UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL DE JESUS MARTINS

Automating Network Management for 5G Microservices-Based Network Slices

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Lisandro Zambenedetti Granville
Coadvisor: Prof. Dr. Juliano Araújo Wickboldt

Porto Alegre
April 2022

CIP — CATALOGING-IN-PUBLICATION

Martins, Rafael de Jesus

Automating Network Management for 5G Microservices-Based Network Slices / Rafael de Jesus Martins. – Porto Alegre: PPGC da UFRGS, 2022.

108 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Lisandro Zambenedetti Granville; Coadvisor: Juliano Araújo Wickboldt.

1. Network management. 2. Microservices. 3. NFV. 4. Network Slices. I. Granville, Lisandro Zambenedetti. II. Wickboldt, Juliano Araújo. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof^a^. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof^a^. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof^a^. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"The true delight is in the finding out
rather than in the knowing."

— ISAAC ASIMOV

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors, professor Lisandro Granville and professor Juliano Wickboldt, for all the teachings and patience they offered me not only during my master's, but throughout last years. I also would like to extend the praise to all the professors in the institute, who were (almost) always kind and insightful whenever a student would need it.

This work could also not be possible without the friendship from my labs colleagues, always open to discussions and helpful in times of need. They are too many to mention, but I know they know who they are. And I hope I had positively impacted you the same way you had impacted me.

Above all, I would especially like to thank my parents, Oscar Morency Otto Martins and Paula de Jesus Martins, and my *fidanzata* Caroline Grazioso (soon to be Martins), for always believing on me and being on my side throughout the years. My life is complete with you, and each day I can only feel more and more fulfilled for having you in my life.

ABSTRACT

Forthcoming 5G networks promise a myriad of new and improved applications such as autonomous driving and smart cities, which impose a drastic shift in how mobile telecommunications must operate. In order to comply with the new requirements, an umbrella of technologies must come together, and solutions based on Network Function Virtualization (NFV) and network slicing, for example, must be carried out. Regarding NFV in particular, the trend towards pulverizing the monolithic software in a microservices-based one carries network management challenges to operators. While traditionally a network function was virtualized through a single monolithic software, the microservices paradigm converts the same function in a number of smaller services that must cooperate to deliver the same functionalities. Thus, the management challenge in the deployment and integration of one or more network management software with the managed microservices is as important as it is delicate, as stringent requirements of 5G applications must be respected. This master's dissertation proposes SWEETEN, the aSsistant for netWork managEmEnT of microsErvices-based VNFs, as a solution for automating the deployment and transparently integrating network management into microservices-based network slices. By enriching their function stack with high-level annotation of the management features they desire, users can easily deploy an augmented stack with both network and management functions. SWEETEN considers the users specification when mapping a solution, which is done through the mapping of requested management features into tools and configurations. The system's usability is demonstrated through two case studies, where SWEETEN is shown to transparently provide monitoring and security solutions for complete network slices, enabling compliance with privacy requirements through minimal low-level interventions from the network slice tenant. The results show how SWEETEN integration of monitoring and security disciplines can assist users in guaranteeing the correct operation of their deployments regardless of the underlying software solutions used.

Keywords: Network management. microservices. NFV. Network Slices.

Automatizando Gerência de Rede em 5G para *Slices* de Rede Baseados em Microserviços

RESUMO

Os novos sistemas 5G prometem uma miríade de novas e melhoradas aplicações tais como carros autônomos e cidades inteligentes, o que impõe uma mudança drástica em como as redes de telecomunicação devem operar. Para atender com os novos requisitos, um guarda-chuva de tecnologias devem ser reunidas, e soluções baseadas em Virtualização de Funções de Rede (NFV, do inglês *Network Function Virtualization*) e *Slicing* de rede, por exemplo, precisam ser realizadas. A respeito especificamente de NFV, a tendência de pulverizar-se o *software* monolítico em um baseado em microserviços traz muitos desafios aos operadores. Enquanto tradicionalmente uma função de rede era virtualizada em um único *software* monolítico, o paradigma de microserviços converte a mesma função em vários serviços menores que precisam cooperar para entregarem as mesmas funcionalidades. Desta forma, o desafio de gerência em implantar-se e integrar-se um ou mais *software* de gerência de rede com o microserviço gerenciado é tão importante quanto é delicado, já que os rigorosos requisitos de aplicações 5G devem ser respeitados. Esta dissertação de mestrado propõe o SWEETEN, um assistente para gerenciamento de rede de VNFs baseadas em microserviços, como uma solução para automatizar a implantação e a integração transparente de soluções de gerência de rede em *slices* de rede baseados em microserviços. Através do enriquecimento de suas especificações de serviços com anotações de alto nível das funcionalidades de gerência requeridas, usuários podem facilmente implantar um *stack* aumentado com tanto as funções de rede quanto as de gerência. O SWEETEN considera a especificação do usuário quando mapeia uma solução, o que é feito através do mapeamento das funcionalidades requeridas em ferramentas e configurações. A usabilidade do sistema é demonstrada através de dois estudos de caso, onde o SWEETEN demonstra prover de forma transparente soluções de monitoramento e segurança para *slices* de rede completos, possibilitando a conformidade com requisitos de privacidade e com mínima intervenção de baixo nível do inquilino do *slice*. Os resultados demonstram como a integração das disciplinas de monitoramento e segurança pode ajudar usuários a garantirem a correta operação de suas implantações, independentemente da escolha de *software* subjacentes para a solução.

Palavras-chave: Gerência de Redes. Microserviços. *Slices* de Rede..

LIST OF FIGURES

Figure 3.1 SWEETEN architecture (MARTINS et al., 2020b)....................................21
Figure 3.2 Set of instantiable templates, from least to most intrusive. 26

Figure 4.1 User dashboard generated from the novice user's specification. 32
Figure 4.2 User dashboard generated from the experienced user's specification. 33

Figure 5.1 Communication flow and bandwidth requirements for radio functions. 36
Figure 5.2 Case Study for Intelligent Healthcare Application 37

Figure 6.1 Time taken to deploy up to 15 BSs in the first case study, with and without using SWEETEN. 42
Figure 6.2 Deployment time overhead for varying replicas count in the second case study. 43
Figure 6.3 CPU usage (in percent) for management containers for up to four concurrent (same VM) base stations in the first case study. 44
Figure 6.4 Computational overhead using different security options in the second case study. 45
Figure 6.5 Network overhead using different security options in the second case study.45
Figure 6.6 Excerpt from user dashboard enabling the requested monitoring features....47
Figure 6.7 Latency monitoring result for two BSs' RX to Softbit communication over a 30-second window (first case study). 48
Figure 6.8 Example dashboard for throughput monitoring of IoT devices, in the second case study. 48

LIST OF TABLES

Table 3.1 Network features and respective tools listings. 22

Table 5.1 Resulting distribution of BSs functions... 36

LIST OF ABBREVIATIONS AND ACRONYMS

5G	Fifth Generation Mobile Network
C-RAN	Cloud Radio Access Network
ETSI	European Telecommunications Standards Institute
HARQ	Hybrid Automatic Repeat reQuest
LTE	Long Term Evolution
MANO	ETSI NFV Management and Orchestration
NETconf	Network Configuration Protocol
NFV	Network Functions Virtualization
NLP	Natural Language Processing
SFC	Service Function Chaining
SNMP	Simple Network Management Protocol
SWEETEN	aSsistant for netWork managEmEnT of microsErvices-based VNFs
VM	Virtual Machine
VNF	Virtualized Network Function
VNFC	Virtualized Network Function Component

CONTENTS

1 INTRODUCTION....11
2 BACKGROUND AND RELATED WORK....16
2.1 Background....16
2.2 Related Work....18
3 SYSTEM DESIGN....21
3.1 General Aspects and User Input....22
3.2 User Images Pre-processing....23
3.3 Management Tools and Templates Mappings....24
3.4 Management Mappings through Instantiable Templates....25
4 PROTOTYPE IMPLEMENTATION....28
4.1 Implementation overview....28
4.2 Tagging....30
4.3 Solution Deployment and User Dashboard....32
5 CASE STUDIES....35
5.1 5G Radio Split for Dynamic C-RAN....35
5.2 Automated Network Management for Intelligent Healthcare....37
6 RESULTS AND DISCUSSION....40
6.1 Experimental Setup and Software....40
6.2 Expressiveness Gains....40
6.3 Deployment Time....41
6.4 Computational and Network Overhead....43
6.5 System's Output....46
7 CONCLUSION AND FUTURE WORK....49
REFERENCES....51
APPENDIX A — RESUMO EXPANDIDO....55
A.1 Contribuições da Dissertação....56
A.2 Principais Resultados Alcançados....57
APPENDIX B — PUBLISHED PAPER (AINA 2020)....58
APPENDIX C — PUBLISHED PAPER (CNSM 2020)....71
APPENDIX D — SUBMITTED PAPER (COMNET)....81

1 INTRODUCTION

Conventional computer networks are relatively static in terms of physical structure with respect to network topology, functionality, and protocols (HAKIRI et al., 2014). These conventional networks extensively employ physical middleboxes (BRIM; CARPENTER, 2002) to perform network functions, such as routing, firewalling, and load balancing. The administration of conventional networks typically relies on traditional network management solutions based, for instance, on SNMP (Simple Network Management Protocol) (FEDOR et al., 1990), NETconf (Network Configuration Protocol) (FEDOR et al., 1990), and Netflow (CLAISE, 2004). These management solutions implement distributed architectures, following the static structure of the underlying managed network. The addition of new middleboxes in a network naturally increases the overall number of devices to be managed, and because middleboxes are implemented with proprietary hardware, the inclusion of new network functions, including their proper configuration and maintenance, often requires a manual and costly effort from the network operator.

To address the effect of intrinsically inflexible middleboxes (leading to additional costs), Network Functions Virtualization (NFV), which is an emerging technology, relies on virtualization to implement and deploy Virtual Network Functions (VNFs) (ETSI, 2013). By decoupling the proprietary hardware from the associated software, NFV enables functions to be run on top of commodity hardware, reducing operational costs and increasing network dynamicity and scalability. Because of its advantages, NFV has quickly became a staple paradigm in the networking field (ZHANG et al., 2019). NFV is often realized with Virtual Machines (VMs), or, recently, with lightweight virtualization technologies based on containers (CZIVA; PEZAROS, 2017). When compared to VMs, VNFs materialized through container virtualization can be deployed faster and more efficiently (FELTER et al., 2015). Containers can create and replicate customized environments, offering isolation for running applications. Because of its enhanced performance, Docker (MERKEL, 2014) has been largely adopted in industry and academia. As network services provisioned by VNFs are vital for the networks' health, properly managing and monitoring Virtualized Network Functions (VNFs) become a mandatory concern in assuring its proper operation.

In another field, *i.e.*, cloud computing, the microservices paradigm advocates towards breaking down applications and end-services in small self-contained functional modules as a solution to the problems faced by monolithic software (DRAGONI et al.,

2017; MARTINS et al., 2020c). While monolithic software is developed and deployed as a single atomic service, microservice-based solutions provide the same high-level service through the cooperation of multiple independent modules. In this case, each module should provide a specific function and runs in a virtual host, and the communication between modules is used to combine the necessary functions and deliver the service correctly. Some possible benefits enjoyed by applications designed with the microservice paradigm includes fine-grained scaling of a service, since only the overloaded modules need to be scaled up (*e.g.*, increasing their computational resources) or out (*e.g.*, replicating the modules over additional hosts), and continuous development and integration, since only the updated modules must be upgraded.

Microservices-based NFV and other novel concepts can emerge as an important mean for new networks to reach their envisioned potential. In this sense, 5G mobile networks exemplify such a case. Unlike previous mobile generations, 5G promises not only to improve data transmission rates but also to enable the coexistence of a myriad of applications with distinct requirements. To achieve that, network slicing of the underlying infrastructure enables several tenants to seamlessly share resources and achieve distinct (potentially conflicting) objectives. The overall system's health relies on the harmonic coexistence of tenants sharing the same infrastructure (SLAMNIK-KRIJEšTORAC et al., 2020). NFV can assist in the provisioning of slices for tenants, but each slice must be individually managed and monitored to guarantee that the tenant's application requirements are being met. Cloud-based monitoring tools often require extensive privileges on the underlying infrastructure to work, which is not wanted or even feasible from the viewpoint of the infrastructure provider. Being designed for higher-level services, these solutions often introduce network overhead because of the additional hop per microservice in a flow, and can hinder their adoption depending on applications requirements[1]. Extreme cases such as edge applications could even suffer from the computational overhead from the additional containers deployed. Moreover, infrastructure owners and tenants may require not only monitoring, but also other network management features (*e.g.*, security and configuration) transparently.

Considering that 5G networks are envisioned to support upcoming mission-critical applications, security becomes a primary concern. Targets can range from governments and industries to ordinary citizens. For example, eavesdropping e-health devices can leak confidential information regarding their users, and thus impose an important security

[1]<https://medium.com/@pklinker/performance-impacts-of-an-istio-service-mesh-63957a0000b>

requirement for the setup (ZHANG; WANG; ZHOU, 2019). Additionally, battery limitations from these devices can result in minimal computational overhead being acceptable for management solutions. A different application with similar security challenges that runs in the cloud, conversely, could make use of more robust management artifacts that would incur in greater overhead overall. Since applications and resulting requirements can vary significantly, and because there is an increasing number of management tools offered for various contexts, correctly choosing and configuring a set of tools for each scenario becomes a challenging task even for experts.

This master dissertation proposes SWEETEN (a**S**sistant for net**W**ork manag**E**m**E**n**T** of micros**E**rvices-based V**N**Fs), a system designed to assist 5G service providers and tenants with configuration and deployment of network management tools along a network slice. By adding high-level management features annotations to their original services stack, SWEETEN can map the necessary tools and configuration to realize the desired features with no hassle for the operator. Available features include monitoring, managing, and securing one or multiple microservices. Because the system is targeted towards VNF management, it is designed to incur in the least overhead possible regarding network and computational resources. Moreover, Natural Language Processing (NLP) is employed to extract meaningful tags from users deployment descriptions. The generated tags are then used to provide tailored solutions for each deployment, so that more robust solutions can be produced for more resourceful deployments, while solutions that prioritize low overhead can be produced for resource-constrained deployments. Additionally, when deemed necessary, operators can specify as many configuration parameters as needed and the system will process them to deliver a solution as aligned as possible with the informed options.

A prototype implementation of SWEETEN is also presented, which is evaluated in two separate case studies. The first case study revolves around a dynamic cloud Radio Access Network (C-RAN) scenario. In this case study, LTE radio functions are split in five containers, as described by Wubben *et al.* (WUBBEN et al., 2014). The prototype is used to manage each function and monitor them assuring the radio requirements are being met. Since the stack for the virtualized radio functions does not need to be altered prior to being fed to the system, continuous development and integration of the virtualized functions is not an impediment for our system, as the updated functions and their respective network management tools are updated transparently.

The second case study focuses on a complete network slice for an intelligent healthcare service. In this service, patients can be monitored by a number of resource-constrained Internet of Things (IoT) health monitors and other resourceful devices in real-time, enhancing quality of human life through the automated execution of mundane tasks (WANG et al., 2018). To achieve that, data collected by said devices is sent to a deep learning module hosted in the cloud, which processes the data from a patient and triggers alarms when events happen. Due to the sensitive nature of the traffic exchanges by the healthcare devices, security in the terms of privacy is a foremost concern for all communication. It should be noted that, since these devices can be resource-constrained, solutions should balance the defensive mechanism effectiveness and the overhead they produce. Moreover, IoT devices in this use case utilize Narrowband IoT (NB-IoT) for their radio-access technology as it offers improved coverage and efficiency in terms of cost and power consumption (XU et al., 2018). Cloud radio access network (C-RAN) is used to deliver connectivity to the application's devices, imposing stringent latency and data rate requirements that must be met throughout the deployment life-cycle and thus implying the need for careful monitoring.

SWEETEN is evaluated through the prototype, and results for both case studies indicate that acceptable overhead is added to the deployment time of the complete solution for different management disciplines. Since the deployment overhead is a one-time cost for the operator, we argue that the observed values are acceptable for the benefits offered by the system. The results also show that the inclusion of monitoring features incur in negligible computational and network overhead throughout the remainder of the lifecycle, while security features incur in greater network overhead. Since network and computational overhead from monitoring solutions were negligible throughout the services lifecycle, including when it is under heavy-stress, operators can quickly diagnose malfunctions and bottlenecks for under-performing services, making it a valuable ally in assuring that services requirements are being met. Notwithstanding, the overhead is much more due to the included management entities themselves and not due to SWEETEN usage, and would thus also be present if a similar management solution were to be manually included by the user, therefore presenting a net gain for the user.

The remainder of the dissertation is organized as follows. In Chapter II, the background and related work are discussed. Chapter III introduces SWEETEN's architecture, discussing its components and main features. Then, in Chapter IV, a prototype implementation for SWEETEN is detailed. Chapter V then presents two case studies: the first one

of a 5G application scenario featuring a microservice-oriented software radio design split into five containers; the second one of a network slice for a intelligent healthcare service. The experiments performed and obtained results when evaluating SWEETEN prototype are discussed in Chapter VI. Finally, Chapter VII presents the concluding remarks and perspectives of future work.

2 BACKGROUND AND RELATED WORK

This chapter presents the related work on the main investigated subjects. First, a background on virtualization technologies, NFV and other important concepts in this study are provided in Section 2.1. The differences and the importance of this thesis proposal when compared to the related work are underlined in Section 2.2.

2.1 Background

Monolithic software can be defined as a software composed by modules that cannot be executed independently (FRANCESCO; LAGO; MALAVOLTA, 2018). Software design has generally followed a monolithic paradigm in which an indivisible software is responsible for realizing a complex service in an integrated manner. Monolithic software still can and should be designed through a composition of specialized modules. However, the different modules in a software following the monolithic paradigm still rely on resource sharing (*e.g.*, memory, CPU) for running in the same machine, which tightly ties all the components as one atomic application. While the monolithic architecture is viable for many applications, recent off-premise and distributed computing offered by cloud services impose the need for a more flexible paradigm in software design.

In the microservices paradigm, systems are designed through independent components called microservices, which provide a system with cohesive and well-defined functionalities (DRAGONI et al., 2017). Context sharing between microservices is done through network messaging, allowing microservices to be deployed along a distributed infrastructure, as well as completely decoupling implementation details and choices (*e.g.,* programming languages) between modules. Microservices introduce many benefits regarding continuous integration and delivery, for example, as updates for individual microservices may be gradually rolled out, and fine-grained scaling of a service, since only the overloaded modules need to be scaled up or out. However, the design also imposes new management challenges in relation to team organization, development practices and infrastructure (MAYER; WEINREICH, 2017), and therefore the correct operation of each microservice must be asserted, and must be that of the composed software as a whole.

Designing and developing software through the microservice paradigm can quickly become hard to manage as complex connection schemes are required among hundreds of microservices. Service meshes recently emerged as a solution for that through the auto-

matic management for microservices connections, as reviewed in the study by Li *et al.* (LI et al., 2019). Among their benefits, service meshes can provide service discovery for the microservices and load balancing among different containers (even using different software versions). On the implementation side, these solutions usually employ an array of lightweight network proxies, which are deployed alongside the application containers and can provide an interface for all incoming and outgoing connections. Some specific scenarios that are much relevant to 5G, such as multi-tenancy, can however present specific challenges that were not part of service meshes design. We argue that the proper management for VNFs and network slices with various requirements, such as the minimal computational and network overhead for IoT applications, must be featured in the management design of a high-level manager. This way, the appropriate management services and configuration can be provided based on the user specification and requirements.

A microservice needs a virtual host to run on, which is typically realized through container virtualization. Containers offer a lighter alternative to Virtual Machines (VMs), since they can run directly on the host system without requiring virtualization layers for an Operating System (OS) that introduces overhead in the process (MARTINS et al., 2020a). In this case, the host system's kernel offers resource isolation features that restrain each container to their own environment. On Linux, these features are mostly achieved through *cgroups* and *namespaces* features. Container orchestrators can be used to deploy and manage complex applications (*e.g.*, composed of multiple containers, deployed over multiple hosts). Docker is an example of a platform for lightweight container virtualization, and studies have shown that running multiple microservices in containers is a viable deployment option, with performance comparable to baseline (JHA et al., 2018). Regarding container orchestration, Kubernetes currently stands out as one of the most widely used platform (BERNSTEIN, 2014). Management tools, such as Prometheus (PROMETHEUS, 2017) and Dynatrace[1], have emerged in this context, offering monitoring solutions to the cloud environment and applications following the microservices paradigm. These tools allow monitoring of cloud systems in varying scales, from a single module to an inter-cloud distributed application. Particularly large applications can leverage service mesh solutions to manage connectivity (and the management concerns that comes with it) among the microservices it comprises (LI et al., 2019). Moreover, machine learning-based scheduling strategies for microservices architecture can leverage monitoring infor-

[1] <https://www.dynatrace.com/>

mation and quickly scale an overworked service, significantly saving time in comparison to traditional algorithms (JINGZE; MINGCHANG; YANG, 2019).

NFV, as defined by ETSI in the Management and Orchestration (MANO) specification (ETSI, 2014), can potentially benefit from the adoption of the microservices paradigm. Modularizing the VNFs forming a Service Function Chaining (SFC) can offer similar benefits to those enjoyed by higher-level distributed cloud applications (CHOWDHURY et al., 2019). Moreover, ETSI's NFV specifications also define that sub-sets of VNF's functionality are implemented by atomic VNF components (VNFC), which themselves can also be designed following the same principles, further benefiting the compound VNFs. However, monitoring and management solutions tailored for cloud applications may not be directly applied to NFV scenarios due to their specificities.

2.2 Related Work

Ciuffoletti (CIUFFOLETTI, 2015) investigates the specification and automation of monitoring infrastructures in a container-based distributed system. The author employs an architecture for monitoring that is comprised of two entities: a sensor, that produces and delivers measurements, and a collector, that specializes the management of those measurements. A simple model that interfaces the user and the container management system is defined, and a prototype implementation that showcases the applicability of the proposal is provided. The work thus focuses solely on the monitoring aspect, while our proposal also covers other management disciplines (*e.g.*, security). Additionally, Ciuffoletti's work considers that the user application and the sensor for the monitoring system run in the same container, which violates the microservices paradigm and can hinder the module development and deployment. Our solution, instead, always considers the application and management microservices as separated containers, even when context sharing is needed, and thus does not breach microservices standards unnecessarily.

The work of Jaramillo *et al.* (JARAMILLO; NGUYEN; SMART, 2016) discusses how Docker can effectively leverage the microservices paradigm through a case study based on a real working model. The authors pose a list with six challenges faced when building a microservice architecture and that contrasts with the multiple advantages offered by microservices design. Specifically, the work highlights the necessity of improvements regarding scalability, automation, and observability. SWEETEN is designed with these challenges in mind, providing automated observability (*i.e.*, a way to visual-

ize health status of microservices to quickly locate and respond to occurring problems) and scalability (*e.g.*, dynamic configuration for multiple microservices), features meeting operators needs.

Li *et al.* (LI et al., 2019) reviewed the state-of-the-art and the challenges for service meshes. Service meshes are emerging solutions that create a dedicated infrastructure layer for handling communication between microservices. Service meshes can offer multiple features such as service discovery, load balancing, and access control. Implementations for service meshes typically rely on deploying an array of network proxies alongside primary containers, intercepting all its connections to provide the features. As pointed out by the authors, edge computing environments and 5G scenarios (*e.g.*, multi-tenancy) incur in specificities that service meshes are not designed to cope with. SWEETEN is designed to work with VNFs with different requirements (*e.g.*, minimal overhead for edge deployments), and management applications (and thus, overhead) are chosen and configured according to high-level user requests and other deployment specifics.

Franco *et al.* (FRANCO; RODRIGUES; STILLER, 2019) introduced a support tool for cybersecurity that focuses on the recommendation of protection services. The authors argue that although a vast number of protections services are offered to network operators and users, the choice for one or more is not trivial for neither. Like SWEETEN, the proposed system can operate with different demands from the user, and recommend protection tools (in their case) for different scenarios. Notably, the proposal is limited in scope to the security discipline, while SWEETEN is designed to consider other network management disciplines.

Chowdhury *et al.* (CHOWDHURY et al., 2019) highlighted the importance for the NFV ecosystem to have VNFs designed through a microservice architecture. In Service Function Chains (SFCs), for example, having monolithic VNFs incurs in unnecessary processing overhead from redundant functionalities. Instead, the redesign of these functions through microservices enable fine-grained resource allocation and independently scalable components, as elements for the orchestration of VNFs in an SFC become also present for the VNF-Components (VNF-C) for any VNF. Among the research challenges documented in the literature, the adequate monitoring of these functions is underlined, as well as questions pertaining performance profiling and overhead trade-offs, all occurring topics in our present research.

Slamnik-Kriještorac *et al.* (SLAMNIK-KRIJEšTORAC et al., 2020) presented an extensive survey on the distributed and heterogeneous resource sharing that is taking place

in 5G networks. The sharing model for 5G and other networks proposed by the authors is classified in three distinct models: technical, business, and geographic. Specifically, the technical model is structured in three layers: infrastructure, orchestration, and service. Although some management concerns for network slices are discussed, they primarily focus on the infrastructure layer, while the examples for service (*e.g.*, Healthcare) and orchestration (*e.g.*, Kubernetes) are left for each layer and case to solve individually. Our study, in contrast, proposes that the network management should consider all layers jointly, and also that this management is a complex task that operators should be assisted with when deploying new network slices.

Kist *et al.* (KIST et al., 2020) proposed a virtualization scheme that allows technologies and instances for different Radio Access Networks (RANs) to be provided as services for network slice tenants. The proposal offers programmability and adaptability for service providers while maintaining isolation between tenants and their slices. An experimental scenario evaluated by the authors comprised of LTE and NarrowBand-IoT (NB-IoT) clients showcases how the proposed system allows the provisioning and management for virtual RANs (vRANs) for providers' network slices. The management aspect is however limited to the RAN infrastructure, and any additional required management aspects (*e.g.*, regarding the application, or other VNFs included in the slice) are left to be determined and deployed by the slice owner.

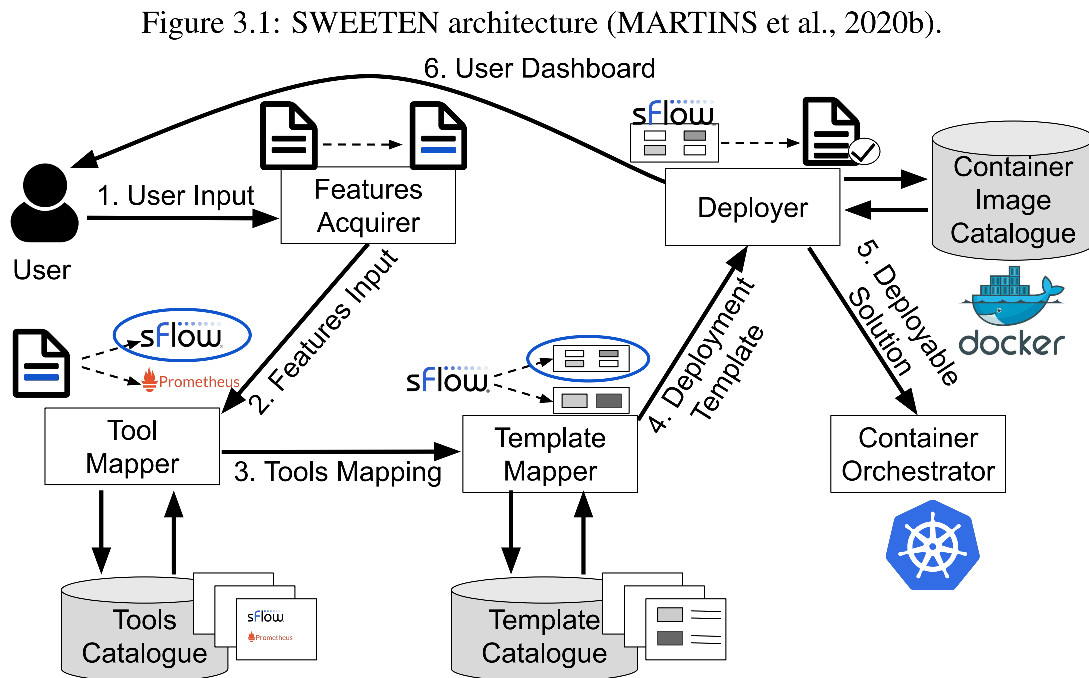
Coelho *et al.* (COELHO et al., 2020) looked into formally defining the network slice designing problem, proposing a framework that considers nested slices and network functions decomposed in smaller services and models the relationship between radio splitting, control and data planes isolation, and core network function placement. Leveraging the reusability of smaller network function services and network slices subnets, a variety of sharing policies that range from total isolation to flat sharing can be used to realize 5G services of any class, and fulfilling the stringent requirements each of them impose. The study therefore focuses on producing a network slice, including necessary network functions, their split and placement, to deliver the demands posed by a number of services. The slice management itself, including the appropriate monitoring of the deployed network functions, is not covered by the slice design and thus is left for the operator to manually determine, configure and deploy the appropriate solutions.

3 SYSTEM DESIGN

Because NFV is critical for upcoming networks, and since the paradigm shift from monolithic VNFs to microservices-based ones is imminent (CHOWDHURY et al., 2019), the burden has increased for network operators. The applications that run on top of a VNF or an SFC can pose stringent requirements for the functions, and the underlying infrastructure can also determine how functions should be managed. The proposed system should reflect the specificities of each scenario in order to properly select and configure the necessary management tools.

This chapter discusses SWEETEN's architecture and design choices. An overview of SWEETEN is presented in Figure 3.1. The remainder of this chapter presents the system progressively. General aspects and user input are discussed in Section 3.1. SWEETEN pre-processing is discussed in Section 3.2. The system's mapping of management tools and configurations is explained in Section 3.3. Finally, Section 3.4 discusses instantiable templates, an important part for the system as they determine how the management tools should be deployed by SWEETEN.

Figure 3.1: SWEETEN architecture (MARTINS et al., 2020b).



Source: Author

3.1 General Aspects and User Input

SWEETEN can be viewed as an automated assistant intended to help tenants of 5G slices to meet management requirements for their slice components. In particular, when these components are designed following the microservices architecture, network management must be thought of while respecting the modularity and isolation envisioned in this architecture. Moreover, a slice can span over thousands of microservices (JAMSHIDI et al., 2018), which makes automated management not only a benefit but a necessity.

The user input in the designed system consists of a specification for user containers, which is augmented by the user to contain requests for management features. These management features can be of multiple disciplines, namely security, monitoring, and administration. Some examples for each discipline are found in Table 3.1.

The existence of multiple tools when mapping a single feature is resolved by the system based on additional user input and deployment information. A single tool can also realize multiple features (*e.g.*, SNMP, for monitoring as well as for administration), which SWEETEN takes into account when selecting the appropriate tools for a deployment. Support for new features (and through different tools) can be added to the system by an expert. The user can adopt varying specification levels when requiring the management features for each service (*e.g.*, choosing to only monitor TCP connections on certain ports, or determining what tool should be used). The system interprets the requested features and maps the appropriate tools and configurations to fulfil all requests.

Table 3.1: Network features and respective tools listings.

	Monitoring	**Security**	**Administration**
Flows	sFlow, NetFlow, Prometheus	Snort (for IDS), OSSEC	-
Traffic	Prometheus, iPerf, SNMP	iptables, nftables	Linux tc
Latency	SmokePing, OWAMP, TWAMP	-	Linux tc
Device	Kubelet (Kubernetes native)	syslog, antivirus utilities	NETCONF, SNMP

With respect to the classification for network management features, three disciplines are considered, as presented in Table 3.1. The *monitoring* discipline encompasses all measurements that can be done to assert a network and its components are behaving as expected (LEE; LEVANTI; KIM, 2014). These measurements can be either passive (*e.g.*,

observing flows in a given interface) or active (*e.g.*, probing a link to check latency and throughput available). The *security* discipline involves all sensitive aspects in a network, including privacy and resilience requirements and the means to guarantee them at a certain level (STALLINGS, 2006). The *administration* discipline is comprised of management tasks and applications that actively alter the network behaviour for one or more device. For example, Netconf protocol can be utilized to reconfigure switches and routers in a network, altering its behavior dynamically (ENNS et al., 2011). Currently supported features are latency and flows, for monitoring; traffic, for security; and device, for administration.

3.2 User Images Pre-processing

The user input is received by SWEETEN through the Features Acquirer module. This module is responsible for retrieving information on all microservices defined in the user specification as well as the management features requested for each microservice. In addition to the nature of the management desired, requirements for the produced solution can be derived from requirements tags. To achieve that, each microservice with a management annotation undergoes three steps:

1. Management feature retrieval, where the requested features and options are extracted from the user specification;

2. Description enrichment, where the information about the user service is augmented;

3. Service tagging, where tags that better describe the service requirements are appended to the specification.

Each requested feature obviously can be realized by a number of tools. In order to allow the best possible match for tools selection and configuration in a later stage within SWEETEN, user's microservices undergoes a tagging process composed by the description enrichment and service tagging processes previously mentioned. For each component in the user input for which a feature is requested, SWEETEN appends tags that can provide some insight about the type of service and its requirements. Tags are later used to differentiate the tools and configuration choices for monitoring a cloud-hosted service from an IoT one; for example, while the former can leverage network and processing resources to employ a robust solution, the later must realize the management necessities with minimal overhead.

To alleviate the burden for the user, SWEETEN can automatically derive tags from the user specification alone without any additional user input. To achieve that, the description for each container that composes a service is processed in the service tagging step. This description is rarely present and descriptive for most containers, so the description is enriched before tags are derived through Natural Language Processing (NLP), as explained more in-depth in the following chapter. NLP is an area of computer science that employs algorithms for learning, understanding, and producing of human language content (HIRSCHBERG; MANNING, 2015). NLP has lately been a tool in various areas with promising results, for example, by providing enterprises with network security insights and suggesting solutions when paired with a neural network model (FRANCO et al., 2020).

3.3 Management Tools and Templates Mappings

Management features required by the users must be realized by a set of management tools. The Tool Mapper module thus is the first one to make selections based on the Features Acquirer output. For each feature required by the user, this module maps to one or more tools that are capable of realizing such features. The listing for these mappings is provided through a Tools Catalogue, which has been already pre-populated by an expert. In the event that more than one tool fits a certain request, tags are considered so that the best fit can be provided. The algorithm for matching the tags to the available tools is a greedy one, so the solution that matches the most tags from the user input is selected each time. The selection algorithm is further explained in the next chapter.

Additionally, each tool must be configured and deployed so that it can perform the intended task correctly. For example, a firewall must be placed in front of a targeted back-end service, while an active latency monitor must be placed alongside the monitored microservice. Moreover, the previously appended tags must be considered when determining the configuration parameters for a given network management tool. A second stage for selection is thus performed by the Template Mapper. A template represents the configuration required by a management tool to be deployed, both with respect to the tools internal configurations and with any necessary cluster definition (MARTINS et al., 2020c). The configuration aspect is also covered by the tags appended to the user input, in a process analogous to the one described for the tools matching.

Occasionally, the correct configuration for some management tool might require some breaching of microservices architectural design during execution. For example, monitoring the active connections for a microservice requires for the the management tool not only to be placed alongside the managed service, but to share its network context too. This is achieved by namespace sharing (PAHL, 2015) between managed and management services, but which is only performed when necessary. In this way, microservices design can be maintained for all applications, and specificities are configured and treated with templates designed for such cases.

3.4 Management Mappings through Instantiable Templates

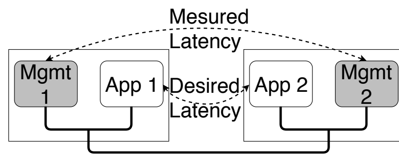
Network management is a discipline that includes, for example, network configuration, fault analysis, performance monitoring, and security assurance. The management of complex networks is thus not to be solved by a single tool but rather by the careful selection and combination of network management software. Their diverse purpose means that network management tools greatly differ with respect to their computational and architectural requirements, *e.g.*, a misplaced firewall is a useless firewall. A typical network management architecture is composed by management agents, that interact directly with managed devices to collect management information and oftentimes set configuration parameters, and a central management entity (*e.g.* an SNMP manager) that monitors and acts on the data collected by agents. Being a distributed application itself, the network management architecture can also be organized as a set of micro-services.

To fulfill user expectations regarding network management features, the appropriate set of tools must be chosen. Since requirements for management tools differ from each other, careful thought is required in their deployment. In this context, our architecture introduces *instantiable templates* for network management architectures. An instantiable template contains the information required so that management containers can be deployed alongside the user application containers, their positioning, and any other configuration needed to realize the management function correctly. Experts can develop new templates as needed, and the new templates can be fed to the architecture's template catalogue.

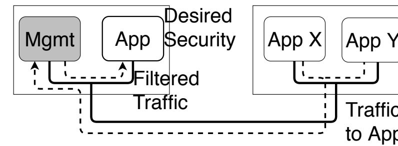
To illustrate the different template compositions, consider the following 4 common network management tasks that our system can realize (Figure 3.2):

Figure 3.2: Set of instantiable templates, from least to most intrusive.

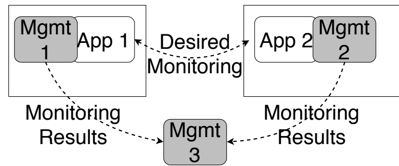
(a) Latency example maintaining complete isolation



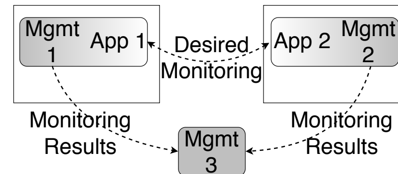
(b) Security example with traffic rerouting



(c) Monitoring example with partial namespace sharing



(d) Configuration example with full namespace sharing



Source: Author

(a) Monitoring the latency between two containers of the user application: for latency sensitive applications, deploying instances of a tool such as OWAMP, one at each of the hosts where the application containers reside, and connecting them through the same local network as their application counterparts can be enough to provide the intended latency measures. Complete isolation between management and user application is maintained, in this situation.

(b) Securing an application through the use of a firewall: the incoming traffic must be routed through the firewall. Only the firewall position in the network is relevant to the deployment of this solution, and thus containers remain isolated. However, traffic must be rerouted and potentially modified through the new firewall function, which could affect the application performance.

(c) Monitoring all network traffic between two containers of an application: NetFlow or similar tools can be used to realize the desired monitoring. However, deploying the containerized agent tool in its own network namespace would be useless, since it would be only monitoring itself. Instead, they must be deployed in the same network namespaces (and host, therefore) of the user applications, so it can correctly perform the desired function. A collector that centralizes monitoring agents data must also be included in the template, as illustrated by the container “Mgmt 3” in Figure 3.1c, but does not require any special isolation or positioning configuration relative to the application containers.

(d) Monitoring and configuring parameters of two containers of an application: this can be done through SNMP, for example, implicating a more intrusive namespace sharing between containers, since SNMP must access network, mount, and other information that would otherwise be isolated. Both application and management containers thus reside in the same set of namespaces, isolated from other systems but not from each other.

4 PROTOTYPE IMPLEMENTATION

While last chapter discussed the architectural design for SWEETEN, this chapter delves into the implementation details for the prototype. Section 4.1 discusses general aspects regarding implementation choices and user input. Section 4.2 explores the process of acquiring the required features. The process of deploying the mapped solution and returning a customized dashboard to the user is discussed in Section 4.3.

4.1 Implementation overview

The prototype was implemented using Python v2.7.17 for the main components in the architecture. Each component (*i.e.*, Features Acquirer, Tool Mapper, Template Mapper, Deployer) was developed as an independent module, and Kubernetes v1.18.5 was used without modifications as the Container Orchestrator. Some minor functions (*e.g.*, getting nodes information for a Kubernetes cluster) were implemented through shell scripts. Python library `PyYAML` v5.3.1[1] was used to read the user input specification, which is then parsed by the Features Acquirer module, and to later write the solution specification that is deployed with Kubernetes. The Tools Catalogue and the Template Catalogue are both materialized through YAML configuration files. That is due to two main reasons. First, the format's readability facilitates the inclusion of new items by experts. Second, it simplifies the generation of a deployable specification from the templates since the language is used by the deployment specification itself. Publicly available DockerHub repository[2] was used as the Container Image Catalogue, and Grafana v7.1[3] is used to produce the customized users dashboard for monitoring functions. An open-source for the prototype is available at <https://github.com/ComputerNetworks-UFRGS/sweeten>.

Docker containers orchestrated by Kubernetes have for the past few years emerged as the most prominent combination for running such complex applications (JAWARNEH et al., 2019), even more so with the discontinued support for alternatives like Docker Swarm[4], and therefore constitute the main container platforms for SWEETEN. The rich ecosystem and widely adoption from industry and academia alike favours both the system's development and its adoption by the wide public (BERNSTEIN, 2014).

---

[1]<https://pyyaml.org/>
[2]DockerHub can be acessed at <https://hub.docker.com/>
[3]Grafana monitoring platform can be found at <https://grafana.com/>
[4]<https://www.mirantis.com/blog/mirantis-acquires-docker-enterprise-platform-business/>

In the Kubernetes architecture, containerized microservices run in entities known as pods. Pods serve as a logical host for containers, sharing storage and network, and being scheduled and deployed together. Pods primary motivation is to support helper programs (*e.g.*, loggers, managers) for the primary container, thus offering a compromise between the microservice paradigm (*e.g.*, decoupled dependencies for each container) and the monolithic benefits (*e.g.*, shared context for monitoring). When mapping SWEETEN architecture templates into Kubernetes, management containers are deployed in the same pods as the managed containers whenever network context sharing is mandatory for the management tool to perform appropriately.

In Kubernetes, the deployment specification is typically realized by one or more YAML configuration files (BEN-KIKI; EVANS; INGERSON, 2009) that specify how the service is composed. For each microservice defined, operators add tags for the management features they expect to attain. As discussed in the previous chapter, operators can employ different abstraction levels when requesting for management features. In that way, an experienced user can go into lower-level configuration specification for how the features should be realized, while a novice user can be less specific and still obtain a proper management solution. An example for the latter is presented in Listing 4.1, where the deployment specification simply determines that flow monitoring and security features must be included for that deployment. The system interprets the requested feature and maps the appropriate tools and configurations to fulfil the request. In this example, the request would be mapped to a template using sFlow (PANCHEN; MCKEE; PHAAL, 2001), even if Prometheus and other equivalent monitoring tools could also fulfil the same feature request, depending on the deployment requirements.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    management:
      monitoring:
        - flows
      security:
        - cryptography
    ...
```

Listing 4.1: Excerpt for a simplified user management service with requirement for management features.

While novice users can request high-level management features more easily, advanced users can specify lower-level configuration parameters that must be observed in the provided solution. This enables operators to promote network management capabilities from a high-level perspective, while also being able to traverse through lower-level configuration parameters when necessary. An example for a more deterministic specification that an experienced user could request is presented in Listing 4.2. Similarly to the previous example (for the novice user), monitoring features are requested. However, unlike the previous example, the user is much more specific in the requests. In this example, the monitoring tool has been specified (*i.e.*, Prometheus), including the need for a specific version. The *scrape_interval* is specified to be set at five seconds. Finally, rather than using the default expression browser for visualization, the user uses nesting specification that *Grafana* should be used for visualization and that its dashboard must listen in the 3030 port (instead of the default 3000). SWEETEN processes the entire user request and adjusts the tools and templates mappings accordingly.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    management:
      - monitoring: flows
        protocol: TCP
        tool: Prometheus
        version: 2.18.0
        scrape_interval: 5s
        dashboard:
          - tool: Grafana
            http_port: 3030
          ...
```

Listing 4.2: Excerpt for a feature request that includes lower-level configuration parameters specification

4.2 Tagging

As mentioned in the previous chapter, SWEETEN can utilize tags associated to microservices and features to optimize the selection of tools and configurations when deploying a solution. Because these tags are seldom provided, SWEETEN applies an

enrichment process for each container declared in the user specification. For the purpose of our proof-of-concept, the description enrichment process is obtained through a simple Google web search query that is automatically requested by the system. This query is composed of the words "define" and the container image name, and the text for the first result is considered by the system. The service tagging process can then run the enriched description through a NLP sub-module to extract the aforementioned tags, as described next.

Among the many available algorithms for NLP, an important aspect that differentiates them is whether they require supervised learning or not. In SWEETEN's case, since we want not only to include current management features but also to allow the system to easily evolve and include new ones, unsupervised learning is preferred. Our model of choice is based on Latent Dirlecht-Allocation (LDA) (BLEI; NG; JORDAN, 2003), an unsupervised machine-learning algorithm that can help find common topics between multiple text documents. In this way, we initialize a database with enriched descriptions for three of the top containers for each category in DockerHub, and stipulate seven different topics to be found. Each topic will contain a weighted list of words that indicates the prevalence of the main words for each topic. For a new document, *i.e.*, the enriched description for the user microservice that is being processed, LDA associates a percentage for each pre-determined topic.

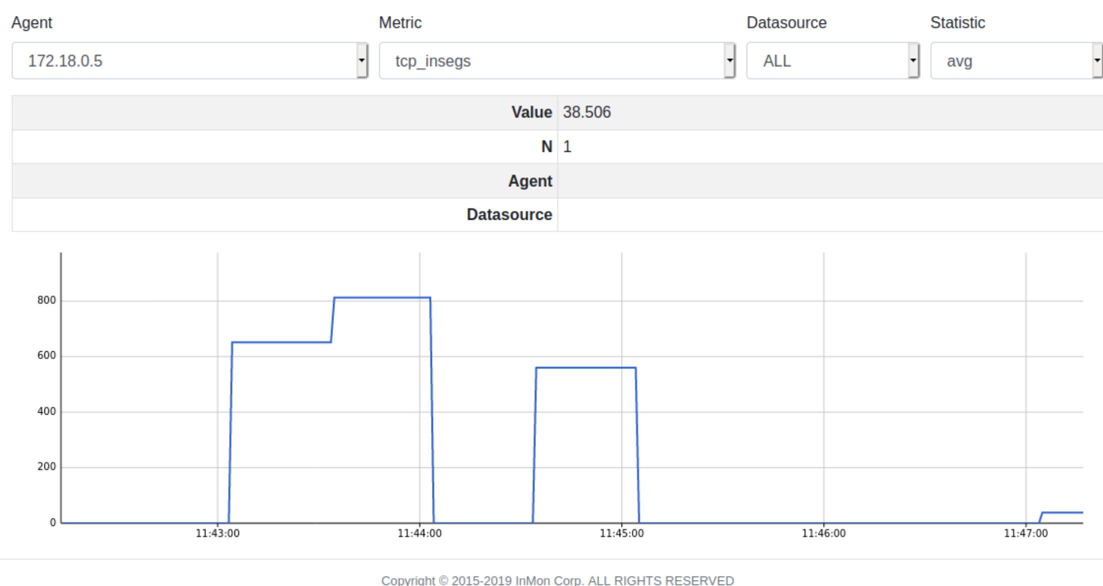
The same process is also performed over the tools and configurations available in the catalogue. When the user input is processed, the features requested are first matched to the tools that can realize them. The matching of topics (and thus of words) between user microservices and management alternatives will determine what is chosen in each instance, by trying to match the highest percentage topic for the user microservice and for the solutions alternatives. If no match can be found, the system defaults to the first solution present in the system. It is important to notice that this work does not intend to propose a new NLP method, neither argue that LDA can outperform other NLP algorithms. Indeed, the choice is an opportunistic one that requires minimal intervention from experts, and can be used to demonstrate how modern models can be integrated into SWEETEN and assist the fine-tailored provisioning of management solutions.

4.3 Solution Deployment and User Dashboard

The elements of the solution must be put together in a deployable specification. Because we chose to use Kubernetes as the system's container orchestrator, the result is composed of two separate YAML (BEN-KIKI; EVANS; INGERSON, 2009) specifications: one for the user services that did not require management features, *i.e.*, services that were already part of the user specification, but that did not require any management feature; and one for the remainder of the user specification plus all the network management tools included by SWEETEN.

During the slice lifecycle, the user can manage their services through a customized dashboard. Based on the user input, SWEETEN can deploy dashboards from different software. An example for a monitoring dashboard by Prometheus [5] provided by SWEETEN for a novice is depicted in Figure 4.1, while for the experienced user a more advanced and fine-tuned Grafana dashboard provided by SWEETEN is depicted in Figure 4.2. Non-visualizing features, such as cryptography introduced by security features, are presented textually for the user's knowledge. Additionally, the user can interact directly with the configured management container through a terminal, so they can still have control after the deployment phase for their slice.

Figure 4.1: User dashboard generated from the novice user's specification.

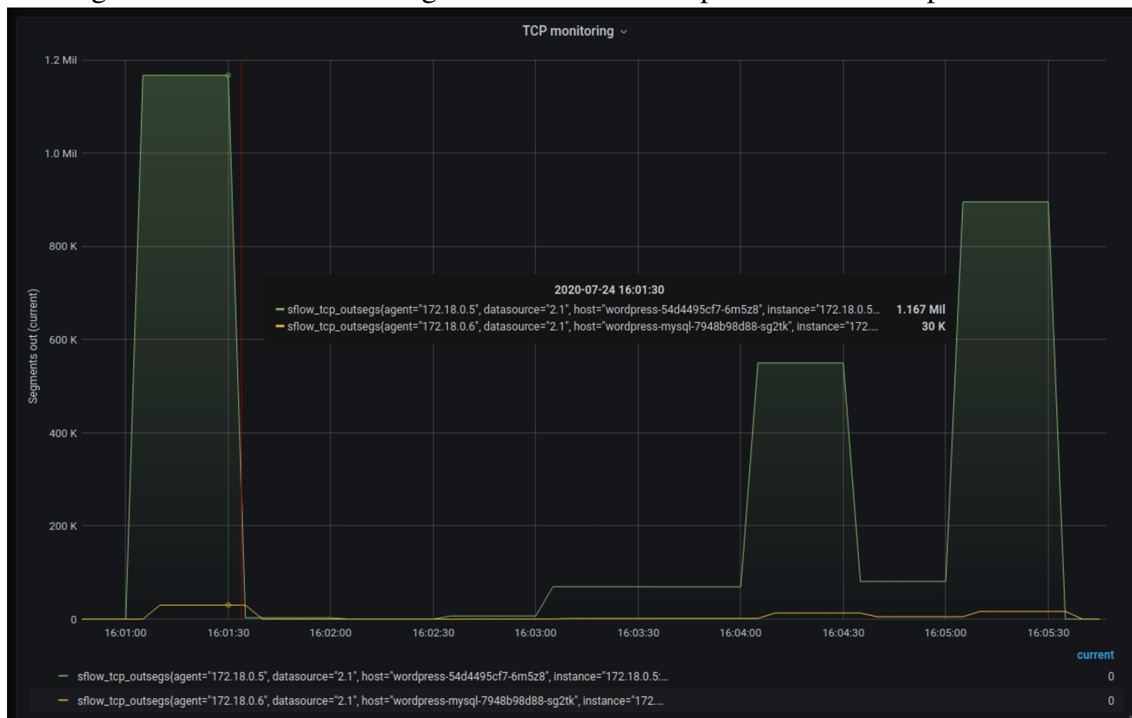


Source: Author

An important aspect that differentiates templates is with respect to container isolation. Container isolation between user application and network management tools should

[5]Prometheus monitoring system can be found at <https://prometheus.io/>

Figure 4.2: User dashboard generated from the experienced user's specification.



Source: Author

be maintained whenever possible. This is realized by the definition of a complete and exclusive set of namespaces for each container deployed, providing resource isolation between processes and containers running in a single system. Usually, in the micro-service paradigm, complete isolation between containers is a welcome feature. However, by carefully breaching certain isolation aspects between specific management tools and the user application they are expected to manage, we can leverage the benefits of the micro-service paradigm while performing the network management deployment needed to realize users' desired features. In this case, two or more containers will share a subset of namespaces, allowing the network management tool to properly perform its function.

Other solutions such as service meshes typically work by appending a sidecar proxy to all containers of interest, *i.e.*, a separate container in the same pod that proxies all connections to and from the primary container, adding management functionalities as needed (LI et al., 2019). Contrarily to that, this master's dissertation proposed system only adds containers to the same pod (and intercepts connections) when the desired management feature requires so (for example, security functionalities that must filter the incoming/outgoing packets), and as indicated by the Management Template Catalogue. When this requirement is not present, appended management functions can reside in the same pod but not proxying the main container connections, or even reside in a separate

pod altogether. An example of the former would be for some passive monitoring functions, with the benefit of lessening the communication overhead from keeping the hop count as low as possible. An example of the latter would be for some active monitoring functions, such as determining the latency between containers located in separate nodes in a cluster, and that thus can be monitored by having the management pods be placed on the same nodes while keeping their context completely independent from the primary container.

5 CASE STUDIES

This chapter proposes two complementary case studies that are used to evaluate SWEETEN's effectiveness. The first study is presented in Section 5.1, and is centered around a 5G radio split for dynamic cloud Radio Access Network (C-RAN), where SWEETEN must deliver management solutions for stringent VNFs requirements. The second study is presented in Section 5.2, and considers a more complete 5G network slice. The study is centered in an intelligent healthcare service, forcing SWEETEN to realize management solutions across a range of elements throughout the network infrastructure.

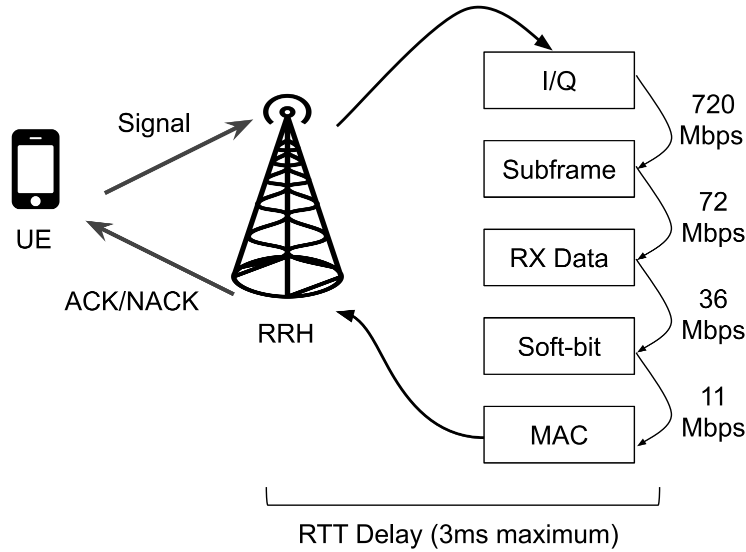
5.1 5G Radio Split for Dynamic C-RAN

Traditionally, network mobile services have been provided by a mobile network operator (MNO). Recently, mobile virtual network operators (MVNOs) have emerged as an alternative for customers. The new virtual providers do not own the physical wireless infrastructure, and must thus lease it from traditional MNOs. Mobile services in turn can be delivered through cloud computing. The various strategies adopted by MNOs can benefit customers and the provider alike (KAMIYAMA; NAKAO, 2019).

In this case study, an MVNO must allocate a number of virtualized Base Stations (BSs) over a region. Being a dynamic C-RAN adopter, the provider makes use of Remote Radio Heads (RRHs) that have their signals processed by Base-Band Units (BBUs). Each BBU is comprised of five forwarding elements: I/Q, Subframe, RX Data, Soft-bit, and MAC (WUBBEN et al., 2014). These elements have stringent requirements regarding bandwidth between the elements and end-to-end latency, as shown in Figure 5.1. In particular, latency requirements limit the maximum distance between an RRH and its BBU, in a relationship that depends on the channel condition and the processing power available (MAROTTA et al., 2018). To assert its compliance to the service terms, the MVNO must properly monitor each BBU closely in order to avoid any violation, with the monitoring overhead itself being kept at minimal levels.

In order to meet the demand for a certain region, the provider must instantiate 15 BSs in the region. To do so, the BBU functions must be allocated along with a central cloud, a regional cloud, and a fog. To maximize the computational resources used both in clouds and in the fog, and to minimize the front-haul data rate, the placement algorithm prioritizes running all BBUs' I/Q and Subframe functions as near to the fog as possible,

Figure 5.1: Communication flow and bandwidth requirements for radio functions.



Source: Author

since both functions are responsible for the majority of the front-haul data rate. The remainder of the BBU functions should be placed on the regional and the central clouds, prioritizing the latter due to its increased computational capacity, whenever latency permits it. Additional functions (such as management entities) should run on the central cloud whenever possible, as to not overload the fog and the regional cloud unnecessarily. The resulting placement for the elements of the 15 BSs is shown in Table 5.1.

Table 5.1: Resulting distribution of BSs functions.

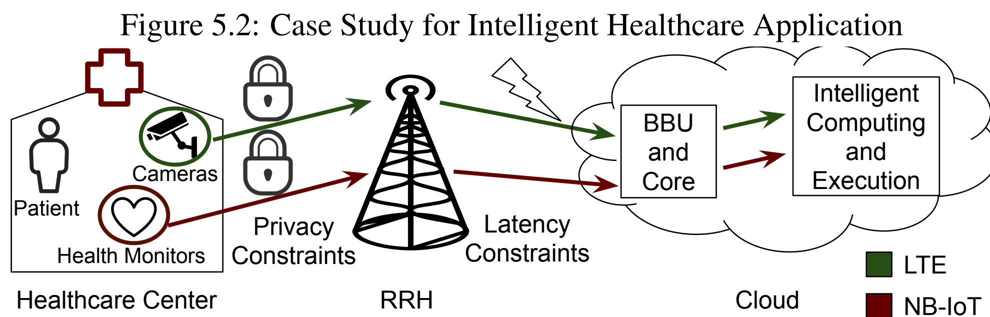
	I/Q	**Subframe**	**RX Data**	**Soft-bit**	**MAC**
Fog	5	5	5	0	0
Regional Cloud	5	5	5	5	0
Central Cloud	5	5	5	10	15

Being the owner of the BS application, the service provider is capable of managing and monitoring each container appropriately. However, the network monitoring is less trivial and it depends on external factors, and due to the stringent requirements, it needs to be properly done. The provider uses our system by tagging the required management features (*i.e.*, the latency and traffic monitoring for all containers) in the deployment specification for the BSs, and SWEETEN deploys the complete solution which includes the tools to realize the required management. The following section presents a separate case study, while the results for both deployments are presented in the next chapter.

5.2 Automated Network Management for Intelligent Healthcare

5G systems promise a series of disruptive advancements for a myriad of applications typically classified in three scenarios. Enhanced Mobile Broadband (eMBB) addresses applications centered in multi-media content, services, and data; Ultra Reliable Low Latency Communications (URLLC) encompasses critical applications that pose stringent requirements such as remote medical surgery; massive Machine Type Communications (mMTC) is characterized by a large number of low-cost devices that transmit a low volume of data (SERIES, 2015). Some of the most technically challenging applications unite requirements from two or even all three scenarios. Healthcare applications can exemplify such a case, where a multitude of health devices of different capabilities and with distinct requirements are used to guarantee the well-being of patients. This use case is depicted in Figure 5.2 and further expanded in the following.

Figure 5.2: Case Study for Intelligent Healthcare Application



Source: Author

Heart rate, respiratory rate, and body temperature monitors are a small sample from a large list of monitoring devices that can be utilized in a patient's health monitoring. The number of IoT devices employed in this scenario can grow significantly, providing abundant data to track patients physiological characteristics but also requiring a more extensive analysis by physicians and other professionals. In this context, these applications can be further benefited by the inclusion of intelligent algorithms to process and automate decisions, triggering alarms and actions whenever abnormalities are detected (WANG et al., 2018). Artificial intelligence (AI) techniques based on novel models such as big data mining and deep learning can process large amount of data at real-time, and then predict and automate tasks at a rate impossible before. While the monitoring part must be performed on-premise, the burden of collecting and processing all the data can be effectively run in the cloud.

Because of the sensitive nature of the data monitored and transferred, security, in particular by the means of privacy, is a foremost concern. As hardware solutions are not always feasible and as new legislation advances the levels of privacy requirements for these applications, guaranteeing a certain security level from a software perspective is a necessity. In certain occasions, resourceful devices are used for patients' monitoring, such as 4K cameras that can record their movements, and paired with deep-learning algorithms can detect facial expressions and gestures of patients and warn healthcare professionals in the event of an anomaly (WANG et al., 2018). However, most monitoring IoT devices are constrained in terms of computational power and battery, and so possible security solutions should account for these limitations and prioritize lighter-weight solutions whenever possible. Less constrained devices, in turn, can afford to employ more advanced and intensive defensive mechanisms, and the provisioning for each case should reflect these characteristics. As manually configuring each security mechanism for every device is not scalable for complex 5G scenarios, automation is a key principle in securing 5G applications and networks (SUN et al., 2020).

In recent years, multiple Low Power Wide Area (LPWA) radio technologies have emerged as options for delivering the scalability required by mMTC applications. From the alternatives, NB-IoT has been shown to offer promising results for healthcare applications (MALIK et al., 2018). NB-IoT is fully compatible with Long Term Evolution (LTE), and can be deployed inside a single LTE physical resource block (PRB) of 180 KHz or inside an LTE guard band, potentially serving up to 50k end-devices per cell (RAZA; KULKARNI; SOORIYABANDARA, 2017). The limited 250 kbps data rate is plenty for heart rate and body temperature monitoring that require only 1 byte for payload every 5 minutes (MALIK et al., 2018), but it is impractical for streaming the video from the deployed cameras. Devices as such that require extensive bandwidth must connect over standard LTE-A network, which is capable of meeting their demands.

Metrics collected by all the devices are reported to an RRH, the radio antenna responsible for communications to and from users' devices. The signals must then be processed by the network core, which is performed by the BBU in a Base Station (BS). Previously, these functions would be performed exclusively by specific-purpose hardware. With the recent advances in virtualization and the expansion of NFV architectures, virtual base stations have been adopted by pioneering MVNOs. Such operators do not own the required wireless physical infrastructure, but instead lease it from traditional mobile network operators. Therefore, the processing modules for wireless services can

be provided by software running in the cloud, enabling different strategies that benefit customers (KAMIYAMA; NAKAO, 2019). In the NB-IoT case, low protocol stack processing requirements and low latency-sensitivity make C-RAN an attractive alternative, so all the BBU processing and higher-layer protocol stacks are implemented by software that runs on the cloud (BEYENE et al., 2017).

A standard LTE-A BBU can be split in different functions (WUBBEN et al., 2014). The different split options offer possibilities of alternating between dedicated hardware and function virtualization, allowing the flexible adoption of functional split in time and location. Noteworthy, 5G specifications pose stringent network requirements for their communications, in particular with respect to data rate and latency. Regarding latency, a maximum delay of around 3ms for transmitting and processing the signal is determined by the Hybrid Automatic Repeat reQuest (HARQ) mechanism adopted in LTE (MAROTTA et al., 2018). There is thus a stringent requirement (*i.e.*, latency) that must be respected by the network slice, and that must be properly monitored too. While the first case study (presented in Section 5.1) considered monitoring challenges for microservice-based VNFs in 5G networks, this second case study further advances the management scope by including security concerns and measurements in the evaluated slice. Moreover, the complete network slice itself is subject of study here, including different radio access technologies and service applications, with new features and requirements that they come with. The following chapter examines and discusses the results for both case studies presented in the current chapter, highlighting the benefits and drawbacks operators can expect when using SWEETEN.

6 RESULTS AND DISCUSSION

This chapter presents the most important results for the two case studies presented in the previous chapter. The setup and additional software used to achieve the results are presented in Section 6.1. The results from both case studies are grouped and presented according to three dimensions: expressiveness, deployment overhead, and computational and network overhead. The results regarding operators' expressiveness gains are presented in Section 6.2. The results for the deployment time needed for each solution are presented in Section 6.3. The results for computational and network overhead are discussed in Section 6.4. Finally, the output visualized by the user is presented in Section 6.5.

6.1 Experimental Setup and Software

For both experiments, the setup consists of four virtual machines (VMs) spread across two physical hosts. Each VM has six CPUs and 16GB of RAM. To account for the variance, each experiment was repeated 30 times. Results are presented with the average and standard deviation, which were calculated and plotted with R (R CORE TEAM, 2021), and specifically the package ggplot2 (WICKHAM, 2016). Memory usage was measured through Linux utility program *free* [1], and likewise, *top* [2] was used for CPU usage measurements. The remainder software utilized comprises SWEETEN's prototype, as has been previously described in Chapter 4.

6.2 Expressiveness Gains

The first analysis regards the expressiveness gains for the operator. Because Kubernetes does not intend to interpret high-level feature requests, the proposed system naturally outperforms what would be required from the operator as manual input. In the first case study, it takes the operator only four lines of high-level feature specification to trigger the deployment of four additional management containers (plus two for each subsequent microservice), as defined by their deployment templates. If the operator were to do it manually, the operator would have to input an additional 157 new lines of specifica-

[1] <https://man7.org/linux/man-pages/man1/free.1.html>
[2] <https://man7.org/linux/man-pages/man1/top.1.html>

tion for the first BS, plus 100 reoccurring lines for each subsequent BS. Even if Kubernetes specification is not designed for this purpose, it would be the available alternative prior to SWEETEN. Being able to do more with less is a recurrent concern for operators (CURTIS-BLACK; WILLIG; GALSTER, 2019). Most importantly, the labour of including these specification lines pales in comparison to the one of determining which precise commands and configuration parameters should be in the lines in the first place. Realistically, hours of work would be spent in finding the correct tools for the job, and properly configuring them manually for the deployment at hand.

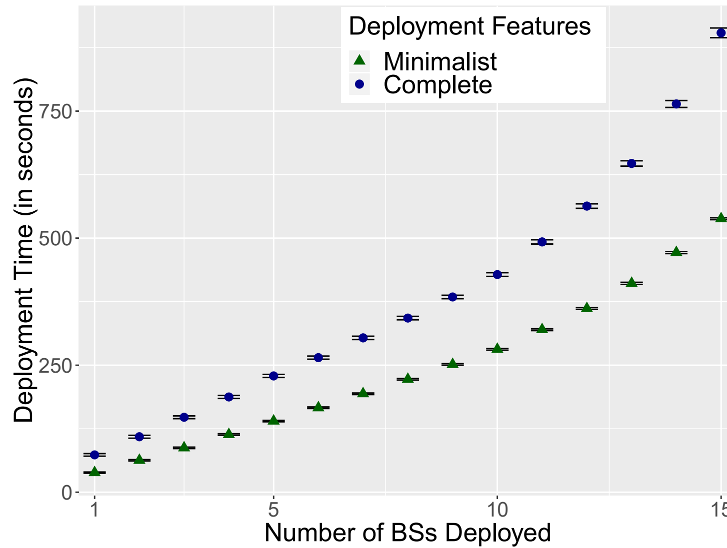
The results for the second case study follow a similar trend. About four lines of high-level feature specification by the user translates to over 30 lines of management deployment specification with respect to security (disregarding the cryptography keys generated and used in the deployment, which have been manually generated prior to the deployment processing). The result is even more prominent with respect to monitoring, where four lines of specification are translated into over 100 lines of management specification that add the required monitoring features. These findings highlight the main benefits offered by SWEETEN for multiple management disciplines.

6.3 Deployment Time

The second analysis regards the deployment overhead in utilizing SWEETEN. To do so, for the first case study it is evaluated the time it takes to deploy the 15 BSs in their initial minimalist state (*i.e.*, with no added management features), and with the complete solution produced by the system. In each case, experiments were run 30 times. The results are presented in Figure 6.1.

On average, the complete deployment took 59.6% more (about 145 seconds) than the minimalist deployment. The evolution of the experiment shows a similar linear pattern for both cases in the earlier stages (*i.e.*, less than 10 BSs). The latter stages shows a disproportional increase in the complete solution in comparison to the minimalist approach. The large number of pods and containers take their toll in the container orchestrator, highlighting the importance of considering the deployments specificities when determining the correct management solutions. Moreover, virtually all of the overhead was due to the additional containers Kubernetes had to deploy and launch, meaning users would still incur in comparable costs if they were to produce a similar solution by other methods. Finally,

Figure 6.1: Time taken to deploy up to 15 BSs in the first case study, with and without using SWEETEN.



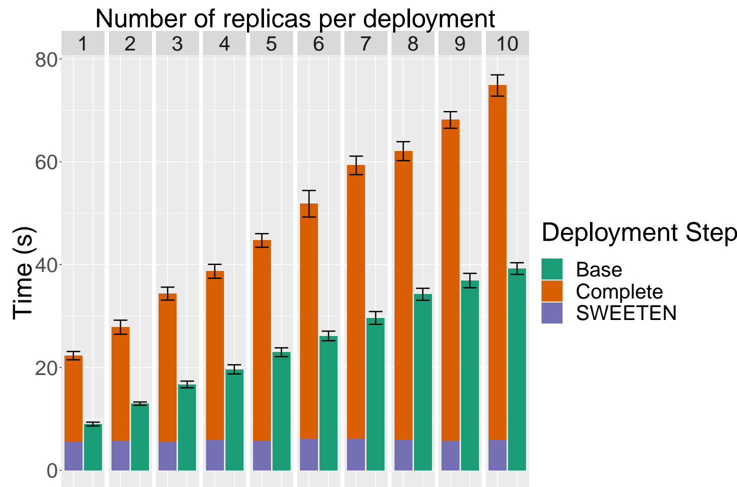
Source: Author

this cost is regarding the deployment of all BSs from scratch, and therefore not a recurring cost.

The analysis for the second case study highlights an important difference when considering SWEETEN's role in the increased overhead. That is because there are two separate types of processing overhead that must be considered. The first one is the overhead introduced by SWEETEN's processing of the user input until the complete solution is produced. The second one is the additional deployment overhead due to the inclusion of the management services (realized by containers), that must be instantiated alongside the original specification. A comparison for these times is presented in Figure 6.2. The system's scalability is evaluated through varying the number of replicas for each deployment in the slice from one to ten.

The results show that SWEETEN's overhead is approximately constant regardless of the replicas count, and it becomes negligible for larger deployments. Larger deployments are precisely the ones that should benefit the most from SWEETEN, as the inclusion of management features throughout a complex slice is burdensome in comparison to a more simplistic one. Most of the overhead is introduced by the inclusion of the additional containers, with the complete solution taking on average 94% more time to be deployed. Two noteworthy points here are that: (1) this is not a recurring cost, as fresh deployments are less frequent than individual updates, which would present a much lighter overhead; (2) the manual inclusion of management containers by an expert user would

Figure 6.2: Deployment time overhead for varying replicas count in the second case study.



Source: Author

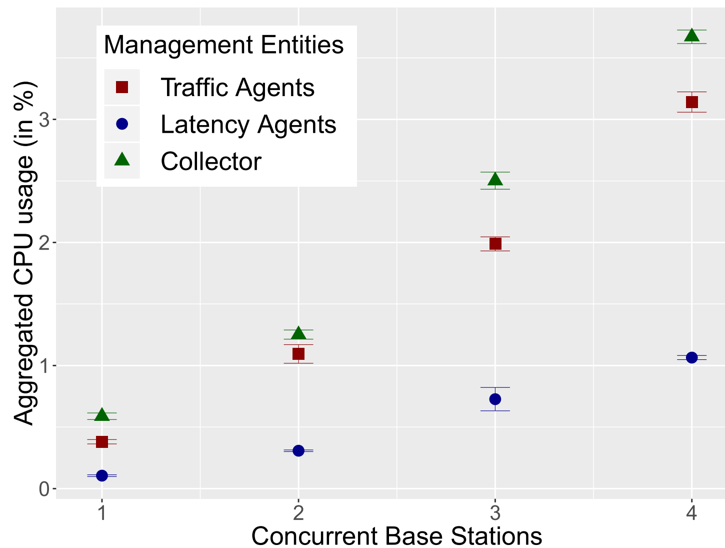
incur in similar overhead for the deployment time. Users could minimize this overhead by including the management software directly into their containers, but doing so would breach the microservice architecture and possibly do more harm than good in the process.

6.4 Computational and Network Overhead

The third analysis focuses on the computational overhead for the remainder of the deployment life-cycle. To assess the CPU usage by management entities included in the deployment, the impact of scaling from one to four BSs in a single VM is evaluated. In this analysis, having all the containers run in a single VM offers a fair comparison for the overhead introduced by each management entity in the architecture. The results presented in Figure 6.3 show how the management entities consume negligible processing for the most part. The collector consumes approximately the same CPU as all the agents combined, but still sits at just over 3.5% for four concurrent BSs. Moreover, since the collector has no strict placement constraints (it only requires to be reachable by the agents), it can be placed in the more resourceful nodes in a deployment with little impact on deployment performance. Between the two types of monitoring agents, it is possible to note that traffic monitoring consumes significantly more CPU than latency monitoring. Still, the sum of all agents for each BS comes at approximately 1% CPU usage, thus the overhead is largely negligible for distributed deployments along the cluster.

The results for memory usage and network overhead follow closely. An average RAM usage of 1.54GB for running the user dashboard and metrics collector, plus 7.38MB

Figure 6.3: CPU usage (in percent) for management containers for up to four concurrent (same VM) base stations in the first case study.



Source: Author

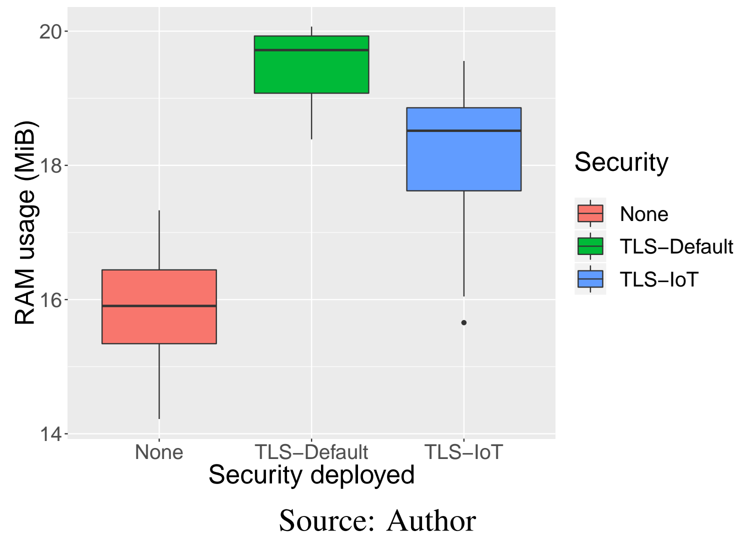
per microservice managed (totalling 36.94MB per BS). The dashboard and collector increased cost are justifiable because they are a unique cost for the entire deployment, and its independence means it can be deployed in the (resourceful) central cloud. In turn, the computational overhead per BS due to management agents is mostly negligible, which not only is imperative due to the stringent requirements of the BS functions but also highlights the scalability of the solution. Regarding the network aspect, management agents introduce an overhead of around 5KB/s for incoming and outgoing traffic per BS. Around 30% of the overhead is due to the latency monitoring probes required for the active measurements. The remainder is mostly due to the periodic reports from agents to the collector. An advanced user could fine-tune parameters to their needs when requesting the features. For example, by increasing reports' scrape time, it is possible to further minimize the communication overhead or decreasing it could allow one to monitor sub-second variations closely.

In the second case study, it is noteworthy that the IoT monitoring devices should try to adopt solutions that incur in minimal overhead because of their resource-constrained nature. Tools and configurations provided by experts to the system's catalogues can therefore feature fine-tuned solutions for systems tagged as such. This way, a TLS configuration using less resource-intensive ciphers can be used for IoT components[3], while more resourceful components can utilize a more robust solution[4]. For comparison purposes,

[3]<https://docs.aws.amazon.com/iot/latest/developerguide/transport-security.html>
[4]<https://www.acunetix.com/blog/articles/tls-ssl-cipher-hardening>

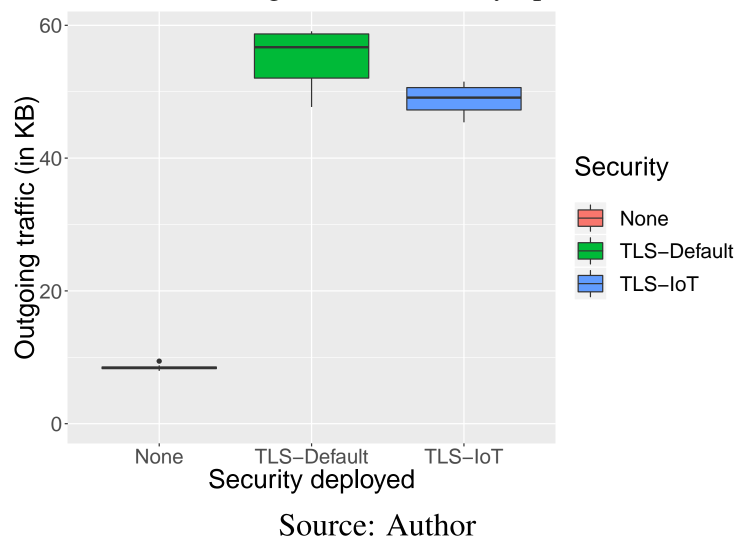
normalized results for the different security options are presented in Figure 6.4 for memory footprint, and in Figure 6.5 for network overhead.

Figure 6.4: Computational overhead using different security options in the second case study.



Source: Author

Figure 6.5: Network overhead using different security options in the second case study.



Source: Author

The results are in-line to what was obtained for the first case study, and indicate that a small computational overhead is added with respect to memory footprint when either the default or the IoT security solution is included. Albeit small, it is also noticeable that the security option recommended for IoT outperforms the default option with respect to overhead. Similar results are also found for network overhead. While it is clear that the overhead is much more noticeable when comparing either security option (*i.e.*, the default one or the IoT one) to having no security deployed, the IoT configuration still leads to lesser overhead in comparison to the default configuration. While the user should still

be mindful that some overhead will be added whenever a security feature is requested, these results show how experts knowledge integrated into the system through different configuration options can be used to produce a more fine-tuned solution for each case.

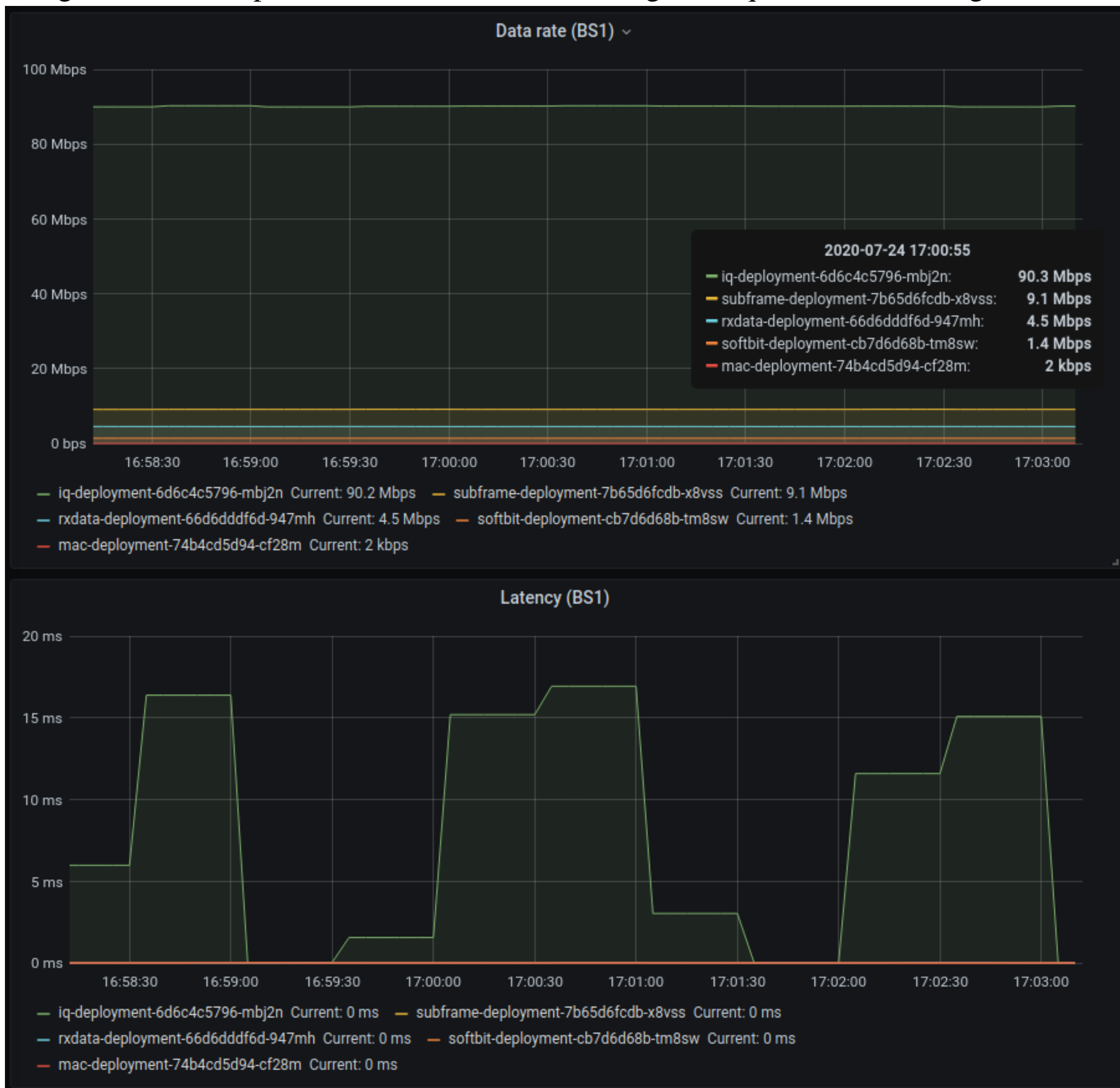
6.5 System's Output

Finally, the fourth analysis showcase the management dashboards and results that the user would have access to. Figure 6.6 shows an excerpt for the customized dashboard that the user for the first case study receives after the complete deployment. For simplicity, the monitoring for a single BS is presented. The dashboard consolidates requested monitoring metrics in a dynamic interface that allows the user easy access to the relevant metrics. As explained previously, users can be more specific in their requests in order to obtain a solution more fine-tuned to their needs. For presentation purposes, the monitoring graphs for the results discussed in the following were re-plotted for specific BSs. Figure 6.7 exemplifies the result for latency monitoring.

The latency result in Figure 6.7 shows the monitoring for two BSs prior and after additional BSs are deployed. The first BS (in green) is deployed over fog (I/Q, Subframe, and RX Data) and regional cloud (Softbit and MAC), while the second BS (in blue) is fully deployed in the central cloud. Prior to the deployment of additional BSs (marked by the vertical line), no latency violations (marked by the horizontal line) are detected for any of the BS. After the deployment of two new BSs (over the three clusters), instability incurs in several violations (four in the figure) for the first BS, while none are for the second BS. The monitoring result alerts the operator that the new deployments are negatively impacting the first BS, and corrective actions must be taken.

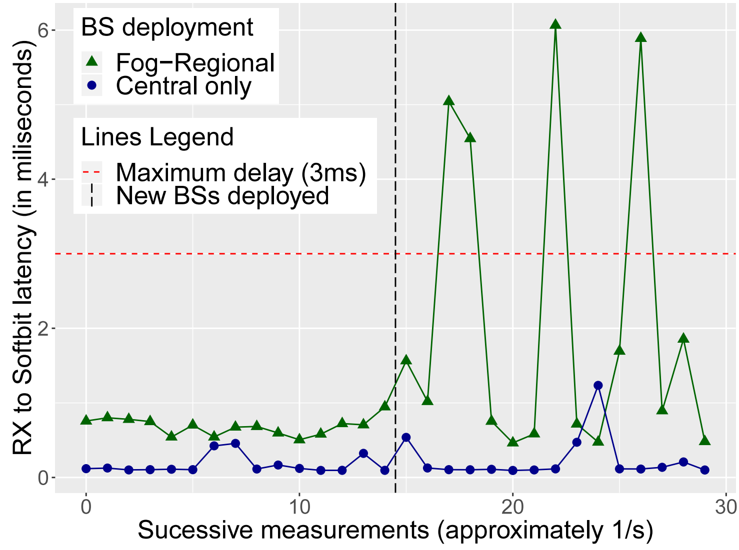
Similarly for the second case study an example for an excerpt of the dashboard provided for monitoring the throughput is illustrated in Figure 6.8. Through this interface, the user can easily monitor multiple services of their slice and quickly identify problems as they occur. Different resources can be configured by the system with different parameters in a transparent manner for the user. For instance, a resource-constrained device can be configured with a lower sampling rate than a more resourceful device, thus introducing less overhead. The user can also edit the graphics in the dashboard and add their own, so they can fine-tune the solution to best fit their needs. The security aspect currently can not be visualized through the dashboard, but the user can still access the service's containers directly and retrieve the solution's logs and configuration files.

Figure 6.6: Excerpt from user dashboard enabling the requested monitoring features.



Source: Author

Figure 6.7: Latency monitoring result for two BSs' RX to Softbit communication over a 30-second window (first case study).



Source: Author

Figure 6.8: Example dashboard for throughput monitoring of IoT devices, in the second case study.

Source: Author

7 CONCLUSION AND FUTURE WORK

5G networks are in the process of being rolled out around the world and will enable disruptive applications and services that were not previously feasible. To realize that, advances presented by NFV, SDN, and network slicing, for example, must be carefully integrated by these networks. With the increasing number of devices and services, network management plays a central role in delivering the resources and features required by each component. For the same reason (*i.e.*, the increasing number of networks components), the configuration and management of all the pieces must be realized in an automated manner.

This master's dissertation proposes the assisted management of network slices using SWEETEN. Initially proposed as a system to assist VNF operators, SWEETEN has been demonstrated in this study as a tool capable of delivering management solutions across a diverse network slice. Through high-level annotations in their slice specification, users are able to effortlessly receive fine-tailored management solutions configured for each of their applications and services. Using NLP, SWEETEN can extract information from each service and use it to determine the best management solution for each case.

A prototype for SWEETEN is evaluated through two case studies. The first case study investigates management challenges, specifically latency and throughput monitoring, in a dynamic C-RAN. The second proposed case study demonstrates how a network slice for intelligent healthcare can include monitoring and security features with ease, even considering the different requirements for each application.

The results show that there is an important expressiveness gain for the user through SWEETEN. Assisting the user in properly deploying complex network slices is a vital point in achieving the dynamism expected from 5G networks. The overhead of SWEETEN with respect to deployment time and computational and network overhead is also evaluated. While the deployment time is noticeably affected by the additional management services included by SWEETEN, it is not a recurring cost (*i.e.*, the slice's deployment) and is easily offset by the management functionalities featured in the slice. In the same way, computational overhead was non-negligible, but small enough that it is adequate for the services included. Network overhead, however, was much higher when cryptography solutions were included in the system, which, while expected, means users' discretion is needed to define whether the overhead is acceptable or not.

As future work, the system can be further developed through the inclusion of additional management disciplines and solutions. A myriad of different services are coming with 5G, and their network requirements can be as varied as the services themselves. It is thus important for a management assistant to be able to cover a variety of cases so that its usefulness is not limited to a small subset of applications, but rather applicable to the whole spectrum that 5G networks entail.

Another aspect that can be worked on the future is the evaluation of different tagging mechanisms for the Features Acquirer module. NLP is a rapid-evolving field, with new and improved algorithms being developed each day. An in-depth evaluation of such algorithms can further improve the Features Acquirer module, in particular, so that a more refined tagging system can help SWEETEN to further fine-tune solutions and configurations for every service.

REFERENCES

BEN-KIKI, O.; EVANS, C.; INGERSON, B. **YAML Ain't Markup Language (YAML) (tm) Version 1.2**. [S.l.], 2009. Available from Internet: <http://www.yaml.org/spec/1.2/spec.html>.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, v. 1, n. 3, p. 81–84, 2014.

BEYENE, Y. D. et al. NB-IoT technology overview and experience from cloud-ran implementation. **IEEE Wireless Communications**, v. 24, n. 3, p. 26–32, 2017.

BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. **the Journal of machine Learning research**, JMLR. org, v. 3, p. 993–1022, 2003.

BRIM, S. W.; CARPENTER, B. E. **Middleboxes: Taxonomy and Issues**. RFC Editor, 2002. RFC 3234. (Request for Comments, 3234). Available from Internet: <https://www.rfc-editor.org/info/rfc3234>.

CHOWDHURY, S. R. et al. Re-architecting NFV ecosystem with microservices: State of the art and research challenges. **IEEE Network**, IEEE, v. 33, n. 3, p. 168–176, 2019.

CIUFFOLETTI, A. Automated deployment of a microservice-based monitoring infrastructure. **Procedia Computer Science**, Elsevier, v. 68, p. 163–172, 2015.

CLAISE, B. **Cisco Systems NetFlow Services Export Version 9**. RFC Editor, 2004. RFC 3954. (Request for Comments, 3954). Available from Internet: <https://www.rfc-editor.org/info/rfc3954>.

COELHO, W. da S. et al. On the impact of novel function mappings, sharing policies, and split settings in network slice design. In: IEEE. **2020 16th International Conference on Network and Service Management (CNSM)**. Izmir, Turkey, 2020. p. 1–9.

CURTIS-BLACK, A.; WILLIG, A.; GALSTER, M. Scout: A framework for querying networks. In: IEEE. **2019 15th International Conference on Network and Service Management (CNSM)**. Halifax, NS, Canada, 2019. p. 1–7.

CZIVA, R.; PEZAROS, D. P. Container network functions: Bringing NFV to the network edge. **IEEE Communications Magazine**, v. 55, n. 6, p. 24–31, June 2017.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: **Present and ulterior software engineering**. [S.l.]: Springer, 2017. p. 195–216.

ENNS, R. et al. **Network Configuration Protocol (NETCONF)**. RFC Editor, 2011. RFC 6241. (Request for Comments, 6241). Available from Internet: <https://www.rfc-editor.org/info/rfc6241>.

ETSI. NFV ISG White Paper #3: Network Operator Perspectives on Industry Progress. In: **SDN and OpenFlow World Congress**. Frankfurt, Germany: ETSI, 2013.

ETSI. **GS NFV-MAN 001 V1. 1.1 Network Function Virtualisation (NFV); Management and Orchestration**. [S.l.]: ETSI, 2014.

FEDOR, M. et al. **Simple Network Management Protocol (SNMP)**. RFC Editor, 1990. RFC 1157. (Request for Comments, 1157). Available from Internet: <https://www.rfc-editor.org/info/rfc1157>.

FELTER, W. et al. An updated performance comparison of virtual machines and linux containers. In: IEEE. **IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. Philadelphia, PA, USA, 2015. p. 171–172.

FRANCESCO, P. D.; LAGO, P.; MALAVOLTA, I. Migrating towards microservice architectures: An industrial survey. In: IEEE. **2018 IEEE International Conference on Software Architecture (ICSA)**. Seattle, WA, USA, 2018. p. 29–2909.

FRANCO, M. F. et al. Secbot: a business-driven conversational agent for cybersecurity planning and management. In: IEEE. **2020 16th International Conference on Network and Service Management (CNSM)**. Izmir, Turkey, 2020. p. 1–7.

FRANCO, M. F.; RODRIGUES, B.; STILLER, B. Mentor: The design and evaluation of a protection services recommender system. In: IEEE. **2019 15th International Conference on Network and Service Management (CNSM)**. Halifax, NS, Canada, 2019. p. 1–7.

HAKIRI, A. et al. Software-defined networking: Challenges and research opportunities for future internet. **Computer Networks**, Elsevier, v. 75, p. 453–471, 2014.

HIRSCHBERG, J.; MANNING, C. D. Advances in natural language processing. **Science**, American Association for the Advancement of Science, v. 349, n. 6245, p. 261–266, 2015.

JAMSHIDI, P. et al. Microservices: The journey so far and challenges ahead. **IEEE Software**, v. 35, n. 3, p. 24–35, 2018.

JARAMILLO, D.; NGUYEN, D. V.; SMART, R. Leveraging microservices architecture by using docker technology. In: IEEE. **SoutheastCon 2016**. Norfolk, VA, USA, 2016. p. 1–5.

JAWARNEH, I. M. A. et al. Container orchestration engines: A thorough functional and performance comparison. In: IEEE. **ICC 2019 - 2019 IEEE International Conference on Communications (ICC)**. Shanghai, China, 2019. p. 1–6.

JHA, D. N. et al. A holistic evaluation of docker containers for interfering microservices. In: IEEE. **2018 IEEE International Conference on Services Computing (SCC)**. San Francisco, CA, USA, 2018. p. 33–40.

JINGZE, L.; MINGCHANG, W.; YANG, Y. A container scheduling strategy based on machine learning in microservice architecture. In: IEEE. **IEEE International Conference on Services Computing (SCC)**. Milan, Italy, 2019. p. 65–71.

KAMIYAMA, N.; NAKAO, A. Analyzing dynamics of mvno market using evolutionary game. In: IEEE. **2019 15th International Conference on Network and Service Management (CNSM)**. Halifax, NS, Canada, 2019. p. 1–6.

KIST, M. et al. AIRTIME: End-to-end virtualization layer for RAN-as-a-service in future multi-service mobile networks. **IEEE Transactions on Mobile Computing**, p. 1–1, 2020.

LEE, S.; LEVANTI, K.; KIM, H. S. Network monitoring: Present and future. **Computer Networks**, Elsevier, v. 65, p. 84–98, 2014.

LI, W. et al. Service mesh: Challenges, state of the art, and future research opportunities. In: IEEE. **2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. San Francisco, CA, USA, 2019. p. 122–1225.

MALIK, H. et al. Narrowband-IoT performance analysis for healthcare applications. **Procedia Computer Science**, v. 130, p. 1077 – 1083, 2018. ISSN 1877-0509. The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops.

MAROTTA, M. A. et al. Characterizing the relation between processing power and distance between bbu and rrh in a cloud ran. **IEEE Wireless Communications Letters**, v. 7, n. 3, p. 472–475, 2018.

MARTINS, R. de J. et al. Virtual network functions migration cost: from identification to prediction. **Computer Networks**, p. 107429, 2020.

MARTINS, R. de J. et al. Sweeten: Automated network management provisioning for 5g microservices-based virtual network functions. In: IEEE. **2020 16th International Conference on Network and Service Management (CNSM)**. Izmir, Turkey, 2020. p. 1–9.

MARTINS, R. de J. et al. Micro-service based network management for distributed applications. In: SPRINGER, CHAM. **International Conference on Advanced Information Networking and Applications**. Caserta, Italy, 2020. p. 922–933.

MAYER, B.; WEINREICH, R. A dashboard for microservice monitoring and management. In: IEEE. **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**. Gothenburg, Sweden, 2017. p. 66–69.

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

PAHL, C. Containerization and the paas cloud. **IEEE Cloud Computing**, IEEE, v. 2, n. 3, p. 24–31, 2015.

PANCHEN, S.; MCKEE, N.; PHAAL, P. **InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks**. RFC Editor, 2001. RFC 3176. (Request for Comments, 3176). Available from Internet: <https://www.rfc-editor.org/info/rfc3176>.

PROMETHEUS. **Prometheus-monitoring system & time series database**. prometheus.io, 2017. Available from Internet: <https://prometheus.io>.

R CORE TEAM. **R: A Language and Environment for Statistical Computing**. Vienna, Austria, 2021. Available from Internet: <https://www.R-project.org/>.

RAZA, U.; KULKARNI, P.; SOORIYABANDARA, M. Low power wide area networks: An overview. **IEEE Communications Surveys & Tutorials**, IEEE, v. 19, n. 2, p. 855–873, 2017.

SERIES, M. Imt vision–framework and overall objectives of the future development of imt for 2020 and beyond. **Recommendation ITU**, v. 2083, 2015.

SLAMNIK-KRIJEšTORAC, N. et al. Sharing distributed and heterogeneous resources toward end-to-end 5g networks: A comprehensive survey and a taxonomy. **IEEE Communications Surveys Tutorials**, p. 1–1, 2020.

STALLINGS, W. **Cryptography and network security, 4/E**. [S.l.]: Pearson Education India, 2006.

SUN, Y. et al. Automated attack and defense framework toward 5g security. **IEEE Network**, v. 34, n. 5, p. 247–253, 2020.

WANG, D. et al. From IoT to 5G I-IoT: The next generation iot-based intelligent algorithms and 5g technologies. **IEEE Communications Magazine**, v. 56, n. 10, p. 114–120, 2018.

WICKHAM, H. **ggplot2: Elegant Graphics for Data Analysis**. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. Available from Internet: <https://ggplot2.tidyverse.org>.

WUBBEN, D. et al. Benefits and impact of cloud computing on 5g signal processing: Flexible centralization through cloud-ran. **IEEE Signal Processing Magazine**, v. 31, n. 6, p. 35–44, 2014.

XU, J. et al. Narrowband internet of things: Evolutions, technologies, and open issues. **IEEE Internet of Things Journal**, v. 5, n. 3, p. 1449–1462, 2018.

ZHANG, C. et al. Towards a virtual network function research agenda: A systematic literature review of vnf design considerations. **Journal of Network and Computer Applications**, Elsevier, v. 146, p. 102417, 2019.

ZHANG, S.; WANG, Y.; ZHOU, W. Towards secure 5g networks: A survey. **Computer Networks**, v. 162, p. 106871, 2019. ISSN 1389-1286.

APPENDIX A — RESUMO EXPANDIDO

As redes de computadores tradicionais são relativamente estáticas no que se refere à topologia de rede, funcionalidades, e protocolos (HAKIRI et al., 2014). Estas redes utilizam-se extensivamente de *middleboxes* físicos (BRIM; CARPENTER, 2002) para realizarem funções de redes, tais como roteamento, funções de *firewall*, e de balanceamento de carga. A administração destas redes tipicamente depende de soluções de gerência de rede tradicionais baseados em protocolos como o SNMP(*Simple Network Management Protocol*) (FEDOR et al., 1990), o NETconf (*Network Configuration Protocol*) (FEDOR et al., 1990), e o Netflow (CLAISE, 2004). A adição de novos *middleboxes* a uma rede naturalmente aumenta o número total de dispositivos gerenciados, e como os *middleboxes* são implementados por hardware proprietário, a inclusão de novas funções de rede, incluindo a sua correta configuração e manutenção, em geral requer um esforço manual, e portanto custoso, da parte do operador de rede.

Em outro campo de pesquisa, *i.e.*, computação na nuvem, o paradigma de microserviços tem advogado pela divisão de aplicações e serviços em micro-modulos auto-contidos, como uma solução aos problemas enfrentados por software monolítico (DRAGONI et al., 2017; MARTINS et al., 2020c). Enquanto um software monolítico é desenvolvido e implantado como um único serviço atômico, soluções baseadas em microserviços provêm o mesmo serviço de alto nível através da cooperação de múltiplos módulos independentes. Neste caso, cada módulo deve prover uma função específica, e rodar em um *host* virtual, e a comunicação entre os módulos é utilizada para combinar as funções necessárias e entregar o serviço de mais alto nível corretamente. Alguns dos benefícios desfrutados por aplicações desenvolvidas utilizando-se do paradigma de microserviços incluem a maior escalabilidade possível para um serviço, já que apenas os módulos mais sobrecarregados precisam ser escalonados quando necessário (seja pelo aumento dos recursos computacionais utilizados pelo módulo, seja pela replicação do módulo em *hosts* adicionais), e desenvolvimento e integração constantes, já que apenas os módulos modificados precisam ser atualizados.

NFV baseada em microserviços e outros novos conceitos emergem como meios importantes para novas redes atingirem seu potencial pretendido. Neste sentido, redes móveis 5G exemplificam tais casos. Diferente das gerações anteriores, a 5G promete não apenas aumentar a taxa de transmissão de dados dos dipositivos conectados, mas também habilitar a coexistência de uma miríade de aplicações com requisitos distintos.

Para alcançar este objetivo, o *slicing* de rede da infrastrutura física subjacente permite que diversos *inquilinos* compartilhem os recursos e atinjam seus objetivos. O bem-estar do sistema depende da coexistência harmoniosa entre os inquilinos que compartilham a mesma infraestrutura (SLAMNIK-KRIJEšTORAC et al., 2020). NFV pode auxiliar no provisionamento de *slices* para os inquilinos, mas cada *slice* precisa ser individualmente gerenciado e monitorado para garantir-se que os requisitos da aplicação estão sendo atendidos. Soluções de monitoramento desenvolvidas para computação na nuvem frequentemente requerem privilégios extensivos à infrastrutura subjacente, o que não é sempre factível do ponto de vista do provedor da infraestrutura. Desenvolvidas para serviços de alto nível, tais soluções de gerência e monitoramento frequentemenete ocorrem em custos adicionais por conta de estrutura utilizada, e portanto podem ter sua adoção impedida por aplicações com requisitos restritivos[1]. Casos mais extremos como de aplicações rodando na *edge* da rede podem sofrer até mesmo do *overhead* causado pelo excedente de *containers* isntanciados. Além disso, os donos da infraestrutura e seus inquilinos podem requerer não somente monitoramento, mas também outras funções de gerência, tais como segurança, de forma transparente.

A.1 Contribuições da Dissertação

A principal contribuição desta dissertação é o SWEETEN (*a**S**sistant for net**W**ork manag**E**m**E**n**T** of micros**E**rvices-based VNFs*), um sistema desenvolvido para auxiliar provedores de 5G e inquilinos de *slices* de rede com a configuração e implantação de ferramentas de gerência de rede junto aos seus *slices* de rede. Através da adição de anotações de alto nível, usuários podem requisitar *features* de gerência aos seus serviços, e o SWEETEN pode mapear as ferramentas e configurações necessárias para realizar a gerência pedida sem outras intervenções do operador. As *features* de gerência incluem monitoramento, segurança, e gerência, que podem ser aplicados para um ou mais microserviços. Como o sistema é projeto para o gerenciamento de VNFs, ele foi desenvolvido de forma a incorrer em um *overhead* tão pequeno quanto possível no que se refere a recursos de rede e computacionais. Além disso, Processamento de Linguagem Natural (NLP) é utilizado para extrair-se anotações descritivas dos serviços instanciados pelo usuário. As anotações geradas são então utilizadas para gerarem soluções adaptadas para cada implantação, de forma que soluções de gerência mais robustas possam ser uti-

[1] <https://medium.com/@pklinker/performance-impacts-of-an-istio-service-mesh-63957a0000b>

lizadas para implantações mais robustas, enquanto soluções que priorizem baixo *overhead* podem ser produzidas para implantações mais limitadas. Adicionalmente, quando julgar necessário, um operador pode especificar parâmetros de configuração manualmente, sendo processados pelo sistema e produzindo uma solução tão alinhada quanto possível com as opções informadas.

A.2 Principais Resultados Alcançados

A eficiência da proposta é observada através de um protótipo, que é avaliado em dois estudos de caso separados. O primeiro apresenta o cenário de uma *dynamic C-RAN*, onde as funções de um *split* de rádio são monitoradas. O segundo estudo de caso foca em um *slice* de rede para um serviço de *healthcare* inteligente, onde diversos dispostivos precisam, além de serem monitorados, terem suas transmissões criptografadas por uma questão de segurança e privacidade.

Os resultados obtidos mostram que existe um ganho de expressividade importante ofertado ao usuário através do uso do SWEETEN. Assistir operadores em instanciar *slices* de rede complexos é um ponto fundamental para se atingir o dinamismo esperado de redes 5G. Os *overheads* observado pelo uso do SWEETEN referente ao tempo de instanciação, e *overhead* computacional e de rede também são avaliados. Enquanto o tempo de instanciação é notávelmente afetado pelos serviços de gerência adicionados pelo SWEETEN, este é um custo não-recorrente (*i.e.*, a instanciação do *slice*), e portanto é justificável considerando-se as *features* de gerência incluídas. Além disso, é um custo referente aos containers adicionados, que também estaria presente em uma solução produzida manualmente pelo operador. O *overhead* computacional foi negligenciável ou suficientemente pequeno para ser considerado aceitável para os serviços incluídos. No caso do *overhead* de rede, por outro lado, a inclusão de soluções de criptografia incorrem em um custo bastante significativo; neste caso, embora seja um resultado esperado (devido à criptografia adicionada), fica a critério do usuário definir se o *overhead* é aceitável ou não, para cada serviço.

APPENDIX B — PUBLISHED PAPER (AINA 2020)

Rafael de Jesus Martins, Rodolfo B. Machado, Ederson Ribas, Jéferson Campos Nobre, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. **Micro-service Based Network Management for Distributed Applications**. In 2020 34th International Conference on Advanced Information Networking and Applications (AINA), AISC 1151, pp. 922–933, April 2020. DOI: 10.1007/978-3-030-44041-1_80.

- **Title**: Micro-service Based Network Management for Distributed Applications.
- **Conference**: 34th International Conference on Advanced Information Networking and Applications (AINA).
- **Qualis / CAPES**: A2.
- **Date**: April 2020.
- **Local**: Caserta, Italy.
- **URL**: <https://link.springer.com/book/10.1007/978-3-030-44041-1>.
- **DOI**: <https://doi.org/10.1007/978-3-030-44041-1_80>.
- **Commentary**: The paper presents our initial vision for SWEETEN, proposing the overall architecture that would later be refined into the present version. The targeted audience was (owners of) high-level applications, as exemplified by the proposed use case. After this work, SWEETEN pivoted to cater for (operators of) Virtual Network Functions (VNFs), particularly in 5G scenarios. This paper also discusses the instantiable management templates necessary to provide the management solutions required by users, and how they relate to management architectures. The instantiable templates were later implemented for Kubernetes on the prototype, as presented in this thesis. A use case for a distributed web application based on docker containers is presented. The evaluation showed acceptable deployment time overhead, and negligible computational and network overhead for using the proposed solution.

Micro-service Based Network Management for Distributed Applications

Rafael de Jesus Martins, Rodolfo B. Hecht, Ederson Ribas Machado, Jéferson Campos Nobre, Juliano Araujo Wickboldt, and Lisandro Zambenedetti Granville

Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil
{rjmartins, jcnobre, jwickboldt, granville}@inf.ufrgs.br
{rodolfo.hecht, ederson.machado}@ufrgs.br

Abstract. Computer networks and their services have become increasingly dynamic with the introduction of concepts such as Network Functions Virtualization (NFV) and cloud computing. To understand and configure such a complex network to their best interests, users must use several management tools that they are not necessarily familiar with. In this paper, we present an architecture that provides network management for distributed applications as easy-to-use micro-services. Our solution is based on container virtualization technologies to offer, to the user, the maximum benefit through minimal cost. We present a proof-of-concept for our architecture through a use case. Our results show that acceptable overhead is added when deploying a solution for a distributed application, and negligible overhead is added by the management tools when the user application is under heavy stress.

Keywords: Network Management · Micro-services

1 Introduction

Conventional computer networks are relatively static in terms of physical structure with respect to network topology, functionality, and protocols. These networks extensively employ physical middleboxes to perform key network functions, such as routing, firewalling, and load balancing. The administration of such networks relies on network management solutions, based for instance on SNMP, NETconf, and Netflow, that are typically implemented through distributed architectures, following the static structure of the underlying managed network.

The adoption of plentiful middleboxes in a network increases the overall number of devices to be managed, and because middleboxes are implemented with proprietary hardware, the inclusion of new network functions, including their proper configuration and maintenance, often requires a manual, thus costly, effort from the network operator. To address these middleboxes limitations, Network Functions Virtualization (NFV) is an emerging technology that relies on virtualization to implement and deploy Virtual Network Functions (VNFs) [5]. By decoupling the proprietary hardware from the associated software, NFV

enables functions to be run on top of commodity hardware, reducing operational costs and increasing network dynamicity and scalability. To achieve that, NFV is often realized with Virtual Machines (VMs), or, recently, with emerging lightweight virtualization technologies based on containers [3]. When compared to VMs, VNFs materialized through container virtualization can be deployed faster and more efficiently [6]. Containers can create and replicate customized environments, offering isolation for running applications. Because of its enhanced performance, Docker [13] has been largely adopted in industry and academia. Nevertheless, the capacity to deploy, manage, and orchestrate NFV container-based application in network environments that are heterogeneous and dynamic remains an open challenge [15].

The dynamic nature of future networks and the ephemeral function virtualization that follows along present new challenges and opportunities for network management [7]. Likewise, the ever-growing infrastructures based on the cloud-fog-edge paradigm is inherently dynamic with respect to hosted services [11]. Moreover, the distributed nature of the cloud paradigm can be leveraged by distributed applications [4]. Cloud applications can enjoy this synergy by being designed through a micro-service paradigm, in which the application is decomposed in smaller interconnected functions [1]. As of now, the burden to deploy, configure, and monitor network management tools for any new application is on its owner, usually, and is not a light one to carry out. Picking the right management tools for each application and guaranteeing their correct functioning throughout scaling events, for example, can become a more difficult task than providing the application itself. Service providers, tenants, and end-users of cloud computing could therefore benefit from the automation of these management tasks.

In this paper, we present an architecture designed to provide network management for distributed applications as micro-services. In our architecture, a tenant or customer can pick network management tools for the network infrastructure serving one or more applications of interest when deploying those applications, and the desired tools are then deployed and configured automatically, transparently to the user. By using container virtualization to deploy the desired management tools, minimal overhead is added to the operation. To assess the feasibility of our proposal, we present an use case for an implementation of our architecture. Our results show that the deployment of a solution with the user application and the necessary network management tools adds an acceptable overhead when compared to the deployment of the user application by itself. A performance analysis of the user application under stress also indicates that negligible overhead is added by the management modules, thus being a valuable tool to understand and manage distributed applications when it is most needed.

The remainder of the paper is organized as follows. In Section II, we present the background and related work. Then, we describe the proposed architecture in Section III. In Section IV, we explain how network management architectures are mirrored in our solution templates. Then, we present a use case and discuss the results regarding the architecture's implementation in Section V. Finally, in Section VII, we present our conclusions and future work.

2 Background and Related Work

The present article proposes micro-service based network management for distributed applications. Thus, it is necessary to present background on micro-service and container as well as some particularities of the chosen implementation. Besides that, we discuss some related works.

Container is a set of one or more processes organized separately from the system. All files required for the execution of such processes are provided by a separate image. In practice, containers are portable and consistent throughout the migration between development, testing and production environments. The containers are light and start very quickly [14]. **Docker** is an example of open platform for lightweight container virtualization platform which exploit improvements in kernel-level namespace support in Linux. These namespaces provide isolation between the host and the container as well as among different containers. Docker is aided by a set of tools and workflows which can be used by developers to deploy and manage containers [8].

Micro-service is an architectural style largely based on decoupled autonomous services that can be developed, deployed and operated independently of each other. Micro-services lead to various challenges in relation to team organization, development practices and infrastructure [12]. In this context, the container architecture proves to be a feasible implementation of micro-services. This tendency to use micro-services architecture has been shown to be allied to the structure of containers in at least 5 essential reasons [1]: to reduce complexity using small services, to scale, remove and deploy parts of the system easily, to improve the flexibility of using different structures and tools, to increase overall scalability, and to improve the resilience of the system.

Ciuffoletti [2] proposed the automated specification and implementation of a monitoring infrastructure in a container-based distributed system. In this work, a simple monitoring infrastructure model was defined to provide an interface between the user and the cloud management system. This model defined a monitoring infrastructure, comprising multiple instances of two basic components, one for measurement and one for data distribution. A proof of concept demonstration was described through the Docker hub, and consisted of two multi-threaded Java applications that implement the two basic components. The reference architecture of the monitoring subsystem is composed of two entities: one that manages data, one type of proxy, another that produces data.

Jaramillo et al. [8] presented a case study to discuss how Docker can effectively help leverage the micro-services architecture with an actual working model. Our work differs from the above by meeting the challenge of observability, *i.e.*, microservices architecture needs a way to visualize the health status of all services in the system to quickly locate and respond to any problem that occurs. The architecture we present, deploy containers with network management tools associated with micro-services available in another set of containers where the applications are installed. Thus, this scheme can monitor, register and manage the network of containers of the main applications avoiding their failures, or even tracking the reason for failures through their records.

Jha *et al.* [9] carried a study on the performance evaluation of Docker containers that perform a heterogeneous set of micro-services at the same time. This study concludes that running multiple micro-services within a container is also a viable deployment option, as it gives comparable (sometimes better) performance than the baseline, except for running multiple similar types of micro-services.

Lv *et al.* [10] proposed a machine learning-based container scheduling strategy for micro-services architecture to adjust the number of containers accurately and quickly in real time, specially when the service load suddenly fluctuates. Data obtained from experiments are used to train a random forest regression model for the prediction of the required service containers in the next time window. By adjusting the number of containers to balance the load pressure of the services, the proposed algorithm saves significant time comparing to traditional algorithms as well as other machine learning algorithms.

3 Network Management as a Micro-service

The aforementioned popularization of cloud computing services and similar distributed computing architectures has enabled the growth of distributed applications. Applications and macro-services can be modularized in lower-level independent services, which are themselves interconnected in a way to provide the application with high-level functionality. For example, the service for a Wordpress Web page can be split in a service running a Web server application and another service running the required database; in another example, a critical service can be replicated among several geo-distributed computing nodes, adding load-balancing and fault-tolerance capabilities to the service with ease. Clients of cloud computing services can then leverage the scalability and robustness offered by the cloud infrastructure transparently, while maintaining their focus on the application itself.

Troubleshooting distributed applications malfunctioning, however, is rarely an easy task. In the previous Wordpress example, a slow response time from the server could be due to a problem in the web server, in the database, or in the communication between them. In this case, the application owner may be well-equipped to monitor how each service is running, but understanding the communication between them could require the installation and configuration of additional network management tools. Similarly, other network management features desired by the application owners, such as security in the form of a firewall or a deep packet inspector, would have to be deployed and configured on top of the application by its owner.

In order to aid application owners with little expertise in the network management discipline, we introduce an architecture that offers the deployment and configuration of such management tools as a micro-service for the user. Our proposal is that application owners should only specify the network management features desired, in addition to the application itself, and would have the low-level work performed by our architecture. Additionally, in order to comply with containerization and micro-services paradigms, management tools should be de-

ployed by the platform only when required, and isolation between application containers and management containers should be maintained whenever possible. With these requirements in mind, we design an architecture that is able to provide network management as a micro-service for distributed application owners. Our architecture and its functioning are presented in Figure 1.

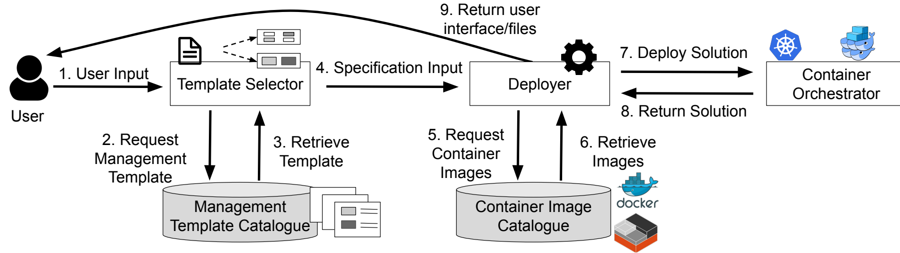


Fig. 1: Proposed architecture, including modules and their interaction.

The **User** of our architecture is the owner of a distributed application who wants to add one or more network management features effortlessly. Their input to the system is composed of a relationship of their regular application and the management features they expect for each service. The user input is passed through the Template Selector module.

The **Template Selector** is a module responsible for interpreting the user expectation regarding network management, and translating them in a relationship between the application modules and the required network management tools, which is mapped through a template. The Template Selector can be implemented to work with different abstraction levels, from low-level configuration parameters to high-level intents. In this work, we implement this module to work with low-level configuration parameters as a proof-of-concept, but the module can be expanded to include different abstraction levels with no impact in the remainder of the architecture. In our architecture, if a user's application is composed of a Web server and a database, for example, and they inform their need to monitor the latency from the Web server to their database, the Template Selector will map the request in a template that includes One-Way Ping (OWPing) and OWAMP, client and server for determining one-way latency, to be deployed and configured alongside the Web server and the database, respectively. To achieve that, the Template Selector will query the **Management Template Catalogue** for a template that fits these requirements. If such a template is available in the catalogue, a complete deployment specification containing both the user's and the management's applications will be provided. Additionally to the management tools required, every available template is also composed of a central monitoring container; when needed, this container also offers users an interface to interact with the management features they previously asked for. The following section covers network management templates specificities in-depth.

The **Deployer** receives the deployment specification and processes it, determining how containers should be distributed, which namespaces must be shared (and by which containers), and any other network configuration needed for the solution to be deployed. When the solution is produced, the Deployer queries the **Container Image Catalogue** for the images needed. Users can provide their own application images when needed, but images for some prominent network management tools are already pre-configured in the system. When the solution is ready to be instantiated, the Deployer triggers the **Container Orchestrator** to deploy the containers. Open-source platforms such as Kubernetes and Docker Swarm are example of container orchestrators, allowing the management and orchestration of Docker Engine clusters. Any additional configuration necessary is performed by the Deployer before returning a user interface to the user.

4 Network Management Architectures as Instantiable Templates

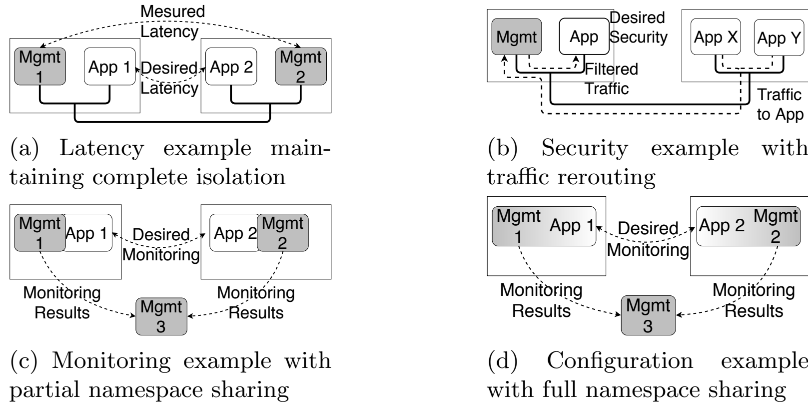
Network management is a discipline that includes, for example, network configuration, fault analysis, performance monitoring, and security assurance. The management of complex networks is thus not to be solved by a single tool but rather by the careful selection and combination of network management software. Their diverse purpose means that network management tools greatly differ with respect to their computational and architectural requirements, *e.g.*, a misplaced firewall is a useless firewall. A typical network management architecture is composed by management agents, that interact directly with managed devices to collect management information and oftentimes set configuration parameters, and a central management entity (*e.g.* an SNMP manager) that monitors and acts on the data collected by agents. Being a distributed application itself, the network management architecture can also be organized as a set of micro-services.

To fulfill user expectations regarding network management features, the appropriate set of tools must be chosen. Since requirements for management tools differ from each other, careful thought is required in their deployment. In this context, our architecture introduces *instantiable templates* for network management architectures. An instantiable template contains the information required so that management containers can be deployed alongside the user application containers, their positioning, and any other configuration needed to realize the management function correctly. Experts can develop new templates as needed, and the new templates can be fed to the architecture's template catalogue.

An important aspect that differentiate templates is with respect to container isolation. Container isolation between user application and network management tools should be maintained whenever possible. This is realized by the definition of a complete and exclusive set of namespaces for each container deployed, providing resource isolation between processes and containers running in a single system. Usually, in the micro-service paradigm, complete isolation between containers is a welcome feature. However, by carefully breaching certain isolation aspects between specific management tools and the user application they are

expected to manage, we can leverage the benefits of the micro-service paradigm while performing the network management deployment needed to realize users' desired features. In this case, two or more containers will share a subset of namespaces, allowing the network management tool to properly perform its function.

To illustrate the different template compositions, consider the following 4 common network management tasks that our system can realize (Figure 2):



(a) Latency example maintaining complete isolation

(b) Security example with traffic rerouting

(c) Monitoring example with partial namespace sharing

(d) Configuration example with full namespace sharing

Fig. 2: Set of instantiable templates, from least to most intrusive.

(a) Monitoring the latency between two containers of the user application: for latency sensitive applications, deploying instances of a tool such as OWAMP, one at each of the hosts where the application containers reside, and connecting them through the same local network as their application counterparts can be enough to provide the intended latency measures. Complete isolation between management and user application is maintained, in this situation.
(b) Securing an application through the use of a firewall: the incoming traffic must be routed through the firewall. Only the firewall position in the network is relevant to the deployment of this solution, and thus containers remain isolated. However, traffic must be rerouted and potentially modified through the new firewall function, which could affect the application performance.
(c) Monitoring all network traffic between two containers of an application: NetFlow or similar tools can be used to realize the desired monitoring. However, deploying the containerized agent tool in its own network namespace would be useless, since it would be only monitoring itself. Instead, they must be deployed in the same network namespaces (and host, therefore) of the user applications, so it can correctly perform the desired function. A collector that centralizes monitoring agents data must also be included in the template, but does not require any special isolation or positioning configuration relative to the application containers.

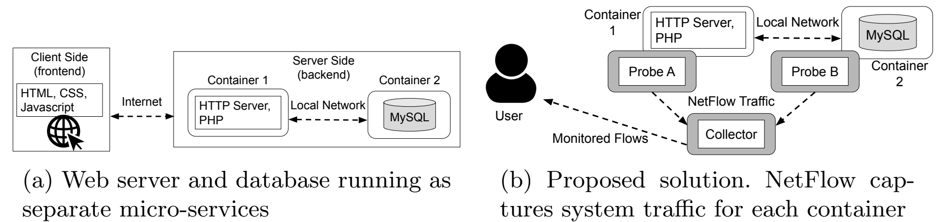
(d) Monitoring and configuring parameters of two containers of an application: this can be done through SNMP, for example, implicating a more intrusive namespace sharing between containers, since SNMP must access network, mount, and other information that would otherwise be isolated. Both application and management containers thus reside in the same set of namespaces, isolated from other systems but not from each other.

5 Use Case: Flow Monitoring for a Distributed Application

In this section, we discuss a use case for a proof-of-concept of our architecture and evaluate its results. The proposed scenario is described in Subsection 5.1 and results are discussed in Subsection 5.2.

5.1 Scenario Description

We consider the use case of a user that needs to run a simple Web application. The back-end of their application is deployed in a distributed architecture, where a Web server runs as a micro-service and responds to clients requests, and a database runs as a separate micro-service, as shown in Figure 3a. When needed, the Web server operates on the database over the network, since both services do not necessarily run on the same host. We use a simple Wordpress instance running over Apache for the Web server micro-service, and the database micro-service is deployed with MySQL. In our scenario, the user informs their desire to monitor existing network flows in both of their services.



(a) Web server and database running as separate micro-services

(b) Proposed solution. NetFlow captures system traffic for each container

Fig. 3: Proposed use case, and solution realized by our architecture.

Our architecture interprets the user's input, and identifies the need to deploy a management architecture based on NetFlow to meet the user's requirements. Based on the selected template, three containers for management will be deployed along the user application (Figure 3b). Two *fprobe* containers will each monitor one of the user's containers, and report their monitoring to a centralized collector. As per the selected template, the network namespaces for the application containers will be shared with the *fprobe* containers. The collector also

offers a processed list of network flows to the user; an experienced user can also directly interact with the management tools and logs, and perform themselves any in-depth analysis they so wish. The proposed solution is shown in Figure 3b.

5.2 Performance Evaluation

Our proof-of-concept is analyzed with respect to two performance aspects. First, we measure the time elapsed to provision of network management containers and their configuration, in comparison to having the user application being deployed by itself. Some overhead here is thus expected, albeit acceptable, since this is a one-time only cost over the life cycle of the application. Second, we must measure the network and computational overhead for having the management tools running alongside the user application. Since users are probably interested in solving bottleneck issues in their system, minimal overhead must be added by our solution for this bottleneck not to be any further narrowed.

Regarding the deployment time, the user application by itself and the complete solution have been deployed 35 times each. To assess the scalability of our solution, the experiment is extended to include an increasing number of application replicas, *i.e.*, multiple instances of the described application, which could be used by an user for fault tolerance or load balancing, for example. Figure 4 presents the time results for all cases. In this test, images for both user application and management solution are available in a local Container Image Catalogue, thus minimizing the network bandwidth effect of having the Deployer download remotely.

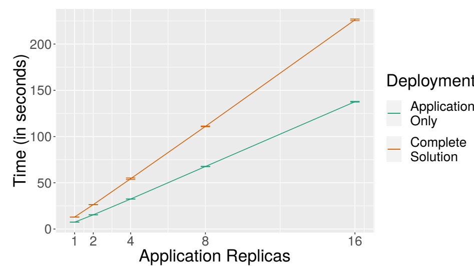
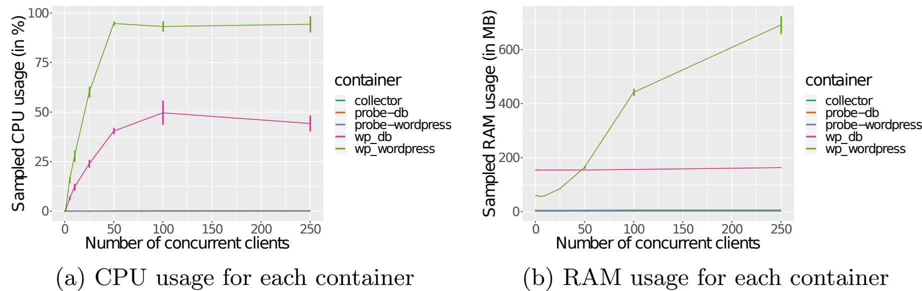


Fig. 4: Deployment time for deploying user application by itself, and with management tools included by our solution.

The results presented in Figure 4 indicate that deploying the user application by itself (*i.e.,* without additional replicas) takes on average 7.41 s to be fulfilled, whereas deploying the complete solution takes on average 12.99 s. Deploying the complete solution therefore results in a 75.3% overhead to the deployment time for a single replica, and proportionally less overhead is added when more replicas are included (64.34% for 16 replicas). The almost imperceptible error bars (for 95% confidence) also indicate the low variance observed throughout the

experiment. Although there is a noticeable overhead added, we argue that the time for deployment is a one-time cost for the user, thus offset by the benefits offered by our solution.

Another important analysis is regarding the overhead introduced for when the user application is being stressed. In this case, to assess the network and computational impact of management tools we instantiate an increasing number of clients that perform requests to the Web server. In order to evaluate how the system fares under different load levels, the number of concurrent clients increases from 0 to 250, and the interval between each client requests are chosen randomly from 0 to 2 seconds. We monitor the increase in CPU and RAM usage by all containers in our deployment, and the total traffic generated by the user application and by the management tools deployed.



(a) CPU usage for each container

(b) RAM usage for each container

Fig. 5: Computational overhead for the system under increasing stress.

Figure 5a shows the results regarding the CPU usage. It is noticeable that the CPU usage for the Web server container (wp_wordpress) explodes from the start, reaching its maximum from 50 concurrent clients onwards, and taking as much CPU resource as possible in order to process all client requests. A similar trend is observed for the database container (wp_db), albeit the maximum CPU used by it is closer to the 60% mark, and occurs at the 100 clients mark. The two results are expected, since the increase in demand for the Web server will rapidly make it consume all available resources; a fraction of these requests will also trigger some operation to the database, thus resulting in an increase for it as well. Most importantly, it is noticeable that the CPU usage by all the management containers (collector, probe-db, probe-wordpress) remains negligible and stable throughout the experiment. This result is important because it indicates that the network management solution can be useful to the application owner when they need it the most, without burdening the system performance itself.

Results with respect to the RAM usage are presented in Figure 5b. The results show for the most part a similar trend to what was observed regarding the CPU usage. RAM usage by the Web server container quickly outgrows all the others combined in order to fulfill all the clients requests. The memory used

by the database in this case is stable throughout the experiment. As with the CPU, the most important result is that the RAM usage by all the management containers is negligible and stable, regardless the number of concurrent clients.

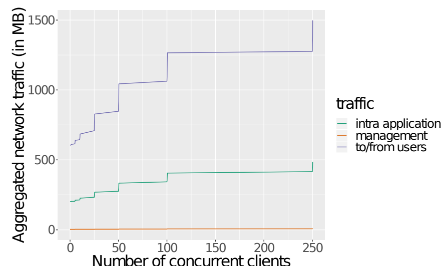


Fig. 6: Accumulated traffic between application containers, application and users, and network management tools.e

Finally, we analyze the results with respect to the traffic generated in the scenario, as presented in Figure 6. Network traffic has been divided in three groups for this analysis: *intra application*, which is the traffic between the Web server and the database; *to/from users*, which is the Web server communication with the users requesting its service; and *management*, which is all traffic generated or consumed by any of the three management containers. Results are shown in terms of total traffic being exchanged by each group, and some values do not start at 0 because traffic have been exchanged prior to the start of the experiment. The increase in the traffic to/from users is the most prominent, which is a straightforward result for the increase in clients throughout the experiment. As a secondary result, Web server and database communication increases, although not as much as in the clients case. Regarding the management overhead, it is noticeable how little impact is added to the overall communication of the system, with the traffic remaining close to 0 throughout the experiment. Paired with the previous result regarding CPU and RAM, we can conclude that the system's performance (*i.e.*, the user application) is not affected by the management tools deployed even when under stress. Thus, our solution is fit to assist application owners in dealing with malfunctions of their system.

6 Conclusions and Future Work

Network management plays an important role in the success of new, dynamic network paradigms. Therefore, the need to automate management solutions and offer them as an easy-to-use service to users is pivotal. In the case of distributed applications, understanding how modules communicate over the network can be the difference between making or breaking an application, but determining the correct tools for each case requires some network knowledge one might not have.

In this paper, we presented an architecture that can easily deploy network management tools for distributed applications. Through our architecture, application owners can indicate what type of management features they expect for each module of their application, and the selection and configuration of management tools is performed automatically. Because our solution is designed using the micro-service paradigm, negligible run-time overhead is added to the user application, and a low-cost overhead in deployment time is offset by the benefits our solution offers. In future work, we expect to further enrich our architecture, with the development of the other modules, and the improvement of the ones that are already in place.

References

1. Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., Steinder, M.: Performance evaluation of microservices architectures using containers. In: IEEE International Symposium on Network Computing and Applications, pp. 27–34 (2015)
2. Ciuffoletti, A.: Automated deployment of a microservice-based monitoring infrastructure. Procedia Computer Science **68**, 163–172 (2015)
3. Cziva, R., Pezaros, D.P.: Container network functions: Bringing nfv to the network edge. IEEE Communications Magazine **55**(6), 24–31 (2017)
4. Dikaiakos, M.D., Katsaros, D., Mehra, P., Pallis, G., Vakali, A.: Cloud computing: Distributed internet computing for it and scientific research. IEEE Internet computing **13**(5), 10–13 (2009)
5. ETSI: NFV ISG White Paper #3: Network Operator Perspectives on Industry Progress. In: SDN and OpenFlow World Congress. Frankfurt, Germany (2013)
6. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172 (2015)
7. Han, B., Gopalakrishnan, V., Ji, L., Lee, S.: NFV: Challenges and opportunities for innovations. IEEE Communications Magazine **53**(2), 90–97 (2015)
8. Jaramillo, D., Nguyen, D.V., Smart, R.: Leveraging microservices architecture by using docker technology. In: SoutheastCon 2016, pp. 1–5. IEEE (2016)
9. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Ranjan, R.: A holistic evaluation of docker containers for interfering microservices. In: 2018 IEEE International Conference on Services Computing (SCC), pp. 33–40. IEEE (2018)
10. Lv, J., Wei, M., Yu, Y.: A container scheduling strategy based on machine learning in microservice architecture. In: IEEE International Conference on Services Computing (SCC), pp. 65–71. IEEE (2019)
11. Mahmud, R., Kotagiri, R., Buyya, R.: Fog computing: A taxonomy, survey and future directions. In: Internet of everything, pp. 103–130. Springer (2018)
12. Mayer, B., Weinreich, R.: A dashboard for microservice monitoring and management. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 66–69. IEEE (2017)
13. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal **2014**(239), 2 (2014)
14. Sill, A.: The design and architecture of microservices. IEEE Cloud Computing **3**(5), 76–80 (2016)
15. Yi, B., Wang, X., Li, K., k. Das, S., Huang, M.: A comprehensive survey of network function virtualization. Computer Networks **133**, 212 – 262 (2018)

APPENDIX C — PUBLISHED PAPER (CNSM 2020)

Rafael de Jesus Martins, Ariel Galante Dalla-Costa, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. **SWEETEN: Automated Network Management Provisioning for 5G Microservices-Based Virtual Network Functions**. In 2020 16th International Conference on Network and Service Management (CNSM), CFP2066L-ART, pp. 1–9, November 2020. DOI: 10.23919/CNSM50824.2020.9269063.

- **Title**: SWEETEN: Automated Network Management Provisioning for 5G Microservices-Based Virtual Network Functions.
- **Conference**: 16th International Conference on Network and Service Management (CNSM).
- **Qualis / CAPES**: A2.
- **Date**: November 2020.
- **Local**: Izmir, Turkey.
- **URL**: <https://ieeexplore.ieee.org/document/9269063>.
- **DOI**: <https://doi.org/10.23919/CNSM50824.2020.9269063>.
- **Commentary**: The paper presents an evolution for the architecture previously proposed, now branded SWEETEN. The user base shifted from high-level user applications to Virtual Network Functions operators, particularly for 5G networks. Due to the stringent requirements pose by use cases in 5G networks, the appropriate and automated management of the network functions is essential for the future. In the implementation side, the focus is shifted towards Kubernetes and other current trends in cloud computing and similar scenarios, and a prototype is presented. A use case for a split base station, as described by Wubben *et al*, is evaluated, where network latency and throughput are monitored. The evaluation demonstrated how much work the operator can leverage using our system, and further showed acceptable overhead (in multiple axis) for using the proposed solution.

SWEETEN: Automated Network Management Provisioning for 5G Microservices-Based Virtual Network Functions

Rafael de Jesus Martins, Ariel Galante Dalla-Costa,
Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville

Institute of Informatics - Department of Applied Computing
Federal University of Rio Grande do Sul (UFRGS) - Porto Alegre, Brazil

{rjmartins, agdcosta, jwickboldt, granville}@inf.ufrgs.br

***Abstract*—Forthcoming 5G systems promise a myriad of new and improved applications, relying on Network Functions Virtualization (NFV) to realize some of 5G's stringent requirements. To guarantee that these requirements are met, network monitoring and management must be deployed and fine-tuned according each application's specificity. As Virtual Network Functions (VNFs) adhere to the microservice paradigm, picking and configuring the right tools is not a trivial task for users. In this paper, we present SWEETEN, a system that assists user to operate a 5G network with the appropriate management tools for the job, in a transparent manner to the user. By enriching their function stack with high-level annotation of the management features they desire, users can easily deploy an augmented stack with both network and management functions. A prototype is presented and evaluated in a dynamic Cloud Radio Access Network (C-RAN) split case study. The evaluation confirms that SWEETEN can assist users in effortlessly deploying complex management solutions, while incurring in acceptable deployment time overhead and negligible computational overhead for throughout the functions life-cycle.**

I. Introduction

Network Functions Virtualization (NFV) has quickly become a staple paradigm in the networking field. By virtualizing network functions – previously only offered by hardware-specific middleboxes –, NFV can offer the dynamism and scalability required by modern applications [1]. As networks service provisioned by such functions are vital for networks' health, properly managing and monitoring Virtualized Network Functions (VNFs) become a mandatory concern in assuring its proper operation.

In another field, *i.e.*, cloud computing, the microservices paradigm advocates towards breaking down applications and end-services in small self-contained functional modules [2]. Management tools, such as Prometheus [3] and Dynatrace[1], have emerged in this context, offering monitoring solutions to the cloud environment and applications following the microservices paradigm. These tools allow monitoring of cloud systems in varying scales, from a single module to an inter-cloud distributed application. Particularly large applications can leverage service mesh solutions to manage connectivity (and the management concerns that comes with it) among the microservices it comprises [4].

NFV, as defined by ETSI in the Management and Orchestration (MANO) specification [5], can potentially benefit from the adoption of the microservices paradigm. Modularizing the VNFs forming a Service Function Chain (SFC) can offer similar benefits to those enjoyed by higher-level distributed cloud applications [6]. Moreover, ETSI's NFV specifications also define that sub-sets of VNF's functionality are implemented by atomic VNF components (VNFC), which themselves can also be designed following the same principles, further benefiting the compound VNFs. However, monitoring and management solutions tailored for cloud applications may not be directly applied to NFV scenarios due to their specificities.

Forthcoming 5G communication systems exemplify some of the specific scenarios that would require tailoring of the aforementioned monitoring and management solutions. Unlike previous mobile generations, 5G promises not only to improve the data transmission rates but also to enable the coexistence of a myriad of applications with distinct requirements. To achieve that, network slicing of the underlying infrastructure should allow several tenants to seamlessly share resources and achieve distinct (potentially conflicting) objectives. The overall system's health relies on the harmonic coexistence of tenants sharing the same infrastructure [7]. NFV can assist in the provisioning of slices for tenants, but each slice must be individually managed and monitored to guarantee that the tenant's application requirements are being met. Previously cited cloud-based monitoring tools often require extensive privileges on the underlying infrastructure to work, which is not wanted or even feasible from the viewpoint the infrastructure provider. Moreover, infrastructure owners and tenants may require not only monitoring, but also other network management features (*e.g.*, security and configuration) transparently.

In this paper, we propose SWEETEN (a**S**sistant for net**W**ork manag**E**m**E**n**T** of micros**E**rvices-based V**N**Fs), a system designed to assist 5G service providers and

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

978-3-903176-31-7 © 2020 IFIP

tenants with configuration and deployment of network management tools along a network slice. By adding high-level management features annotations to their original services stack, SWEETEN can map the necessary tools and configuration to realize the desired features with no hassle for the operator. Because the system is targeted towards VNF management, it is designed to incur in the least overhead possible regarding network and computational resources. Available features include monitoring, managing, and securing one or multiple microservices. When deemed necessary, operators can specify as many configuration parameters as needed and the system will process them to deliver a solution as aligned as possible with the informed options.

We also present a prototype implementation of SWEETEN, which is evaluated in a dynamic cloud Radio Access Network (C-RAN) case study. In this case study, LTE radio functions are split in five containers, as described by Wubben *et al.* [8]. The prototype is used to manage each function and monitor them assuring the radio requirements are being met. Since the stack for the virtualized radio functions does not need to be altered prior to being fed to the system, continuous development and integration of the virtualized functions is not an impediment for our system, as the updated functions and their respective network management tools are updated transparently. We evaluate our system through the prototype, which indicates that acceptable overhead is added to the deployment time of the complete solution, and negligible computational and network overhead is added throughout the remainder of the lifecycle.

The remainder of the paper is organized as follows. In Section II, we present some background information on microservices and container-based virtualization and discuss related work. In Section III, we introduce SWEETEN's architecture discussing its main features and detailing our prototype implementation. Then, in Section IV, we present a case study of a 5G application scenario featuring a microservice-oriented software radio design split into five containers. We discuss the experiments performed and obtained results when evaluating our system prototype in Section V. Finally, in Section VI, we present concluding remarks and perspectives of future work.

II. Background & Related Work

The microservice paradigm has emerged in the context of cloud computing as a solution to the problems faced by monolithic software [9]. While monolithic software is developed and deployed as a single atomic service, microservice-based solutions provide the same high-level service through the cooperation of multiple independent modules. In this case, each module should provide a specific function and runs in a virtual host, and the communication between modules is used to combine the necessary functions and deliver the service correctly. Some possible benefits enjoyed by applications designed with the microservice paradigm includes fine-grained scaling of a service, since only the overloaded modules need to be scaled up or out, and continuous development and integration, since only the updated modules must be upgraded.

A microservice needs a virtual host to run on, which is typically realized through container virtualization. Containers offer a lighter alternative to Virtual Machines (VMs), since they can run directly on the host system without requiring virtualization layers for an Operating System (OS) that introduces overhead in the process [10]. In this case, the host system's kernel offers resource isolation features that restrain each container to their own environment. In Linux, these features are mostly achieved through *cgroups* and *namespaces* features. Containers orchestrators can be used to deploy and manage complex applications (*e.g.*, composed of multiple containers, deployed over multiple hosts). Docker is an example of a platform for lightweight container virtualization, while Kubernetes currently stands out as one of the most widely used container orchestration platforms [11].

Ciuffoletti [12] investigates the specification and automation of monitoring infrastructures in a container-based distributed system. The author employs an architecture for monitoring that is comprised of two entities: a sensor, that produces and delivers measurements, and a collector, that specializes the management of those measurements. A simple model that interfaces the user and the container management system is defined, and a prototype implementation that showcases the applicability of the proposal is provided. The work thus focuses solely on the monitoring aspect, while our proposal also covers other management disciplines (*e.g.*, security). Additionally, Ciuffoletti's work considers that the user application and the sensor for the monitoring system run in the same container, which violates the microservices paradigm and can hinder the module development and deployment. Our solution, instead, always considers the application and management microservices as separated containers, even when context sharing is needed, and thus does not breach microservices standards unnecessarily.

The work of Jaramillo *et al.* [13] discusses how Docker can effectively leverage the microservices paradigm through a case study based on a real working model. The authors pose a list with six challenges faced when building a microservice architecture and that contrasts with the multiple advantages offered by microservices design. Specifically, the work highlights the necessity of improvements regarding scalability, automation, and observability. SWEETEN is designed with these challenges in mind, providing automated observability (*i.e.*, a way to visualize health status of microservices to quickly locate and respond to occurring problems) and scalability (*e.g.*, dynamic configuration for multiple microservices), features meeting operators needs.

Li *et al.* [4] reviewed the state-of-the-art and the challenges for service meshes. Service meshes are emerging solutions that create a dedicated infrastructure layer for handling communication between microservices. Service meshes can offer multiple features such as service discovery, load balancing, and access control. Implementations for service meshes typically rely on deploying an array of network proxies

alongside primary containers, intercepting all its connections to provide the features. As pointed out by the authors, edge computing environments and 5G scenarios (*e.g.*, multi-tenancy) incur in specificities that service meshes are not designed to cope with. SWEETEN is designed to work with VNFs with different requirements (*e.g.*, minimal overhead for edge deployments), and management applications (and thus, overhead) are chosen and configured according to high-level user requests and other deployment specifics.

Franco *et al.* [14] introduced a support tool for cybersecurity that focuses on the recommendation of protection services. The authors argue that although a vast number of protections services are offered to network operators and users, the choice for one or more is not trivial for neither. Like SWEETEN, the proposed system can operate with different demands from the user, and recommends protection tools (in their case) for different scenarios. Notably, the proposal is limited in scope to the security discipline, while SWEETEN is designed to consider other network management disciplines.

III. System Design and Implementation

In this section, we discuss the system architecture and our design choices. SWEETEN's architecture is an evolution of the one previously proposed [2], but in this case specifically shifting the system target towards different users and use cases, namely from cloud applications to containerized VNFs and their operators, incurring in important system choices as described in the following.

Because NFV is critical for upcoming networks, and since the paradigm shift from monolithic VNFs to microservices-based ones is imminent [6], the burden has increased for network operators. The applications that run on top of a VNF or an SFC can pose stringent requirements for the functions, and the underlying infrastructure can also determine how functions should be managed. The proposed system should reflect the specificities of each scenario in order to properly select and configure the necessary management tools. The system architecture targets minimizing the operator effort in specifying the tools and configurations necessary, while still allowing them to be manually specified when needed.

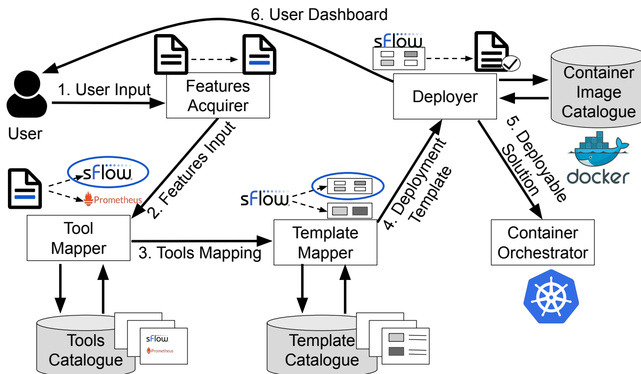


Fig. 1. SWEETEN architecture.

SWEETEN architecture is presented in Figure 1. The user is a VNF operator that augments his/her initial deployment definition (*i.e.*, the containers that compose the VNFs) with specification for the management features expected to attain. Management features are specified in three categories: monitoring, security, and administration. Each category provides a list of features that users can specify in their requirements. A non-exhaustive list of features and respective tool selection is presented in Table I.

The existence of multiple tools when mapping a single feature is resolved by the system based on additional user input and deployment information. A single tool can also realize multiple features (*e.g.*, SNMP, for monitoring as well as for administration), which SWEETEN takes into account when selecting the appropriate tools for a deployment. Currently supported features are latency and flows, for monitoring; traffic, for security; and device, for administration. Support for new features (and through different tools) can be added to the system by an expert. The user can adopt varying specification levels when requiring the management features for each service (*e.g.*, choosing to only monitor TCP connections on certain ports, or determining what tool should be used). The system interprets the requested features and maps the appropriate tools and configurations to fulfil all requests.

TABLE I
Network features and respective tools listings.

	Monitoring	Security	Administration
Flows	sFlow, NetFlow, Prometheus	Snort (for IDS), OSSEC	-
Traffic	Prometheus, iPerf, SNMP	iptables, nftables	Linux tc
Latency	SmokePing, OWAMP, TWAMP	-	Linux tc
Device	Kubelet (Kubernetes native)	syslog, antivirus utilities	NETCONF, SNMP

First, the **Features Acquirer** parses the input specification, determining what management features are required by each service. The features definition is passed on to the **Tool Mapper**, which determines what tools are required so that the required features can be materialized. As aforementioned, operators can employ varying abstraction levels when requesting management features, specifying tools, and even configuration parameters when necessary. In this case, the Tool Mapper fixes the selections accordingly.

Determining the non-specified tools and configurations is achieved by querying the **Tools Catalogue**, which provides the options of management tools sets that are able to realize the features specification. Each tool may also provide additional information with respect to its operation (*e.g.*, overhead, scalability). From these tools options, the Tool Mapper selects the appropriate set of tools considering the remaining of the user specification and other deployment specificities. For example, edge deployments may require minimal overhead, and thus the tool selection must reflect that; conversely, complex cloud deployments may prioritize more sophisticated

tools that can provide higher-level utilities, incurring in a different tool selection.

The selected tools along with the remainder of the specification input are passed on to the **Template Mapper**, which maps the tools to instantiable templates from a **Template Catalogue**. This step is necessary to provide a deployment template based on a tool's architecture, and that later can be deployed on top of the user's application. In the sFlow example, its architecture determines that agents must be deployed with monitored entities, and a collector must be deployed to aggregate agents' metrics. Each agent therefore strongly depends on the entity it must monitor, including the need of sharing network context with the monitored entity. The collector, however, does not have such requirement, and thus its deployment is much more flexible. A deployment template is thus generated with an specification on how to deploy the selected tools along with the user's application, and passed on to the Deployer.

The **Deployer** is responsible for piercing together the deployment specification based on the determined management templates and the user application. While the previous components extract the management information from the initial input and provide a template for deploying the necessary tools, the Deployer composes a deployable specification with all the containers for application and network management. A central management container running a customized dashboard for the user is also deployed for the user's convenience. Images for the required containers are fetched from the **Container Image Catalogue**, and the complete solution is then fed to a **Container Orchestrator**, such as Kubernetes, and the resulting deployment (in the form of a customized user dashboard) is returned to the user.

A. *Prototype's software choices*

Microservices typically run inside lightweight containers. Docker containers orchestrated by Kubernetes have for the past few years emerged as the most prominent combination for running such complex applications [15], even more so with the discontinued support for alternatives like Docker Swarm [2], and therefore constitute the main container platforms for our system. The rich ecosystem and widely adoption from industry and academia alike, on the one hand, favours the system's development, and on the other hand, favours its adoption by the wide public.

In the Kubernetes architecture, containerized microservices run in entities known as pods. Pods serve as a logical host for containers, having them shared storage and network, and being scheduled and deployed together. Pods primary motivation is to support helper programs (*e.g.*, loggers, managers) for the primary container, thus offering a compromise between the microservice paradigm (*e.g.*, decoupled dependencies for each container) and the monolithic benefits (*e.g.*, shared context for monitoring). When mapping our architecture templates into Kubernetes, management containers are deployed in the same pods as the managed containers whenever network context sharing is mandatory for the management tool to perform appropriately.

In Kubernetes, the deployment specification is typically realized by one or more YAML configuration files [16] that specify how the service is composed. For each microservice defined, operators add tags for the management features they expect to attain. As previously discussed, operators can employ different abstraction levels when requesting for management features. In that way, an experienced user can go into lower-level configuration specification for how the features should be realized, while a novice user can be less specific and still obtain a proper management solution. An example for the latter is presented in Listing 1, where the deployment specification simply determines that TCP flow monitoring must be included for that deployment. The system interprets the requested feature and maps the appropriate tools and configurations to fulfil the request. In this example, the request would be mapped to a template using sFlow [17], even if Prometheus and other equivalent monitoring tools could also fulfil the same feature request, depending on the deployment requirements. The deployment produced by SWEETEN returns a simplified user dashboard, as exemplified in Figure 2.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    management:
      monitoring:
        - flows:TCP
    ...
```

Listing 1. Summarized example of feature request through annotations by a novice user.

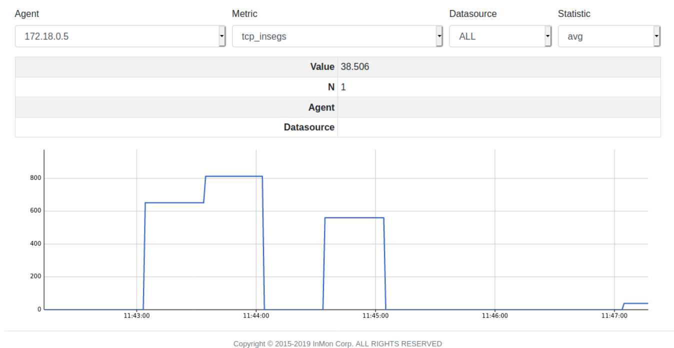


Fig. 2. User dashboard generated from the novice user's specification.

An example for a more deterministic specification that an experienced user could request is presented in Listing 2. Similarly to the previous example (for the novice user), monitoring features are requested. However, unlike the previous example, the user is much more specific in the requests. In this example, the monitoring tool has been specified (*i.e.*, Prometheus), including the need for a specific

[2]https://www.mirantis.com/blog/mirantis-acquires-docker-enterprise-platform-business/

version. The *scrape_interval* is specified to be set at five seconds. Finally, rather than using the default expression browser for visualization, the user uses nesting specification that *Grafana* should be used for visualization and that its dashboard must listen in the 3030 port (instead of the default 3000). SWEETEN processes the entire user request and adjusts the tools and templates mappings accordingly, resulting in a more fine-tuned user dashboard, as exemplified by Figure 3.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    management:
      monitoring:
        - flows:TCP
          tool: Prometheus
          version: 2.18.0
          scrape_interval: 5s
          dashboard:
            - tool: Grafana
              http_port: 3030
          ...
```

Listing 2. Summarized example of feature request through annotations by an experienced user.

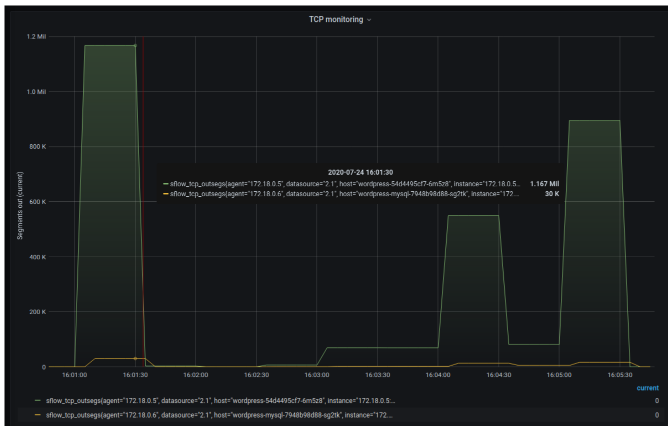


Fig. 3. User dashboard generated from the experienced user's specification.

Other solutions such as service meshes typically work by appending a sidecar proxy to all containers of interest, *i.e.*, a separate container in the same pod that proxies all connections to and from the primary container, adding management functionalities as needed [4]. Contrarily to that, our system only adds containers to the same pod (and intercepts connections) when the desired management features require so (for example, security functionalities that must filter the incoming/outgoing packets), and as indicated by the Management Template Catalogue. When this requirement is not present, appended management functions can reside in the same pod but not proxying the main container connections, or even reside in a separate pod altogether. An example of the former would be for some passive monitoring functions, with the benefit of lessening the communication overhead from keeping the hop count as low as possible. An example of the latter would be for some active monitoring functions, such as determining the latency between containers located in separate nodes in a cluster, and that thus can be monitored by having the management pods be placed on the same nodes while keeping their context completely independent from the primary container.

Our prototype was implemented using Python v2.7.17 for the main components in the architecture. Each component (*i.e.*, Features Acquirer, Tool Mapper, Template Mapper, Deployer) was developed as an independent module, and Kubernetes v1.18.5 was used without modifications as the Container Orchestrator. Some minor functions (*e.g.*, getting nodes information for a Kubernetes cluster) were implemented through shell scripts. Python library `PyYAML` v5.3.1 [3] was used to read the user input specification, which is then parsed by the Features Acquirer module, and to later write the solution specification that is deployed with Kubernetes. The Tools Catalogue and the Template Catalogue are both materialized through YAML configuration files. That is so because, on the one hand, the format's readability facilitates the inclusion of new items by experts, and on the other hand, it simplifies the generation of a deployable specification from the templates, since the language is used by the deployment specification itself. Publicly available dockerhub repository [4] was used as the Container Image Catalogue, and Grafana v7.1 [5] is used to produce the customized users dashboard for monitoring functions.

IV. Case Study: 5G Radio Split

Traditionally, network mobile services have been provided by a mobile network operator (MNO). Recently, mobile virtual network operators (MVNOs) have emerged as an alternative for customers. The new providers do not own the physical wireless infrastructure, and must thus lease it from traditional MNOs. Mobile services in turn can be delivered through cloud computing. The various strategies adopted by MNOs can benefit customers and the provider alike [18].

In this case study, an MVNO must allocate a number of virtualized Base Stations (BSs) over a region. Being a dynamic C-RAN adopter, the provider makes use of Remote Radio Heads (RRHs) that have their signals processed by Base-Band Units (BBUs). Each BBU is comprised of five forwarding elements: I/Q, Subframe, RX Data, Soft-bit, and MAC [8]. These elements have stringent requirements regarding bandwidth between the elements and end-to-end latency, as shown in Figure 4. In particular, delay requirements limit the maximum distance between an RRH and its BBU, in a relationship that depends on the channel condition and the processing power available [19]. To assert its compliance to the service terms, the provider must properly monitor each BBU closely in order to avoid any violation, with the monitoring overhead itself being kept at minimal levels.

[3]https://pyyaml.org/
[4]https://hub.docker.com/
[5]https://grafana.com/

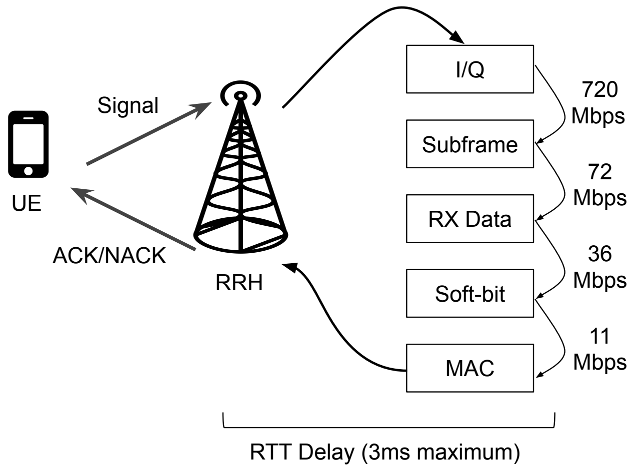


Fig. 4. Communication flow and bandwidth requirements for radio functions.

The provider must instantiate 15 BSs for a region. To do so, the BBU functions must be allocated along with a central cloud, a regional cloud, and a fog. To maximize the computational resources used both in clouds and in the fog, and to minimize the front-haul data rate, the placement algorithm prioritizes running all BBUs' I/Q and Subframe functions as near to the fog as possible, since both functions are responsible for the majority of the front-haul data rate. The remainder of the BBU functions should be placed on the regional and the central clouds, prioritizing the latter due to its increased computational capacity, whenever latency permits it. Additional functions (such as management entities) should run on the central cloud whenever possible, as to not overload the fog and the regional cloud unnecessarily. The resulting placement for the elements of the 15 BSs is shown in Table II.

TABLE II
RESULTING DISTRIBUTION OF BSs FUNCTIONS.

	I/Q	Subframe	RX Data	Soft-bit	MAC
Fog	5	5	5	0	0
Regional Cloud	5	5	5	5	0
Central Cloud	5	5	5	10	15

Being the owner of the BS application, the service provider is capable of managing and monitoring each container appropriately. However, the network monitoring is less trivial and it depends on external factors, and due to the stringent requirements, it needs to be properly done. The provider uses our system by tagging the required management features (*i.e.*, the latency and traffic monitoring for all containers) in the deployment specification for the BSs, and SWEETEN deploys the complete solution which includes the tools to realize the required management.

V. Experiment and Discussion of Results

We assert SWEETEN qualities through the proposed use case in two aspects: the management benefits offered and the overhead introduced. It is noteworthy that 5G applications can require diverse management features, and thus results for different use cases can incur varied benefits and overhead. For example, an e-health application can pose stringent requirements regarding availability and mobility [20], and therefore must be reflected on the network management solutions selected and configured by SWEETEN.

Our first analysis regards the expressiveness gains for the operator. Because Kubernetes does not intend to interpret high-level feature requests, our system naturally outperforms what would be required from the operator manually. In this case study, it takes the operator only four lines of high-level feature specification to trigger the deployment of four additional management containers (plus two for each subsequent microservice), as defined by their deployment templates. If the operator were to do it manually, the operator would have to input an additional 157 new lines of specification for the first BS, plus 100 reoccurring lines for each subsequent BS. Even if Kubernetes specification is not designed for this purpose, it would be the available alternative prior to our system. Being able to do more with less is a recurrent concern for operators [21]. Most important, the labour of including these specification lines pales in comparison to the one of determining what should be in the lines in the first place. Realistically, hours of work would be spent in finding the correct tools for the job, and properly configuring them manually for the deployment at hand.

Our second analysis regards the deployment overhead in utilizing our solution. To do so, we evaluate the time it takes to deploy the 15 BSs in their initial minimalist state (*i.e.*, with no added management features), and with the complete solution produced by our system. In each case, experiments were run 30 times. The results are presented in Figure 5.

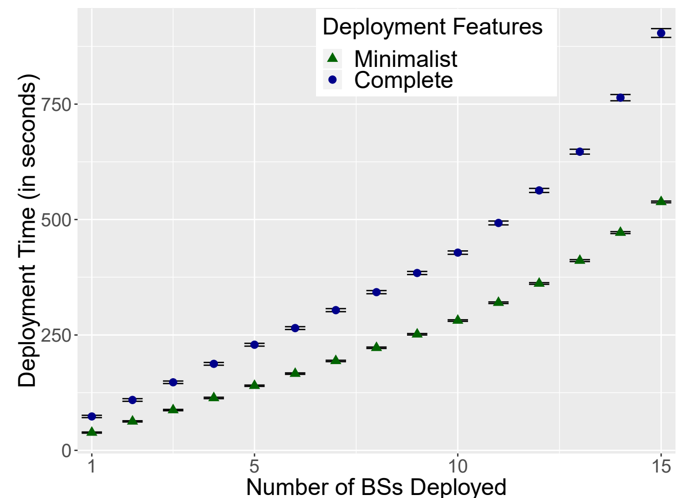


Fig. 5. Time taken to deploy up to 15 BSs, with and without using our system.

On average, the complete deployment took 59.6% more (about 145 seconds) than the minimalist deployment. The

evolution of the experiment shows a similar linear pattern for both cases in the earlier stages (*i.e.*, less than 10 BSs). The latter stages shows a disproportional increase in the complete solution in comparison to the minimalist approach. The large number of pods and containers take their toll in the container orchestrator, highlighting the importance of considering the deployments specificities when determining the correct management solutions. Moreover, virtually all of the overhead was due to the additional containers Kubernetes had to deploy and launch, meaning users would still incur in comparable costs if they were to produce a similar solution by other methods. Finally, this cost is regarding the deployment of all BSs from scratch, and therefore not a recurring cost.

Next, our third analysis focuses on the computational overhead for the remainder of the deployment life-cycle. To assess the CPU usage by management entities included in our deployment, we evaluate the impact of scaling from one to four BSs in a single VM. In this analysis, having all the containers run in a single VM offers a fair comparison for the overhead introduced by each management entity in the architecture. The results presented in Figure 6 show how the management entities consume negligible processing for the most part. The collector consumes approximately the same CPU as all the agents combined, but still sits at just over 3.5% for four concurrent BSs. Moreover, since the collector has no strict placement constraints (it only requires to be reachable by the agents), it can be placed in the more resourceful nodes in a deployment with little impact on deployment performance. Between the two types of monitoring agents, it is possible to note that traffic monitoring consumes significantly more CPU than latency monitoring. Still, the sum of all agents for each BS comes at approximately 1% CPU usage, thus the overhead is largely negligible for distributed deployments along the cluster.

The results for memory usage and network overhead follow closely. An average RAM usage of 1.54GB for running the user dashboard and metrics collector, plus 7.38MB per microservice managed (totalling 36.94MB per BS). The dashboard and collector increased cost are justifiable because they are a unique cost for the entire deployment, and its independence means it can be deployed in the (resourceful) central cloud. In turn, the computational overhead per BS due to management agents is mostly negligible, which not only is imperative due to the stringent requirements of the BS functions but also highlights the scalability of the solution. Regarding the network aspect, management agents introduce an overhead of around 5KB/s for incoming and outgoing traffic per BS. Around 30% of the overhead is due to the latency monitoring probes required for the active measurements. The remainder is mostly due to the periodic reports from agents to the collector. An advanced user could fine-tune parameters to their needs when requesting the features. For example, by increasing reports' scrape time, it is possible to further minimize the communication overhead or decreasing it could allow one to monitor sub-second variations closely.

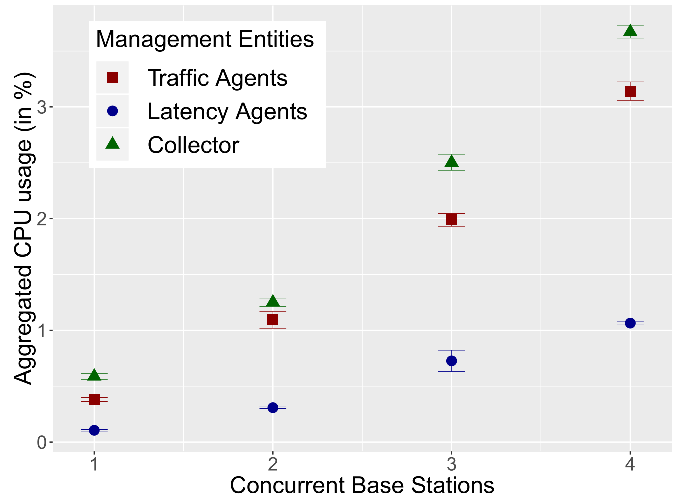


Fig. 6. CPU usage (in percent) for management containers for up to four concurrent (same VM) base stations.

Finally, our fourth analysis showcase the monitoring results that would assist the operator in detecting network issues when they occur. Figure 7 shows an excerpt for the customized dashboard that the user receives after a complete deployment. For simplicity, the monitoring for a single BS is presented. The dashboard consolidates requested monitoring metrics in a dynamic interface that allows the user easy access to the relevant metrics. As explained previously, the user can be more specific in their requests in order to obtain a solution more fine-tuned to their needs. For presentation purposes, the monitoring graphs for the results discussed in the following were re-plotted for specific BSs. Figure 8 exemplifies the result for latency monitoring, while Figure 9 does the same for the traffic monitoring.

The latency result in Figure 8 shows the monitoring for two BSs prior and after additional BSs are deployed. The first BS (in green) is deployed over fog (I/Q, Subframe, and RX Data) and regional cloud (Softbit and MAC), while the second BS (in blue) is fully deployed in the central cloud. Prior to the deployment of additional BSs (marked by the vertical line), no latency violations (marked by the horizontal line) are detected for any of the BS. After the deployment of two new BSs (over the three clusters), instability incurs in several violations (four in the figure) for the first BS, while none are for the second BS. The monitoring result alerts the operator that the new deployments are negatively impacting the first BS, and corrective actions must be taken.

Figure 9 shows the result for the traffic monitoring for a BS I/Q data rate in two moments, for a total period of 300 seconds. In a first moment, three other BSs are deployed, and the monitoring results indicate that achieved data rates are in accordance with the function's requirements. In a second moment (by the 160 seconds mark), eight new BSs are deployed over the same regions as the monitored BS (vertical dashed line in the graph). The monitoring shows how the I/Q

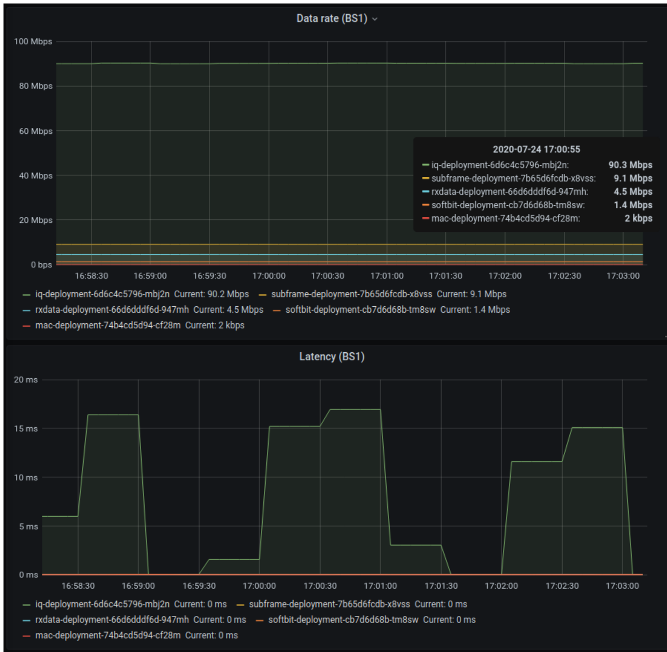


Fig. 7. Excerpt from user dashboard enabling the requested monitoring features.

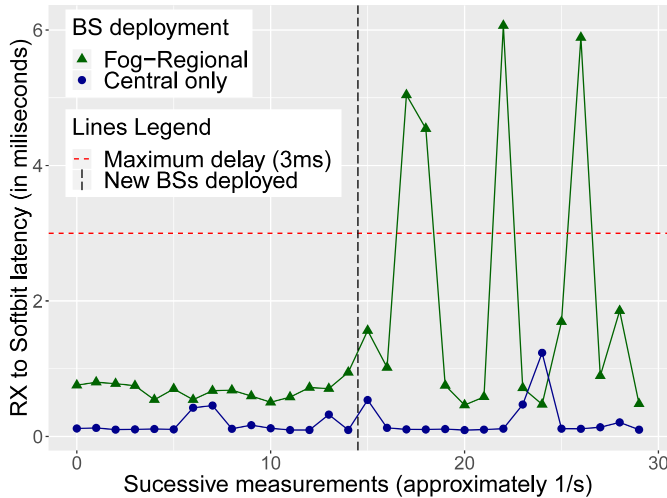


Fig. 8. Latency monitoring result for two BSs' RX to Softbit communication over a 30-second window.

throughput for the BS declines as a result. In this case, the operator can pinpoint the BS malfunctioning to the bottleneck created by the additional BSs deployed, and that pushed the infrastructure beyond its limits.

VI. Conclusions and Future Work

NFV plays a major role in new networks such as 5G, and thus it is imperative that they are properly managed. The diverse requirements that VNFs can present, their redesign following microservice paradigm, and the different scenarios that must be contemplated, makes choosing and configuring the right management tools appropriately a

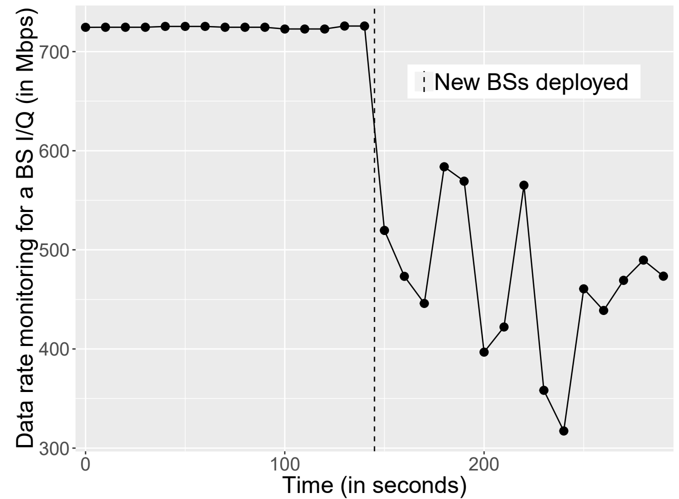


Fig. 9. Data rate result for monitoring a BS's IQ prior and after new BSs deployment.

non-trivial task to their users. We propose SWEETEN, a system designed to assist microservices-based VNFs users in including network management features to their deployments. Users augment their deployment specification with high-level annotations, which SWEETEN architecture maps into tools and configurations that complies to deployments specificities, producing a deployable specification that materializes the user's management needs.

We evaluate SWEETEN with a prototype in a dynamic C-RAN case study. Primarily, the results show that effort from the operator to configure and deploy management tools appropriately is greatly reduced. Our results also indicate that non-negligible overhead is added to the deployment time of the complete solution, but since new deployments are infrequent the added overhead is considered acceptable. Additionally, negligible computational overhead is added throughout the remainder of the services life-cycle, which is imperative due to the stringent requirements of the functions deployed. The management features added are shown to assist the operator in monitoring the correct functioning of their deployments.

As future work, we intend to continue developing the system with the support for new network management features and the development of the accompanying templates. Due to the diversity of management features and tools, and the ubiquity of Kubernetes in different environments, covering distinct use cases (*e.g.*, IoT devices and edge deployments) can enrich the system and its usefulness to a wider audience.

Acknowledgment

This study was partially funded by CAPES - Finance Code 001. We also thank the funding of CNPq, Research Productivity Scholarship grants ref. 313893/2018-7 and 312392/2017-6. This research was also partially funded by project ref. 423275/2016-0 from CNPq entitled "NFV-MENTOR (NFV ManageENT & ORchestration)."

REFERENCES

[1] S. Marinova, T. Lin, H. Bannazadeh, and A. Leon-Garcia, "End-to-end network slicing for future wireless in multi-region cloud platforms," *Computer Networks*, vol. 177, p. 107298, 2020.

[2] R. de Jesus Martins, R. B. Hecht, E. R. Machado, J. C. Nobre, J. A. Wickboldt, and L. Z. Granville, "Micro-service Based Network Management for Distributed Applications," in *34th International Conference on Advanced Information Networking and Applications (AINA)*. Springer, 2020, pp. 922–933.

[3] Prometheus Authors, "Prometheus-monitoring system & time series database," 2017. [Online]. Available: https://prometheus.io/

[4] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in *13th IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–127.

[5] ETSI, NFVISG, "GS NFV-MAN 001 v1.1.1 Network Function Virtualisation (NFV); Management and Orchestration," 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[6] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba, "Re-Architecting NFV Ecosystem with Microservices: State of the Art and Research Challenges," *IEEE Network*, vol. 33, no. 3, pp. 168–176, 2019.

[7] N. Slamnik-Kriještorac, H. Kremo, M. Ruffini, and J. M. Marquez-Barja, "Sharing Distributed and Heterogeneous Resources toward End-to-End 5G networks: A Comprehensive Survey and a Taxonomy," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1592–1628, 2020.

[8] D. Wubben, P. Rost, J. S. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy, and G. Fettweis, "Benefits and Impact of Cloud Computing on 5G Signal Processing: Flexible centralization through cloud-RAN," *IEEE Signal Processing Magazine*, vol. 31, no. 6, pp. 35–44, 2014.

[9] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[20] J. Lloret, L. Parra, M. Taha, and J. Tomás, "An architecture and protocol for smart continuous eHealth monitoring using 5G," *Computer Networks*, vol. 129, pp. 340–351, 2017.

[10] R. de Jesus Martins, C. B. Both, J. A. Wickboldt, and L. Z. Granville, "Virtual Network Functions Migration Cost: from Identification to Prediction," *Computer Networks*, vol. 181, p. 107429, 2020.

[11] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[12] A. Ciuffoletti, "Automated deployment of a microservice-based monitoring infrastructure," *Procedia Computer Science*, vol. 68, pp. 163–172, 2015.

[13] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *SoutheastCon 2016*. IEEE, 2016, pp. 1–5.

[14] M. F. Franco, B. Rodrigues, and B. Stiller, "MENTOR: The Design and Evaluation of a Protection Services Recommender System," in *15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–7.

[15] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container Orchestration Engines: A Thorough Functional and Performance Comparison," in *53rd IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[16] O. Ben-Kiki, C. Evans, and B. Ingerson, "YAML Ain't Markup Language (YAML™) Version 1.1," *Working Draft*, vol. 11, 2009. [Online]. Available: https://yaml.org/spec/1.1/index.html

[17] P. Phaal, S. Panchen, and N. McKee, "RFC 3176 - InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," 2001.

[18] N. Kamiyama and A. Nakao, "Analyzing Dynamics of MVNO Market Using Evolutionary Game," in *15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–6.

[19] M. A. Marotta, H. Ahmadi, J. Rochol, L. DaSilva, and C. B. Both, "Characterizing the Relation Between Processing Power and Distance Between BBU and RRH in a Cloud RAN," *IEEE Wireless Communications Letters*, vol. 7, no. 3, pp. 472–475, 2018.

[21] A. Curtis-Black, A. Willig, and M. Galster, "Scout: A Framework for Querying Networks," in *15th International Conference on Network and Service Management (CNSM)*. IEEE, 2019, pp. 1–7.

APPENDIX D — SUBMITTED PAPER (COMNET)

Rafael de Jesus Martins, Juliano Araujo Wickboldt, Lisandro Zambenedetti Granville. **Assisted Monitoring and Security Provisioning for 5G Microservices-Based Network Slices with SWEETEN**. Submitted to Elsevier Computer Networks (ComNet), currently under review.

- **Title**: Assisted Monitoring and Security Provisioning for 5G Microservices-Based Network Slices with SWEETEN.
- **Journal**: Elsevier Computer Networks.
- **Qualis / CAPES**: A2.
- **Commentary**: This paper presents the next stage for SWEETEN, where the scope is extended in various axis. First, not only the network managament of VNF's is considered, but also the management for complete network slices. Second, in addition to the monitoring features evaluated in previous papers, security features are also included in the discussion and evaluation in this study. Third, NLP is included in the system's workflow so that the system can intelligently retrieve tags for each service, determining what kind of service it is and its inherent characteristics. The prototype is evaluated through a intelligent healthcare case study, and results corroborate to what was observed in the previous papers, with important additions specially regarding security features and a more detailed analysis of the system's overhead.

Assisted Monitoring and Security Provisioning for 5G Microservices-Based Network Slices with SWEETEN

Rafael de Jesus Martins*, Juliano Araújo Wickboldt, Lisandro Zambenedetti Granville

Federal University of Rio Grande do Sul, Brazil

Abstract

5G networks have imposed a drastic shift in how mobile telecommunications must operate. In order to comply with the new requirements, solutions based on Network Function Virtualization (NFV) and network slicing must be carried out. Regarding NFV in particular, the trend towards pulverizing the monolithic software in a microservices-based one carries network management challenges to operators. The deployment and integration of one or more network management software with the managed services is as important as it is delicate, as stringent requirements of 5G applications must be respected. In this paper, we propose SWEETEN as a solution for automating the deployment and transparently integrating network management solutions from different management disciplines, in this case, monitoring and security. Demonstrating its usability through a intelligent healthcare use case, SWEETEN is shown to transparently provide monitoring and security solutions for a complete network slice, enabling compliance with privacy requirements through minimal low-level interventions from the network slice tenant. The results show how SWEETEN integration of monitoring and security disciplines can assist users in guaranteeing the correct operation of their deployments regardless of the underlying software solutions used.

Keywords: 5G, NFV, Network Management, Microservices

*Corresponding author

Email addresses: rjmartins@inf.ufrgs.br (Rafael de Jesus Martins), jwickboldt@inf.ufrgs.br (Juliano Araújo Wickboldt), granville@inf.ufrgs.br (Lisandro Zambenedetti Granville)

1. Introduction

Telecommunications have been undergoing massive evolution in the last years with the specification and launch of 5G networks. While previous generations mostly focused on improving customers data rate, 5G networks cover a wide range of applications with diverse requirements by offering disruptive improvements in reliability, device density, and coverage, to cite a few. In order to comply with such requirements, 5G networks must employ modern techniques for network slicing and Network Function Virtualization (NFV) [1].

Since its inception, NFV has drawn attention from academia and industry because of the benefits it offers in comparison to traditional middlebox appliances (*e.g.*, firewalls, deep packet inspectors (DPIs)). NFV decouples the proprietary hardware from the associated software, enabling the network functions to run on top of commodity hardware [2]. This shift enables dynamism and scalability much needed for 5G networks, all the while reducing operational costs for mobile carriers. In turn, these virtual appliances represented by Virtual Network Functions (VNFs) present their own challenges, in particular when a function is pulverized in multiple microservices that must cooperate to deliver the network service appropriately [3]. In this case, the 5G application's requirements must be carefully considered by the VNF manager, for the VNF as a whole and for the individual microservices comprising it.

Network management is a complex process that can include disciplines such as monitoring, securing, and configuring network devices. With the virtualization of network functions and further with the shift from monolithic software to microservices-based ones, networks become increasingly reliant on proper management of their components [4]. More then ever, individual pieces of the network must be properly managed so that the myriad of applications that 5G enables can be truly experienced by the end user. Newer well-rounded management solutions such as service meshes can provide a variety of network management capabilities to multiple applications interconnected through microservices. These technologies often rely on deploying a separate container with every microservice for the managed applications, which interfaces all connections from and to the application container, and therefore are known as sidecar proxies [5]. Such solutions,

however, are mostly fit for complex deployments, and often introduce network overhead (due to the additional hop per microservice in a flow) that can hinder their adoption depending on applications requirements[1]. Extreme cases such as edge applications could even suffer from the computational overhead from the additional containers deployed. Specific requirements for each application must therefore be considered when provisioning the network management capabilities, as there is no one-size-fits-all solution for all different applications thus far.

Considering that 5G networks are envisioned to support upcoming mission-critical applications, security becomes a primary concern. Targets can range from governments and industries to ordinary citizens. For example, eavesdropping e-health devices can leak confidential information regarding their users, and thus impose an important security requirement for the setup [6]. Additionally, battery limitations from these devices can result in minimal computational overhead being acceptable for management solutions. A different application with similar security challenges that runs in the cloud, conversely, could make use of more robust management artifacts that would incur in greater overhead overall. Since applications and resulting requirements can vary significantly, and because there is an increasing number of management tools offered for various contexts, correctly choosing and configuring a set of tools for each scenario becomes a challenging task even for experts.

In this paper, we present how SWEETEN (a**S**sistant for net**W**ork manag**E**m**E**n**T** of micros**E**rvices-based V**N**Fs) can help VNF operators and network slice tenants by including management features from multiple disciplines (*e.g.*, monitoring and security) in operators' and tenants' deployments in a transparent manner. This work is a direct evolution on our previous one [7], where managed entities were limited to VNFs and only monitoring was implemented as a management discipline. Now complete network slices are covered by SWEETEN, which is also able to provide both security and monitoring solutions. By augmenting their deployment specification with high-level management features request, tenants can enrich their deployed entities with automatically chosen and configured management tools and have their management needs fulfilled. When deemed

[1] https://medium.com/@pklinker/performance-impacts-of-an-istio-service-mesh-63957a0000b

necessary, the user can specify lower-level configuration parameters in order to fine-tune how the complete solution should be put together by SWEETEN. Natural language processing (NLP) is employed to extract meaningful tags from users deployment descriptions, which are then used to provide tailored solutions for each deployment. After the deployment phase, the system provides the user with an integrated dashboard and an API that allows the operator to manage their VNFs and network slice from a high-level perspective, regardless of the specific management tools selection and their interfaces at the lower-level. In this respect, SWEETEN's main contributions presented in this paper are:

- Automated configuration and deployment of network management tools following user's high-level specification, offering management solutions for novice users with ease;

- Varying abstraction levels in the specification are allowed, enabling fine-tuning of the solution by advanced users;

- Integrated dashboard, allowing transparent management for all entities of interest regardless of underlying software configured to realize the network management required.

We implemented a prototype to evaluate SWEETEN in terms of providing management capabilities to an intelligent healthcare use case. In this use case, patients can be monitored by a number of resource-constrained Internet of Things (IoT) health monitors and other resourceful devices in real-time, enhancing quality of human life through the automated execution of mundane tasks [8]. To achieve that, data collected by said devices is sent to a deep learning module hosted in the cloud, which processes the data from a patient and triggers alarms when events happen. Due to the sensitive nature of the traffic exchanges by the healthcare devices, security in the terms of privacy is a foremost concern for all communication. It should be noted that, since these devices can be resource-constrained, solutions should balance the defensive mechanism effectiveness and the overhead they produce. Moreover, IoT devices in this use case utilize Narrowband IoT (NB-IoT) for their radio-access technology as it offers improved coverage and efficiency in terms of cost and power consumption [9]. Cloud radio access network (C-RAN) is used to deliver connectivity to the application's devices, imposing stringent latency

and data rate requirements that must be met throughout the deployment life-cycle and thus implying the need for careful monitoring.

By using SWEETEN, the results show that operators can request management features and come up with solutions from a high-level perspective, with no need for expertise in specific tools and configurations that would typically constitute a burdensome work for experts. Through the inclusion of annotations to their specification, users receive management solutions configured for their needs. The additional management entities deployed represent a significant overhead regarding the deployment time, but considered acceptable for the benefits offered and due to its infrequent occurrence. Minimal computational overhead was also perceived for when the solution is deployed, while a much more prominent overhead is present regarding network overhead. Notwithstanding, the overhead is much more due to the included management entities themselves and not due to SWEETEN usage, and would thus also be present if a similar management solution were to be manually included by the user, therefore presenting a net gain for the user.

The remainder of this paper is organized as follows. In Section II, we present background and related work. Then, we describe a proposed architecture in Section III. In Section IV, we present a use case that illustrates management challenges faced by novel applications in 5G networks management. Then, we evaluate our proposal with the use case, discussing the results in Section V. Finally, we present our conclusions and future work in Section VI.

2. Background & Related Work

Because this study spans over a few non-trivial concepts, contextualizing each of them and how they relate to each other is necessary. To this end, Subsection 2.1 presents the main characteristics of the microservices paradigm and current efforts towards managing software that follows such paradigm. Network slicing as a technique for delivering 5G use cases is discussed in Subsection 2.2.

2.1. Microservices

Monolithic software can be defined as a software composed by modules that cannot be executed independently [10]. Software design has generally followed a monolithic paradigm in which an indivisible software is responsible

for realizing a complex service in an integrated manner. Monolithic software still can and should be designed through a composition of specialized modules. However, the different modules in a software following the monolithic paradigm still rely on resource sharing (*e.g.*, memory, CPU) for running in the same machine, which tightly ties all the components as one atomic application. While the monolithic architecture is viable for many applications, recent off-premise and distributed computing offered by cloud services impose the need for a more flexible paradigm in software design. In the microservices paradigm, systems are designed through independent components called microservices, which provide a system with cohesive and well-defined functionalities [4]. Context sharing between microservices is done through network messaging, allowing microservices to be deployed along a distributed infrastructure, as well as completely decoupling implementation details and choices (*e.g.,* programming languages) between modules. Microservices introduce many benefits regarding continuous integration and delivery, for example, as updates for individual microservices may be gradually rolled out. However, the design also imposes new management challenges that must assert the correct operation of each microservice, and of the composed software as a whole.

Designing and developing software through the microservice paradigm can quickly become hard to manage as complex connection schemes are required among hundreds of microservices. Service meshes recently emerged as a solution for that through the automatic management for microservices connections, as reviewed in the study by Li *et al.* [5]. Among their benefits, service meshes can provide service discovery for the microservices and load balancing among different containers (even using different software versions). On the implementation side, these solutions usually employ an array of lightweight network proxies, which are deployed alongside the application containers and can provide an interface for all incoming and outgoing connections. Some specific scenarios that are much relevant to 5G, such as multi-tenancy, can however present specific challenges that were not part of service meshes design. SWEETEN is designed to provide management for VNFs and network slices with various requirements, such as the minimal computational and network overhead for IoT applications, and thus can provide the appropriate management services and configuration based on the user specification and high-level feature annotations.

Chowdhury *et al.* [3] highlighted the importance for the NFV ecosystem to have VNFs designed through a microservice architecture. In Service Function

Chains (SFCs), for example, having monolithic VNFs incurs in unnecessary processing overhead from redundant functionalities. Instead, the redesign of these functions through microservices enable fine-grained resource allocation and independently scalable components, as elements for the orchestration of VNFs in an SFC become also present for the VNF-Components (VNF-C) for any VNF. Among the research challenges documented in the literature, the adequate monitoring of these functions is underlined, as well as questions pertaining performance profiling and overhead trade-offs, all occurring topics in our present research.

2.2. Network Slicing

Network slicing has emerged as a cornerstone for evolving 5G networks. While 4G and previous generation relied on a one-fits-all architecture to serve mobile network costumers, 5G covers a plethora of services with diverse requirements, and so the system itself must be customized to meet each customer's needs. With network slicing, the common network infrastructure can be harmoniously shared among multiple tenants, allowing their diverse requirements to be met while providing isolation between slices. End-to-end network slices can span over various network layers and heterogeneous technologies (*e.g.*, RAN, core, and cloud), and can facilitate service delivery to customers while also enabling efficient networking and service convergence [11]. Network slicing helps providing the much needed dynamism and scalability in 5G networks, as customized end-to-end slices can be created on-demand and thus provide a cost-efficient manner to serve customers.

Slamnik-Kriještorac *et al.* [12] presented an extensive survey on the distributed and heterogeneous resource sharing that is taking place in 5G networks. The sharing model for 5G and other networks proposed by the authors is classified in three distinct models: technical, business, and geographic. Specifically, the technical model is structured in three layers: infrastructure, orchestration, and service. Although some management concerns for network slices are discussed, they primarily focus on the infrastructure layer, while the examples for service (*e.g.*, Healthcare) and orchestration (*e.g.*, Kubernetes) are left for each layer and case to solve individually. Our study, in contrast, proposes that the network management should consider all layers jointly, and also that this management is a complex task that operators should be assisted with when deploying new network slices.

Kist *et al.* [13] proposed a virtualization scheme that allows technologies and instances for different Radio Access Networks (RANs) to be provided as services for network slice tenants. The proposal offers programmability and adaptability for service providers while maintaining isolation between tenants and their slices. An experimental scenario evaluated by the authors comprised of LTE and NarrowBand-IoT (NB-IoT) clients showcases how the proposed system allows the provisioning and management for virtual RANs (vRANs) for providers' network slices. The management aspect is however limited to the RAN infrastructure, and any additional required management aspects (*e.g.*, regarding the application, or other VNFs included in the slice) are left to be determined and deployed by the slice owner.

Coelho *et al.* [14] looked into formally defining the network slice designing problem, proposing a framework that considers nested slices and network functions decomposed in smaller services and models the relationship between radio splitting, control and data planes isolation, and core network function placement. Leveraging the reusability of smaller network function services and network slices subnets, a variety of sharing policies that range from total isolation to flat sharing can be used to realize 5G services of any class, and fulfilling the stringent requirements each of them impose. The study therefore focuses on producing a network slice, including necessary network functions, their split and placement, to deliver the demands posed by a number of services. The slice management itself, including the appropriate monitoring of the deployed network functions, is not covered by the slice design and thus is left for the operator to manually determine, configure and deploy the appropriate solutions.

3. SWEETEN design and implementation

In this section, we review SWEETEN design, which has been previously introduced and evaluated in the context of network monitoring [7]. Moreover, additional implementation details are now presented in-depth for all the architectural components that comprise the system. An overview of SWEETEN is presented in Figure 1. The remainder of this section presents the system progressively. General aspects and user input are discussed in Subsection 3.1. SWEETEN pre-processing is discussed in Subsection 3.2. The system's mapping of management tools and configurations is explained in Subsection 3.3. Finally, the process of deploying the mapped solution and returning a customized dashboard to the user is discussed in Subsection 3.4.

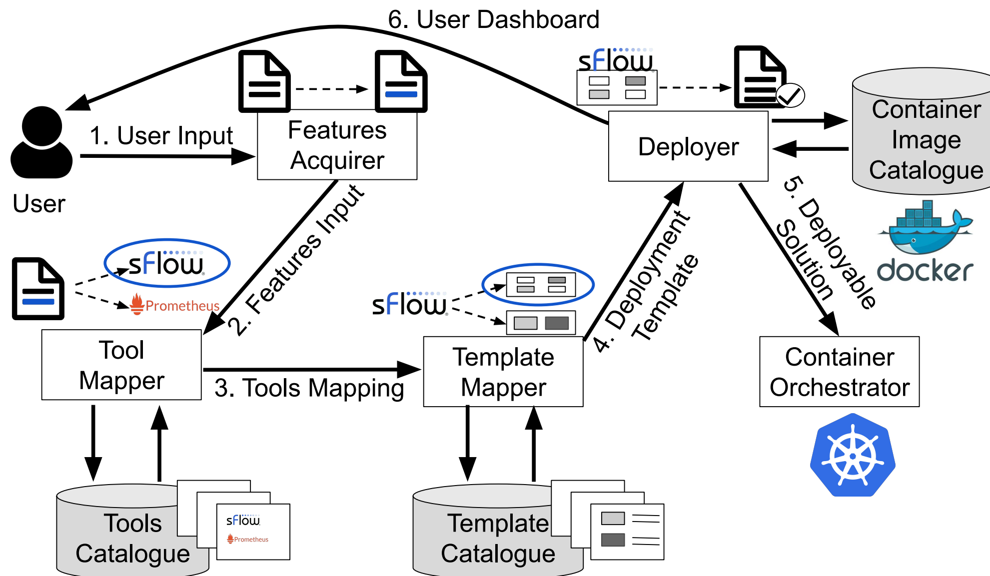


Figure 1: SWEETEN architecture [7].

3.1. General Aspects and User Input

SWEETEN can be viewed as an automated assistant intended to help tenants of 5G slices to meet management requirements for their slice components. In particular, when these components are designed following the microservices architecture, network management must be thought of while respecting the modularity and isolation envisioned in this architecture. Moreover, a slice can span over thousands of microservices [15], which makes automated management not only a benefit but a necessity.

The user input in the designed system consists of a specification for user containers, which is augmented by the user to contain requests for management features. These management features can be of multiple disciplines, namely security, monitoring, and administration. Some examples for each discipline are found in Table 1.

With respect to the classification for network management features, three disciplines are considered, as presented in Table 1. The *monitoring* discipline encompasses all measurements that can be done to assert a network and its components are behaving as expected [16]. These measurements can be either passive (*e.g.*, observing flows in a given interface) or active (*e.g.*, probing a link to check latency and throughput available). The *security* discipline

Table 1: Network features and respective tools listings.

	Monitoring	**Security**	**Administration**
Flows	sFlow, NetFlow, Prometheus	Snort (for IDS), OSSEC	-
Traffic	Prometheus, iPerf, SNMP	iptables, nftables	Linux tc
Latency	SmokePing, OWAMP, TWAMP	-	Linux tc
Device	Kubelet (Kubernetes native)	syslog, antivirus utilities	NETCONF, SNMP

involves all sensitive aspects in a network, including privacy and resilience requirements and the means to guarantee them at a certain level [17]. The *administration* discipline is comprised of management tasks and applications that actively alter the network behaviour for one or more device. For example, Netconf protocol can be utilized to reconfigure switches and routers in a network, altering its behavior dynamically [18].

Having presented the user options for management features requests, an excerpt from a user specification with requests for both monitoring and security features is presented in Listing 1. While novice users can request high-level management features more easily, advanced users can specify lower-level configuration parameters that must be observed in the provided solution. This enables operators to promote network management capabilities from a high-level perspective, while also being able to traverse through lower-level configuration parameters when necessary. An example for such parameters addition is presented in Listing 2, where in addition to specifying which tool should be used (*i.e.*, Prometheus), configuration for the measurement intervals and the dashboard are provided by the user.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    management:
      monitoring:
        - flows
      security:
        - cryptography
    ...
```

Listing 1: Excerpt for a simplified user management service with requirement for management features.

```yaml
monitoring:
  - flows: TCP
    tool: Prometheus
    scrape_interval: 1s
    dashboard:
      - tool: Grafana
        http_port: 3333
    ...
```

Listing 2: Excerpt for a feature request that includes lower-level configuration parameters specification

3.2. User Images Pre-processing

The user input is received by SWEETEN through the Features Acquirer module. This module is responsible for retrieving information on all microservices defined in the user specification as well as the management features requested for each microservice.In addition to the nature of the management desired, requirements for the produced solution can be derived from requirements tags. To achieve that, each microservice with a management annotation undergoes three steps:

1. Management feature retrieval, where the requested features and options are extracted from the user specification.
2. Description enrichment, where the information about the user service is augmented.
3. Service tagging, where tags that better describe the service requirements are appended to the specification.

Each requested feature obviously can be realized by a number of tools. In order to allow the best possible match for tools selection and configuration

in a later stage within SWEETEN, user's microservices undergoes a tagging process composed by the description enrichment and service tagging processes previously mentioned. For each component in the user input for which a feature is requested, SWEETEN appends tags that can provide some insight about the type of service and its requirements. Tags are later used to differentiate the tools and configuration choices for monitoring a cloud-hosted service from an IoT one; for example, while the former can leverage network and processing resources to employ a robust solution, the later must realize the management necessities with minimal overhead.

To alleviate the burden for the user, SWEETEN can automatically derive tags from the user specification alone without any additional user input. To achieve that, the description for each container that composes a service is processed in the service tagging step. This description is rarely present and descriptive for most containers, so the description is enriched before tags are derived. For the purpose of our proof-of-concept, the description enrichment process is obtained through a simple Google web search query that is automatically requested by the system. This query is composed of the words "define" and the container image name, and the text for the first result is considered by the system. The service tagging process can then run the enriched description through a Natural Language Processing sub-module to extract the aforementioned tags, as described next.

Natural Language Processing (NLP) is an area of computer science that employs algorithms for learning, understanding, and producing of human language content [19]. NLP has lately been a tool in various areas with promising results, for example, by providing enterprises with network security insights and suggesting solutions when paired with a neural network model [20].

Among the many available algorithms for NLP, an important aspect that differentiates them is whether they require supervised learning or not. In SWEETEN's case, since we want not only to include current management features but also to allow the system to easily evolve and include new ones, unsupervised learning is preferred. Our model of choice is based on Latent Dirlecht-Allocation (LDA) [21], an unsupervised machine-learning algorithm that can help find common topics between multiple text documents. In this way, we initialize a database with enriched descriptions for three of

the top containers for each category in DockerHub [2], and stipulate seven different topics to be found. Each topic will contain a weighted list of words that indicates the prevalence of the main words for each topic. For a new document, *i.e.*, the enriched description for the user microservice that is being processed, LDA associates a percentage for each pre-determined topic; later, the relevant words for the selected topics (*e.g.*, indicating the resourcefulness of an object) are used to determine which solutions SWEETEN should prioritize, as explained next.

3.3. Management Tools and Templates Mappings

Management features required by the users must be realized by a set of management tools. The Tool Mapper module thus is the first one to make selections based on the Features Acquirer output. For each feature required by the user, this module maps to one or more tools that are capable of realizing such features. The listing for these mappings are provided through a Tools Catalogue, which has been already pre-populated by an expert. In the event that more than one tool are fit for a certain request, tags are considered so that the best fit can be provided. The algorithm for matching the tags to the available tools is a greedy one, so the solution that matches the most tags from the user input is selected each time.

Additionally, each tool must be configured and deployed so that it can perform the intended task correctly. For example, a firewall must be placed in front of a targeted back-end service, while an active latency monitor must be placed alongside the monitored microservice. Moreover, the previously appended tags must be considered when determining the configuration parameters for a given network management tool. A second stage for selection is thus performed by the Template Mapper. A template represents the configuration required by a management tool to be deployed, both with respect to the tools internal configurations and with any necessary cluster definition [22]. The configuration aspect is also covered by the tags appended to the user input, in a process analogous to the one described for the tools matching.

Occasionally, the correct configuration for some management tool might require some breaching of microservices architectural design during execution. For example, monitoring the active connections for a microservice

[2] `https://hub.docker.com/`

requires for the the management tool not only to be placed alongside the managed service, but to share its network context too. This is achieved by namespace [23] sharing between managed and management services, but which is only performed when necessary. In this way, microservices design can be maintained for all applications, and specificities are configured and treated with templates designed for such cases.

3.4. Solution Deployment and User Dashboard

The elements of the solution must be put together in a deployable specification. Because we chose to use Kubernetes as the system's container orchestrator, the result is composed of two separate YAML [24] specifications: one for the user services that did not require management features, *i.e.*, services that were already part of the user specification, but that did not require any management feature; and one for the remainder of the user specification plus all the network management tools included by SWEETEN. We chose Kubernetes because it is the most widespread platform for containers in industry and academia alike [25], which in the one hand offers an active community and a rich environment, and the other hand enables SWEETEN to be used by a wide audience.

Finally, during the slice lifecycle, the user can manage their services through a customized dashboard. Based on the user input, SWEETEN can deploy dashboards from different software. An example for a monitoring dashboard by Prometheus [26] provided by SWEETEN for a novice user is depicted in Figure 2, while a more advanced Grafana dashboard provided by SWEETEN is depicted in Figure 3. Non-visualizing features, such as cryptography introduced by security features, are presented textually for the user's knowledge. Additionally, the user can interact directly with the configured management container through a terminal, so they can still have control after the deployment phase for their slice.

4. Automated Network Management for an Intelligent Healthcare Use Case

5G systems promise a series of disruptive advancements for a myriad of applications typically classified in three scenarios. Enhanced Mobile Broadband (eMBB) addresses applications centered in multi-media content, services, and data; Ultra Reliable Low Latency Communications (URLLC) encompasses critical applications that pose stringent requirements such as

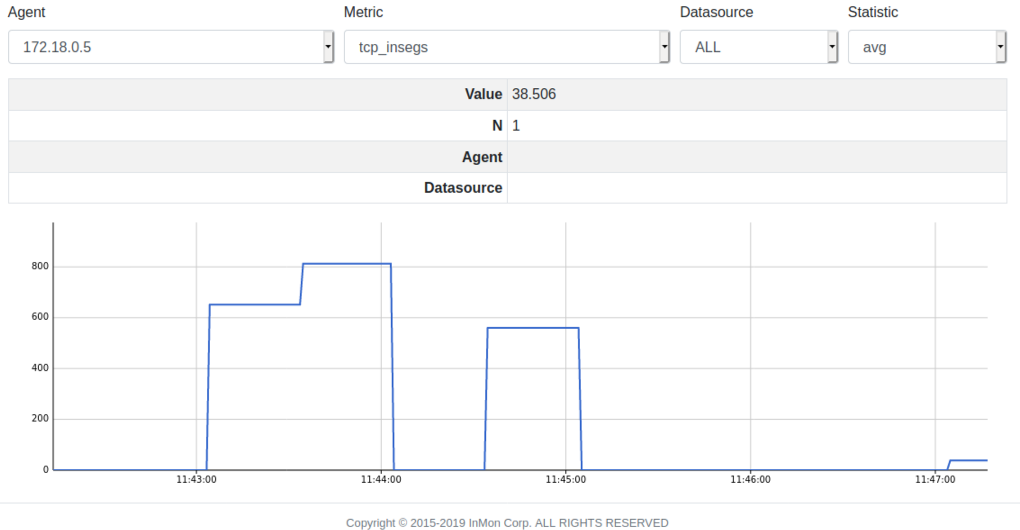


Figure 2: User dashboard generated from the novice user's specification.

remote medical surgery; massive Machine Type Communications (mMTC) is characterized by a large number of low-cost devices that transmit a low volume of data [27]. Some of the most technically challenging applications unite requirements from two or even all three scenarios. Healthcare applications can exemplify such a case, where a multitude of health devices of different capabilities and with distinct requirements are used to guarantee the well-being of patients. This use case is depicted in Figure 4 and further expanded in the following.

Heart rate, respiratory rate, and body temperature monitors are a small sample from a large list of monitoring devices that can be utilized in a patient's health monitoring. The number of IoT devices employed in this scenario can grow significantly, providing abundant data to track patients physiological characteristics but also requiring a more extensive analysis by physicians and other professionals. In this context, these applications can be further benefited by the inclusion of intelligent algorithms to process and automate decisions, triggering alarms and actions whenever abnormalities are detected [8]. Artificial intelligence (AI) techniques based on novel models such as big data mining and deep learning can process large amount of data at real-time, and then predict and automate tasks at a rate impossible before. While the monitoring part must be performed on-premise, *i.e.*, in

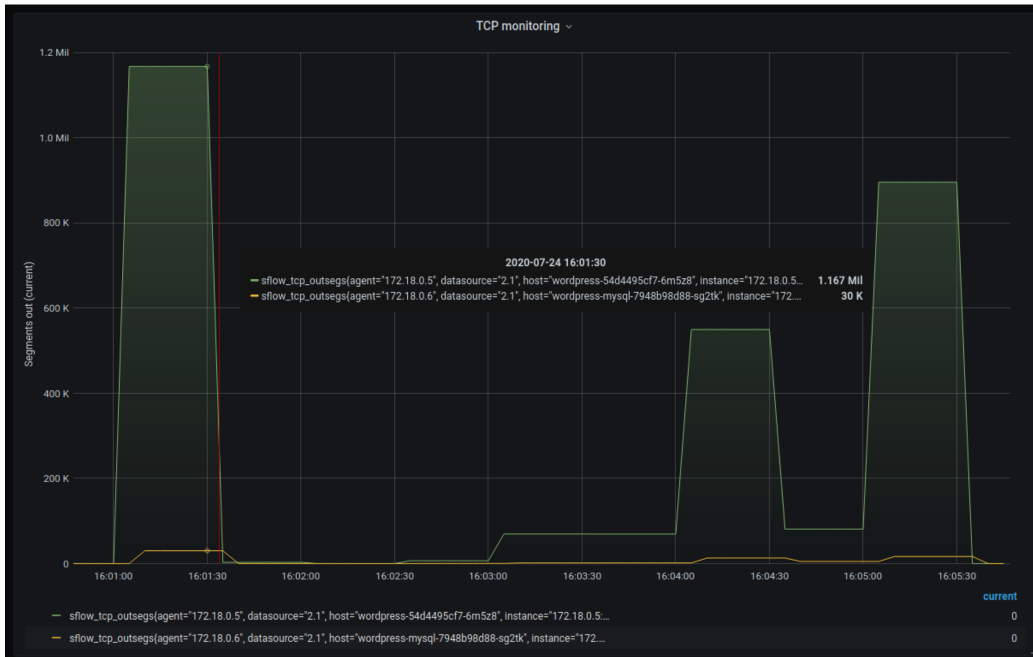


Figure 3: User dashboard generated from the experienced user's specification.

the healthcare center where the patients are located, the burden of collecting and processing all the data can be effectively run in the cloud.

Because of the sensitive nature of the data monitored and transferred, security, in particular by the means of privacy, is a foremost concern. As hardware solutions are not always feasible and as new legislation advances the levels of privacy requirements for these applications, guaranteeing a certain security level from a software perspective is a necessity. In certain occasions, resourceful devices are used for patients' monitoring, such as 4K cameras that can record their movements, and paired with deep-learning algorithms can detect facial expressions and gestures of patients and warn healthcare professionals in the event of an anomaly [8]. However, most monitoring IoT devices are constrained in terms of computational power and battery, and so possible security solutions should account for these limitations and prioritize lighter-weight solutions whenever possible. Less constrained devices, in turn, can afford to employ more advanced and intensive defensive mechanisms, and the provisioning for each case should reflect these characteristics. As a

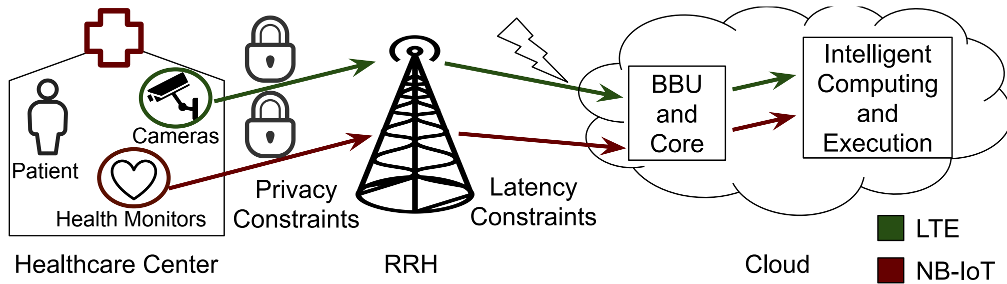


Figure 4: Use Case for Intelligent Healthcare Application

manual security approach is not feasible for complex 5G scenarios, security automation is a key principle in securing 5G applications and networks [28].

In recent years, multiple Low Power Wide Area (LPWA) radio technologies have emerged as options for delivering the scalability required by mMTC applications. From the alternatives, NB-IoT has been shown to offer promising results for healthcare applications [29]. NB-IoT is fully compatible with Long Term Evolution (LTE), and can be deployed inside a single LTE physical resource block (PRB) of 180 KHz or inside an LTE guard band, potentially serving up to 50k end-devices per cell [30]. The limited 250 kbps data rate is plenty for hear rate and body temperature monitoring that require only 1 byte for payload every 5 minutes [29], but it is impractical for streaming the video from the deployed cameras. Devices as such that require extensive bandwidth must connect over standard LTE-A network, which is capable of meeting their demands.

Metrics collected by all the devices are reported to a Remote Radio Head (RRH), the radio antenna responsible for communications to and from users' devices. The signals must then be processed by the network core, which is performed by the Base-Band Unit (BBU) in a Base Station (BS). Previously, these functions would be performed exclusively by specific-purpose hardware. With the recent advances in virtualization and the expansion of NFV architectures, virtual base stations have been adopted by pioneering virtual mobile network operators. Such operators do not own the required wireless physical infrastructure, but instead lease it from traditional mobile network operators. The processing modules for wireless services, in turn, can be provided by software running in the cloud, enabling different strategies that benefit customers [31]. In the NB-IoT case, low protocol stack processing requirements and low latency-sensitivity make C-RAN an

attractive alternative, so all the BBU processing and higher-layer protocol stacks are implemented by software that runs on the cloud[32].

A standard LTE-A BBU can be split in different functions [33]. The different split options offer possibilities of alternating between dedicated hardware and function virtualization, allowing the flexible adoption of functional split in time and location. Noteworthy, 5G specifications pose stringent network requirements for their communications, in particular with respect to data rate and latency. Regarding latency, a maximum delay of around 3ms for transmitting and processing the signal is determined by the Hybrid Automatic Repeat reQuest (HARQ) mechanism adopted in LTE [34]. There is thus a stringent requirement (*i.e.*, latency) that must be respected by the network slice, and that must be properly monitored too. While our previous work considered monitoring challenges for microservice-based VNFs in 5G networks, the current study further advances the management scope by including security concerns and measurements in the evaluated slice. Moreover, the complete network slice itself is subject of study here, including different radio access technologies and service applications, with new features and requirements that they come with.

5. Results and Discussion

Although there are already some promising NB-IoT solution in development [35, 32], for stability and reproducibility of our results we do not utilize any specific implementation, and simply consider the traffic patterns for these deployments. With respect to the security demanded in the user's specification, SWEETEN can leverage the previously populated catalogues and produce different results for each requiring application. In the present use case, it is noteworthy that the IoT monitoring devices should try to adopt solutions that incur in minimal overhead because of their resource-constrained nature. Tools and configurations provided by experts to the system's catalogues can therefore feature fine-tuned solutions for systems tagged as such. This way, a TLS configuration using less resource-intensive ciphers can be used for IoT components[3], while more resourceful components can utilize a more robust solution[4]. For comparison purposes, normalized

[3] `https://docs.aws.amazon.com/iot/latest/developerguide/transport-security.html`

[4] `https://www.acunetix.com/blog/articles/tls-ssl-cipher-hardening`

results for the different security options are presented in Figure 5 for memory footprint, and in Figure 6 for network overhead.

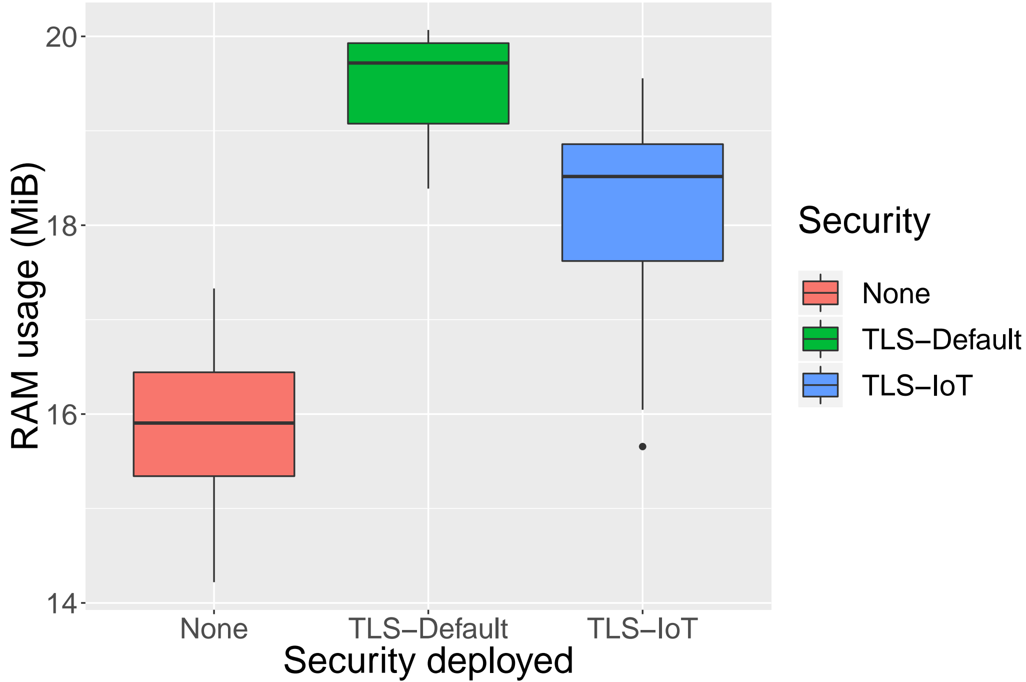


Figure 5: Computational overhead using different security options

The results indicate that a small computational overhead is added with respect to memory footprint when either the default or the IoT security solution is included. Albeit small, it is also noticeable that the security option recommended for IoT outperforms the default option with respect to overhead. Similar results are also found for network overhead. While it is clear that the overhead is much more noticeable when comparing either security option (*i.e.*, the default one or the IoT one) to having no security deployed, the IoT configuration still leads to lesser overhead in comparison to the default configuration. While the user should still be mindful that some overhead will be added whenever a security feature is requested, these results show how experts knowledge integrated into the system through different configuration options can be used to produce a more fine-tuned solution for each case.

Another overhead aspect that we consider is the one added to the deployment time of the slice specification. Here, there are two separate

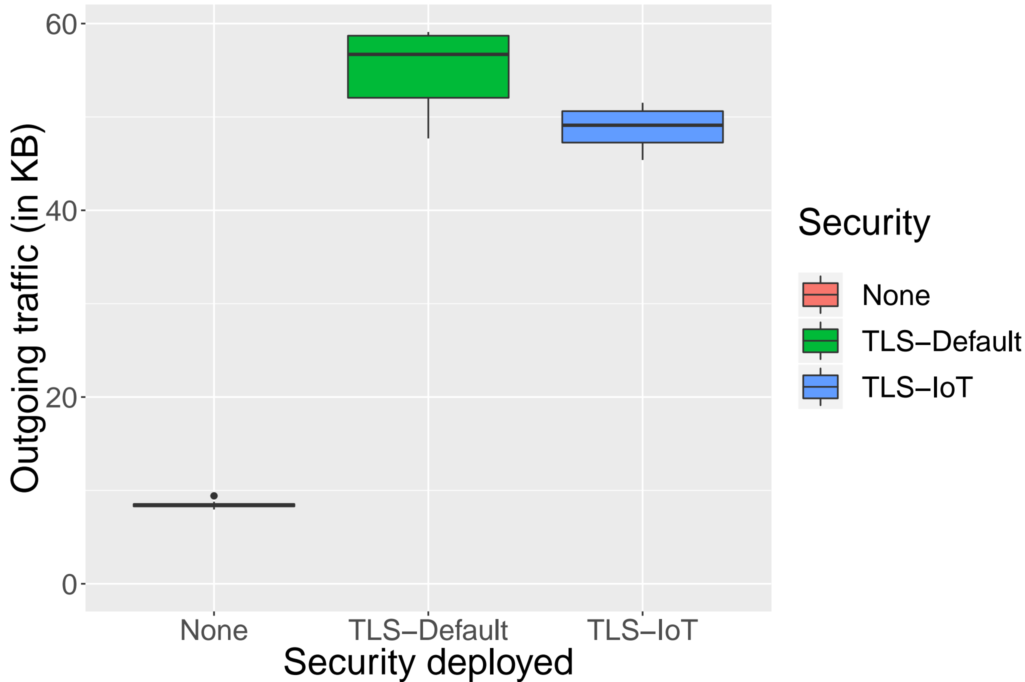


Figure 6: Network overhead using different security options

types of processing overhead that must be considered. The first one is the overhead introduced by SWEETEN's processing of the user input until the complete solution is produced, as has been explained in-depth in Section 3. The second one is the additional deployment overhead due to the inclusion of the management services (realized by containers), that must be instantiated alongside the original specification. A comparison for these times is presented in Figure 7. We evaluate the system's scalability through varying the number of replicas for each deployment in the slice from one to ten.

The results show that SWEETEN's overhead is approximately constant regardless of the replicas count, and it becomes negligible for larger deployments. Larger deployments are precisely the ones that should benefit the most from SWEETEN, as the inclusion of management features throughout a complex slice is burdensome in comparison to a more simplistic slice. Most of the overhead is introduced by the inclusion of the additional containers, with the complete solution taking on average 94% more time to be deployed. Two noteworthy points here are that: (1) this is not a recurring cost, as fresh deployments are less frequent than individual updates,

which would present a much lighter overhead; (2) the manual inclusion of management containers by an expert user would incur in similar overhead for the deployment time. Users could minimize this overhead by including the management software directly into their containers, but doing so would breach the microservice architecture and possibly do more harm than good in the process. The overhead results are also aligned to what we observed previously for a different network slice where only monitoring management was featured [7].

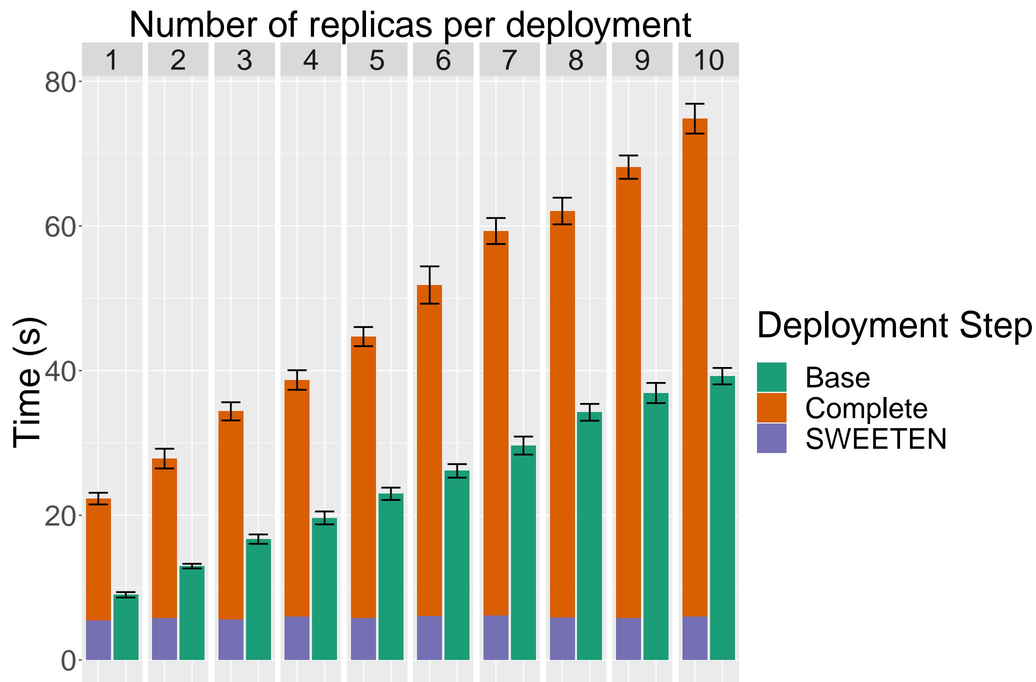


Figure 7: Deployment time overhead for varying replicas count

We also illustrate how the user receives their monitoring information. An example for an excerpt of the dashboard provided for monitoring the throughput is illustrated in Figure 8. Through this interface, the user can easily monitor multiple services of their slice and quickly identify problems as they occur. Different resources can be configured by the system with different parameters in a transparent manner for the user. For instance, a resource-constrained device can be configured with a lower sampling rate than a more resourceful device, thus introducing less overhead. The user can also edit

the graphics in the dashboard and add their own, so they can fine-tune the solution to best fit their needs.

Figure 8: Example dashboard for throughput monitoring

We also analyze the system performance with respect to the expressiveness offered to the user. About four lines of high-level feature specification by the user is translated to over 30 lines of management deployment specification with respect to security (disregarding the cryptography keys generated and used in the deployment). The result is even more prominent with respect to monitoring, where four lines of specification are translated into over 100 lines of management specification that add the required monitoring features. These findings are inline with what was reported in our previous work [7], further reinforcing the benefits offered by SWEETEN for multiple management disciplines.

6. Conclusion and Future Work

5G networks are in the process of being rolled out around the world and will enable disruptive applications and services that were not previously feasible. To realize that, advances presented by NFV, SDN, and network

slicing, for example, must be carefully integrated by these networks. With the increasing number of devices and services, network management plays a central role in delivering the resources and features required by each component. For the same reason (*i.e.*, the increasing number of networks components), the configuration and management of all the pieces must be realized in an automated manner.

In this work, we investigate the assisted management of network slices through the use of SWEETEN. Initially proposed as a system to assist VNF operators, SWEETEN has been demonstrated in this study as a tool capable of delivering management solutions across a diverse network slice. Through high-level annotations in their slice specification, users are able to effortlessly receive fine-tailored management solutions configured for each of their applications and services. The proposed use case demonstrates how a network slice for intelligent healthcare can include monitoring and security features with ease, even considering the different requirements for each application.

Our results show that there is an important expressiveness gain for the user through SWEETEN. Assisting the user in properly deploying complex network slices is a vital point in achieving the dynamism expected from 5G networks. We also evaluated the overhead of SWEETEN with respect to deployment time, and computational and network overhead. While the deployment time is noticeably affected by the additional management services included by SWEETEN, it is not a recurring cost (*i.e.*, the slice's deployment) and is easily offset by the management functionalities featured in the slice. In the same way, computational overhead was non-negligible, but small enough that it is adequate for the services included. Network overhead, however, was much higher when cryptography solutions were included in the system, which, while expected, to users' discretion is needed to define whether the overhead is acceptable or not.

In the future, we plan to further develop the system through the inclusion of additional management disciplines and solutions. A myriad of different services are coming with 5G, and their network requirements can be as varied as the services themselves. It is thus important for a management assistant to be able to cover a variety of cases so that its usefulness is not limited to a small subset of applications. Moreover, we also intend to evaluate different tagging mechanisms for the Features Acquirer module, so that a more refined tagging system can help SWEETEN to further fine-tune solutions and configurations for every service.

Acknowledgment

This study was partially funded by CAPES - Finance Code 001. We also thank the funding of CNPq, Research Productivity Scholarship grants ref. 313893/2018-7 and 312392/2017-6.

References

[1] A. A. Barakabitze, A. Ahmad, R. Mijumbi, A. Hines, 5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges, Computer Networks 167 (2020) 106984.

[2] ETSI, NFVISG, GS NFV-MAN 001 v1. 1.1 Network Function Virtualisation (NFV); Management and Orchestration, 2014. URL: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf.

[3] S. R. Chowdhury, M. A. Salahuddin, N. Limam, R. Boutaba, Re-architecting NFV ecosystem with microservices: State of the art and research challenges, IEEE Network 33 (2019) 168–176.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, in: Present and ulterior software engineering, Springer, 2017, pp. 195–216.

[5] W. Li, Y. Lemieux, J. Gao, Z. Zhao, Y. Han, Service mesh: Challenges, state of the art, and future research opportunities, in: 13th IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2019, pp. 122–127.

[6] S. Zhang, Y. Wang, W. Zhou, Towards secure 5g networks: A survey, Computer Networks 162 (2019) 106871. URL: http://www.sciencedirect.com/science/article/pii/S138912861830817X. doi:https://doi.org/10.1016/j.comnet.2019.106871.

[7] R. de Jesus Martins, A. G. Dalla-Costa, J. A. Wickboldt, L. Z. Granville, Sweeten: Automated network management provisioning for 5g microservices-based virtual network functions, in: 2020 16th International Conference on Network and Service Management (CNSM), IEEE, 2020, pp. 1–9.

[8] D. Wang, D. Chen, B. Song, N. Guizani, X. Yu, X. Du, From iot to 5g i-iot: The next generation iot-based intelligent algorithms and 5g technologies, IEEE Communications Magazine 56 (2018) 114–120. doi:10.1109/MCOM.2018.1701310.

[9] J. Xu, J. Yao, L. Wang, Z. Ming, K. Wu, L. Chen, Narrowband internet of things: Evolutions, technologies, and open issues, IEEE Internet of Things Journal 5 (2018) 1449–1462. doi:10.1109/JIOT.2017.2783374.

[10] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards microservice architectures: An industrial survey, in: 2018 IEEE International Conference on Software Architecture (ICSA), 2018, pp. 29–2909. doi:10.1109/ICSA.2018.00012.

[11] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, H. Flinck, Network slicing and softwarization: A survey on principles, enabling technologies, and solutions, IEEE Communications Surveys Tutorials 20 (2018) 2429–2453. doi:10.1109/COMST.2018.2815638.

[12] N. Slamnik-Kriještorac, H. Kremo, M. Ruffini, J. M. Marquez-Barja, Sharing distributed and heterogeneous resources toward end-to-end 5G networks: A comprehensive survey and a taxonomy, IEEE Communications Surveys & Tutorials 22 (2020) 1592–1628.

[13] M. Kist, J. F. Santos, D. Collins, J. Rochol, L. A. Dasilva, C. B. Both, Airtime: End-to-end virtualization layer for ran-as-a-service in future multi-service mobile networks, IEEE Transactions on Mobile Computing (2020) 1–1. doi:10.1109/TMC.2020.3046535.

[14] W. da Silva Coelho, A. Benhamiche, N. Perrot, S. Secci, On the impact of novel function mappings, sharing policies, and split settings in network slice design, in: 2020 16th International Conference on Network and Service Management (CNSM), IEEE, 2020, pp. 1–9.

[15] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, S. Tilkov, Microservices: The journey so far and challenges ahead, IEEE Software 35 (2018) 24–35. doi:10.1109/MS.2018.2141039.

[16] S. Lee, K. Levanti, H. S. Kim, Network monitoring: Present and future, Computer Networks 65 (2014) 84–98.

[17] W. Stallings, Cryptography and network security, 4/E, Pearson Education India, 2006.

[18] R. Enns, Netconf configuration protocol, RFC 6241 (2006).

[19] J. Hirschberg, C. D. Manning, Advances in natural language processing, Science 349 (2015) 261–266.

[20] M. F. Franco, B. Rodrigues, E. J. Scheid, A. Jacobs, C. Killer, L. Z. Granville, B. Stiller, Secbot: a business-driven conversational agent for cybersecurity planning and management, in: 2020 16th International Conference on Network and Service Management (CNSM), IEEE, 2020, pp. 1–7.

[21] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, the Journal of machine Learning research 3 (2003) 993–1022.

[22] R. de Jesus Martins, R. B. Hecht, E. R. Machado, J. C. Nobre, J. A. Wickboldt, L. Z. Granville, Micro-service Based Network Management for Distributed Applications, in: 34th International Conference on Advanced Information Networking and Applications (AINA), Springer, 2020, pp. 922–933.

[23] C. Pahl, Containerization and the paas cloud, IEEE Cloud Computing 2 (2015) 24–31.

[24] O. Ben-Kiki, C. Evans, B. Ingerson, Yaml ain't markup language (yaml™) version 1.1, Working Draft 11 (2009). URL: `https://yaml.org/spec/1.1/index.html`.

[25] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Computing 1 (2014) 81–84.

[26] Prometheus Authors, Prometheus-monitoring system & time series database, 2017. URL: `https://prometheus.io/`.

[27] M. Series, Imt vision–framework and overall objectives of the future development of imt for 2020 and beyond, Recommendation ITU 2083 (2015).

[28] Y. Sun, Z. Tian, M. Li, C. Zhu, N. Guizani, Automated attack and defense framework toward 5g security, IEEE Network 34 (2020) 247–253. doi:10.1109/MNET.011.1900635.

[29] H. Malik, M. M. Alam, Y. L. Moullec, A. Kuusik, Narrowband-iot performance analysis for healthcare applications, Procedia Computer Science 130 (2018) 1077 – 1083. URL: http://www.sciencedirect.com/science/article/pii/S1877050918305192. doi:https://doi.org/10.1016/j.procs.2018.04.156, the 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops.

[30] U. Raza, P. Kulkarni, M. Sooriyabandara, Low power wide area networks: An overview, IEEE Communications Surveys & Tutorials 19 (2017) 855–873.

[31] N. Kamiyama, A. Nakao, Analyzing dynamics of mvno market using evolutionary game, in: 15th International Conference on Network and Service Management (CNSM), IEEE, 2019, pp. 1–6.

[32] Y. D. Beyene, R. Jantti, O. Tirkkonen, K. Ruttik, S. Iraji, A. Larmo, T. Tirronen, a. J. Torsner, Nb-iot technology overview and experience from cloud-ran implementation, IEEE Wireless Communications 24 (2017) 26–32. doi:10.1109/MWC.2017.1600418.

[33] D. Wubben, P. Rost, J. S. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy, G. Fettweis, Benefits and impact of cloud computing on 5G signal processing: Flexible centralization through cloud-ran, IEEE Signal Processing Magazine 31 (2014) 35–44.

[34] M. A. Marotta, H. Ahmadi, J. Rochol, L. DaSilva, C. B. Both, Characterizing the relation between processing power and distance between bbu and rrh in a cloud ran, IEEE Wireless Communications Letters 7 (2018) 472–475.

[35] C.-Y. Ho, R.-G. Cheng, J.-W. Chen, C.-S. Liu, Open nb-iot network in a pc, in: 2019 IEEE Globecom Workshops (GC Wkshps), IEEE, 2019, pp. 1–6.