NFV Service Dynamicity with a DevOps Approach: Insights from a Use-case Realization

Steven Van Rossem*, Xuejun Cai†, Ivano Cerrato‡, Per Danielsson§, Felicián Németh¶,
Bertrand Pechenot‖, István Pelle¶, Fulvio Risso‡, Sachin Sharma*, Pontus Sköldström‖ and Wolfgang John†
*Ghent University iMinds, Department of Information Technology. Email: steven.vanrossem@intec.ugent.be
†Ericsson Research, Cloud Technologies, Sweden ‡Politecnico di Torino, Department of Control and Computer Engineering
§SICS Swedish ICT AB ¶Budapest University of Technology and Economics ‖Acreo Swedish ICT AB

***Abstract*—This experience paper describes the process of leveraging the NFV orchestration platform built in the EU FP7 project UNIFY to deploy a dynamic network service exemplified by an elastic router. Elasticity is realized by scaling dataplane resources as a function of traffic load. To achieve this, the service includes a custom scaling logic and monitoring capabilities. An automated monitoring framework not only triggers elastic scaling, but also a troubleshooting process which detects and analyzes anomalies, pro-actively aiding both dev and ops personnel. Such a DevOps-inspired approach enables a shorter update cycle to the running service. We highlight multiple learnings yielded throughout the prototype realization, focussing on the functional areas of service decomposition and scaling; programmable monitoring; and automated troubleshooting. Such practical insights will contribute to solving challenges such as agile deployment and efficient resource usage in future NFV platforms.**

I. Introduction

Using new possibilities enabled by SDN and NFV, a NFV service production environment has been devised in the EU FP7 project UNIFY[1]. It aims at unifying cloud and carrier network resources in a common orchestration framework, used to efficiently deploy and operate NFV-based services. The platform needs to support new service dynamics, with more flexibility compared to current available cloud platforms. Services deployed in currently available cloud environments have limited options to implement automated elasticity actions like scaling. Commercial cloud providers are heavily focused on web services handling requests and following an N-tier architecture (e.g. web server + databases). OpenStack also follows these commercial cloud features, implementing horizontal and vertical scalability described by its Heat template and triggered by the Ceilometer engine[1]. These services follow a relatively straightforward decomposition strategy where an auto-scaling approach consists of cloning servers and putting a load balancer in front of them. This is in contrast to the NFV services we intend to deploy in the UNIFY framework. There, part of the service might be about rapid packet handling. Thus, adding a load-balancer in front of the packet processing node is not the most ideal scaling strategy to minimize processing delay. Also stateful VNFs need to be supported, so state migration is an important part of the scaling procedure. The scaling of NFV services requires in-depth knowledge of the service, and needs to be controllable/programmable by the service developer. Furthermore, with the support of programmable monitoring and troubleshooting tools, the service platform becomes dynamically adapted to support specialized services. Specialized metrics can be analyzed to optimize scaling triggers and possible failures can be automatically detected and debugged. This is inspired by DevOps principles, allowing updates to the operational service in a very short loop. After introducing the UNIFY NFV framework in Section II, we outline the realization of the elastic router service within this framework in Section III. The main contribution of this paper is in Section IV, where we discuss the lessons learned by implementing three main functionalities:

- A **customized scaling procedure**, made possible by the UNIFY orchestration framework.
- A **programmable monitoring framework**, supporting the deployed service in an automated way by triggering scaling.
- An **automated troubleshooting process** offering short update cycles to fix service failures.

II. The UNIFY Framework

The base platform to deploy the NFV-based service is a PoC following an ETSI NFV MANO-conform architecture - i.e. the UNIFY architecture [2] shown in Fig. 1. Compared with ETSI MANO (left side of Fig. 1), the VNF Manager (VNFM) and Network Function Virtualization Orchestrator (NFVO) support the same high-level functionality. VNFM is responsible for life-cycle management of individual VNFs, whereas the life-cycle management of the whole service is delegated to the NFVO. ETSI's Virtualized Infrastructure Managers (VIMs), i.e., the controllers/managers of compute, storage and network resources, are replaced by the UNIFY Orchestration Layer [3], logically centralizing all resource management functions. Within the UNIFY framework however, it is argued that a central management entity such as the VNFM or NFVO at the service layer is too generic to implement optimized resource control for any service [4]. Therefore an interface connecting the Virtual Network Function (VNF) directly to the Orchestrator is installed. This is depicted in Fig. 1 and enables a fully closed control loop allowing VNF and service developers to autonomously control their resource needs, closer and more adapted to the actual execution environment of the service. This enables new service dynamics, different

S. Sharma has changed affiliation to NEC Labs Europe since project end.
[1]https://www.fp7-unify.eu/

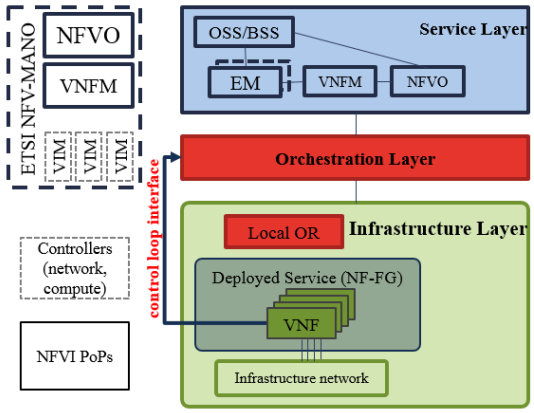

Fig. 1. Basic UNIFY architecture (right side) compared to ETSI NFV-MANO (left-side). The control loop interface enables network services to dynamically control its resources.

from current (commercially) available cloud platforms. In the following section, we describe how this platform is used to support an elastic network service with custom scaling and DevOps-inspired monitoring and troubleshooting support [5].

III. The Elastic Router Service

From a top-level perspective (Fig. 2), the Elastic Router (ER) example is a service that routes traffic between four Service Access Points (SAPs), described by a Service Graph (SG). The SG is translated into a Network Function-Functional Graph (NF-FG) by the Service Layer. This NF-FG is a more practical description of the service, describing the discrete VNFs, SAPs and links to be deployed. The derived NF-FG also contains annotated instructions how to deploy and configure monitoring components (expressed as monitoring intents we call MEASURE [5]). The Orchestration Layer in Fig. 2 deploys the NF-FG on the available infrastructure. In this integrated prototype, both Service and Orchestration Layer are implemented in the ESCAPE[2] orchestration software [6]. The infrastructure nodes are Universal Nodes (UN) [3], also developed in UNIFY[3]. A UN is a high performing compute node enabling the deployment of different VNF types (e.g., various container and VM versions), well suited to run on commodity hardware in data centers and even on resource-constrained hardware such as home/SOHO CPEs. Each UN has a local orchestrator (LO), which deploys the received NF-FG locally. We extend an early ER prototype [7] to be deployed on the UNIFY platform supported by our SP-DevOps monitoring and troubleshooting framework [5]. More implementation details about the complete prototype can be found in [8].

In addition to decomposition at the deployment time of the service, automatic scaling is possible at any time, with no noticeable impact or interruption of the service's quality. During the service's lifetime, the control plane (*Ctrl App* VNF) is responsible to determine how to scale in/out data plane components, implemented as Open vSwitch processes (*ovs* VNFs). The scaling trigger is based on load measurements by the monitoring framework. The actual resource allocation is performed in the UNIFY architecture through the Orchestrator via the control loop interface.

The lifetime of the ER service in our prototype implementation consists of distinct stages, as illustrated in Fig. 2:

1) At initial service deployment, a SG is sent to the Service Layer. This SG is a high-level description of the elastic router (i.e., the external interfaces connected to a monolithic router function).
2) At instantiation, the Service Layer generates an NF-FG based on the SG, supported by the Network Function - Information Base (NF-IB), a database containing the decomposed topology of the ER (see Section IV-A).
3) The NF-FG is passed to a (global) orchestrator which calculates the infrastructure mapping, deciding where the VNFs will be deployed. The local orchestrator (LO) in the infrastructure will deploy the VNFs and trigger the deployment of the monitoring components defined in the monitoring intents (MEASURE) annotated to the NF-FG (see Section IV-B).
4) At runtime, the service can scale elastically. A scaling condition derived from the MEASURE annotations triggers the ER *Ctrl App* VNF to generate a new, scaled version of the NF-FG. Via the control loop interface, the new NF-FG is sent to the orchestrator, which can deploy the new VNFs on the available infrastructure (see Section IV-A).
5) During its lifetime, the service is continuously monitored. Besides scaling, also failure conditions are detected and used to trigger automated troubleshooting (Section IV-C).

IV. Experiences and Insights

The following subsections present the challenges encountered while implementing this demonstrator prototype and integrating the service and monitoring/troubleshooting components with the UNIFY orchestration platform and UNs[4]. We present the learnings structured along three functional areas: service decomposition and scaling; programmable monitoring; and automated troubleshooting.

A. Customized Decomposition and Scaling

Customizable service decomposition and scaling required the integration of several different interfaces and descriptor formats in this prototype. At deployment, customized decompositions of the SG can be stored in the NF-IB, while during the service's lifetime, the *Ctrl App* communicates different service updates over the control loop interface. In addition to this, also monitoring-related aspects need to be included in the service descriptors. These different decomposition and scaling procedures are illustrated in Fig. 2. A short description of the different main interface formats is given here:

[2]Source code available: https://github.com/hsnlab/escape
[3]Source code available: https://github.com/netgroup-polito/un-orchestrator

[4]A related demo paper was submitted to the IM demo session [9]. A video of the demo deployment is available at https://youtu.be/jSxyKBZkVes

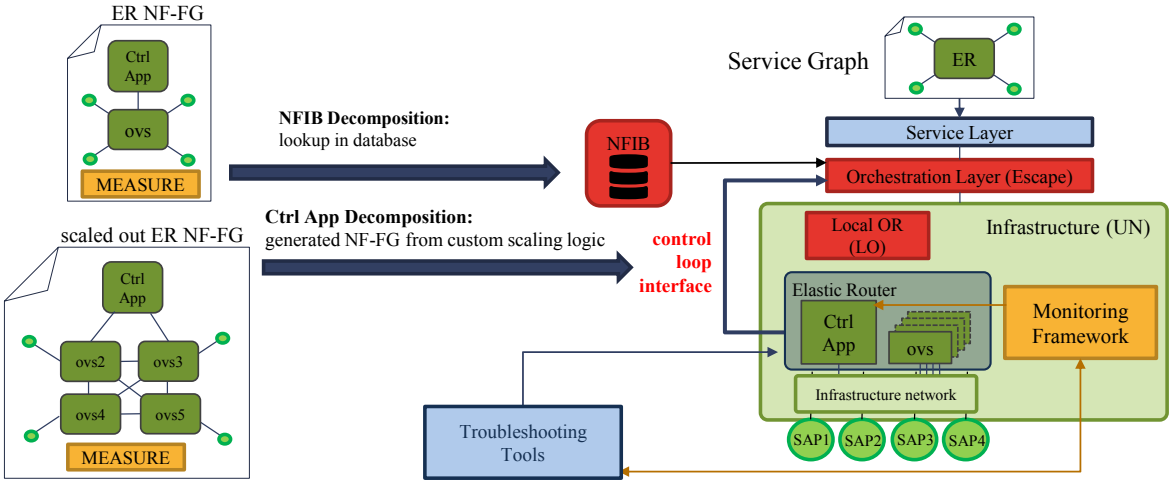

Fig. 2. The UNIFY architecture incl. monitoring and troubleshooting tools, implementing a service-specific scaling mechanism of an Elastic Router (ER).

- The NF-IB is a Neo4j DB which requires a dedicated API and formalism to store and retrieve service decompositions.
- The UNIFY framework defines an XML-based NF-FG format, based on a YANG model, described in [10].
- Both ESCAPE and the UN use two (syntactical) JSON-based formalisms to describe the service to be deployed.
- Service-specific monitoring intents are annotated to the NF-FG in the MEASURE language.
- A novel, secure and carrier-grade message bus (DoubleDecker [11]) is used for communication between orchestration, service, monitoring, and troubleshooting components in the UNIFY platform (more details in Section IV-B.)

This also shows that further standardization regarding service descriptors is needed. Existing efforts (such as TOSCA, ETSI [12]) are still work in progress, while UNIFY proposed a novel format - the NF-FG [8]. Implementing the entire service platform gave further insights on how to use the UNIFY NF-FG to enable the needed service flexibility and configurations.

Learning 1: Enable VNF configuration. For the ER use case, we used Docker as lightweight virtualization environment to run the *Ctrl App* and *ovs* VNFs; this allows the VNF configuration mechanism to be entirely implemented in the Local Orchestrator (LO) without introducing, in the Service Layer, a module dedicated for this purpose. Two types of configuration were necessary to deploy this service: configuration of the IP addresses of the VNF virtual interfaces, and configuration of the VNF software executed inside the Docker container. Both kinds of information must be specified in the NF-FG, and are set up in the container by the LO during the instantiation of the container itself; particularly, VNF-specific configuration is provided by means of Linux environment variables. This means that to fully implement the closed control loop, also configuration mechanisms beyond resource control are needed in the Orchestration Layer.

Learning 2: Scaling without service interruption through the control loop. To achieve zero-downtime, the scaled topology cannot be sent as a single NF-FG update, but must consist out of several incremented updates, via a make-before-break procedure. This ensures that the service's packet flows can be seamlessly re-routed to a set of newly scaled, pre-deployed and pre-configured VNFs. The old, un-scaled VNFs and traffic routes are removed after the new ones are in place. An intermediate topology is created during the scaling procedure where the new, additional data planes (*ovs* VNFs) are deployed next to the original ones. The control loop interface allows the Orchestrator to receive the different NF-FG updates, calculated by the *Ctrl App* during this make-before-break scaling procedure. The LO calculates the difference between the currently deployed and the new version of the NF-FG, hence leaving unaffected VNFs and flow rules to continue their operations during the service scaling. Finally, after state transfer, the old data planes are removed, and traffic from the SAPs is re-routed to the newly scaled data plane topology.

Learning 3: Optimizing scaling speed. In our implementation, scaling-out time was in the order of 25 secs (scaling one *ovs* VNF to four on a single UN hardware node, as illustrated in Fig. 2, see also [9]). This time can be subdivided in:

1) Deploying/removing new compute resources (*ovs* VNFs).
2) State transfer between the old and the newly scaled VNFs.
3) Deploying network updates (updating flow-rules).

The most time is spend on point 3: processing the multiple NF-FG updates and installing the changed service graph edges during the scaling (re-connecting the SAPs and new VNFs). State transfer will also take longer if large flow-tables need to be transferred between *ovs* VNFs, but we did not examine this in our experiment. From an implementation point of view, the processing of the NF-FGs (creating XML-files in the *Ctrl App* and translating them to flow-rule updates in the LO) is the largest bottleneck which we need to examine further.

Learning 4: Modifications to the Service Description (NF-FG). The original NF-FG representation[10] had to be extended in order to support the ER use case. This consolidates the previous two learnings. Particularly, we added the possibility: *(i)* to specify a configuration for the VNFs that are part of the service; *(ii)* to specify a service-specific monitoring intent (in the MEASURE language) and *(iii)* to require that one

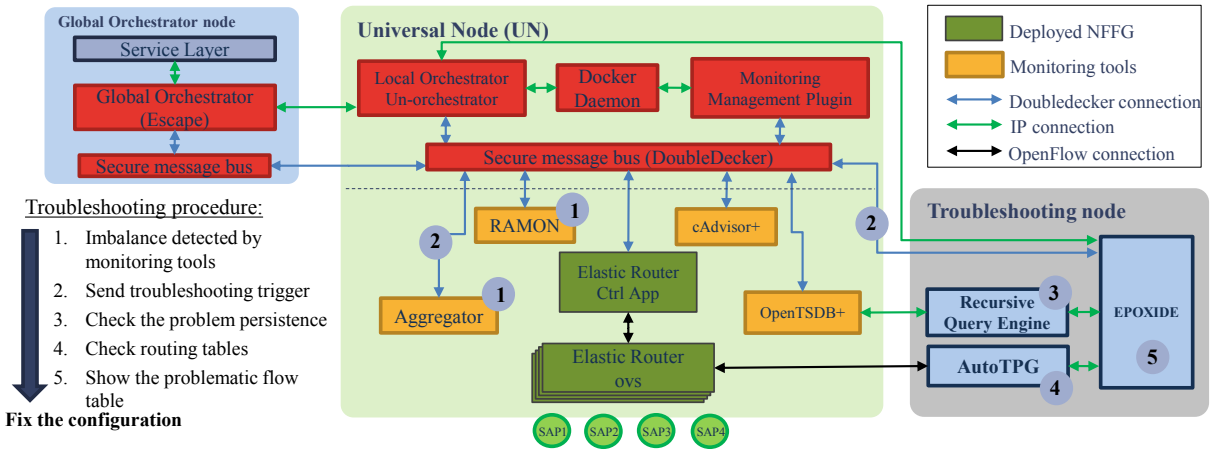

Fig. 3. The detailed system architecture of the prototyped UNIFY orchestration platform, including monitoring and troubleshooting framework.

(or more) VNF(s) is connected to the control loop interface.

We chose a generic approach by making the LO agnostic to the configuration parameters and monitoring configuration. This allows VNFs to define their own configuration parameters (in the form `key=value` pairs) and any update to annotated parts of the NF-FG, like the MEASURE language, can be supported smoothly in the UNIFY architecture.

Learning 5: Security aspects of the control loop. The NF-FG specifies a port for the *Ctrl App* VNF that is connected to control loop interface. This interface is not connected to the rest of the NF-FG, but it is attached to the UN's public interface to the Internet. This security threat would be mitigated if the control loop interface would be implemented through proper connections in the NF-FG. It allows to connect the control loop interface to multiple VNFs, through a firewall that only enables the VNF to talk with the allowed entities (e.g. orchestration, monitoring, and troubleshooting components).

B. A Programmable Monitoring Framework

The ER Ctrl App takes scaling decisions based on triggers from monitoring components. The monitoring framework we apply is a realization of an SP-DevOps observability process as discussed in [5] and is composed of multiple components (see Fig. 3): *(i)* A Monitoring management plugin (MMP) represents a monitoring orchestrator inside a UN. It supports the LO by receiving the MEASURE monitoring intents and translating them accordingly into dynamically instantiated and configured monitoring components (yellow in Fig. 3) realized as Docker containers. *(ii)* DoubleDecker, our secure, scalable and carrier-grade messaging system based on ZeroMQ. In this use case, DoubleDecker[5] is set up with a two-layer broker hierarchy, enabling transport of opaque data between clients in the infrastructure (i.e. the UN), the global orchestration layers and the troubleshooting node. *(iii)* A novel, probabilistic monitoring tool (RAMON[6]) used to estimate the overload risk of links. It does so by keeping lightweight counters in the data plane (i.e. at the port) and models a probability distribution of the traffic in the control plane to estimate the overload risk [13]. In this setup, RAMON monitors the ER ports representing the external SAPs. *(iv)* The OpenSource Docker resource monitoring tool cAdvisor, used to gather CPU and memory usage data of the VNF components (ER containers, green in Fig. 3). *(v)* An aggregator, realizing a generic local aggregation function for monitoring results. This component aggregates raw metrics streamed via DoubleDecker from local monitoring functions (RAMON and cAdvisor) and generates events (e.g., scaling or troubleshooting triggers) that in turn are published on DoubleDecker. *(vi)* A monitoring database (realized in OpenTSDB) which stores all raw-metrics generated by the monitoring functions. It is used as a local logging facility allowing Root Cause Analysis (RCA) during troubleshooting processes (Section IV-C).

By integrating this monitoring framework into our prototype of a UNIFY service platform, we gained valuable practical insights, which we are summarizing below:

Learning 6: Tight integration of VNF and monitoring orchestration and management. In our initial system design we planned to perform orchestration of VNFs and MFs separately: next to the LO responsible for deploying service VNFs (e.g. the ER containers) we utilize a MMP for orchestration and management of monitoring. During the integration work, however, it became apparent that a tighter integration between VNF and monitoring orchestration would be favorable. The following issues had to be dealt with during prototype development:

(i) As an NF-FG contains abstracted representations of VNFs, the final mapping onto infrastructure realization of the VNF can only happen at the infrastructure layer (e.g. a compute/network controller or the UN LO). Thus, for each newly deployed/updated VNF, the LO has to inform the MMP about dynamically assigned realization details such as container ID, IP and port assignments. Only after receiving these details, the MMP is able to properly deploy and configure monitoring functions to observe the relevant service components.

(ii) Dynamic VNFs (like the ER *Ctrl App*) need to be connected with monitoring components to receive events (like

[5]Source code available: https://github.com/Acreo/DoubleDecker
[6]Source code available: https://github.com/nigsics/ramon/

scaling triggers). This requires the LO to match VNF monitoring requirements with available monitoring capabilities and metrics managed by the MMP.

(iii) There are resource dependencies, since both service (VNFs) and monitoring components utilize the same infrastructure resources. This implies that decisions to deploy monitoring components by the MMP effect the resource availability assumed by the LO.

(iv) As Monitoring represents infrastructure capabilities which need to be mapped to service requirements (e.g. SLAs or policies), they need to be taken into account already during service deployment and thus need to be exposed to higher orchestration and service layers alongside other resource descriptions (like compute, storage or network capabilities).

(v) Synchronization between LO and MMP is required in order to avoid monitoring of components that are not active. For instance, the LO must inform the MMP about new, updated, or removed VNFs to avoid generation of false alarms by pausing or reconfiguration of monitoring functions.

Learning 7: VNF/MF life-cycle issues. We ran into issues concerning life-cycle management of individual components (i.e. VNFs) and the service they form. The state of a deployed VNF has to be tracked in order to avoid race conditions e.g., in configuration or activation. For instance, the starting time of a container/VM is not fixed nor bounded, hence the exact time in which that VNF can be configured or activated may vary. Consequently, we learned that the component needs to inform the LO about its current state, i.e. that the application itself has to report when it is actually ready. This becomes very relevant when orchestrating multiple, interdependent components as part of a dynamic service where even small synchronization glitches may lead to errors such as packet losses.

The tracking required for the orchestration, configuration, and activation to function correctly can likely be generalized for both VNFs and monitoring functions to a simple state machine (e.g. `STARTING`, `READY`, `ACTIVE`, `PAUSED`, `STOPPING`). The `ACTIVE` state could also be used to monitor the components liveness, preferably with some watchdogs, as liveness cannot be assessed only by monitoring containers/VMs from the outside world.

Learning 8: Management of the messaging system. Even if the developed messaging system greatly simplified the networking code needed for interconnecting the various components, many problems appeared regarding its management.

For instance, the current pub/sub mechanism does not mandate any registration to use a particular topic - each client can create any topic by simply publishing on it. While this simplifies the use of the messaging system, it makes much simpler to create (by mistake) the wrong topic. This could potentially be solved by tracking active topics or having a separate topic creation and registration process.

Another management related issue is the distribution of public/private keys for authentication with the bus and end-to-end encryption between clients. The keys are currently stored in files which have to be added to the containers either at build time or during start-up. This makes the system somewhat fragile and it is difficult to move containers between different environments or tenants. It is not clear how to best solve this problem, especially in shared virtualized environment where different tenants may not trust each other.

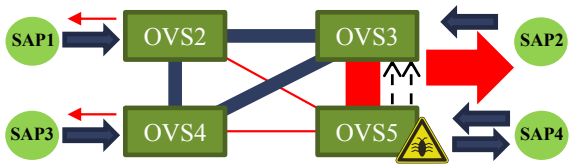

Fig. 4. The effect of a routing table bug in $OVS5$ on the traffic load: $SAP1$ and $SAP3$: decreased overload risk; $OVS2$ and $OVS4$: decreased CPU load; $SAP2$: increased overload risk; $OVS3$: increased CPU load; $OVS5$: unchanged CPU load; $SAP4$: unchanged overload risk.

Learning 9: Complex metrics require special care. An innovation used in this prototype is probabilistic monitoring by RAMON. The advantage of this approach is higher scalability, lower overhead and better observability compared to common monitoring practices. However, probabilistic metrics (e.g. overload risk) are more complex to understand and interpret by a human users, so care needs to be taken when presenting these metrics to make them intuitively understandable and useful.

C. Automated Troubleshooting Mechanisms

The ER *Ctrl App* balances traffic between its datapath instances (i.e. *ovs* VNFs). To demonstrate and test a UNIFY SP-DevOps troubleshooting process [5], we introduced a bug in the routing table of an ovs instance to cause an erroneous imbalance (Fig. 4). In real scenarios, this can be caused by an error in the *Ctrl App* that migrated the routing tables after scaling, the data plane software, or switch firmware.

In order to debug such a complex system, we relied on our EPOXIDE troubleshooting framework [14]. This Emacs based software[7] harnesses the power of special purpose troubleshooting tools by offering an integrated platform to automate certain parts of the troubleshooting process. EPOXIDE is controlled by a troubleshooting graph (TSG)—a formal description of troubleshooting processes—where graph nodes are wrappers for troubleshooting tools or processing functions and edges direct the data flow among these. Additionally, EPOXIDE allows developers to observe the output of each node in a separate Emacs buffer.

EPOXIDE instrumented the following tools to detect problems and analyze symptoms in a dynamically changing topology of the Elastic Router (Fig. 3): *(i)* The Monitoring components RAMON, cAdvisor and Aggregator are used to continuously analyze the performance of the ER *ovs* instances and log this in the OpenTSDB database. When detecting an imbalance, the components trigger EPOXIDE using DoubleDecker messaging. *(ii)* Our novel Recursive Query Engine (RQE)[15] efficiently provides aggregated metrics of historical monitoring retrieved from OpenTSDB instances. It communicates with EPOXIDE (i.e., it receives both queries and relevant deployment details) through a REST API. *(iii)* AutoTPG[8] is

[7]Source code available: https://github.com/nemethf/epoxide

[8]Source code available: http://users.intec.ugent.be/unify/autoTPG/

a verification tool with a REST API that actively probes the Flow-Match Header part of OpenFlow rules for finding erratic behavior in flow matching [16]. It generates test packets to identify flow-matching errors at run-time. To optimize resources, it should only be run during troubleshooting, therefore we deployed AutoTPG as a secondary OpenFlow controller. *(iv)* The `ovs-ofctl` in an auxiliary tool of *ovs* and it allows monitoring and administering OpenFlow switches. In this scenario, it is used for querying *ovs* flow tables.

The applied TSG describes the following sequence of the troubleshooting process. First, EPOXIDE actively listens on DoubleDecker, capturing proactively essential data that later can be used during debugging, as well as troubleshooting triggers by the monitoring framework (steps 1–2 in Fig. 3). Then, by configuring—with automatically queried data from the UN—and calling RQE, the problem's persistence is validated: false positive alarms can be eliminated by analyzing recent CPU utilization. In step 4, EPOXIDE automatically queries *ovs* VNFs in the ER service, connects them one-by-one to AutoTPG, initiates the verification of their flow tables, and evaluates the results. When a switch is marked faulty, EPOXIDE queries its flow table to allow further analysis for troubleshooting personnel. To help tracking the troubleshooting process, EPOXIDE logs events and gives notifications to highlight important events, while also having support for the grouping of specific node outputs together and showing them in a structured manner.

Learning 10: Develop user-friendly troubleshooting. Though EPOXIDE makes the job of troubleshooting easier by parsing and processing data, it comes at the cost of creating a long chain of reading, decoding, and transformation nodes in the TSG. We learned that (i) while EPOXIDE effectively excels at processing of human readable data, JSON/XML processing is limited by the Emacs core. Complex troubleshooting scenarios yield complex graphs in our domain specific language for describing TSG's and these might be hard to follow using the text description. A graphical method to connect nodes might prove to be more beneficial. (ii) Similarly, a high number of different messages can bombard the troubleshooting operators who—without proper tools—could easily miss the relevant ones. (iii) EPOXIDE's graph traversal methods enable operators to further analyze failure symptoms in the tested service. It captures events on DoubleDecker, evaluates them, and provides logging and notification options. (iv) Conditional branches help distinguishing between failure modes and views can filter out unnecessary data, highlighting only the most important ones. (v) The accessibility of historical data helped to reason about the status of the troubleshooting process and the possibility of manual interventions made it easy to test conditional branches.

V. CONCLUSIONS

The implicit flexibility of NFV-based services is challenging the capabilities of current orchestration frameworks. In UNIFY, we proposed a modular NFV platform with novel features. However, only the actual implementation of use-cases can provide important insights to evaluate the modularity of the framework and identify pain points. Thus, we realized an elastic router service to explore and verify the set of functional blocks needed to support the main stages during the lifetime of such NFV-based service. This includes service-specific scaling combined with programmable monitoring and troubleshooting tools, thereby adopting DevOps principles where a fast cycle of debugging and updating a deployed service is a key asset. Based on our experiences while realizing this UNIFY service production platform, we identified and discussed several important insights related to NFV service deployment and operations. We are certain that this type of practical learnings can positively influence ongoing development and advancements in the field of NFV-based network service deployment and operations.

ACKNOWLEDGMENT

This work is supported by UNIFY, a research project partially funded by the European Community under the 7th Framework Program (grant agreement no. 619609). The views expressed here are those of the authors only.

REFERENCES

[1] H. Arabnejad *et al.*, "An auto-scaling cloud controller using fuzzy q-learning-implementation in openstack," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2016.

[2] B. Sonkoly, R. Szabo *et al.*, "Unifying cloud and carrier network resources: An architectural view," in *2015 IEEE GLOBECOM*, 2015.

[3] I. Cerrato, A. Palesandro, F. Risso, M. Suñé, V. Vercellone, and H. Woesner, "Toward dynamic virtualized network services in telecom operator networks," *Computer Networks*, vol. 92, pp. 380–395, 2015.

[4] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar, "Elastic network functions: opportunities and challenges," *IEEE Network*, vol. 29, no. 3, pp. 15–21, 2015.

[5] G. Marchetto, R. Sisto, W. John *et al.*, "Final Service Provider DevOps concept and evaluation," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1610.02387

[6] B. Sonkoly, J. Czentye *et al.*, "Multi-domain service orchestration over networks and clouds: A unified approach," in *SIGCOMM*. ACM, 2015.

[7] S. Van Rossem *et al.*, "Deploying elastic routing capability in an sdn/nfv-enabled environment," in *IEEE NFV-SDN Conference*. IEEE, 2015.

[8] "Deliverable 3.5: Programmability framework prototype report," UNIFY Project, Tech. Rep. D3.5, Jun. 2016. [Online]. Available: https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables

[9] S. van Rossem *et al.*, "NFV Service Dynamicity with a DevOps Approach: Demonstrating Zero-Touch Deployment & Operations," in *2017 IFIP/IEEE IM, submitted to Demo Session*, 2017.

[10] R. Szabo, Z. Qiang, and M. Kind, "Recursive virtualization and programming for network and cloud resources," IETF, Internet-Draft, Sep. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-irtf-nfvrg-unify-recursive-programming-00

[11] W. John, C. Meirosu *et al.*, "Scalable software defined monitoring for service provider devops," in *2015 IEEE EWSDN*.

[12] J. Garay *et al.*, "Service description in the nfv revolution: Trends, challenges and a way forward," *IEEE Communications Magazine*, 2016.

[13] P. Kreuger and R. Steinert, "Scalable in-network rate monitoring," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 866–869.

[14] I. Pelle *et al.*, "One tool to rule them all: A modular troubleshooting framework for sdn (and other) networks," in *Proc. of the 1st ACM SIGCOMM Symp. on Software Defined Networking Research*, 2015.

[15] X. Cai *et al.*, "Recursively querying monitoring data in nfv environments," in *NetSoft Conference, 2016 IEEE*, 2016.

[16] S. Sharma *et al.*, "Verification of flow matching functionality in the forwarding plane of openflow networks," in *IEICE Transactions on Communications*, vol. E98-B, 2015, pp. 2190–2201.