do Magistério Superior**, em 01/09/2020, às 10:53, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539,

Dedico esse trabalho a todas as mulheres que lutaram por todos direitos que temos hoje, principalmente pelo o libertador direito de estudar.

Agradecimentos

Agradeço aos meus pais, Adriana Guardieiro Pereira Rezende e Diom Gande Rezende, que nunca mediram esforços para que eu conseguisse avançar nos meus estudos. Obrigada Adriana por não me deixar duvidar do seu amor não importasse qual seriam os resultados das minhas empreitadas, por ter me incentivado desde sempre a ser o melhor que eu podia e por sempre me fazer peixe em momentos importantes para que eu ficasse mais inteligente. Obrigada Diom por sempre poder contar com você a qualquer momento e por ser meu exemplo de disciplina e dedicação. O amor e a dedicação de vocês é a fundação de cada conquista que tive e terei. Agradeço ao meu irmão, Diom Gande Rezende Filho, por ser meu confidente sincero e sempre me mostrar o quanto eu podia melhorar, obrigada pelos lanchinhos que me levava na universidade e pelos cafés, ideias e planos que compartilha comigo.

Ao meu orientador, Rafael Pasquini, tenho muita gratidão pela paciência, disponibilidade, gentileza e empenho em sempre abrir caminhos e me guiar em meu desenvolvimento acadêmico durante o mestrado. Agradeço profundamente à minha coorientadora, Raquel Fialho de Queiroz Lafetá, por me ajudar a estruturar e apresentar meu raciocínio de maneira mais clara e lógica e por ser um fundamental ponto de apoio emocional durante todo esse período. Agradeço também ao meu parceiro de mestrado Gustavo Silveira por me mostrar que quando a execução acompanha o planejamento podemos ter grandes resultados.

Agradeço muito a todas professoras que tive desde os primeiros anos da minha educação até hoje. Vocês foram meus maiores exemplos para que eu sempre continuasse me desenvolvendo em meus estudos. Sou grata também ao professor Rodrigo Miani, por ser a alegria dos alunos nas manhãs de sexta-feira nas aulas de Segurança da Informação. Com certeza seus conselhos, leveza, motivação e ensinamentos ficarão registrados.

Este projeto de mestrado foi financiado com recursos da 4º chamada colaborativa BR-EU no contexto do H2020, registrados no acordo 777067 (*NECOS - Novel Enablers for Cloud Slicing*), que é fomentado pelo Ministério da Ciência e Tecnologia no lado Brasileiro e pela Comissão Europeia de Tecnologia no lado Europeu.

"If I have seen further, it is by standing upon the shoulders of giants."
(Sir Isaac Newton)

Resumo

Este trabalho, inserido no contexto do projeto *Novel Enablers for Cloud Slices* (NECOS), visava a proposta de um orquestrador de recursos de nuvem de provedores federados. Dada a natureza do projeto NECOS, o orquestrador deveria gerenciar parcelas de recursos dessa nuvem federada, chamadas de fatias. Graças a diversidade dos recursos que poderiam compor essa fatia e visando manter o *Service Level Agreement* dos clientes da plataforma para com seus usuários finais, foi desenvolvida uma estratégia automaticamente customizável de orquestração, baseada em aprendizado de máquina. Redes Neurais Recorrentes foram usadas para prever valores futuros de uma métrica indicadora de performance. Com base nessa predição, o orquestrador deveria disparar ações de redimensionamento dos recursos da fatia, tanto no sentido de aumentar a capacidade para acomodar altas cargas, quanto reduzir essa capacidade a fim de economizar recursos em caso de baixa demanda. A criação do protótipo do orquestrador aliada aos experimentos executados mostram que é possível, viável e adequado o uso da estratégia proposta.

Palavras-chave: Orquestração. Nuvem. Fatias de nuvem. Predição. Redes Neurais Recorrentes. Aprendizado de Máquina.

Cloud-Network Slices Orchestration Driven by Service-level Metrics Prediction from Infrastructure Monitoring

Aryadne Guardieiro Pereira Rezende

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

Abstract

This work was developed in the context of the Novel Enablers for Cloud Slices (NECOS) project. This dissertation aimed to propose an orchestrator of cloud-network resources from federated providers. Given the nature of the NECOS project, the orchestrator should manage portions of the resources inside this federation, which were called slices. Due to the diversity of resources that could make up each slice and aiming to maintain the Service Level Agreement of the platform's clients towards their end-users, an automatically customizable orchestration strategy was developed, based on machine learning. Recurrent Neural Networks were used to predict future values of a key performance indicator, previously chosen by the client. Based on this prediction, the orchestrator triggered slice resizing actions, both in the sense of increasing the capacity of the slice to accommodate high loads, as well as reducing this capacity to save resources in case of low demand. The creation of the orchestrator prototype allied with the experiments carried out showed that it is possible, viable, and appropriate to use the proposed strategy.

Keywords: Orchestration. Cloud. Cloud Slice. Forecasting. Machine Learning.

List of Figures

List of Tables

Acronyms list

API Application Programming Interface

CSV Comma Separated Values

IoT Internet of Things

IMA Infrastructure and Monitoring Abstraction

KPI Key Performance Indicator

LSDC Lightweight Software Defined Cloud

MAPE Monitoring, Analysis, Planning, and Execution

ML Machine Learning

NECOS Novel Enablers for Cloud Slicing

QoS Quality of Service

RNN Recurrent Neural Network

SLA Service Level Agreement

SLO Service Level Objective

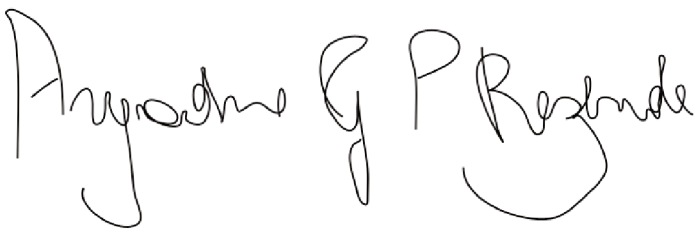
SRO Slice Resource Orchestrator

VM Virtual Machine

List of Algorithms

Contents

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my dissertation, and that their permissions allow availability such as being deposited in public digital libraries.



Aryadne Guardieiro Pereira Rezende

Chapter 1

Introduction

Cloud Computing is the paradigm that has emerged of the computational resource offer in a-pay-as-you-go manner, in which the service payment is calculated based on its usage during some period. One attribute of this paradigm is elasticity, which allows users to acquire or release a certain amount of computational resources, according to their needs (NETTO et al., 2014).

Cloud Computing offers services in three levels: infrastructure, (*IaaS - Infrastructure as a Service*), platform (*PaaS - Platform as a Service*) and software (*SaaS - Software as a Service*), wherein each layer there are thousands of resource providers. Alongside these layers is the *Cloud Slice* concept. A slice can be defined as a set of virtual or physical resources (network, processing, and storage) that can accommodate service components, in an integrated fashion, independent of other slices (FREITAS et al., 2018). A large number of possible resource combinations and the high complexity of integration make it hard for an organization to deploy its services using different computing, network, and storage providers. The choice of resources to run a service among many resource and providers options and then to integrate them makes new services launching time-consuming and expensive (SILVA et al., 2018).

To address this issue, it was created the Novel Enablers for Cloud Slicing (NECOS) project (NECOS, 2018b), which was the background of this dissertation. The main project's goal was to investigate a new business model called *Slice as a Service*. Inside the project, it was tested the on-demand slice creation, in a multi-domain environment, and its automatic reconfiguration. To reach this goal, three main requirements were gathered:

1. To make application, network, and service innovation and integration easier and quicker;

2. To develop management systems more integrated;

3. To improve energy and cost efficiency and interoperability among different domains.

The first requirement regards the difficulty of dealing with the high coupling of data centers, cloud, and network systems. Static connections among data centers are migrating to software-defined connections, where storage, processing, and communication are being virtualized. At the same time, the second requirement refers to the data centers management integration and it aims to facilitate the products and services combination, once that, currently, this is done individually in each provider. Lastly, the third requirement comes from the need of saving costs using the most appropriate configuration, combined with automation of currently manual tasks, usually expensive and imprecise.

Following the workflow defined in NECOS, first, the tenant (most likely a service provider) gives a description of the slice's configuration that is needed. After this resource set has been allocated, from one or more resource providers, the tenant can deploy its service(s) and provide them to its end users. Due to the dynamic nature of the majority of cloud-hosted applications, the tenant's demand for resources can increase or decrease depending on its service usage.

If the resource need is increasing, more resources should be allocated, which can be a time expensive task (ZANELA et al., 2019). If the resource needs to be reduced, to save money and electricity, for example, it is interesting to release the previously allocated resources. For this reason, can be useful to anticipate resource usage to allocate or release resources accordingly. In this context, this dissertation works upon the third requirement above, building a slice orchestrator that forecasts if a slice configuration should be adjusted, acting autonomously, during the slice run-time. The strategy used to predict when this change should happen is to forecast the Service Level Agreement (SLA) being delivered to the end-users of the tenant's service.

It is possible to use a simple orchestration, guided by the monitoring of a metric, where whenever this metric reaches a threshold it causes an elasticity operation. One weakness of this method is that cloud computing workload may change rapidly over time and a scalable application requires time to perform the elasticity request (VAZQUEZ; KRISHNAN; JOHN, 2015). For this reason, to use a proactive approach to foresee future demands avoids resource wasting during non-peak hours and decreases the risk of bad quality or denying of service for online users, thus improving their product experience.

To accomplish this goal, this dissertation implements a computer system where a slice is profiled and then a model of its behavior is generated, using a Recurrent Neural Network (RNN). This model is capable of forecasting a tenant's service Key Performance Indicator (KPI), which tells whether the service is being delivered as the tenant requires. This forecast is done t seconds beforehand, where t is a predefined parameter. Once the scaling of a slice comes with an overhead, it is necessary a t large enough to preemptively scale to the correct configuration, alleviating this way some or all of the elasticity overhead.

1.1 Motivation

In October of 2001, IBM issued a manifesto attesting that the main challenge of Information Technology would be an imminent software complexity crisis. To support this claim, the company used applications that weighed thousands of lines of code and were dependent on highly skilled professionals to be installed, configured, tweaked, and maintained. These reasons could turn the dream of pervasive computing in all areas a nightmare (KEPHART; CHESS, 2003).

Reinforcing this concern, it has been estimated by Gartner (VELOSA et al., 2014) that the number of connected devices in 2020 will exceed 25 billion. Managing resources and information produced by so many pieces of equipment will be hard problems to handle manually. Thus, one possible way out would be autonomous computing. This name is inspired by the human autonomic nervous system, and carries the idea of automatic adaptation in response to stimuli (KEPHART; CHESS, 2003). In this sense, the use of a standalone slice orchestrator emerges as a solution to be analyzed.

Since NECOS will act at the slice level, all levels of the cloud stack (infrastructure, platform, or software) can be touched. According to (RANJAN et al., 2015), the orchestrator must take action that is holistic taking into account all cloud service layers, benefiting the application as a whole. In addition to the concern with all levels of a particular slice, when multiple providers are considered, this problem gets another dimension that can be investigated, which aims to combine resources from multiple providers at any level of the cloud stack.

In Qu, Calheiros e Buyya (2018), there is a survey on how the automatic scaling of cloud resources in the state of the art is handled. In this article, the authors point out that automatic resource scaling can be seen as an automatic control problem, and therefore we can use the Monitoring, Analysis, Planning, and Execution (MAPE) cycle to solve it. Figure 1 shows this control loop. Each step and its challenges are defined as follows:

Monitoring: In this step, metrics are collected and can be transformed into performance indicators. Then, they will be used to determine which scaling strategies will be used and how they should be executed. In this context, there are two main challenges. The first is **selecting which are the most significant performance indicators**, which increase the accuracy of resource estimation and decrease information traffic costs on the network. The second is to **define a monitoring interval** of these indicators, which also impacts the amount of information on the network and also influences how sensitive the changes will be to the orchestrator.

Analysis: At this stage the orchestrator determines, through the collected data, whether it is necessary to make any changes to the resources already allocated or not. It is needed to define the **scaling time**, which can be proactive or reactive to changes. If the method is proactive, a **load estimation strategy** shall be chosen. It is also

needed to evaluate if the system will support **change adaptation**, or will consist of fixed rules. Another relevant point to be evaluated is to mitigate the oscillation in the distribution of resources because increasing or decreasing their allocation in a short time causes waste and loss of SLA.

Planning: Here, the resources that will be provided or released in the next resize action are estimated. It is important to select the right amount to reduce financial costs. Difficulties at this stage include **estimating which and what amount of resources** will be needed for the current or future situation of the service, and also **how to combine them**. The challenges of this step make it an optimization problem with a large space of possibilities, being an NP-hard problem to generate a perfect provisioning plan.

Execution: At the end of the MAPE cycle, execution will be the realization of the provision plan produced, which is done by performing resizing actions through each service provider's Application Programming Interface (API). When it comes to applications that can be deployed in different data centers, or if actions at different tiers need to be performed, different APIs need to be dealt with, which increases the degree of difficulty in building a generic orchestrator.

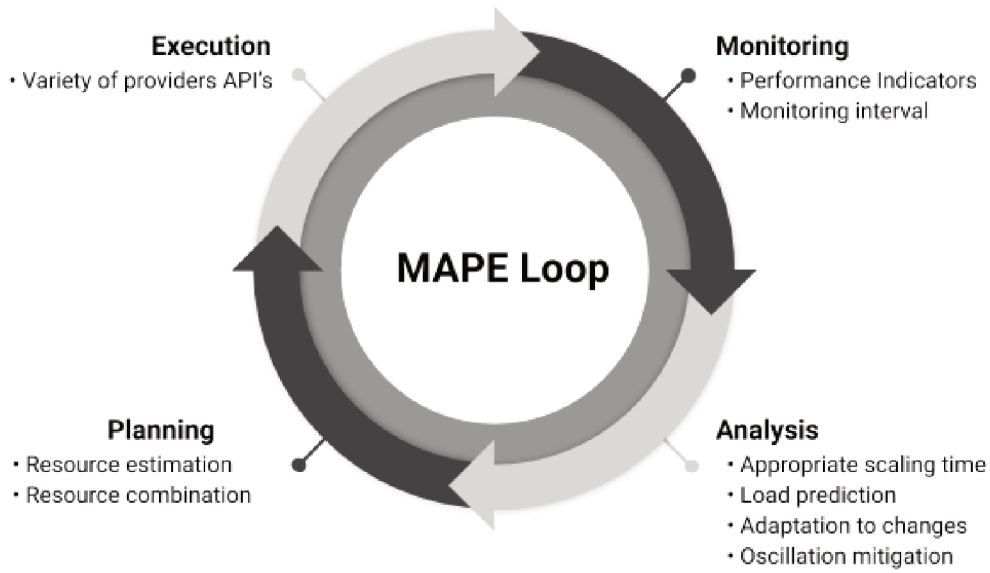


Figure 1 – MAPE loop and challenges of each phase.

Due to the variety and complexity of these challenges, specific objectives were selected to be investigated in this work. They are portrayed in the next section.

1.2 Research Goals and Challenges

As pictured in Figure 1, there are several research challenges in cloud orchestration in each phase of the MAPE cycle. This work selected as subject the "appropriate scaling time" issue, from the Analysis step. Besides, there are some additional challenges inherited from the NECOS project context:

1. NECOS has a multi-tenant architecture. Two restrictions come up from this feature:
   a) The orchestration of each slice must happen isolated.
   b) Different kinds of services can run on top of the slice. Thus, to perform a good orchestration, the orchestration needs to adapt its resource management strategy to each slice.
2. NECOS aims to reduce the deployment time of the service on a slice. For this reason, the orchestration configuration should be as automatic as possible, avoiding the need to have a specialist to configure the orchestration rules.
3. Since scaling the resources inside the slice takes time, a truly effective approach should do it before it is needed, avoiding the prejudicial effects of not having the right configuration when necessary. However, it can not be too much beforehand, which would result in resource waste.
4. Finally, a tenant may not have an online probe running all the time collecting real end user's metrics, which is used as an orchestration metric. This means that the orchestrator can not rely on the end-user's side metrics to forecast the tenant's service KPI during the slice run-time.

Having in mind these conditions, the overall goal of this dissertation was to explore the slice resource orchestration problem in the NECOS context, proposing a slice orchestrator architecture to keep the SLA agreed between the tenants and their end-users while saving resources when possible. This macro objective was broken in the following way:

1. Propose an orchestration architecture with the support of multi-slice, where each slice is orchestrated in isolation.
2. Develop an automatized and customized orchestration strategy for each slice.
3. Investigate forecasting approaches that allow foreseeing the service quality been delivered to the end-users, respecting the NECOS formerly described restrictions, previously and precisely enough to perform the elasticity actions proactively.
4. Lastly, implement a prototype of this architecture to perform experiments to validate the selected approaches.

1.3 Hypotheses

Considering the whole scenario presented so far, the following hypotheses, and their motivating questions, were evaluated in this dissertation:

H1 - It is possible to forecast the selected tenant's service KPI using only the provider (slice's infrastructure) side metrics.

Q1.1 - Is it possible to use a forecasting method that uses only the providers' monitoring metrics to estimate the selected tenant's service KPI?

H2 - It is possible to propose an automatic and proactive slices orchestrator, which avoids breaches in the tenants' SLAs, while also saves resources.

Q2.1 - Does the orchestrator adapt properly (maintain SLA and keep the cost as low as possible) in situations where resources need to be added?

Q2.2 - Does the orchestrator adapt properly (maintain SLA and keep the cost as low as possible) in situations where it is necessary to remove resources?

Q2.3 - Can the configuration changes be done previously enough to avoid SLA breaches?

The results found about these assumptions are shown in Chapter 4. The next section shows the dissertation's scientific outcomes so far.

1.4 Contributions

Firstly, this work contributed to the core of the NECOS project, providing an architecture for a smart and automatic orchestrator, which was demonstrated in the final review of the project at UNICAMP (NECOS, 2019b). The orchestrator prototype, its documentation, along with a real monitoring information dataset, are available as open-source at the NECOS-associated repository[1].

[1] <https://gitlab.com/necos/demos/mlo>

1.5 Dissertation Organization

Chapter 2 aims to give context about the project where this dissertation was developed, it also introduces the state of art and concepts about resource orchestration in cloud environments and gives notions about time series forecasting, as it was one tool used here to reach the described goals. Then, Chapter 3 explains the solution found to the target research challenges. It shows, in a top-down manner, how the orchestration's workflows were embodied in an orchestration architecture, and how from such an orchestration architecture was derived a management cycle, using a high-level codding language. Chapter 4 shows how the proposed solution was evaluated, and the experiment results that were obtained from it. Finally, it is given a conclusion which ties the objectives, hypothesis, and experiment results.

Chapter 2

Background

The central topics for this dissertation were: the NECOS project, resource orchestration, and time series forecasting methods. Section 2.1 presents the fundamental concepts related to NECOS and an architectural overview. Section 2.2 shows different approaches used currently in resource orchestration, and also the concepts related to it. At last, Section 2.3 focus on time series forecasting and the method used here to perform this task.

2.1 The NECOS Project

The NECOS project vision is that "computation, storage and networking resources have to be considered as a whole to be allocated to service requests" (NECOS, 2018b). To do so, the project coined a new business model: The Slice as a Service. In this model, a grouping of resources from different resource providers is managed as a whole to accommodate service components on top of each slice, independently of other slices (NECOS, 2019a). The Slice as a Service approach provides an adaptable control plane, supporting features for creating, scaling up or down, and deleting slices, as well as adapting slices at run-time while considering service requirements and current cloud resource conditions.

There are three key roles inside NECOS: the Resource Provider, the Slice Provider, and the Slice Tenant. The first owns the physical resources and infrastructure (network/ cloud/ data center) and provides them. The second is typically a telecommunication service provider, which is the owner or tenant of the infrastructures from which cloud network slices can be created. And finally, a slice tenant is the user of a specific slice, in which its services are hosted.

Another important concept is the Cloud-Network Slice, which is a set of infrastructures (network, cloud, data center) components/network functions, infrastructure resources (i.e., connectivity, compute, and storage manageable resources), and service functions that have attributes specifically designed to meet the needs of an industry vertical or a service. The Cloud-Network Slice key concepts are:

- A cloud-network slice supports at least one type of service.
- A cloud-network slice may consist of cross-domain components from separate domains in the same or different administrations, or components applicable to the infrastructure.
- A collection of cloud-network slice parts from separate domains is combined, connected through network slices, and finally aggregated to form an end-to-end cloud-network slice.

With the aim to make this kind of business model possible, the functional architecture, shown in Figure 2, was designed (NECOS, 2019a). It contains three main high-level sub-systems: The NECOS Lightweight Software Defined Cloud (LSDC) Slice Provider (colored in blue), the Resource Marketplace (in yellow), and the Resource Providers (in green).

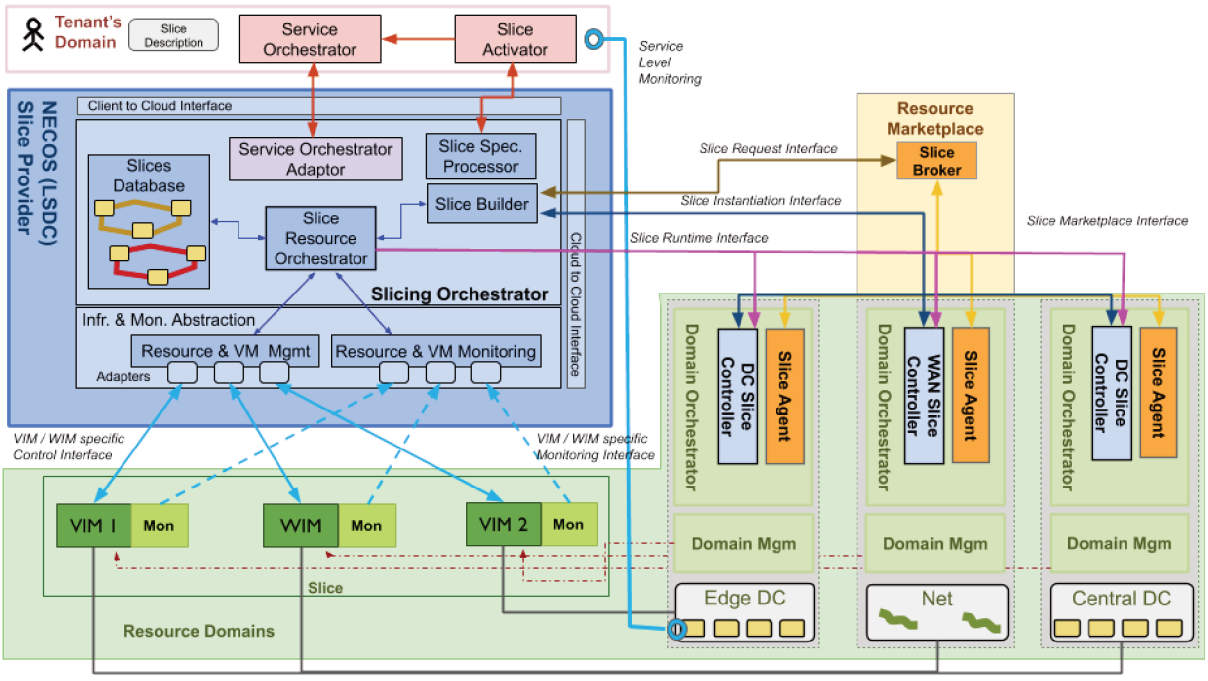


Figure 2 – NECOS functional architecture.

The NECOS (LSDC) Slice Provider is the sub-system that allows for the creation of full end-to-end Slices from a set of constituent Slice Parts. In NECOS, a slice looks the same as the full set of federated resources, with the main attribute being that the domains look a lot smaller, once they were partitioned in slices. For this reason, the slice is called "LSDC" - Lightweight Software Defined Cloud (NECOS, 2019a).

The NECOS (LSDC) Slice Provider presents a northbound API compatible with a tenant's Service Orchestrator, thus enabling tenants to operate on the full infrastructure, or to choose to interact with Slice as a Service providers, using NECOS. When requesting a slice from a NECOS provider, there is a Slice Builder component that goes out to a specially designed and configured Resource Marketplace that can find Slice parts across various participating Resource Domains, based on a Slice Specification.

Within the NECOS (LSDC) Slice Provider, the Slice Resource Orchestrator (SRO) module has the following attributions:

1. To combine the Slice Parts that make up a slice into a single aggregated slice.

2. To orchestrate the running end-to-end Slices, including the run-time management of their lifecycle.

3. To manage the service elements across the slice parts that make up the full end-to-end slice.

4. To place and embed VMs and virtual links for the services into the resource domains.

This dissertation explored the attribution 2. In this context, the aim was to perform the slice orchestration during run-time, guided by the service quality being delivered to the tenants' end-users, while investigating the problem of intelligent and automatic slice reconfiguration. The proposed orchestration solution is explained in details in Chapter 3, and its experimental outcomes are shown in Chapter 4.

The next relevant module for this dissertation in this architecture is the Infrastructure and Monitoring Abstraction (IMA). It is responsible for interacting with the actual remote cloud elements. Through it the Slice Provider can interact with various remote VIMs, WIMs, and monitoring sub-systems in a generic way, using plugin adaptors with the relevant API interactions. The IMA allows the SRO to interact with the remote clouds in order to provision the actual tenant services and to monitor the remote resources running those services (via additional monitoring data that is not available via the Service Level Monitoring Interface).

The SRO will take care of performing the slice life-cycle management, i.e., continuously checking whether the allocated slice is capable of fulfilling the requirements that were initially requested by the Tenant. With this goal in mind, the SRO must have access to a set of monitoring measurements (coming from each slice part) that provides information about respective slice resource utilization patterns (e.g., the number of available cores, memory, network delay/loss, etc.). This information set will be at the granularity of the slice and will not directly be linked to any of the service instances running on that slice. The monitoring information is expected to be propagated to the SRO via the underlying monitoring abstraction implemented by the IMA.

A key concept inside NECOS is elasticity. It is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources (computing, networking, and storage) in an autonomic manner NECOS (2018a). The goal is that at each point in time the available resources match the current demand as closely as possible. The types of elasticity and each workflow are showed in.

Vertical elasticity is defined as the ability to resize slice parts dynamically, as needed, to adapt the slicing part to demand changes. In other words, it means to resize the resources inside an already allocated slice part. For example, this expresses the ability to augment the bandwidth among hosts in a particular slice part of a data center when the demand for the services supported by the slice increases.

On the other hand, **horizontal elasticity** is the capability of creating or removing slice parts dynamically, using resources of the same or other(s) provider(s), depending on the workload evolution. For example, as the service workload moves from an area to another in the globe, maybe because of time zones, a slice part under a lighter workload can be removed, while another slice part can be created in the heavier load area, maybe being composed by resources from different providers. It is important to notice that vertical elasticity is preferred whenever possible because of its set up smaller cost, once the configuration and contracts are more simple than for horizontal elasticity, and using more infrastructure from the same provider usually facilitates to get better costs. The next section presents other definitions of resource orchestration and also presents how it is done in related works.

2.2 Resource Orchestration

The aim of this section is to define the key concepts around resource orchestration and also compare the most related current orchestrators with the proposed one. The resource orchestration problem can be observed from different angles, depending on the resources to be managed. The works cited in (QU; CALHEIROS; BUYYA, 2018), for example, deal with orchestrating virtual machines. In (CASALICCHIO; PERCIBALLI, 2017) and (CASALICCHIO, 2019), container-level orchestration is analyzed. (CARNEVALE et al., 2018) describes an osmotic orchestrator that manages devices in the Internet of Things (IoT) context. On the other hand, the work (SCIANCALEPORE; CIRILLO; COSTA-PEREZ, 2017) illustrates a slices orchestrator in a 5G background.

In (QU; CALHEIROS; BUYYA, 2018) a pertinent theme is introduced in cloud orchestration: **resource auto-scaling**, or automatic resource scaling. It takes advantage of the elasticity facilitated by virtualization and deals with the challenge of scaling and providing only the necessary amount of resources required for a given application at a given time automatically. It provides a compilation of the latest research on cloud computing, focusing on automatic scaling in a Virtual Machine (VM) level.

The approach used to analyze resource orchestration in (QU; CALHEIROS; BUYYA, 2018), is to treat the auto-scaling problem as a classic automatic control problem, the Monitoring, Analysis, Planning, and Execution (MAPE) loop described previously in Section 1.1, Figure 1. In this control loop, a controller should dynamically adjust the type and amount of resources allocated in order to comply with the SLA or satisfy a Service Level Objective (SLO).

In the context of a cloud service, a **SLA** is used to describe and set service level objectives (WEES et al., 2014). In general, a **SLO** is a service objective linked to metrics. A **metric** can be described as "a defined measurement method and measurement scale, which is used in relation to a quantitative service level objective" (WEES et al., 2014). In an SLO, metrics are applied to establish boundaries and margins of errors, related to the behavior of the cloud service and any limitations. Thus, metrics can be used at run-time for service monitoring, balancing, or remediation. A single **measurement** of a metric is the value that this metric has at a certain point in time.

According to Qu, Calheiros e Buyya (2018), to meet a SLO, four types of **elasticity** operations can be performed. Regarding enlarge resources, the **scaling up** operation increases the internal resources of a VM (number of CPUs, memory capacity, etc.), while the **scaling out** creates more VMs. When it is needed to shrink resources, the **scaling down** action can be applied to reduce the internal resources of a VM, or a **scaling in** operation can be performed to reduce the number of VMs assigned to the application. Operations of scaling out and scaling in are considered horizontal scaling, whereas scaling up or down is considered vertical scaling.

By raising the level of abstraction, instead of scaling VMs, it is possible to orchestrate containers that run inside these machines or directly into metal. A **container** can be defined as an isolated and portable environment in which one can install an application, add libraries, binaries, and even a basic configuration of how the application should be executed (CASALICCHIO; PERCIBALLI, 2017). Executing an **image**, which is a description of a container, is the same as creating an **instance** of it. One can create or delete copies of these instances depending on the needs of the application users. These copies are defined as **replicas**.

Instances of a container are managed locally by a **container manager** (CASALICCHIO, 2019), for example: Docker (MERKEL, 2014), Apache Mesos (MESOSPHERE, 2018a), and Amazon ECS (AMAZON WEB SERVICES, 2018). On the other hand, **container orchestrators** allow the selection, deployment, monitoring, and dynamic control of containers in a cloud (CASALICCHIO, 2019) environment. This type of orchestrator should be concerned with multi-node resource control, scaling instances within the cluster, load balancing, health checking, fault tolerance, and automatic scaling of active instances. In this way, container managers act at a local level, while orchestrators act at a *cluster* level, managing instances among multiple nodes on a network.

Examples of such orchestrators are: Kubernetes (KUBERNETES AUTHORS, 2018), Docker Swarm (DOCKER INC, 2018), and Mesosphere Marathon (MESOSPHERE, 2018b). In contrast with these orchestrators, the one proposed here deals with resources at a broader level, the slice level. Thus, it may operate over any layer involved in the slice. It means that an elasticity operation that triggers changes in a lower level of the slice can bubble to higher levels, and also be applied to containers. However, due to the complexity of the slice context and time constraints, the orchestrator presented here is only concerned with the automatic scaling aspect.

Regarding the auto-scaling aspect, Kubernetes has an implementation of a Horizontal Pod Autoscaler. A Pod is a basic container organization level, which represents a process running within the cluster (KUBERNETES, 2018). It encapsulates one or more containers, which will share storage resources, an IP, and other settings defined in the Pod description.

However, problems with the metrics used by this scaler were pointed out in Casalicchio e Perciballi (2017). The article evaluated the best types of metrics for predicting the required amount of Pods according to current demand. Two types of metrics were defined: relative, which measures the portion of resources that each container uses, and absolute, which represents the actual resource utilization of the physical system or VM. The article proposed an algorithm that uses only absolute metrics. It also demonstrated that for the benchmarks tested, absolute metrics are better at estimating the number of replicas needed than relative metrics used natively by Kubernetes. The orchestrator designed in this dissertation works with low-level metrics, as recommended by Casalicchio e Perciballi (2017). The results in Chapter 4 showed that this approach is indeed promising also in a slice context.

A new paradigm in the state of the art, that emerges from the relationship between the computing structures: IoT, edge (small data centers at the network edge), and cloud computing, is the Osmotic Computing (CARNEVALE et al., 2018). The purpose of this computing model is to enable the automatic deployment of microservices in highly distributed and federated environments interconnected between edge and cloud (VILLARI et al., 2016) structures. Taking the term borrowed from chemistry, osmotic computing aims to migrate microservices, usually in the form of containers, between cloud and edge in an organic and automatic manner. The possibility of migration of resources among parts of the same provider infrastructure or different providers is what correlates this paradigm with NECOS.

Carnevale et al. (2018) illustrates the architecture of an orchestrator capable of deploying microservices in an osmotic environment, where applications can migrate between layers: Cloud, edge, and IoT. Their work focuses on the abstraction of services, which are transformed into containerized “microelements” (MELS), and also on the abstract proposal of an orchestrator architecture. Orchestration is treated as a multi-objective op-

timization problem, taking into account power consumption, cost, and availability metrics. In contrast with this dissertation, the orchestration here is guided by one main objective: To ensure that a given tenant's service KPI is being kept under the agreed SLO, shrinking the allocated resources when possible to avoid waste. Also, besides the orchestrator architecture, this dissertation presents its implementation and experimental results.

The osmotic orchestrator's architecture presented in Carnevale et al. (2018) can be seen in Figure 3. On it, IoT devices and MELS are registered in a dashboard and then orchestrated by a Smart Orchestrator. Their Smart Orchestrator has the following modules: container manager, streaming management, training module, and prediction module.

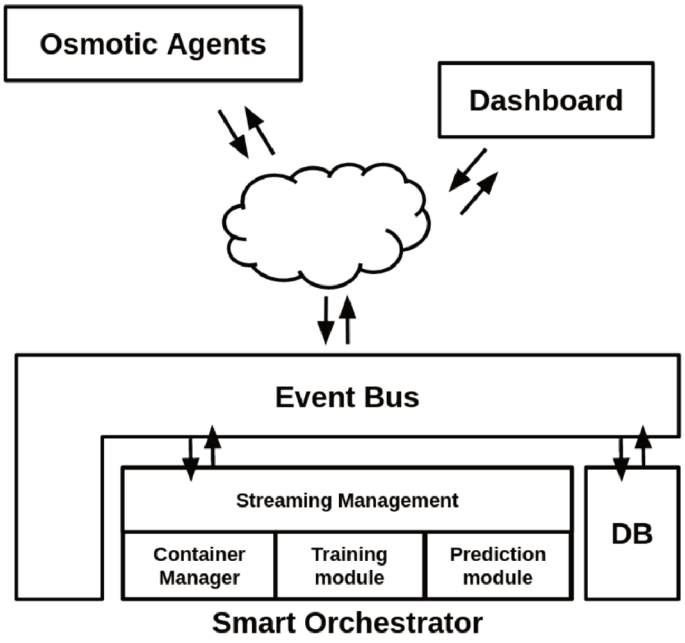


Figure 3 – Osmotic Orchestrator proposed at (CARNEVALE et al., 2018).

The streaming module should preprocess the metrics sent by devices to facilitate training and prediction tasks. The training and prediction modules would be responsible for the osmotic orchestrator learning. The training module would use Deep Learning (LECUN; BENGIO; HINTON, 2015) to learn patterns from metrics sent by Osmotic Agents to generate dynamic manifests for more assertive MELS deployments. Manifest templates would be saved to a database (DB module). The prediction module would then use the previous templates to generate new manifests that might be more useful in deploying MELS at different levels (IoT, Edge, Fog, and or Cloud).

The architecture proposed here, in this dissertation, has some similar features. Although there is a training module, it is used to create a model to forecast a tenant's service KPI. The strategy used to perform this forecast also uses a neural network, however, its used a RNN, which is described later in this chapter. There is also a module similar to the prediction module, in this case, doing forecasts in the future.

About slice orchestration, Sciancalepore, Cirillo e Costa-Perez (2017) splits resources into two management fronts: network and IoT devices. In this sense, it creates network and IoT device maximization functions based on priority. Because it is a rule-based orchestrator, this approach is not adaptable to variations in network and IoT device usage. Different from this one, the approach used here does not rely on fixed rules, being more flexible to deal with cloud applications which often face changes in their demands.

Closing this section, the Autonomous Computing paradigm, a term coined by IBM (KEPHART; CHESS, 2003) reflects the idea that computer systems can manage themselves using high-level goals given by administrators. This term comes from the autonomous systems found in nature, which hierarchically coordinate simpler autonomous systems, ranging from molecular machines within cells to societies and the international market.

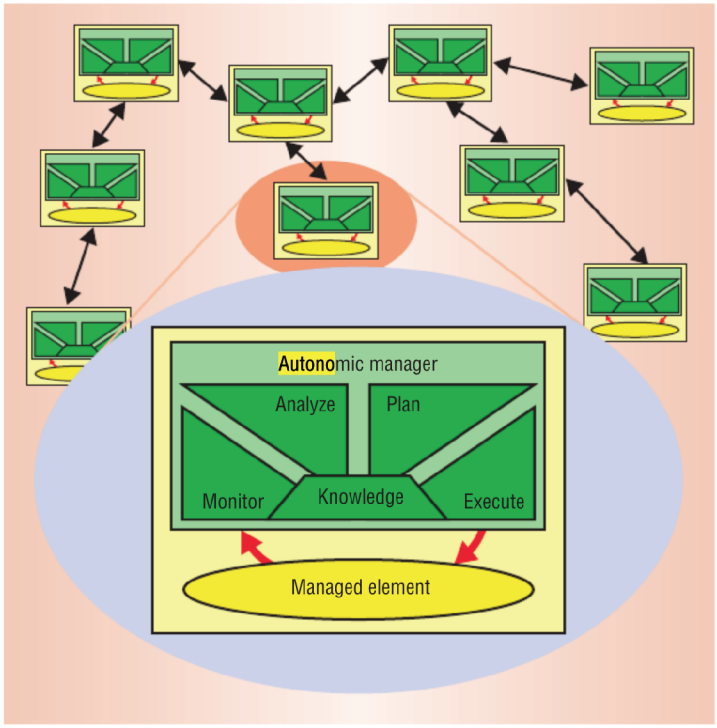


Figure 4 – Autonomous element structure (KEPHART; CHESS, 2003).

Figure 4 shows the schema of an autonomic element, which is composed of an autonomic manager and usually only one managed element. The standalone manager communicates with the outside environment by getting resources for itself and making it available to other standalone elements according to their functionality. It also receives self-level instructions that guide its management. On the other hand, a managed element can be seen as any common non-standalone system: a hardware resource (CPU, storage, printer), or software (a database, a large legacy system), or in the case of NECOS, a slice, or components of a slice.

In order to implement cloud resource orchestration Kephart e Chess (2003) suggests a comprehensive control loop composed of four phases: Monitoring, Analysis, Planning, and Execution – the MAPE loop. At the first, infrastructure usage metrics are collected, the second step inspects this data, checking if the application is healthy, the third generates actions to keep the application running nicely (if needed), and the last will apply the actions generated by the former.

Kephart e Chess (2003) also coined the predecessor of the MAPE cycle, the MAPE-K, that can be seen inside of the autonomic manager, in Figure 4. The MAPE-K cycle includes a Knowledge step, that is pervasive to the other stages of the cycle, used for learning and dynamic adjustments of all other steps.

NECOS orchestration approach fits on this one. The IMA module is responsible for the Monitoring part, but also includes a feature selection step, which is responsible for picking only the most relevant metrics to be collected for each slice. The SRO takes care of the Analysis and Planning phases, receiving the information collected by the IMA, estimating the future values for the tenant's KPI, and then using this intelligence to choose which actions are needed to apply. The chosen actions are then executed by the DC Slice Controller or WAN Slice Controller modules.

The Quality of Service (QoS) prediction from service provider metrics is analyzed in Pasquini e Stadler (2017). There, an artificial intelligence model is trained to predict the QoS being delivered to end-users. The way in which it is trained allows us to use only infrastructure metrics to do the service KPI prediction. This dissertation builds upon this work once it forecasts the KPI seconds ahead in the future, also using only service provider metrics to do the forecast during the run-time.

The last section of this chapter defines concepts that are used in time series forecasting and also, concepts related to Recurrent Neural Networks, used in this dissertation to perform the forecast.

2.3 Time Series Forecasting

Vazquez, Krishnan e John (2015) endorses how important is the use of time series forecasting in cloud computing applications, pointing out the benefits of predicting the infrastructure demand beforehand. The benefits include: to avoid the loss of potential sales, to keep the quality of service for current clients, and also to prevent the denial of the running service. According to Hyndman e Athanasopoulos (2018), a time series is "anything that is observed sequentially over time", being the time lag between each observation constant or variable.

A time series is a set of measurements of a metric, starting in a point at time 1 to t. This real measurements can be followed by forecasted values, which are separated from

the last collected measurement t by a time lag l:

$$ts = \{y_1, y_2, y_t, ..., y'_{t+l}\} \tag{1}$$

To perform the prediction of the future behavior of a variable y, a forecasting method is used, which can predict one or more values after the last observation y_t. As stated by Hyndman e Athanasopoulos (2018), the most straightforward way to perform this prediction is to use the past measurements of y to build a forecasting model like:

$$y_{t+1} = f(y_t, y_{t-1}, y_{t-2}, y_{t-3}, ..., error) \tag{2}$$

Where a function f should process the given observations and also use an "error" parameter, to allow for random variation and the effects of relevant variables that are not included in the model. Although this kind of model can represent the trends and seasonal patterns, it will not be able to use new information about other factors that affect the variable's behavior. Some methods that use this strategy are decomposition models, ARIMA models, exponential smoothings, and also moving averages.

Another kind of model uses predictor variables that explain the values of the forecasted variable, instead of its values. For this reason they are called explanatory models. This type of model can be represented as:

$$y_{t+1} = f(x_t^1, x_t^2, x_t^3, ..., x_t^k, error) \tag{3}$$

where x_t^k is the value of the predictor variable k at time t. It is possible to see it takes one measurement of each k predictor variables along with an error parameter to predict y_{t+1}. The choice of which model to use in forecasting relies on the resources and data available, the accuracy of the competing models, and the way in which the forecasting model is to be used (HYNDMAN; ATHANASOPOULOS, 2018).

The method used in this dissertation to generate an explanatory forecasting model is a Recurrent Neural Network. The reason for this choice is detailed in Chapter 3. An RNN is "a type of artificial neural network, elaborated to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, or numerical times series data emanating from sensors, stock markets, and government agencies" (NICHOLSON, 2019). The decision making of an RNN is a combination of the predictor variables and also the previous outputs did by the own RNN. This additional memory makes this kind of neural network a model that uses the recent past to determine its responses, just like a person that reads each letter of this phrase in sequence and then uses the previous information to extract the meaning of this text.

The basic structure of a neural network is two or more sequences of "neurons" layers, as can be seen in Figure 5 extracted from (HYNDMAN; ATHANASOPOULOS, 2018). This kind of neural network is called a multilayer feed-forward network. The first layer

receives the predictor's variables, also called inputs, and after extracting information about its values in the hidden middle layer(s), in form of weights, generates one or more outputs for the next layer(s).

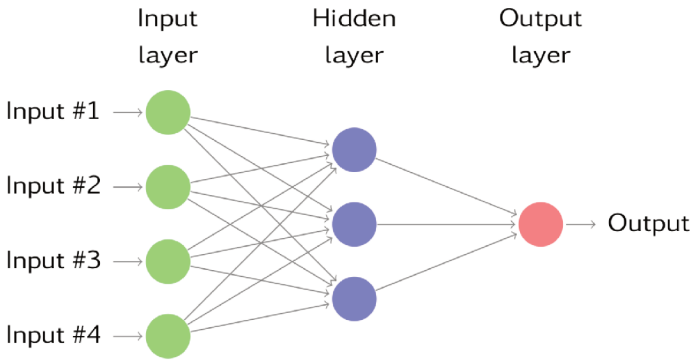


Figure 5 – Abstraction of a multilayer feed-forward neural network (HYNDMAN; ATHANASOPOULOS, 2018).

The main difference between a regular multilayer feed-forward neural network and a RNN can be seen comparing Figure 5 and Figure 6, from Nicholson (2019). In Figure 6, each x is an input example, w is the weight that filters the input, a is the activation of the hidden layer (a combination of weighted input and the previous hidden state), and b is the output of the hidden layer after it has been transformed. It is also possible to see the intermediate step in an RNN, where the output from each neuron given by b is passed to the neighbor to be one of its inputs. The sharing of this value is what gives the RNN memory ability.

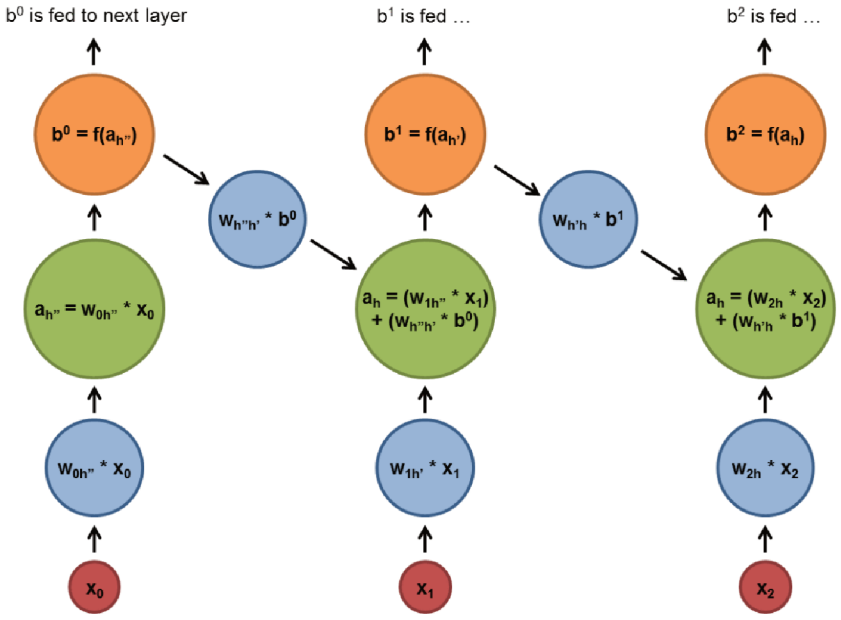


Figure 6 – How recurrent neural networks work (NICHOLSON, 2019).

Besides the description of the advantages to using forecasting in cloud orchestration, Vazquez, Krishnan e John (2015) also compares different forecasting methods including ARIMA models, exponential smoothing, and neural networks. To do so, traces of two services the Intel Netbatch and Google Cluster Data under a certain workload were recorded to be used as inputs to the methods to create a forecasting model. After performing experiments like the evaluation of training sets of different sizes, estimation of values that were not in the training set, and multiple-point forecasting, they concluded that none of the methods is optimal for all situations, but the neural network performance was at least in the average of the other methods.

The differences between Vazquez, Krishnan e John (2015) and this dissertation are: they did not use the infrastructure metrics where the services were run, they used only the service side metrics to perform the forecast. Also, the neural network used (HYNDMAN; KHANDAKAR et al., 2007) was a traditional feed-forward neural network, while here an RNN, which is a more sophisticated method, is used.

Chapter 3

Proposal

The proposition is presented in a top-down approach. Section 3.1 shows the high-level workflows executed to perform the machine learning-based slice orchestration. One step down in the level of abstraction, Section 3.2 shows the orchestration modules created to perform the workflows. It also shows the connection between the modules and how they work together to orchestrate the slices. Finally, Section 3.3 goes deeper into the central orchestration control loop, explaining it in a high-level code definition.

3.1 Orchestration Workflows

Section 2.1 defines the two kinds of elasticity concepts inside NECOS. The horizontal elasticity incorporates two actions that are out of the scope of this dissertation: the creation and the decommission of a slice part, which is a simplification of the slice creation and decommission. For this reason, this dissertation focuses only on the vertical elasticity, that updates the resource in an already allocated slice part.

As stated in Section 1.1, the choice of the best reconfiguration plan can be a very expansive combination problem. To overcome this problem, a flavor approach was used. Before the slice creation, the tenant is supposed to choose a set of possible slice configurations that the slice can assume in terms of resource amount. Each configuration in the set is a **slice flavor**. Each slice needs to have at least one flavor to start with. The relationship between a slice and a flavor is shown in the class diagram of Figure 7.

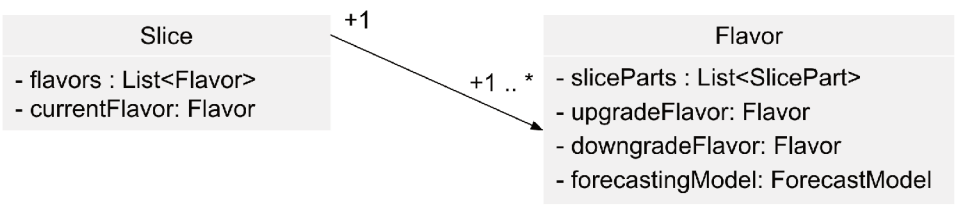


Figure 7 – Slice and flavor relationship.

Each slice flavor needs to have at most one flavor to upgrade to and at most one flavor to downgrade to, forming a linked list, where the flavors are linked between each other. Figure 8 shows three possible flavors for a hypothetical slice. The slice has three slice parts, in red, and each slice part has its resources, in green. In case of need, Flavor 1 can upgrade to Flavor 2, which has more resources in the top slice part. If even more resource is needed, Flavor 2 can upgrade to Flavor 3. In case of demand decrease, Flavor 3 can downgrade to Flavor 2, and then Flavor 2 to Flavor 1.

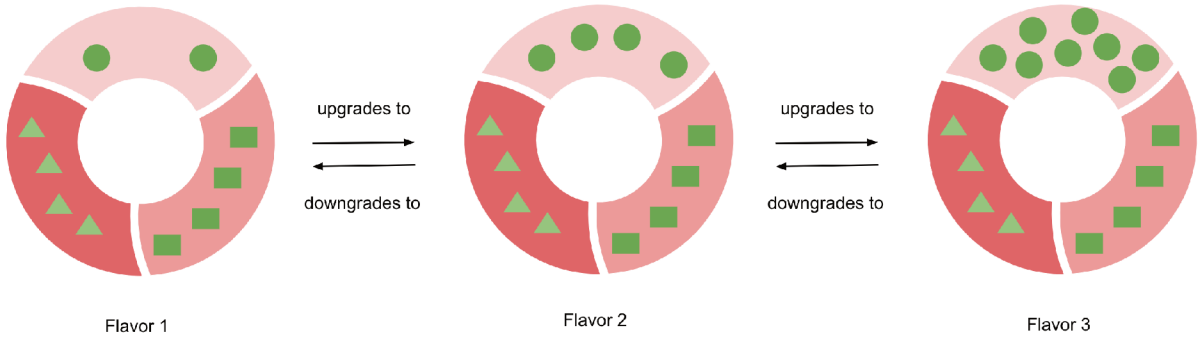


Figure 8 – Flavors upgrade and downgrade hierarchy.

It would be possible to develop a more sophisticated mechanism that would analyze what flavor was better for the current demand without respect to a predefined order. However, as the focus of the dissertation is not in the MAPE planning phase, this simpler approach is used.

However, the remarkable feature in this orchestration proposal is to predict when an elasticity action is going to be needed, which means when the workload over the slice requires a scaling action to be performed. The strategy used here is to learn how each flavor's infrastructure behaves under a certain workload. This learning is kept in the attribute *forecastingModel* showed in the Flavor class, in Figure 7. Before the orchestration is performed, each flavor is required to go under a profiling phase.

Figure 9 shows how the profiling phase works. The diagrams' notation is a green circle for a workflow starting point, orange circle for an ending point. A rounded gray rectangle indicates an action, a group of actions is included in a dashed rectangle with the name of the responsible NECOS component. A diamond shape is a gateway, and a diamond shape with an X inside means that only one action is going to be performed, meaning an exclusive gateway.

According to the flow shown in Figure 9, first, the tenant should collect measurements of the chosen tenant's service KPI, while its service is under a varying, and realistic, load (step 1). This tenant's service KPI can be collected by customized client software, and the load can be either real or artificial. After collecting this data, the tenant can request

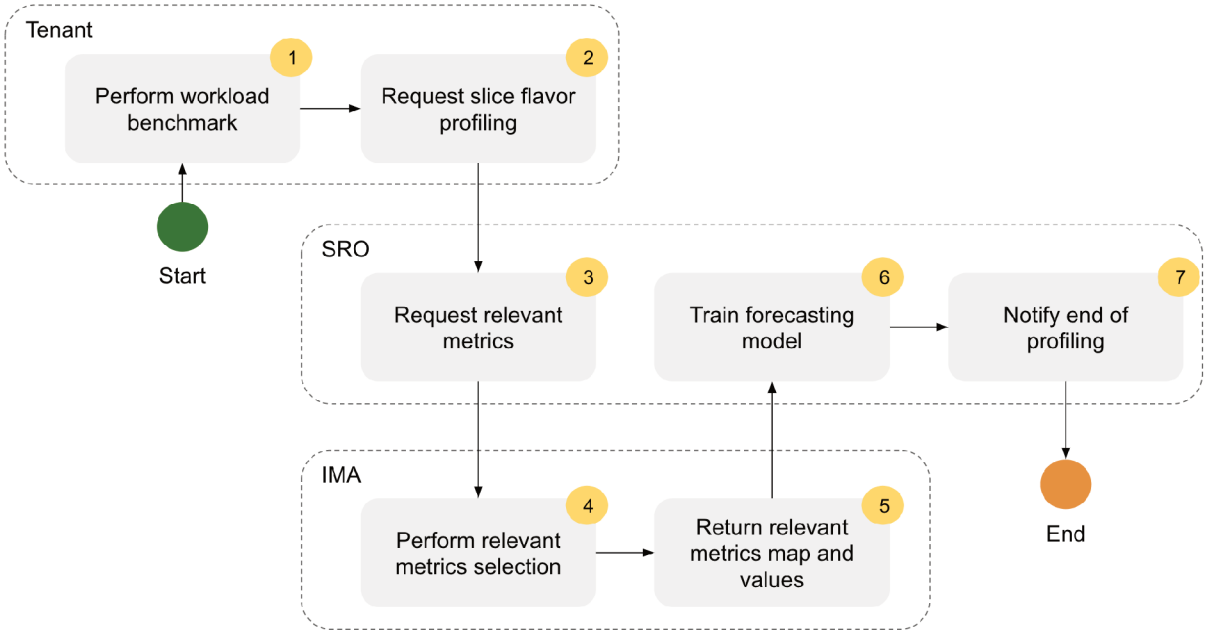


Figure 9 – Profiling Workflow.

the profiling of that flavor, sending the collection of the tenant's service measurements recorded during the benchmark. Specifically, the tenant needs to send the time series of its service's KPI, while requesting the flavor profiling (step 2). Then, the SRO connects the information from the service side, which is not always available, and the infrastructure information, which is always available.

As the amount of all infrastructure metrics can result in a huge data set very quickly (MARQUES et al., 2019), and the Slice Resource Orchestrator (SRO) needs to perform forecasting in real-time, it is necessary to request only the relevant metrics from the monitoring module (step 3). The responsible for this feature selection is the Infrastructure and Monitoring module (IMA), as it is in charge of collecting the infrastructure metrics, once knowing which are the relevant metrics can help to avoid processing unnecessary data (step 4). The IMA after performing this selection, using the relationship between the tenant's service KPI time series and the infrastructure metrics, returns the relevant feature names and their time series during the moment of the tenant's benchmark (step 5). Steps 4 and 5 were not subjects of this work.

Having the tenant's service KPI and the infrastructure metrics time series, the orchestrator can train a model (step 6), that relates both, which will allow the SRO to perform real-time forecasting of the KPI only having the infrastructure data during run time. In the end, a response is sent to a callback given by the tenant's informing that the profiling phase for that flavor is finished.

After profiling each flavor, the automatic resource orchestration can be performed. Figure 10 shows the orchestration workflow. Initially, the tenant makes a request to

the SRO to allow the automatic orchestration (step 1). The SRO informs the IMA to collect only the relevant metrics for that flavor (step 2). The IMA publishes the relevant metrics to an information bus that SRO (step 3) has access to. The SRO consumes the measurements from the bus until filling a window of measurements (step 4). Having this window full, forecasting can be performed (step 5).

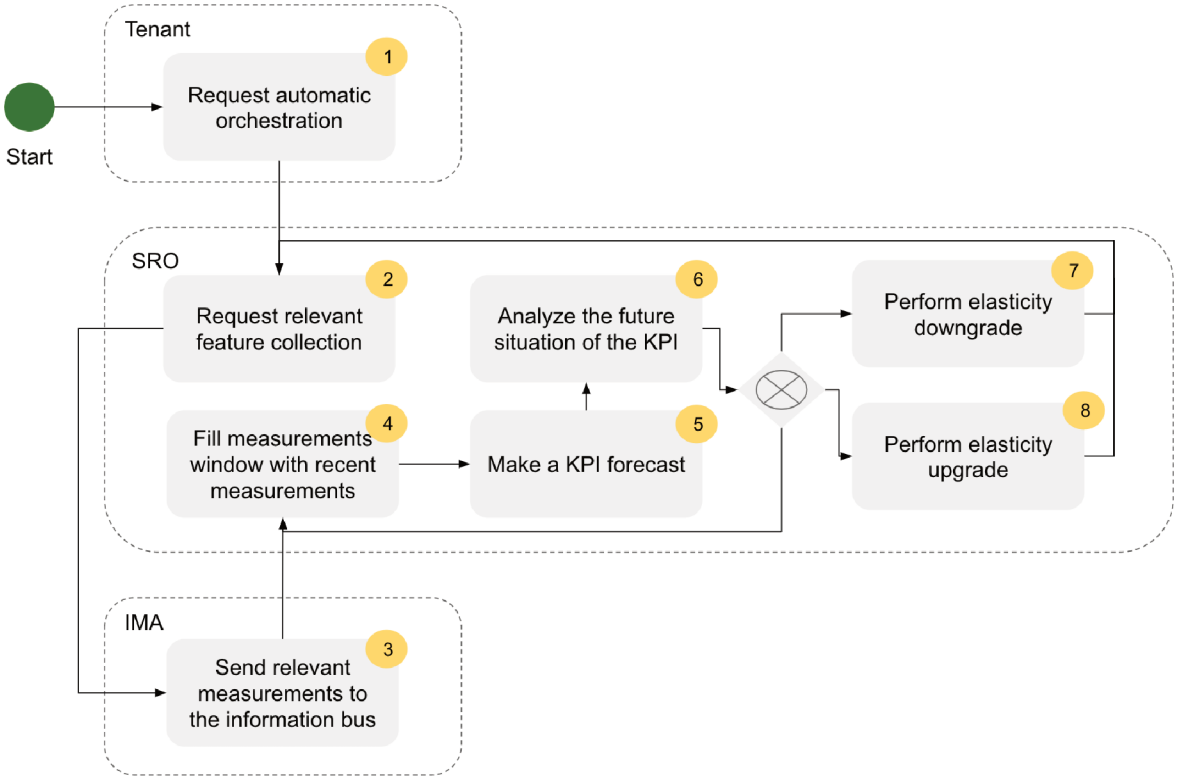


Figure 10 – Elasticity Workflow.

If the automatic orchestration is enabled, the value produced by the forecasting is analyzed (step 6) and three actions are possible:

(a) To trigger an elasticity downgrade (step 7), if the tenant's service KPI value is "too good" so the end-user is not paying enough to receive it. In this context, "too good" is defined as the KPI crossing a threshold given by the tenant, depending on one operator that the tenant picks (the script 3.3 shows how this evaluation can be done). It means to downgrade the flavor, and by consequence, the resources and the infrastructure costs. The value being "too good" depends on the service, and it is a parameter given by the tenant during the slice creation.

(b) To trigger an elasticity upgrade (step 8), if the value is "too bad" that the tenant can have its service compromised, and needs more resources to avoid it. It means to upgrade the flavor, and as result, increase resources and costs. The badness of the tenant's service KPI is also given as a parameter during the slice creation.

(c) The last possible outcome is to keep the same configuration, once the forecasted tenant's service KPI value is under conformance, so no action is required and the loop can continue to step (4).

If one of the options (a) or (b) is chosen, a different set of relevant metrics may be needed, resulting in a request for relevant feature collection. On the other hand, if (c) is chosen, then the loop continues with the measurements window being updated with newly published measurements, and so on. This workflow lacks an end state because it should be "infinity" while the slice run time lasts.

Next, Section 3.2 presents more details about the orchestration implementation, in terms of orchestrator components, their relationships, and tasks.

3.2 Orchestrator Architecture

The architecture shown in Figure 11 is responsible for performing the previous workflows. The components developed during this work are bounded by dashed lines, the others are represented to improve the visualization of the workflow steps. Both workflows start with the tenant performing a request to the SRO through the **SRO Server** component. The SRO Server is the SRO entry point. The main endpoints exposed by this server are:

profile_flavor - Receives a flavor profiling request, at step 1 of the Profiling Workflow (Figure 9). The request contains a tenant's service KPI measurements file, in a Comma Separated Values (CSV) format. This file is called the "X file", once it is the independent variables file. Also, it is one of the two input files that are necessary to build the forecasting model. The consistency of this file is checked (format, maximum size), and then it is delivered to the Slice Controller to be processed.

register_metrics_flavor - Once the request of the relevant metrics to the IMA is asynchronous, this endpoint is a callback that is used at step 5, of the Profiling Workflow. It receives a file, called "Y file", with the relevant infrastructure metrics measurements, that were collected during the tenant's benchmark. The Y file is sent to the Slice Controller to be processed along with the X file.

start_sla_forecasting - This endpoint is exposed to allow the tenant to visualize, at the Slice Monitoring Dashboard, the tenant's service KPI forecasting values, without enabling the automatic orchestration. It triggers a call to a task inside the Slice Controller to perform the KPI forecasting and then publish the values in the Information Bus.

start_orchestration - Responsible to start the automatic orchestration as shown in Figure 10, step 1. It enables the automatic orchestration option inside the Slice

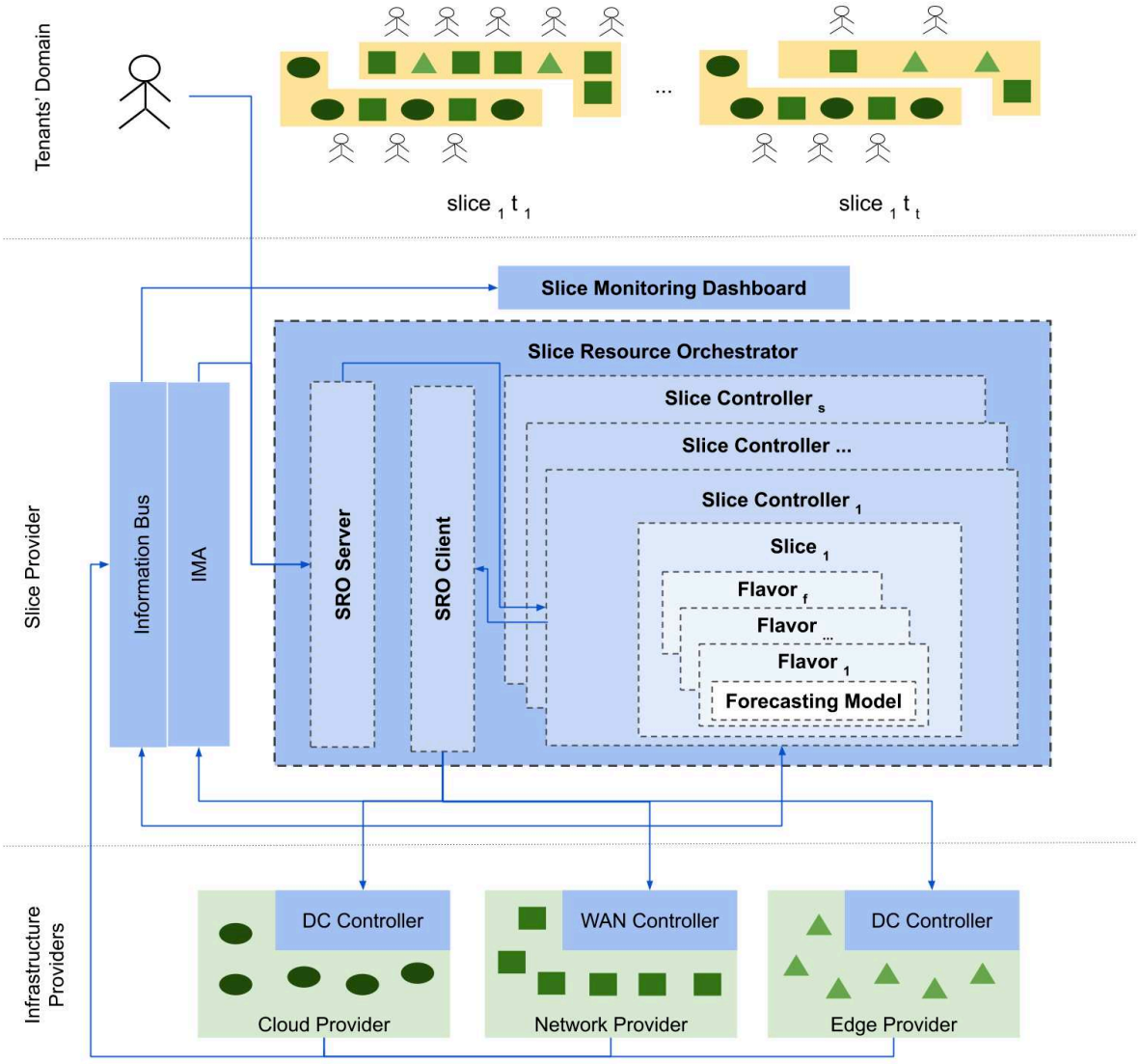


Figure 11 – SRO modules in NECOS scenario. The allocated slice 1 with its resources is shown in yellow, NECOS related modules in blue, and resource providers in green.

Controller, allowing changes in the slice flavor based on the tenant's service KPI forecasting.

elasticity_upgrade - Available if the tenant wants to perform a flavor upgrade manually. Triggers the actions as in step 7 of the Elasticity Workflow.

elasticity_downgrade - As the previous one, defined if the tenant wants to perform a flavor downgrade manually. It starts the actions as in step 8 of the Elasticity Workflow.

While the SRO Server receives tasks requests, the **SRO Client** is responsible to make tasks requests to other components. For example, it requests the metrics selection to

the IMA, at step 3 of the Profiling Workflow in Figure 9. Also, it is used by the Slice Controller to communicate with other components. It handles the request to change the metrics set being collected when a flavor switching is performed, once two flavors of the same slice can have different sets of most relevant features (step 2 of Elasticity Workflow in Figure 10). When an elasticity action needs to be executed, it is also responsible for requesting the elasticity changes to the appropriate resource provider (steps 7 and 8 of Elasticity Workflow in Figure 10). These two modules are the SRO interface to the external world.

As in an autonomous element (Section 2.2), where each element has an autonomous control loop, here, each slice has its control loop being executed by its own **Slice Controller** instance. After the forecasting or automatic orchestration being enable, this component should consume the measurements of the most relevant metrics from the information bus, which is the step 4 of the Elasticity Workflow in Figure 10. It is defined as the **consume_measurements** task, and the goal is to fill a window W like the matrix structure:

$$W = \begin{bmatrix} x_{t,1} & x_{t,2} & \cdots & x_{t,k} \\ x_{t+l,1} & x_{t+l,2} & \cdots & x_{t+l,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{t+h,1} & x_{t+h,2} & \cdots & x_{t+h,k} \end{bmatrix}$$

where each $x_{i,j}$ is a measurement of the relevant metric j at time i. All measurements in the same row were collected at the same time. All measurements from the same column belong to the same infrastructure metric. This window has size $h \times k$, where h is the number of samples needed for each metric, which means, the history of the previous measurements necessary to forecast, and k is the number of metrics that were selected in the profiling phase. The k parameter is selected by the IMA, after performing an analysis of the correlation among the infrastructure metrics and the tenant's service KPI. The l indicates the time lag passed between two measurements of the same metric being collected. For example, a metric can be measured every 5 seconds, so $l = 5$.

After having a full window of infrastructure metrics, the next task is to **forecast_kpi**. It is done using the model trained at the flavor profiling phase. How the training is performed is shown in Section 3.4. The Slice Controller selects the current slice flavor and uses its **Forecasting Model** component, passing to it the filled window. This component returns the forecasted value, which goes to the **analyze_kpi** task, in the Slice Controler. The behavior of this task was already described in step 6 description of the Elasticity Workflow in Figure 10. If the analysis requires any elasticity action, which means in our context a flavor change, the infrastructure updates are requested using the

SRO Client component, which asks the appropriate resource providers to perform the necessary changes.

Finally, the tenant can follow what is happening with its slice using a **Slice Monitoring Dashboard** which summarizes the information that is being published in the information bus. In the dashboard are presented the forecasted tenant's service KPI values, the current most relevant metrics measurements, and the last actions performed by the orchestrator.

The next section shows the control loop performed by the Slice Controller of each slice in detail.

3.3 The Slice Control Loop

In this section, pseudo code is used to show in detail some important tasks cited before. The first piece of code is the Algorithm 3.1. It shows the control loop process which is the core of the slice orchestration and runs inside the Slice Controller component. It loops through the measurements collected, reading the messages published by the IMA in the information bus. The aim is to fill a line from the window W, grouping all metrics collected at the same timestamp. The *current_measurements* is the variable that keeps the measurements set.

```
1  current_timestamp = now()
2  window = empty_window()
3  current_measurements = empty_measurements_list()
4
5  for message in information_bus:
6    if current_flavor.is_relevant_metric(message.metric):
7      if message.timestamp == current_timestamp:
8        current_measurements.add(message.measurement)
9      else if message.timestamp > current_timestamp:
10       current_measurements = empty_measurements_list()
11       current_measurements.add(message.metric, message.measurement)
12       current_timestamp = message.timestamp
13
14   if current_measurements.is_completed():
15       window.update_window(current_measurements)
16       if window.is_full():
17         kpi_value = current_flavor.forecasting_model.forecast_kpi(window)
18         publish_forecast(kpi_value, current_timestamp)
19         current_measurements = empty_measurements_list()
20         if is_orchestration_enabled:
```

```
21              analyze_kpi(kpi_value)
```

Algorithm 3.1 – Slice Control Loop.

A message contains the metric name, the measurement value, and the timestamp marking the time the measurement was collected. A message is processed if it is a relevant metric for the current flavor (messages from previous flavors could be in the information bus). A measurement is inserted inside the *current_measurements* if it has the same timestamp as the others (line 7). Line 9 means that if the current measurement is newer than the others inside the *current_measurements*, a new list of *current_measurements* of that timestamp should be completed.

When a line of the window is completed, it is inserted into it (line 15). When the window is completed, a tenant's service KPI forecasting can be performed (line 17). The current flavor of the slice has a forecasting model that is used to perform this prediction. If the automatic orchestration is enabled (line 20) this value is analyzed and based on it, and elasticity action may be performed. This evaluation is detailed at the Algorithm 3.2.

In this function, line 2 shows that if some elasticity action is being done, until it is finished, no other elasticity action can be performed, once the values being collected from the infrastructure do not represent the flavor.

```
1 analyze_kpi(kpi_value):
2   if not is_performing_elasticity:
3     if current_flavor.is_too_good(kpi_value):
4       request_elasticity(''downgrade'')
5     if current_flavor.is_too_bad(kpi_value):
6       request_elasticity(''upgrade'')
```

Algorithm 3.2 – analyze_kpi function.

At the lines 3 and 5 the functions *is_too_good* and *is_too_bad* define the border values that the tenant's service KPI should not cross. For example, consider that a tenant chose as the KPI the download velocity being delivered to its end users. This way, the tenant stipulates that 100 Mb/s of download throughput is the highest velocity that it provides. On the other hand, it guarantees at least 10 Mb/s. In this context, the forecasted value of 200 is greater than 100, so it is "too good" once the clients are not paying by it. On the other side, 5 is "too bad", because the clients are expecting at least 10 Mb/s. In this case, being "too good" is a $kpi\_value > 100$ relationship and a being "too bad" is a $kpi\_value < 10$.

The relationship being greater than a threshold is good and being less than a threshold is bad does not work for all cases. If the tenant's service KPI represents a response

time, for example, being great than a value can indicate that the system is too slow, so an elasticity upgrade should be performed, while if the value is too low, and elasticity downgrade should take place, once the end-users may not be paying enough to receive it.

This way, the logic behind the *is_too_good* function can be defined as shown at the Algorithm 3.3, where the *good_operator* and the *good_threshold* are previously chosen by the tenant, when establishing the SLO's, and the *good_operator* belongs to the $valid\_operators = \{<, \leq, =, \geq, >\}$ set. The test performed at the line 4 depends on another parameter provided by the tenant, the "too_good_occurrences", and it is used to avoid the elasticity being triggered because of punctual oscillations at the tenant's service KPI forecasting. This value is reset after a period of time, also provided by the tenant.

```
1 is_too_good(kpi_value):
2   if good_operator(kpi_value, good_threshold):
3     too_good_occurrences = too_good_occurrences + 1
4     if too_good_occurrences > allowed_times_too_good:
5       return True
6   return False
```

Algorithm 3.3 – Function inside Flavor that evaluates if a forecasted tenant's service KPI value is too good.

The *is_too_bad* function behavior is analogous to the Algorithm 3.3. Continuing at the Algorithm 3.2, if the return of the functions "is_too_good" or "is_too_bad" is positive, an elasticity action is requested. Details about this request are shown at Algorithm 3.4. If the operation is a downgrade, line 2, the algorithm checks if there is a downgrade flavor for that current flavor. If it has, the global flag *is_performing_elasticity* is set to true and a message of elasticity is published at the information bus. Next, a function at the Flavor component is called so it can call the actions in the providers to downgrade the flavor. The global variable *new_flavor* holds temporarily the next slice flavor. The behavior when a flavor upgrade is performed is similar.

```
1 request_elasticity(operation):
2   if operation == ''downgrade'':
3     if current_flavor.has_downgrade_flavor():
4       is_performing_elasticity = True
5       publish_sro_log(''Requesting elasticity downgrade'')
6       current_flavor.request_downgrade()
7       new_flavor = current_flavor.downgrade_flavor
8     else:
9       publish_sro_log(''Current flavor does not allow downgrade'')
```

```
10
11    if operation == ''upgrade'':
12      if current_flavor.has_upgrade_flavor():
13        is_performing_elasticity = True
14        publish_sro_log(''Requesting elasticity upgrade.'')
15        current_flavor.request_upgrade()
16        new_flavor = current_flavor.upgrade_flavor
17      else:
18        publish_sro_log(''Current flavor does not allow upgrade'')
```

Algorithm 3.4 – Elasticity request code.

Finally, when the providers return that the elasticity actions were completed, a callback function, shown at Algorithm 3.5 is called. It sets the *is_performing_elasticity* flag to False, once the elasticity is completed, it resets the measurements window then updates the *current_flavor* variable to the new current slice flavor, and finally, it publishes the message that the elasticity action previously requested was completed.

```
1  elasticity_callback()
2    is_performing_elasticity = False
3    window = empty_window()
4    current_flavor = new_flavor
5    publish_sro_log(''Elasticity completed'')
```

Algorithm 3.5 – Elasticity callback.

All these actions are possible only because at the profiling phase a machine learning model was trained to perform the forecasting. The next section gives details about how the forecasting model used at line 17 of Algorithm 3.1 was trained.

3.4 Forecasting Model

Here, the forecasting model is trained to answer the question: "Given a sequence of infrastructure monitoring measurements, which is the most probable tenant's service KPI value in s seconds?". This question regards the two hypotheses from Section 1.3, once it requires that the forecast model uses only the infrastructure metrics (first hypotheses) and predicts the future (second hypotheses). To do so, the intelligent model first needs to learn how to associate the monitoring metrics with the tenant's service KPI. In machine learning, this process is called as "training phase". However, before training the model to perform the forecast, the training data needs to be prepared, which is called data preprocessing. First, the previously received files X and Y are read by the SRO and then

have their values under a standardization process. It is done by subtracting the mean and dividing by the standard deviation of each metric (TENSORFLOW, 2019). The standardization is needed because some metrics can have very different scales, and doing that process gives the same meaning to all values: how much they are different from the standard deviation (FROST, 2020).

Different from "vanilla neural networks" that are fed with a vector containing a single measurement of each metric, the RNN input is a sequence of measurements of each metric. The reason to give this piece of time series as input is to provide context for the RNN to perform a more precise forecast. Therefore, the X file is broken in windows of size h, where h is the window size. Each window is then associated with the future tenant's service KPI value which is the target, which is shown in detail next. This is a "labeling" process, where the labels are in the Y file. The label is the value that we want our model to "guess" from the features that we have.

In our case, it means that if a window has the values starting at the timestamp t to $t+h$, the target is the y value in the Y file with timestamp $t+h+s$, where s is the distance in the future that we want to forecast. Figure 12 shows the labeling phase. There, it is possible to see in the middle a data set instance, which is the window W extracted from the X file and its corresponding label, from Y file. In the figure, it is also possible to see that the Y file starts at $h+s$, instead of 1, which represents the first timestamp of the X file, because the first label labels the window metrics from 1 to h.

After the files being preprocessed in instances, each instance needs to be digested by the RNN more than one time, to be effectively learned. Thus, the training instances are shuffled and grouped in batches of size B. After having the dataset in batches the training phase can start. It is shown in Figure 13. The RNN architecture is in purple, in which each circle is a neuron. The architecture used here was inspired by the RNN shown in Tensorflow (2019), but other architectures could be applied as well. The RNN has three layers, one input layer, one hidden layer, and one output layer. The input layer size depends on the window size of h. The middle layer, which is the "hidden" layer has a fixed size of 32 neurons. One neuron is used in the output because just one value needs to be estimated by this RNN.

The pink arrows that connect the neurons in the hidden layer show the recurrent element of the neural network, in which case one neuron gives to the next its output. This is what gives the RNN the ability to predict based on the past context.

Before the training starts, the RNN is initialized with random weights in its neurons. Each input neuron processes a vector from the window and passes the processed value in the flow of the black arrows. Again, the hidden layer process these values, and the outcome of each neuron is passed to the following neuron of the layer, which is shown by the pink arrows. After that, all values are once more combined in the output layer, by the single neuron. The produced final output (y') is going to be compared with the

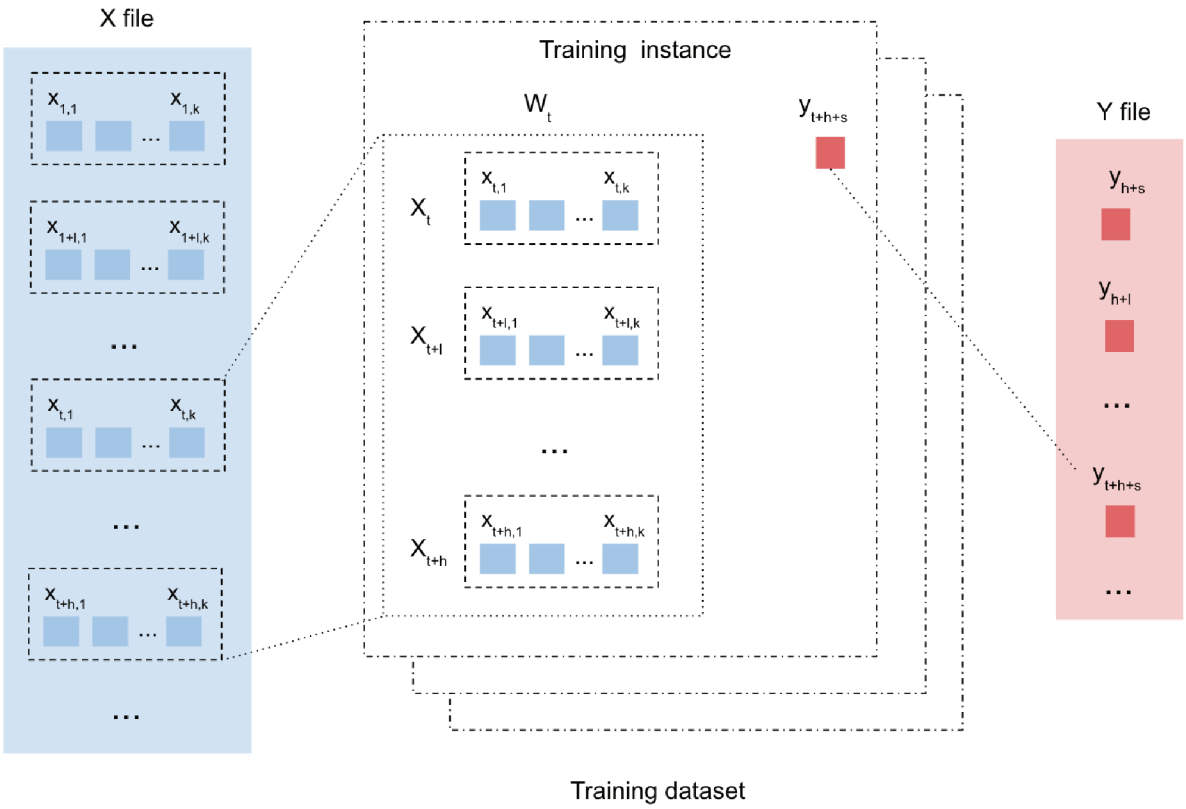


Figure 12 – Dataset during pre-process phase.

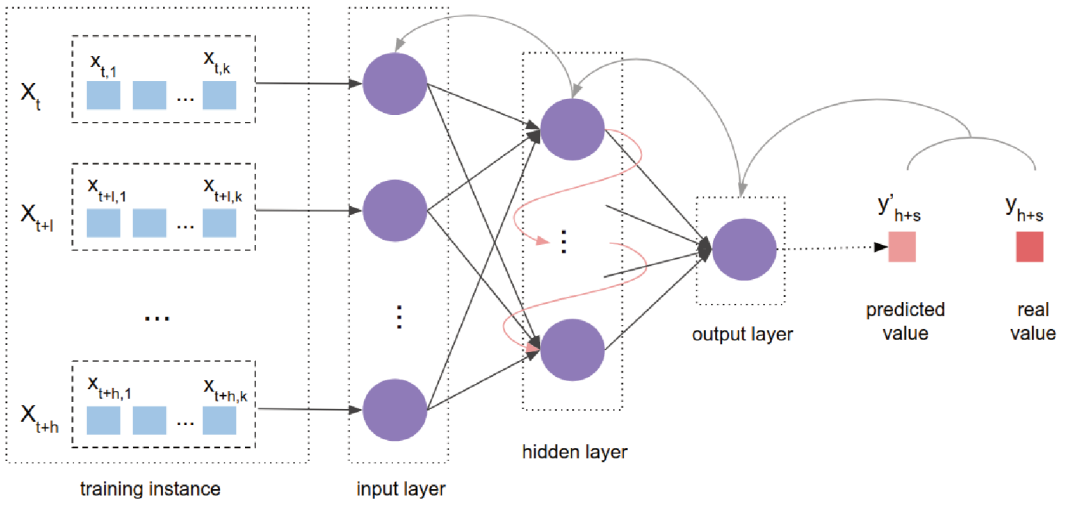


Figure 13 – Training phase schema.

real value from the training instance (y), and using the difference of these two values, the RNN adjusts its neuron weights, from the right to left. This process is known as backpropagation, and it is illustrated by the gray arrows. All this process continues for each training instance in each batch for a predefined number of times, and the expectation is that the RNN weights converge to a number that can predict the target value with a

certain precision. Finally, after the model has been trained, it is not going to need the label value y anymore. The only necessary input is the recent window of metrics, which is the recent infrastructure history to perform the forecasting.

The presented orchestration approach is totally agnostic to the service running on top of the cloud-network slice, which makes it suitable for many kinds of applications. Another advantage is that it is proactive, being capable to adjust the infrastructure before an SLA breach. Also, after profiling the flavors and set the thresholds, it runs automatically after triggered, free of manual interaction.

Next, the following chapter shows the implementation of the SRO prototype and its behavior. It shows the testbed built to prove the proposed SRO architecture. It also shows results like the RNN training convergence, behavior of the forecasting values for each slice flavor, and the elasticity triggering based on the forecasting values.

Chapter 4

Experimental Results and Analysis

This chapter shows how the intelligent slice orchestration architecture proposed before can evaluate the hypotheses from Section 1.3. To do so, a testbed, which is a platform of experiments was set up to provide a slice, its elasticity, and monitoring metrics. Section 4.1 shows more details about this testbed. Section 4.2 shows how the experiments were run on this platform and also, the results obtained from each experiment. The discussion about the outcomes of the experiments is provided as they are presented.

4.1 Evaluation Method

In order to embody the proposed architecture shown in Figure 11 and make the experimental process closer to reality, a real private cloud infrastructure was set up. This way, it was possible to collect real service and infrastructure metrics and also apply load and elasticity actions on them. Figure 14 shows this testbed and Table 1 shows the configuration of each used machine. In blue, are the NECOS components. In green, the Resource Domains division, showing an infrastructure provider simulation. In purple is the third part software used. They include:

Granafa: An open-source analytics and monitoring tool that facilitates the visualization of time series databases (GRAFANA LABS, 2020). It is used as the visualization part of the Slice Monitoring Dashboard. The version used was 6.2.4.

InfluxDB: A database optimized to store time series in real-time (INFLUXDATA INC, 2020) and open source. It groups and stores the monitoring and orchestration data that Grafana displays. This system was installed in the 1.7.6 version.

Kafka: A streaming platform, similar to a message queue. It stores the streams of records in a fault-tolerant and durable way (APACHE KAFKA, 2017). Kafka can be deployed in a cluster of one or more servers, storing the streams in categories called topics. It is used as the information bus of the orchestration architecture. The

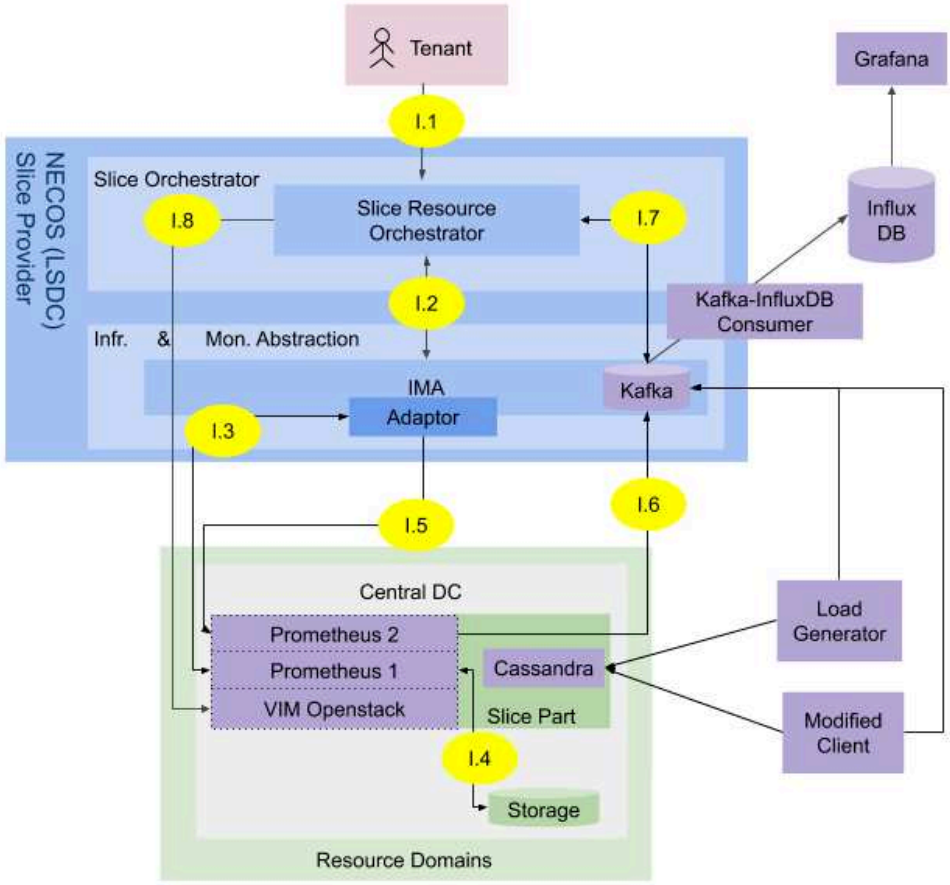


Figure 14 – Test bed.

SRO is subscribed to the slice metrics' topic and publishes its forecast to another specific topic. Version 2.11-2.0.0 is used.

Kafka-InfluxDB-Consumer: An open-source Kafka consumer for InfluxDB written in Python (ENDLER, 2018). After being subscribed to receive topics like the SRO forecasting results and monitoring selected metrics from IMA, it reads the information published in Kafka and stores it in the InfluxDB database.

Prometheus: An open-source systems monitoring used to collect the slice metrics (THE LINUX FOUNDATION, 2020). It collects the server-related metrics associated with CPU, memory, and network. Its function is to collect the metrics that are provided by the IMA. The installed version is 2.2.1.

Cassandra: It is a NoSQL database management system, open-source, and free (THE APACHE SOFTWARE FOUNDATION, 2016). In the testbed, it represents the Tenant service running on top of the slice. The Cassandra version used is the 3.11.3.

Load Generator: It is a python code that was programmed to generate client processes that make requests to the Cassandra service (CUNHA, 2019). It is configurable so

Components hosted	OS	RAM	HD	Processor
IMA	Ubuntu 16.04	16GB	500GB	8 vCPUs
SRO	Ubuntu 16.04	16GB	500GB	8 vCPUs
Influx DB, Kafka-InfluxDB	Ubuntu 16.04	16GB	500GB	8 vCPUs
Grafana	Ubuntu 16.04	16GB	500GB	8 vCPUs
Kafka Node 1	Ubuntu 16.04	4GB	20GB	1 vCPU
Kafka Node 2	Ubuntu 16.04	4GB	20GB	1 vCPU
Kafka Node 3	Ubuntu 16.04	4GB	20GB	1 vCPU
Kafka Node 4	Ubuntu 16.04	4GB	20GB	1 vCPU
Kafka Node 5	Ubuntu 16.04	4GB	20GB	1 vCPU
Load Generator	Ubuntu 16.04	8GB	20GB	2 vCPUs
Modified Client	Ubuntu 16.04	4GB	20GB	2 vCPUs
Prometheus 1, Prometheus 2	Ubuntu 16.04	8GB	500GB	4 vCPUs

Table 1 – Testbed machine configurations.

the client creation can follow a certain distribution like a Gaussian, for example.

Modified Client: It is a program written in Java, that was used to perform queries to the Cassandra service (CUNHA, 2019), as an end-user would do. It also records the tenant's service KPI measurements, and export them as the Y file in .csv format.

VIM Openstack: It is a cloud operating system (VEXXHOST, 2020), and its role was to create all the VMs employed in the testbed. The version used was Openstack Queens, 3.14.0.

The slice holding the Cassandra service was composed of one slice part which had 5 servers, forming a Cassandra cluster. Each cluster node had 1 vCPU, 4GB of RAM, and 50GB of disk. This slice could have two different flavors, and the difference was in the network traffic police. In the first flavor, which was the "smaller flavor", a traffic control police had been configured to allow only 10Mbit to pass through the main network interface of each cluster node. This was done by using the Linux *tc* command:

```
sudo tc qdisc add dev ens3 root tbf rate 10Mbit latency 1ms
                        burst 10000
```

Such a command uses the *iproute* package. The latency was 1ms because the aiming was not simulating a distant network package traveling. The burst of 10000 was chosen to not compromise the package processing. In the "bigger flavor" the traffic restriction was 30Mbit, with the same other parameters. The 10Mbit and 30Mbit values were chosen once they could induce variations in the chosen tenant's service KPI. The KPI was the 99^{th} percentile of the reading response time, in milliseconds. It indicates that the reading response time of 99% of the reading requests performed by the Modified Client is under

this value, as a maximum mark. Section 4.2 shows more details about the behavior of this tenant's service KPI in both flavors.

To perform the profiling workflow, a tenant's service KPI file (Y file) is needed for each slice flavor. In order to generate it, three main modules shown before were used: the Cassandra Service, hosted in a cluster, the Modified Client, and the Load Generator. Figure 15 shows their interactions. In the center are the five nodes that compose the Cassandra Cluster, which was connected by a bandwidth b. This bandwidth varied (10Mbit or 30Mbit) depending on the selected cloud-network slice flavor. The Modified Client, on the right, was responsible by generate the Y file with the tenant's service KPI measurements. This means that it performed requests to the service and recorded the tenant's service KPI that was delivered to it, the 99^{th} percentile of the reading response time. On the other hand, the Load Generator created and destroyed clients that also performed requests to the service, in a wavy form. Meanwhile, the underlying Prometheus 1 instance (omitted) was collecting all the infrastructure metrics available in the Cassandra Cluster, and keeping them in the local storage, which were used later to generate the X file.

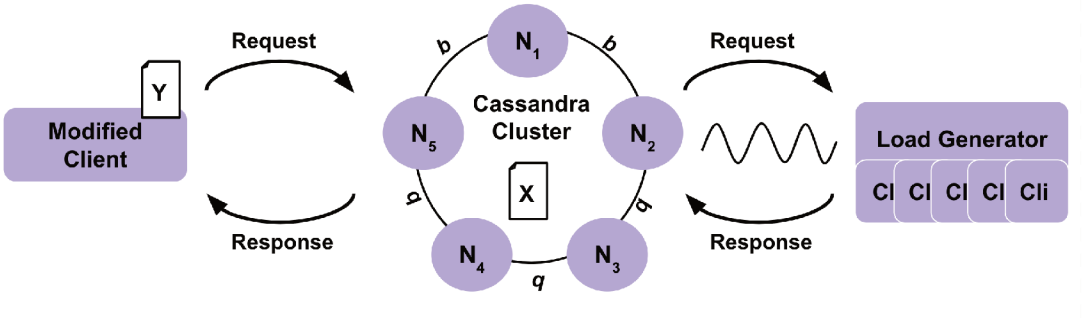


Figure 15 – Closer view of the interaction among the Cassandra Cluster, the Modified Client and the Load Generator.

After this required step, the profiling phase was started. It is represented by the interaction (I.1) shown in Figure 14 in yellow. This interaction was triggered by the following HTTP POST request:

```
curl -X POST "http://{SRO_IP:SRO_PORT}/slice/{sliceId}/
flavor/{flavorId}/profile?url_to_answer={callbackEndpoint}
&sla_metric_name={KPI}&timestamp_label={timestampField}" -H
"Content-Type: multipart/form-data" -F "file=@{yFile.csv}"
```

using the *curl* program, a command-line software used to perform requests from the terminal. The parameters were: The IP of the machine running the SRO (SRO_IP), the SRO server port (SRO_PORT), the slice identifier ($sliceId$), the flavor identifier ($flavorId$), the URL to be called to inform the profiling phase result ($callbackEndpoint$), the KPI

name (KPI), the name of the field in the file that represents the measurements' timestamps ($timestampField$), and finally, the Y file being transferred to the SRO ($yFile.csv$).

After receiving this request, the SRO called the IMA to select the most relevant metrics of the infrastructure to train the flavor model, also passing the Y file (interaction I.2 in Figure 14). The IMA then requests to Prometheus 1 all the previously collected metrics to perform the metrics selection (interaction I.3 in Figure 14). Next, the IMA returns the selected metrics in a .csv file, which was the X file (interaction I.2 in Figure 14). The SRO then trains the flavor model that represented that flavor under the given load.

Once both flavors had been profiled, the SRO could accept forecasting requests or orchestration requests (interaction I.1 in Figure 14). For the forecasting, the request would be:

```
curl -X POST
http://{SRO_IP:SRO_PORT}/slice/{sliceId}/start_sla_forecast
```

or to enable forecasting based orchestration:

```
curl -X POST
http://{SRO_IP:SRO_PORT}/slice/{sliceId}/start_orchestration
```

When the forecasting request is received, the SRO requests the IMA to publish real-time measurements of the K selected metrics (interaction I.2 in Figure 14). The IMA requests the Prometheus 2 to send the measurements to Kafka (interactions I.5, I.4, and I.6 in Figure 14). The SRO then consumes the real-time data, in order to forecast the tenant's service KPI s seconds in the future (interaction I.7 in Figure 14). The SRO publishes its forecasted values to Kafka, which are plot in a Grafana dashboard (interaction I.7 in Figure 14). If the orchestration request is performed, the SRO may execute actions in the infrastructure to keep the selected tenant's service KPI in compliance (interaction I.8 in Figure 14), which means, to keep its values inside the acceptable interval previously defined.

To know whether the forecasting models were representing well the flavors conditions, it was needed to measure their accuracy. According to Hyndman e Athanasopoulos (2018), it is possible to measure the forecast accuracy by summarizing the forecast errors. In their book, they show a percentage metric called Mean Absolute Percentage Error (MAPE), which is a unit-free metric and can compare forecast performances between datasets. This MAPE metric was chosen to measure the models' accuracy in this dissertation. Note that from now on, the MAPE acronym will be used to designate this metric and not the MAPE loop mentioned earlier in this work. It can be calculated doing:

$$MAPE = \frac{\sum_1^n \frac{|y_i' - y_i|}{y_i} * 100}{n} \tag{4}$$

where y' stands for the forecasted value at time i, y_i is the real value for the same time i and n is the number of samples. Each trained model is going to have its MAPE compared with the MAPE of a naïve method, a simple mean of the training KPI values, which is going to be used as a baseline. This is done to answer the questions: Is the RNN prediction better than a simple and cheap arithmetic mean? Does the RNN provide a better estimation that makes it worth it to be used?

The next section brings the experimental results obtained using the experimental strategy and the proposed testbed. It shows how the tenant's service KPI floats under different flavor and load conditions, how the trained models perform for each flavor, and finally, how the orchestration could keep the KPI in conformance.

4.2 Experiments

To validate the hypotheses at Section 1.3, this section presents three kinds of experiment and their results. The first hypothesis, H1, aimed to check if it was possible to create a model that predicted the tenant's KPI using only infrastructure data, and H2 hoped to verify if the model could be used in a run time orchestration to maintain the SLA and also save costs. The experiments worked as a three steps pipeline, where the output of one was used as input by the next:

1. Put the testbed under load to gather the infrastructure and client metrics for each slice flavor. It represents steps 1 to 5 of the Profiling Workflow (Figure 9). It was used to collect data for hypothesis H1.

2. With the collected data, to train different RNN models for each flavor, varying the window size and forecast horizon. Also for each flavor, select the model with the best MAPE value (step 6 of the Profiling Workflow). This aimed to validate the hypothesis H1.

3. Finally, after deploying the selected models at the SRO, the third experiment consisted of running the orchestrator and check if the proposed orchestration did obey the hypothesis H2 (Elasticity Workflow - Figure 10).

For experiment number 1, the first step is to get the Y file for each slice flavor. The first experiment to get the Y file for Flavor 1 is shown in Figure 16. The load applied to the service is shown in pink, while the 99^{th} Percentile of Reading Response Time measurements is shown in green. The tenant's service KPI axis is on the left, whereas the right axis shows how many users were sending requests.

In this experiment, a load floating from two to twelve clients in a time interval (sinusoidal load pattern) of twenty minutes was applied to the Cassandra service for three hours, generating nine repetitions of this pattern. As can be seen in the plot, there is

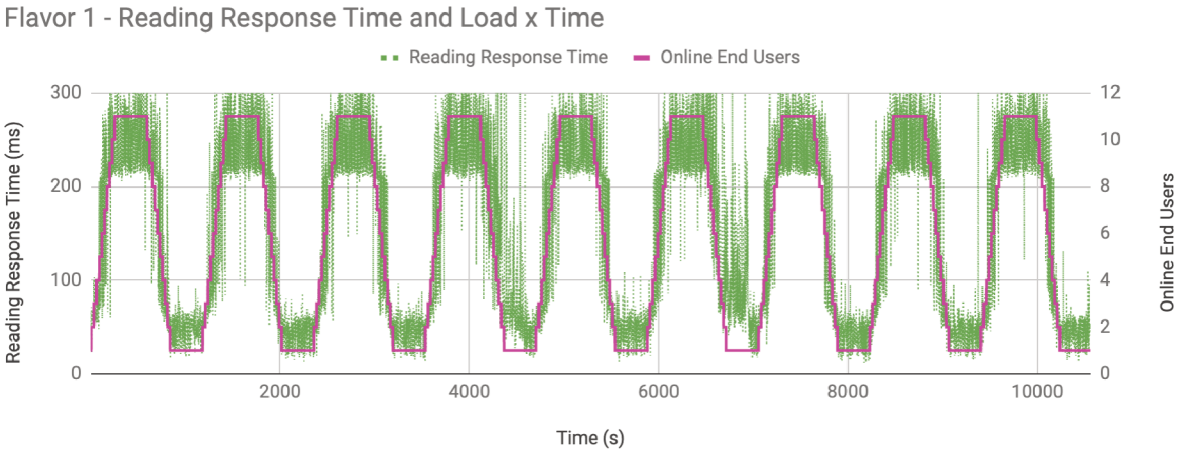


Figure 16 – Experiment executed for three hours to get the Flavor 1 Y file. The green curve represents the tenant's service KPI measurements while the pink represents the load applied to the Cassandra system.

a strong relationship between the tenant's service KPI and the load applied, which was the reason to choose it in this exploratory work. In addition, the similarity of the shapes during the load repetitions shows that this relation is reproducible and not a random event.

Next, the slice Flavor 2 was exposed to the same conditions, and the results are presented in Figure 17. It is possible to see that because more network resource was available (from 10 Mbit/s restriction to 30Mbit/s in the main network interface) the reading response time wave was lower, which means smaller reading response times. Again, the similarity between tenant's service KPI and load holds and is reproducible.

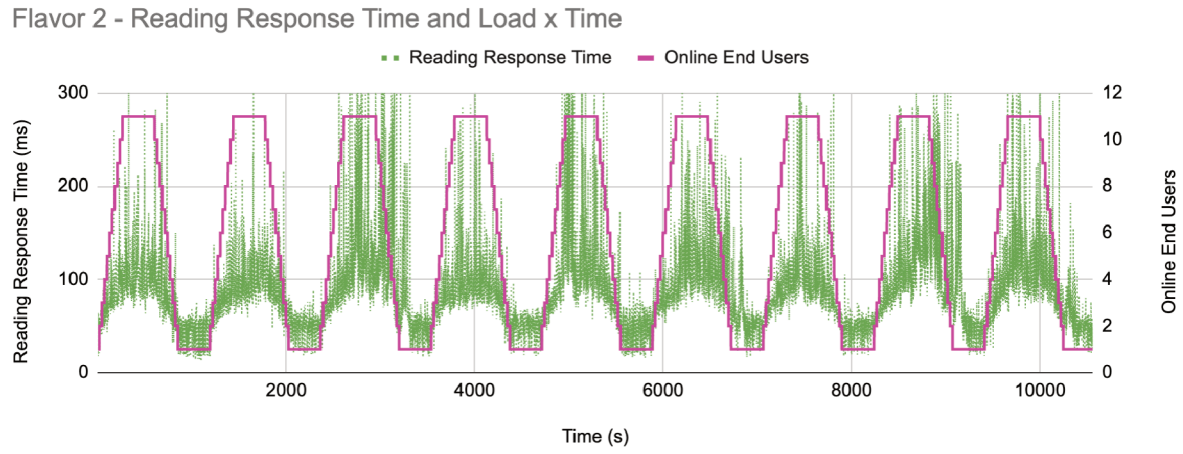


Figure 17 – Experiment executed during three hours to get the Flavor 2 Y file. The green curve represents the tenant's service KPI measurements while the pink represents the load applied to the Cassandra system.

During both experiments, Prometheus 1 was collecting the infrastructure monitoring data, to be extracted by IMA in the form of the X file. The X file from Flavor 1 was requested, as in steps 3-5 of the Profiling Workflow (Figure 9). With this and the client data (Y file) it was possible to train models to forecast the tenant's service KPI (Figure 9, step 6).

Aiming to understand the RNN behavior under different conditions, 8 models were trained for each flavor, with different window sizes and forecasting horizons. As it is a common choice to train AI algorithms, all models were trained with 70% of the experimental data, leaving 30% of unseen measurements for validation. The Flavor 1 models predictions are shown in Figure 18. In all figures, in the left column (a, c, e, and g), the models were trained using a window size of 240s. The right column (letters b, d, f, and h) shows the performance for a wider window, 480 seconds large. This kind of experiment was done to observe the effect of more past information on the RNN performance.

In each line of the Figures, a different forecasting horizon is used: 1s (a, b), 30s (c, d), 60s (e, f), and 120s (g, h). This variation aimed to explore how good the model is to look ahead in the future, providing information beforehand. In each plot, the real tenant's service KPI values are plotted in green and the forecasting in blue. Finally, in orange is the mean of the real values, which represents our naïve method, used as a baseline to compare with the RNN performance.

On each plot of Figure 18 we see a remarkable resemblance of shape between the real values and the forecasted ones. Both, the forecasted shape and real shape present two distinct and well-concentrated areas: a ceiling and a floor. Another peculiarity is that the fluctuations of the real values for the ceiling area are wider than for the floor area. Also, those shapes have a straight transition from the ceiling to the floor, and vice-versa. It is possible to see that with a larger range of future forecasting, the top of the shape in blue becomes disturbed (plots g and f). It may indicate that the larger horizon makes the predictions more vulnerable to data fluctuations. The loss of the shape while rising the curve for the plot h, may indicate that a larger window may increase the forecasting problems.

As the forecasting horizon gets larger, the plots tend to shift right, which starts to be visible in plots e and f and it is even more clear in plots g and h. This shift may indicate that the model becomes late as it has to predict ahead in the future.

To help us to see how good the forecasting is compared with the naïve model, the MAPE metric described in Equation 4 was used. As it represents an error metric, a smaller value means better model performance. For improving the visualization the MAPEs of each model was put together in the plot of Figure 19. The first values for each bar represent the RNN with 240s history (blue). This model had the best performance accordingly with the MAPE indicator for the majority forecasting horizons, except by the 1s horizon, where it has a quite small worse behavior comparing with the RNN with 480s history (red). The

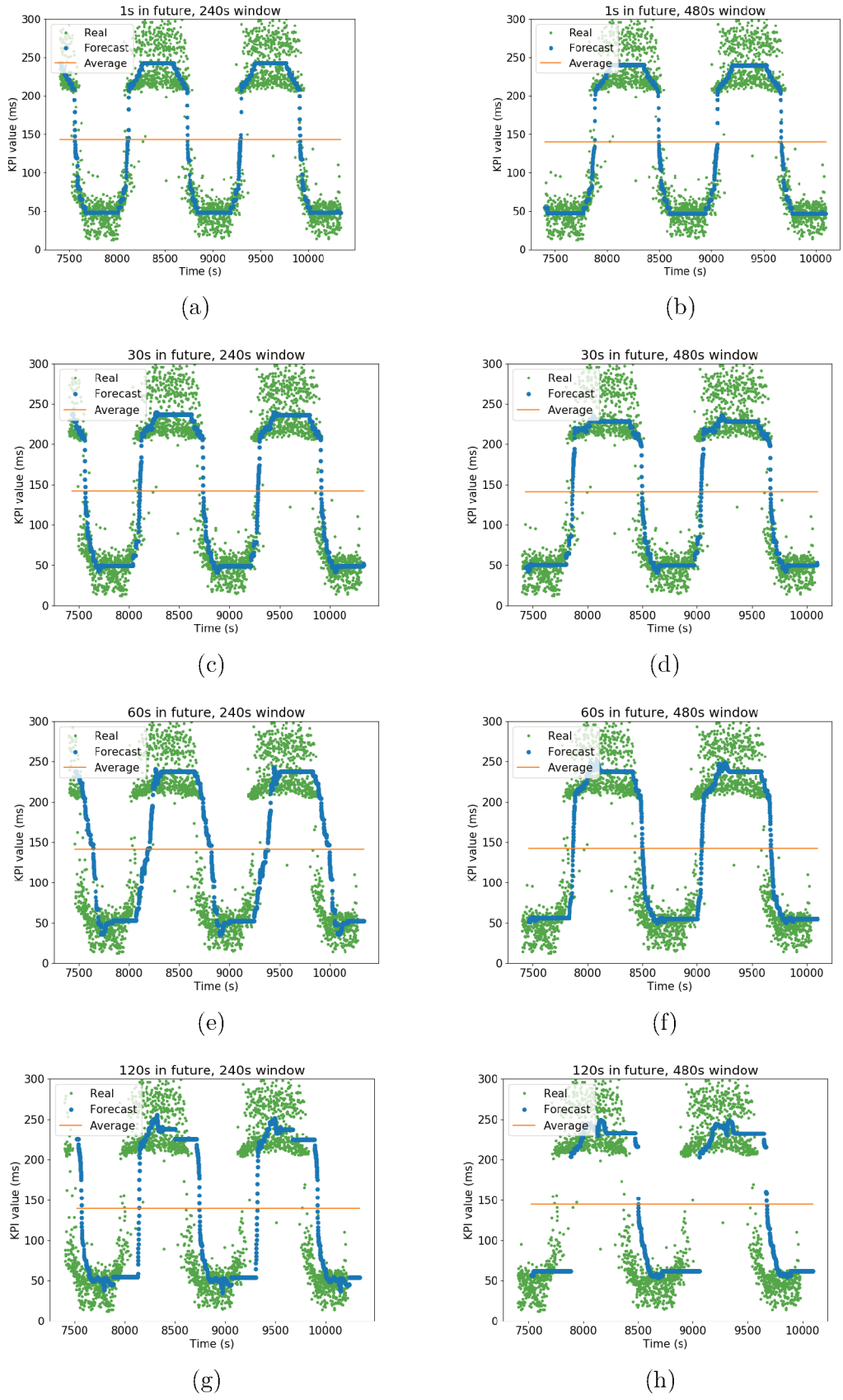


Figure 18 – Forecasting provided by models trained for Flavor 1. At left, 240s of window size and 480 seconds at right. The forecasting horizon varied for each line starting in 1, 30, 60, and finally 120 seconds.

RNN with a window size of 480s (red) was the second better model for almost all cases. It shows that a larger window did not represent, in general, improvements to the forecasting. The naïve methods were the two worse methods, about 5x worse than the RNNs for both windows sizes, which indicates that our Machine Learning (ML) implementation delivers a better tenant's service KPI prediction than the naïve method.

Although it can be seen as an expected result, we consider it an important result, since it shows that the ML solution considerably improves the predictions and, in such a case, it justifies the overall complexity associated with such a solution. We also would like to emphasize that our proposed method is agnostic to the service and the cloud-network slice infrastructure configuration, which also grants value to the proposal.

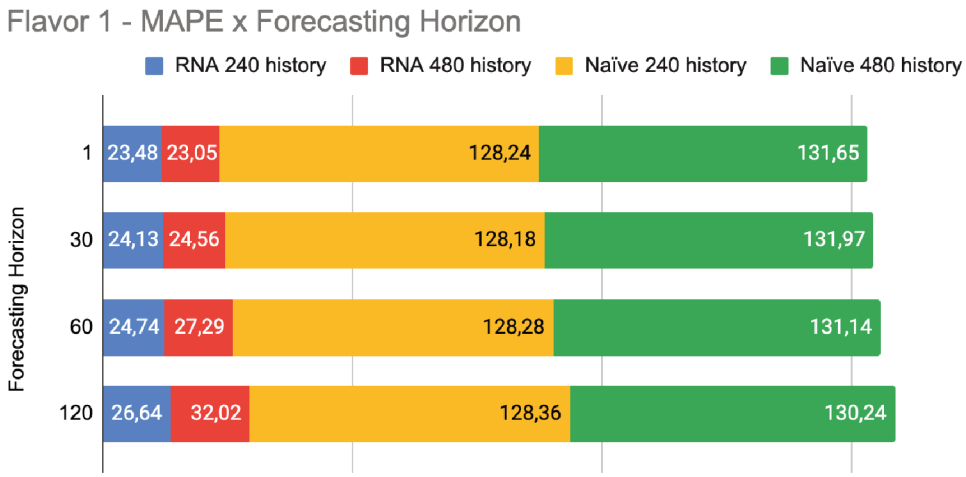


Figure 19 – MAPE for Flavor 1 models.

An analogous analysis was performed with the Flavor 2 models, which are shown in Figure 20. It is effortless to see the similarity with the blue and green curves, which demonstrates also a strong relationship between the models' predicted values and the real ones. A shift in the blue curves is also presented as the forecasting horizon increased, being more visible with a wider window size (h). The predicted values' curves weren't as smooth as the ones seen for Flavor 1, maybe because the real data set for Flavor 2 had more diffuse measurements.

One more time, the MAPE comparison is done, now for the Flavor 2 models, and it is shown in Figure 21. This time, the naïve methods delivered a smaller MAPE than for Flavor 1. However, they were about twice times worse than the RNN's predictions, on average. Comparing the RNN's, one more time a wider window size did not deliver the best performance in the majority of the cases. Essentially, Flavor 2 has more network bandwidth and the same number of clients were used in the load generator as for Flavor 1. So, actually, Flavor 2 was not pushed to the boundaries of its capacity and the dataset is not as rich as desirable, which is expected to be the reason why the naïve method closely approximates the predictions of the RNN. Also, as the majority of the real values

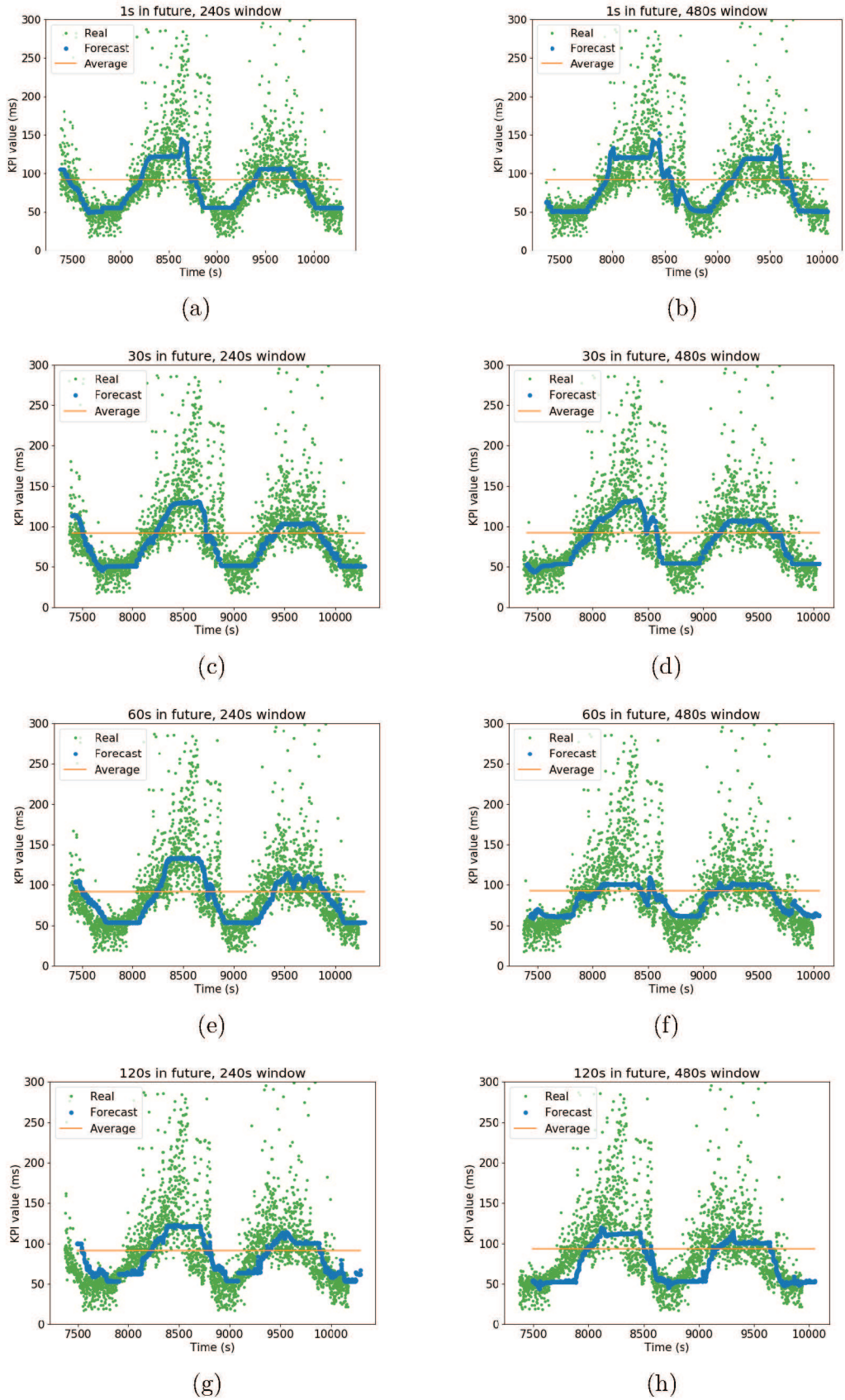


Figure 20 – Models trained for Flavor 2 under the same conditions of previously shown models.

for the Flavor's 2 tenant's service KPI had a smaller range, the naïve method, which is a simple mean, becomes closer to the real points, making the error metric smaller. It is important to clarify that our decision was to not use a different number of clients (loads) while profiling the different flavors because we did not intend to apply even larger flavors.

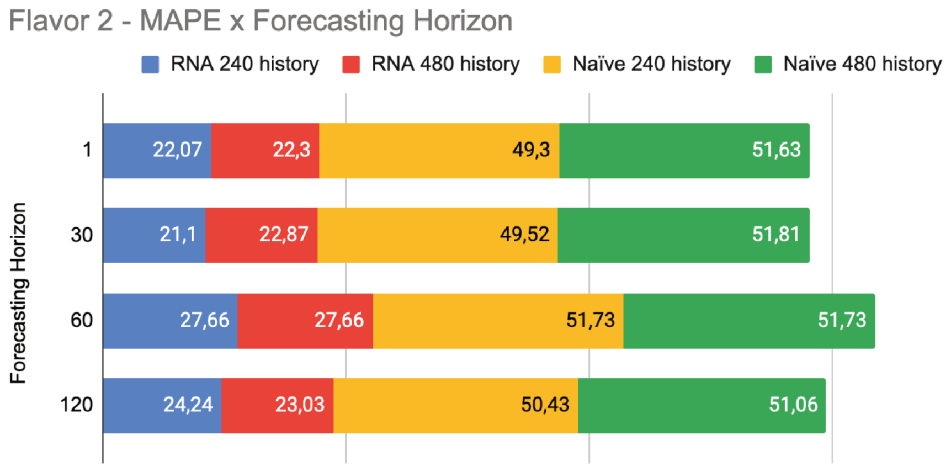


Figure 21 – MAPE for Flavor 2 models.

It was proven until here that "it is possible to forecast the selected tenant's service KPI using only the provider side metrics", the first hypothesis of Section 1.3. The final step to validate the proposed orchestration approach was to demonstrate that using two of the previously trained models (one for each flavor) could be used to provoke elasticity actions, in an adequate manner. As stated in Section 1.3, an adequate orchestration means to keep the resource usage as low as possible, and therefore the costs, but without breaches in the SLA. Another point to highlight is the restriction to use only the provider's data during the slice run time, step 3 of the Elasticity Workflow (Figure 10).

Therefore, two experiments were run, aiming to observe the orchestration behavior during the two possible orchestration actions: elasticity upgrade and downgrade. The forecasting models used for both flavors were the 30s ahead models, once the elasticity actions take about 2.39s to be performed in our infrastructure, and the 30s models showed a good performance, which was closer to the real curve and the smallest MAPE indicator. To demonstrate the elasticity upgrade case, an upper bound for the tenant's service KPI of 180ms of Reading Response Time was set. This way, when the model forecasted a value equals or greater than this value, the orchestrator should perform the elasticity upgrade. In order to test the upgrade, we started with Flavor 1, the smallest, making a request to the SRO orchestration endpoint.

Figure 22 shows what happened with the real and forecasted tenant's service KPI values (green and blue dots). As the load increased (stair shape in yellow), the predicted KPI started to grow as well. As it crossed the 180ms threshold (blue star) near to 200s, the elasticity upgrade action was triggered by the SRO, which put the slice in Flavor 2 set

up. After a few seconds, the real and predicted KPI was kept under the threshold, and the model's predictions continued to follow a pattern similar to the real KPI, with the 0.99 percentile of real values kept under 180ms. This means that if the client's SLO was to keep the 0.99 percentile of the Reading Response Time (tenant's service KPI) under 180ms, it would have been accomplished.

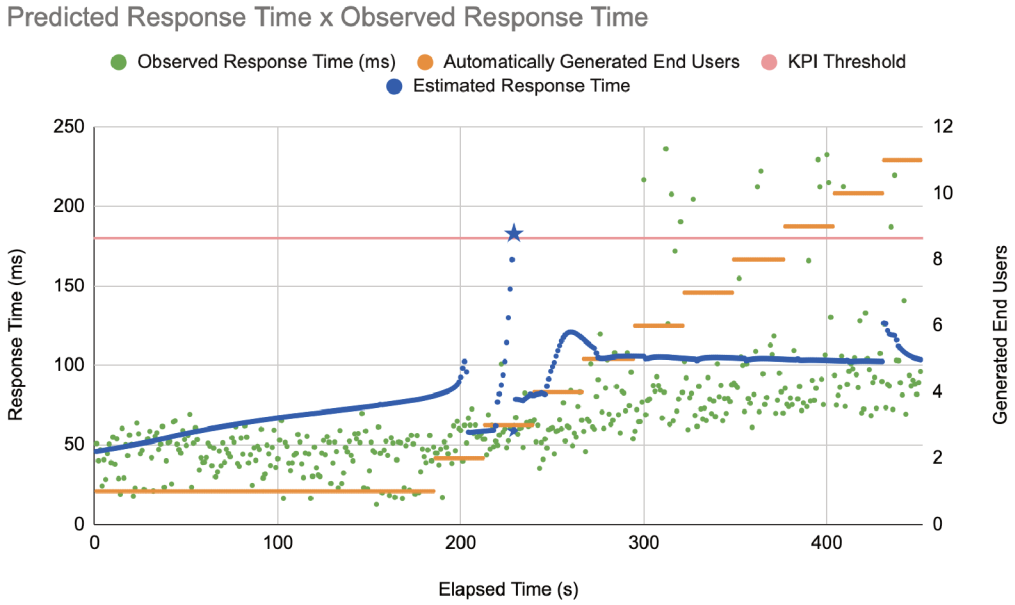


Figure 22 – Elasticity upgrade triggered by forecasted value crossing the upper threshold (blue star).

Advancing in the orchestration analysis, Figure 23 shows the elasticity downgrade point of view. The lower bound was set to 10ms. In this case, the lower bound means that the slice would be delivering a "too good" tenant's service KPI, and the used resources are being wasted. This way, the slice starts with the Flavor 2 configured. As the load started to decrease, the ML model predicted a tenant's service KPI value smaller than the threshold, which is shown as a star, near to 200 seconds. This prediction triggered an elasticity downgrade action, putting the slice back in Flavor 1. This time, the 0.99 percentile of the real values were kept above the tenant's service KPI threshold, which could represent another client's SLO. It is possible to see that the predicted values started to grow again after this action. As the KPI values were growing again, a situation very similar to the one showed in Figure 22 took place again, which formed an orchestration cycle. This cycle can be watched in real-time in a video available at NECOS (2020).

Lastly, the time cost of this orchestration approach can be considered in two parts, the time to perform the profiling phase and how long it takes to perform a prediction and to apply the orchestration actions. The profiling phase includes 6 hours to carry out the loading experiments of Figures 16 and 17, 10 hours of feature selection for the two flavors, and 20 minutes to generate each of the two 30s forecasting models. Such a process adds up to 16:40h. To perform a single prediction, the window needs to be full, which takes 4

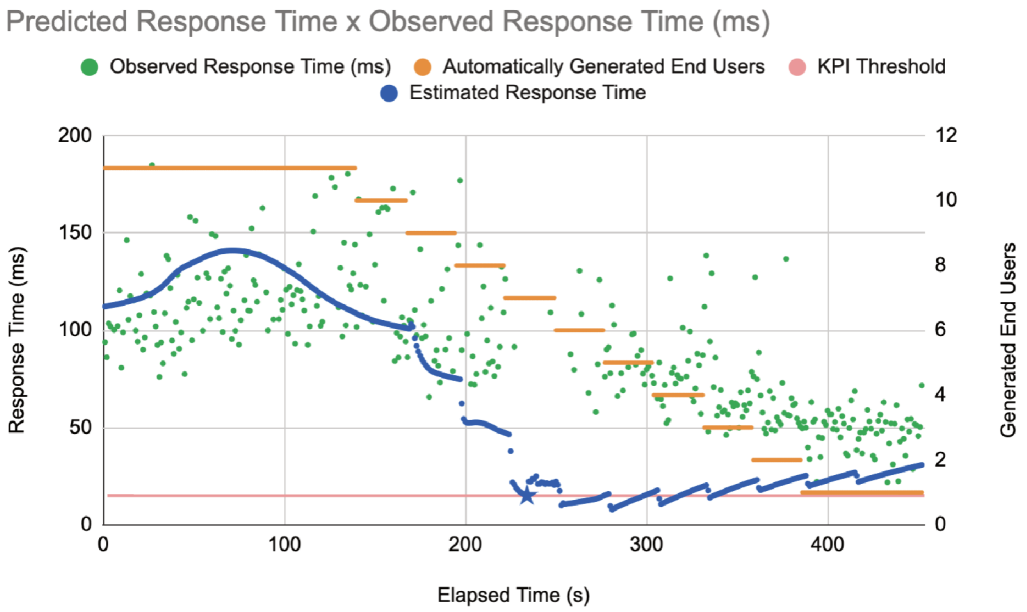


Figure 23 – Elasticity downgraded triggered when the prediction reaches the bottom threshold (blue star).

minutes for the first prediction. Later, it takes less than a second, on average, to perform forecasting. As stated before, to execute a scaling action it took on average 2.49s. This means that the offline preparation phase took about 16:40, however, during run time, the orchestration itself is quite lightweight taking about 4 seconds in the worse case (when a scaling action is needed) and less than one second in general cases (just performed a forecast).

This section concludes that the Hypotheses at Section 1.3 were valid, once the orchestration manages satisfactorily the infrastructure, considering pretty tight SLO's, as shown in Figures 22 and 23. Another very important aspect to highlight is that the elasticity actions were triggered before the real tenant's service KPI crossed the thresholds, proving to be a working proactive by prediction orchestration method.

A final remark about the results is that we proved that it is feasible to use infrastructure metrics to predict service-level KPIs for cloud-network slices. We used supervised learning techniques under the offline learning approach since this dissertation fits an exploratory phase in the development, in which phase the idea was to prove the solution as feasible. We consider that the entire process can benefit from on-line techniques and other ML methods, for example, reinforcement learning techniques, but they are left as future work. What we have achieved can be seen as a baseline implementation framework that can be really deployed and used for orchestration.

Chapter 5

Conclusion

As it was stated at the beginning of this document, at Section 1.2, the main goal of this project was to propose a slice orchestrator architecture, inside the context of the NECOS project. The main objective of this orchestration approach was to keep the SLA that was established between the NECOS tenant and its end users managing the cloud-network slice in a way that a defined SLO could be kept. This primary goal was broken into four steps which were shown in Section 1.2 and were concluded in the following way:

1. To propose an orchestration architecture. This goal was reached as shown by the multi-slice orchestration architecture designed and presented in Section 3.2.
2. To create an automatized and customized orchestration strategy for each slice. To do so, it was adopted an approach using the training of ML models for each slice configuration (flavors), which was detailed in Section 3.4.
3. To investigate forecasting approaches in orchestration, respecting the NECOS restriction to use only the provider's infrastructure metrics. This was achieved by exploring the RNN as described in Section 4.2.
4. To implement a prototype of the suggested orchestration architecture. This prototype was built and its behavior was portrayed in Section 4.2.

All these goals were used to prove that the hypotheses of Section 1.3 hold. The flavors forecasting models plots and metrics showed in the Section 4.2 convince that it was possible to use ML models to forecast tenant's service KPI values in the future using only infrastructure metrics (Hypotheses 1) and, finally, that these models could be used to stimulate an automatic orchestration of a slice.

5.1 Main Contributions

The biggest contribution of this dissertation is to prove that the provider's side metrics can give a good prediction in the future of the quality been delivered to the end user's

point of view. Also, this work provided an orchestrator prototype that is agnostic to service and slice configuration in an open-source form and also datasets to future works [1]. This way, other works can explore other prediction methods using the data sets, other infrastructure configurations, or even explore other services.

5.2 Future Work

During the development of this orchestration architecture, three points of attention came up. The first is that when a change of flavor happens, the previous window is filled with the last flavor's data, which can cause the prediction to lose some precision. Future work could explore more this situation, analyzing the impacts, and maybe proposing solutions. Secondly, the forecasting can be truly compromised by the latency/failure of the network or any other service involved in the metrics collection and publication. It would be necessary to explore techniques to mitigate the orchestration unavailability or even fallback options if it happens. The third is to automate the choice of the machine learning model, which could take into consideration the precision metric, the time to perform forecasting, the time to be trained, etc.

Besides these two subjects, other kinds of machine learning techniques could be analyzed to generate better models, or even to explore other RNN architectures and parameterization. More complex test cases involving more slice parts or other applications could also be explored, and new techniques of slice setting reconfiguration more complex than predefined flavors could be investigated as well.

5.3 Contributions in Bibliographic Production

Two articles were published at the SBRC 37th Brazilian Symposium on Computer Networks and Distributed Systems at the Workshop of Theory, Technologies, and Applications of Slicing for Infrastructure Softwarization (WSLICE), in 2019. The first paper was titled "Arcabouço para Orquestração Osmótica de Cloud Slices" (GUARDIEIRO et al., 2019) and the second "Arcabouço de um sistema inteligente de monitoramento para cloudslices" (MARQUES et al., 2019). Also, in the same event, one poster titled: "A. Towards osmotic orchestration of cloud slices" was presented at the Latin American Student Workshop on Data Communication Networks (GUARDIEIRO, 2019).

[1] https://gitlab.com/necos/demos/mlo

Bibliography

AMAZON WEB SERVICES. **Amazon Elastic Container Service**. 2018. <https://aws.amazon.com/pt/ecs/>. Online; accessed 12 dec. 2018.

APACHE KAFKA. **Apache Kafka a Distributed Streaming Platform - Introduction**. 2017. <https://kafka.apache.org/intro>. Online; accessed 12 jan. 2020.

CARNEVALE, L. et al. From the cloud to edge and iot: a smart orchestration architecture for enabling osmotic computing. In: IEEE. **2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)**. [S.l.], 2018. p. 419–424. <https://doi.org/10.1109/WAINA.2018.00122>.

CASALICCHIO, E. Container orchestration: A survey. In: **Systems Modeling: Methodologies and Tools**. [S.l.]: Springer, 2019. p. 221–235. <https://doi.org/10.1007/978-3-319-92378-9_14>.

CASALICCHIO, E.; PERCIBALLI, V. Auto-scaling of containers: The impact of relative and absolute metrics. In: IEEE. **Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on**. [S.l.], 2017. p. 207–214. <https://doi.org/10.1109/FAS-W.2017.149>.

CUNHA, I. R. d. **Construction of a tool to support prediction of Cassandra response times from infrastructure metrics**. 2019. Computer Science Bachelor Monograph. <https://repositorio.ufu.br/handle/123456789/26441>.

DOCKER INC. **Swarm mode overview**. 2018. <https://docs.docker.com/engine/swarm/>. Online; accessed 12 dec. 2018.

ENDLER, M. **Kafka-InfluxDB**. 2018. <https://github.com/mre/kafka-influxdb>. Online; accessed 12 jan. 2020.

FREITAS, L. A. et al. Slicing and allocation of transformable resources for the deployment of multiple virtualized infrastructure managers (vims). In: IEEE. **2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)**. [S.l.], 2018. p. 424–432. <https://doi.org/10.1109/NETSOFT.2018.8459990>.

FROST, J. **Standardization**. 2020. <https://statisticsbyjim.com/glossary/standardization/>. Online; accessed 23 nov. 2020.

GRAFANA LABS. **The open observability platform**. 2020. <https://grafana.com/>. Online; accessed 12 jan. 2020.

GUARDIEIRO, A. Towards osmotic orchestration of cloud slices. Poster at LANCOMM Student Workshop (Latin American Student Workshop on Data Communication Networks). 2019.

GUARDIEIRO, A. et al. Arcabouço para orquestração osmótica de cloud slices. In: SBC. **Workshop of Theory, Technologies, and Applications of Slicing for Infrastructure Softwarization (WSLICE), 2019 SBRC 37th Brazilian Symposium on Computer Networks and Distributed Systems**. [S.l.], 2019. p. 30–43. <https://doi.org/10.5753/wslice.2019.7720>.

HYNDMAN, R. J.; ATHANASOPOULOS, G. **Forecasting: principles and practice**. Melbourne, Australia: OTexts, 2018. <https://otexts.com/fpp2/>. Online; accessed 12 mar. 2019.

HYNDMAN, R. J.; KHANDAKAR, Y. et al. **Automatic time series for forecasting: the forecast package for R**. [S.l.]: Monash University, Department of Econometrics and Business Statistics, 2007. <https://doi.org/10.18637/jss.v027.i03>.

INFLUXDATA INC. **Real-time visibility into stacks, sensors and systems**. 2020. <https://www.influxdata.com/>. Online; accessed 12 jan. 2020.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, IEEE, n. 1, p. 41–50, 2003. <https://doi.org/10.1109/MC.2003.1160055>.

KUBERNETES. **Pod Overview**. 2018. <https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>. Online; accessed 21 dec. 2018.

KUBERNETES AUTHORS. **Production-Grade Container Orchestration**. 2018. <https://kubernetes.io/>. Online; accessed 12 dec. 2018.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, Nature Publishing Group, v. 521, n. 7553, p. 436, 2015. <https://doi.org/10.1038/nature14539>.

MARQUES, G. et al. Arcabouço de um sistema inteligente de monitoramento para cloud slices. In: SBC. **Workshop of Theory, Technologies, and Applications of Slicing for Infrastructure Softwarization (WSLICE), 2019 SBRC 37th Brazilian Symposium on Computer Networks and Distributed Systems**. [S.l.], 2019. p. 58–70. <https://doi.org/10.5753/wslice.2019.7722>.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, Belltown Media, v. 2014, n. 239, p. 2, 2014.

MESOSPHERE. **Apache Mesos**. 2018. <http://mesos.apache.org/>. Online; accessed 12 dec. 2018.

_____. **Marathon: A container orchestration platform for Mesos and DC/OS**. 2018. <https://mesosphere.github.io/marathon/>. Online; accessed at 12-12-2018.

NECOS. **D5.1: Architectural update, Monitoring and Control Policies Frameworks**. 2018. <http://www.maps.upc.edu/public/D5.1_final.pdf>. Online; accessed 12 mar. 2019.

_____. **Novel Enablers for Cloud Slicing**. 2018. <http://www.h2020-necos.eu/>. Online; accessed 12 dec. 2018.

_____. **D3.2: NECOS System Architecture and Platform Specification. V2**. 2019. <http://www.maps.upc.edu/public/necos_d3.2.v4.11_final_web.pdf>. Online; accessed 17 dec. 2019.

_____. **NECOS Final Review**. 2019. <http://www.h2020-necos.eu/ufrn-telecomday-2019-2-2/>. Online; accessed at 11-17-2020.

_____. **Machine Learning Based Orchestration of Slices**. 2020. <http://www.maps.upc.edu/public/MLO_demo_video_with_audio.mp4>. Online; accessed at 06-24-2020.

NETTO, M. A. et al. Evaluating auto-scaling strategies for cloud computing environments. In: IEEE. **Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium**. [S.l.], 2014. p. 187–196. <https://doi.org/10.1109/MASCOTS.2014.32>.

NICHOLSON, C. **A Beginner's Guide to LSTMs and Recurrent Neural Networks**. 2019. <https://pathmind.com/wiki/lstm>. Online; accessed 29 dec. 2019.

PASQUINI, R.; STADLER, R. Learning end-to-end application qos from openflow switch statistics. In: IEEE. **Network Softwarization (NetSoft), 2017 IEEE Conference**. [S.l.], 2017. p. 1–9. <https://doi.org/10.1109/NETSOFT.2017.8004198>.

QU, C.; CALHEIROS, R. N.; BUYYA, R. Auto-scaling web applications in clouds: A taxonomy and survey. **ACM Computing Surveys (CSUR)**, ACM, v. 51, n. 4, p. 73, 2018. <https://doi.org/10.1145/3148149>.

RANJAN, R. et al. Cloud resource orchestration programming: overview, issues, and directions. **IEEE Internet Computing**, IEEE, v. 19, n. 5, p. 46–56, 2015. <https://doi.org/10.1109/MIC.2015.20>.

SCIANCALEPORE, V.; CIRILLO, F.; COSTA-PEREZ, X. Slice as a service (slaas) optimal iot slice resources orchestration. In: IEEE. **GLOBECOM 2017-2017 IEEE Global Communications Conference**. [S.l.], 2017. p. 1–7. <https://doi.org/10.1109/GLOCOM.2017.8254529>.

SILVA, F. S. D. et al. Necos project: Towards lightweight slicing of cloud federated infrastructures. In: IEEE. **2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)**. [S.l.], 2018. p. 406–414. <https://doi.org/10.1109/NETSOFT.2018.8460008>.

TENSORFLOW. **Time series forecasting**. 2019. <https://www.tensorflow.org/tutorials/structured_data/time_series>. Online; accessed 26 jul. 2020.

THE APACHE SOFTWARE FOUNDATION. **Apache Cassandra**. 2016. <http://cassandra.apache.org/>. Online; accessed 12 jan. 2020.

THE LINUX FOUNDATION. **Prometheus**. 2020. <https://prometheus.io/>. Online; accessed 12 jan. 2020.

VAZQUEZ, C.; KRISHNAN, R.; JOHN, E. Time series forecasting of cloud data center workloads for dynamic resource provisioning. **JoWUA**, v. 6, n. 3, p. 87–110, 2015.

VELOSA, A. et al. Predicts 2015: The internet of things. **Gartner: Stamford, CT, USA**, 2014.

VEXXHOST. **Open source software for creating private and public clouds**. 2020. <https://www.openstack.org/>. Online; accessed 12 jan. 2020.

VILLARI, M. et al. Osmotic computing: A new paradigm for edge/cloud integration. **IEEE Cloud Computing**, IEEE, v. 3, n. 6, p. 76–83, 2016. <https://doi.org/10.1109/MCC.2016.124>.

WEES, A. Van der et al. Cloud service level agreement standardisation guidelines. **Cloud Select Industry Group-Subgroup on Service Level Agreement (C-SIGSLA), Tech. Rep**, p. 2, 2014.

ZANELA, E. H. et al. Proposta de vim on-demand para fatiamento de nuvem. In: SBC. **Workshop of Theory, Technologies, and Applications of Slicing for Infrastructure Softwarization (WSLICE), 2019 SBRC 37th Brazilian Symposium on Computer Networks and Distributed Systems**. [S.l.], 2019. p. 2–15. <https://doi.org/10.5753/wslice.2019.7718>.