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.

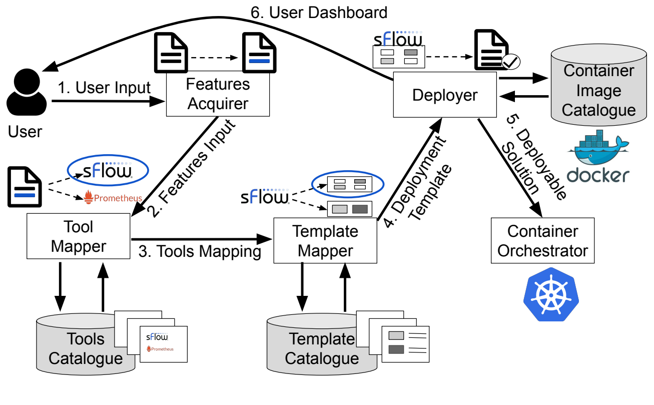


Fig. 1. SWEETEN architecture.

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

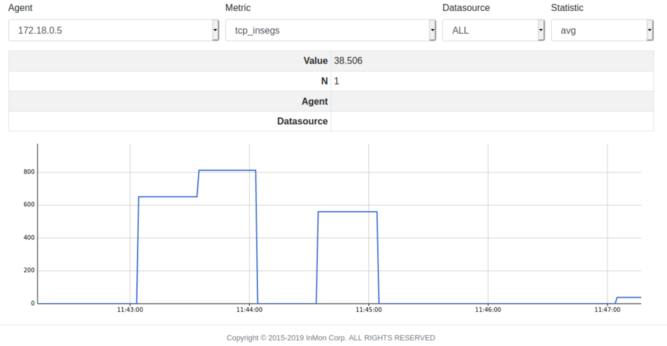
Microservices typically run inside lightweight containers. Docker containers orchestrated by Kubernetes have for the past few years emerged as the most prominent combination for running such complex applications [15], even more so with the discontinued support for alternatives like Docker Swarm [2], 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.



Agent: 172.18.0.5
Metric: tcp_insegs
Datasource: ALL
Statistic: avg

Value	38.506
N	1
Agent	
Datasource	

Copyright © 2015-2019 InMon Corp. ALL RIGHTS RESERVED

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.

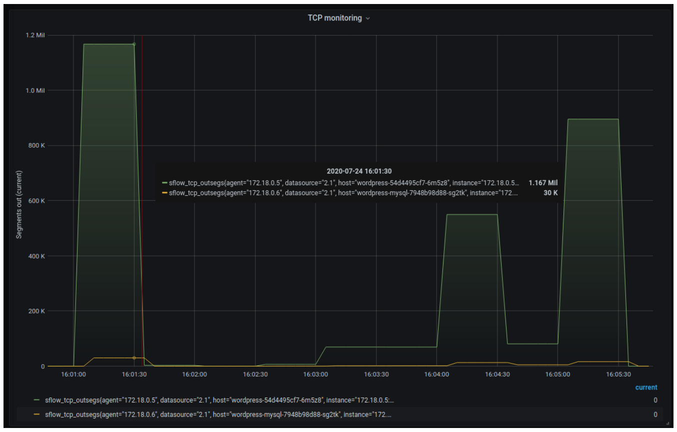

TCP monitoring
1.2 Mil
1.0 Mil
800 K
600 K
400 K
200 K
0
Segments out (current)
16:01:00
16:01:30
16:02:00
16:02:30
16:03:00
16:03:30
16:04:00
16:04:30
16:05:00
16:05:30
2020-07-24 16:01:30
sflow_tcp_outsegs{agent="172.18.0.5", datasource="2.1", host="wordpress-54d4495cf7-6m5z8", instance="172.18.0.5... 1.167 Mil
sflow_tcp_outsegs{agent="172.18.0.6", datasource="2.1", host="wordpress-mysql-7948b98d88-sg2tk", instance="172.... 30 K
current
sflow_tcp_outsegs{agent="172.18.0.5", datasource="2.1", host="wordpress-54d4495cf7-6m5z8", instance="172.18.0.5:... 0
sflow_tcp_outsegs{agent="172.18.0.6", datasource="2.1", host="wordpress-mysql-7948b98d88-sg2tk", instance="172.... 0

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/

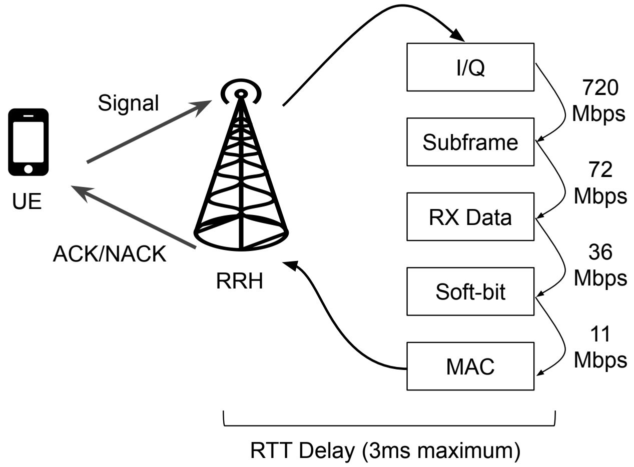


Fig. 4. Communication flow and bandwidth requirements for radio functions.

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.

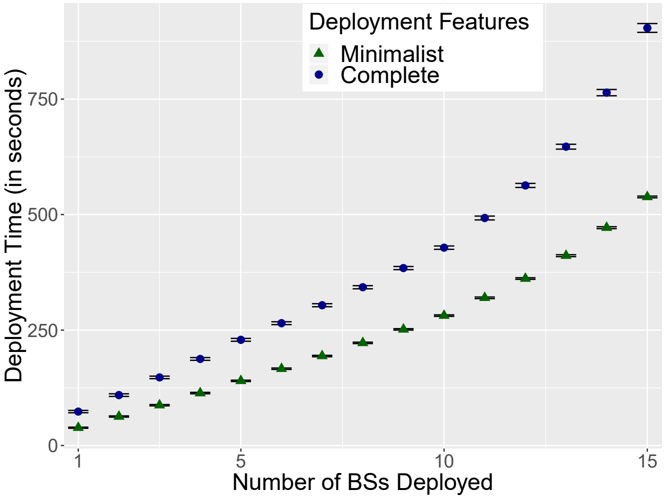


Fig. 5. Time taken to deploy up to 15 BSs, with and without using our system.

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.

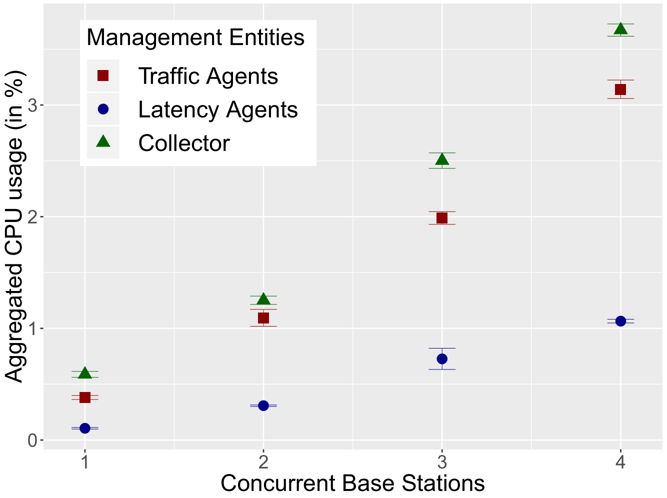


Fig. 6. CPU usage (in percent) for management containers for up to four concurrent (same VM) base stations.

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

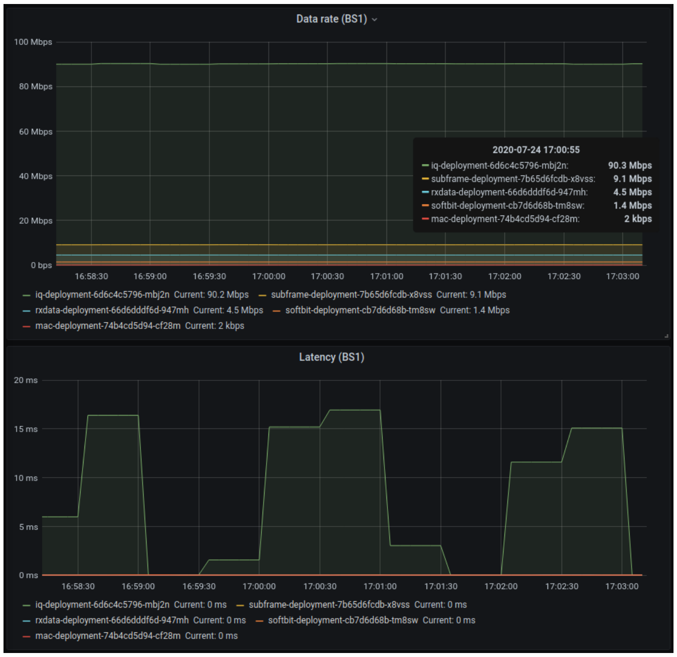

Data rate (BS1)
100 Mbps
80 Mbps
60 Mbps
40 Mbps
20 Mbps
0 bps
2020-07-24 17:00:55
iq-deployment-6d6c4c5796-mbj2n: 90.3 Mbps
subframe-deployment-7b65d6fcdb-x8vss: 9.1 Mbps
rxdata-deployment-66d6dddf6d-947mh: 4.5 Mbps
softbit-deployment-cb7d6d68b-tm8sw: 1.4 Mbps
mac-deployment-74b4cd5d94-cf28m: 2 kbps
16:58:30
16:59:00
16:59:30
17:00:00
17:00:30
17:01:00
17:01:30
17:02:00
17:02:30
17:03:00
iq-deployment-6d6c4c5796-mbj2n Current: 90.2 Mbps
subframe-deployment-7b65d6fcdb-x8vss Current: 9.1 Mbps
rxdata-deployment-66d6dddf6d-947mh Current: 4.5 Mbps
softbit-deployment-cb7d6d68b-tm8sw Current: 1.4 Mbps
mac-deployment-74b4cd5d94-cf28m Current: 2 kbps
Latency (BS1)
20 ms
15 ms
10 ms
5 ms
0 ms
16:58:30
16:59:00
16:59:30
17:00:00
17:00:30
17:01:00
17:01:30
17:02:00
17:02:30
17:03:00
iq-deployment-6d6c4c5796-mbj2n Current: 0 ms
subframe-deployment-7b65d6fcdb-x8vss Current: 0 ms
rxdata-deployment-66d6dddf6d-947mh Current: 0 ms
softbit-deployment-cb7d6d68b-tm8sw Current: 0 ms
mac-deployment-74b4cd5d94-cf28m Current: 0 ms

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

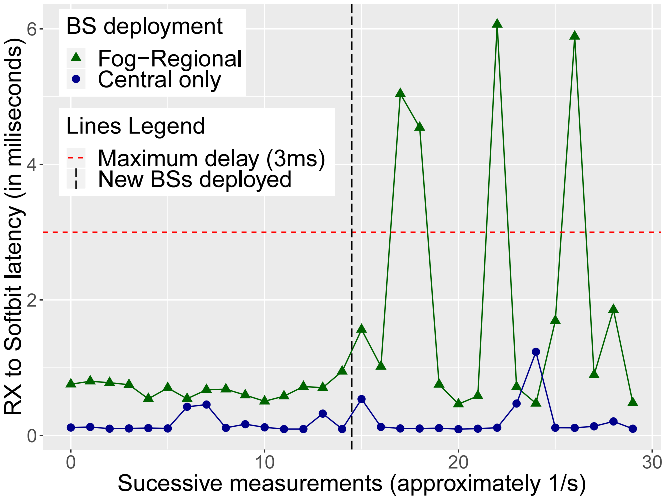


Fig. 8. Latency monitoring result for two BSs' RX to Softbit communication over a 30-second window.

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 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.

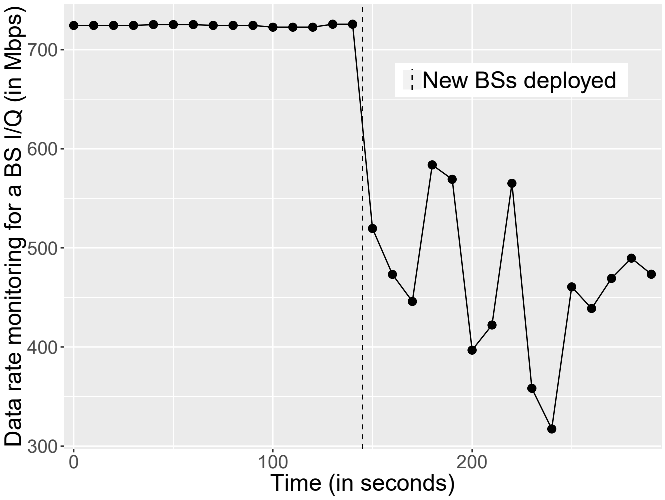


Fig. 9. Data rate result for monitoring a BS's IQ prior and after new BSs deployment.

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.