SUPERFLUIDITY

a super-fluid, cloud-native, converged edge system

Research and Innovation Action GA 671566

Deliverable D6.1:

System Orchestration and Management Design and Implementation

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	31 Jan 2018
Actual Date of Delivery to the EU:	31 March 2018
Workpackage Contributing to the Deliverable:	WP6
Editor(s):	Luis Tomas (Red Hat) Daniel Mellado (Red Hat)
Author(s):	Carlos Parada, Isabel Borges, Francisco Fontes (Altice Labs), George Tsolis (Citrix), Michael McGrath, Vincenzo Riccobene (Intel), John Thomson, Julian Chesterfield, Joel Atherley, Manos Ragiadakos (OnApp), Luis Tomas, Livnat Peer, Daniel Mellado (Red Hat), Erez Biton (ALU-IL), Claudio Pisa, Francesco Lombardo, Stefano Salsano, Luca Chiaraviglio, Mohammad Shojafar, Lavinia Amorosi (CNIT), Costin Raiciu, Radu Stoenescu, Matei Popovici, Dragos Dumitrescu (UPB)
Internal Reviewer(s)	Stefano Salsano, Maria Bianco (CNIT)

Abstract: This deliverable reports the efforts of the three WP6 Tasks. It concludes all the efforts done at WP6, finalizing the control framework, the SLA-base deployment design, and completes the symbolic execution checking and anomaly detection tools implementation.

Keyword List: Orchestration, Management, VMs, Containers

Executive Summary

The document provides the results of all the work carried out by the WP6 of the Superfluidity Project. The WP6, named "*System Orchestration and Management Tools*", focused on orchestration and management actions taken at a distributed system level in the Superfluidity architecture. The WP6 activities were split in three tasks, namely T6.1 *"Provisioning and Control Framework"*, T6.2 *"Access-Agnostic SLA-Based Network Service Deployment"*, T6.3 *"Automated Security Verification Framework"*. This single deliverable reports on all the activities and the results of the three tasks for the whole project duration. In order to help the reader to understand the main results without being overwhelmed by the details, we have structured this deliverable as follows. The initial part of the document (around 50 pages from section 1 to section 5) provides a comprehensive overview of the results. This should be detailed enough to understand the value of the contributions and how these contributions are related to the Superfluidity architecture and in general to the evolution of 5G networking. Then we provide three annexes, one for each task, gathering the detailed reports on the activities performed in the project lifetime and on the achieved results. In these annexes, we also include information that was presented in the previous internal deliverables, considering that this document is the only public deliverable of the WP6.

INDEX

List of Figures

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
API	Application Programming Interface
CNI	Container Network Interface
DNS	Domain Name System
EC2	Elastic Compute Cloud
EPC	Enhanced Packet Core
ETSI	European Telecommunications Standards Institute
GTP	GPRS Tunning Protocol
GUI	Graphical User Interface
HOT	Heat Orchestration Template
HTTP	Hypertext Transfer Protocol
ISG	Industry Specification Group
JSON	Javascript Object Notation
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LBaaS	Load Balancing as-a-Service
LB	Load Balancer
LTE	Long Term Evolution
MAC	Media Access Control
MANO	Management and Orchestration
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
NAT	Network Address Translation
NEMO	NEtwork MOdeling
NFV	Network Function Virtualization
NIC	Network Interface Controller
NFV	Network Function Virtualization
NFVO	NFV Orchestrator

NFVI	NFV Infrastructure
NS	Network Service
NSD	Network Service Descriptor
OSM	Open Source MANO
OSS	Operation Support Systems
OvS	Open virtual Switch
QoS	Quality of Service
RAN	Radio Access Network
RDCL	RFB Description and Composition Language
REST	Representational State Transfer
REE	RFB Execution Environment
RFB	Reusable Functional Blocks
SDN	Software Defined Networking
SEFL	Symbolic Execution Friendly Language
SLA	Service Level Agreement
SSH	Secure Shell
TOF	Traffic Offloading Function
TOSCA	Topology and Orchestration Specification for Cloud Applications
UE	User Equipment
URI	Uniform Resource Identifier
VDU	Virtual Deployment Unit
VLAN	Virtual Local Area Network
VIM	Virtual Infrastructure Manager
VIP	Virtual IP
VNF	Virtualized Network Function
VNFD	VNF Descriptor
VNFM	VNF Manager
VM	Virtual Machine
YAML	YAML Ain't Markup Language

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

WP6 focuses on the orchestration and management actions at a distributed system level, building upon WP5 advances. The WP main objectives are resource provisioning, and control and management of network functions as well as applications located at the edge (in this case MEC). To achieve this we propose a framework that targets dynamic scaling and resource allocation, traffic load balancing between virtual functions, or automatic recovery upon hardware failure, among others.

The main objectives from the DoW that the WP's tasks target are:

- OBJ1: design of SLA based network function descriptors
- OBJ2: design of cross management domains resource allocation and placement algorithms
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks in the overall system
- OBJ4: development of platform middleboxes and services
- OBJ5: design of a security framework that allows efficient enforcement of operator policies

There are several points where Superfluidity have contributed/focused to achieve these goals:

- Service behaviour models to support autonomous policy management reacting to current status of the system. For instance, detecting an application having noisy neighbor problems and reacting to it by either performing (QoS) bandwidth limitation, migration or load balancing actions.
- Make OpenStack suitable for C-RAN/MEC components by improving network performance as well as allowing mixed VM and Container environments as well as the possibility of running containers inside VMs or on baremetal nodes.
- Management and Placement in a distributed environment with a system-wide overview as well as with synchronization between the different sites.
- Security framework that enables networking policy assurance checkings, as well as bugs finding.

The document is structured as follows. First an overview of the proposed architecture and its components is presented in section 2. After that, the sections 3, 4 and 5 highlight the main achievements of tasks 6.1, 6.2 and 6.3, respectively. On top of that, there are 3 annexes (1 for each task) attached to present with more details the work carried out as part of each task, as well as the information presented in the previous internal deliverables.

2 System Orchestration and Management Tools Architecture

Some major international operators and vendors started the ETSI ISG (Industry Specification Group) on Mobile Edge Computing (MEC), recently re-branded as “Multi-access Edge Computing”. The MEC paradigm advocates for the deployment of virtualized network services on distributed access infrastructures, which are placed next to base stations and aggregation points, and run on x86 commodity servers. In other words, MEC enables services to run at the edge of the network, so that they can benefit from higher bandwidth and low latency.

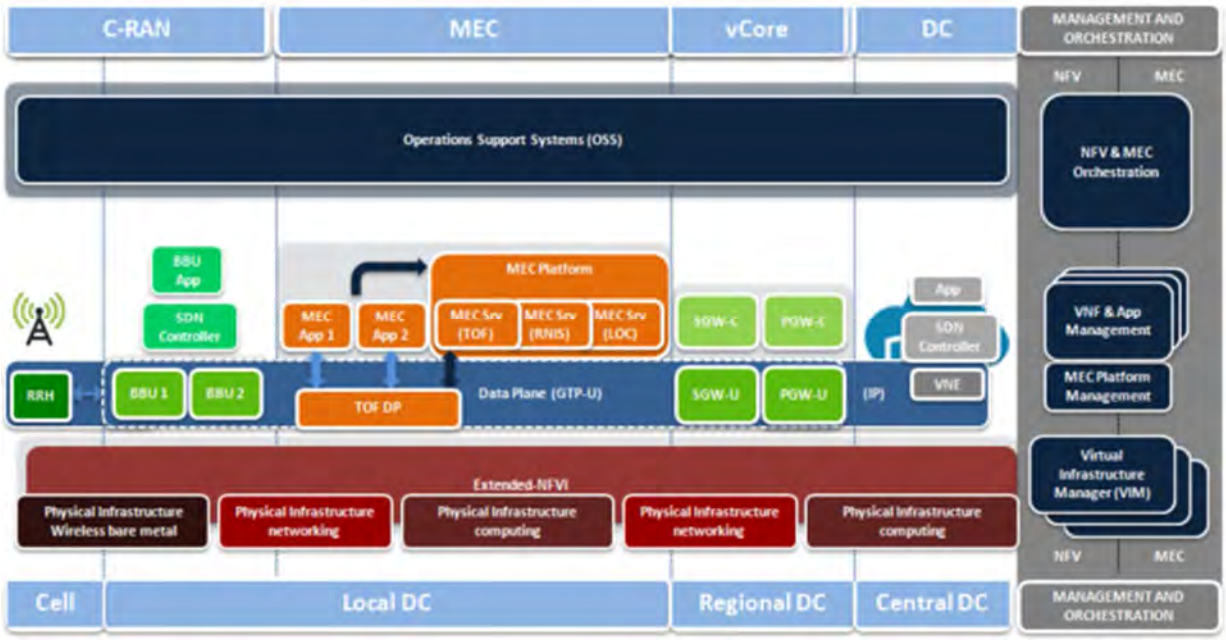


Figure 1: NFV Overview

In such scenario, some network services and applications may be possibly deployed in a specific edge of the network (MEC App and MEC Srv in orange boxes in the above figure). This creates new challenges. On the one hand, the network services reaction to current situations (e.g., spikes in the amount of traffic handled by some specific VNFs/NS) needs to be extremely fast. The application lifecycle management, including instantiation, migration, scaling, and so on, needs to be quick enough to provide a good user experience. On the other hand, the amount of available computational resources at the edge is notably limited when compared to central data centers. Therefore, they must be used efficiently, which results in careful planning of virtualization overheads (time-wise and resource-wise).

Based on the current status of available upstream components (OpenStack, Kubernetes, OSM, ManageIQ, ODL, etc.), the requirement analysis, and the management and orchestration study carried out (see **Annex A, Section A2**, specially section A2.4), we think VMs may not always be a proper

approach for all the needs. Instead, other solutions such as the unikernel VMs and containers should be used. So, we forecast a mixed containers and VMs scenario, at least for the following years. Note, even though there is a high interest in moving more and more functionality to containers over the next years, the priority so far is still set on new applications rather than the legacy ones. And not all the applications will/can be migrated at the same time. On top of that, there is a belief that containers and virtualization are essentially the same thing, while they are not. Although they have a lot in common, they have some differences too. They should be seen as complementary, rather than competitive technologies. For example, VMs can be a perfect environment for running containerized workload (it is already fairly common to run Kubernetes or OpenShift on top of OpenStack VMs), providing a more secure environment for running containers, as well as higher flexibility and even improved fault tolerance, and also taking advantage of accelerated application deployment and management through containers. This is commonly referred to as "nested containers".

Considering the gathered requirements and the state of the art revision, as well as this blend of VMs and Containers, we have proposed and evolved at Superfluidity the next orchestration framework composed of different components at the different actuation levels:

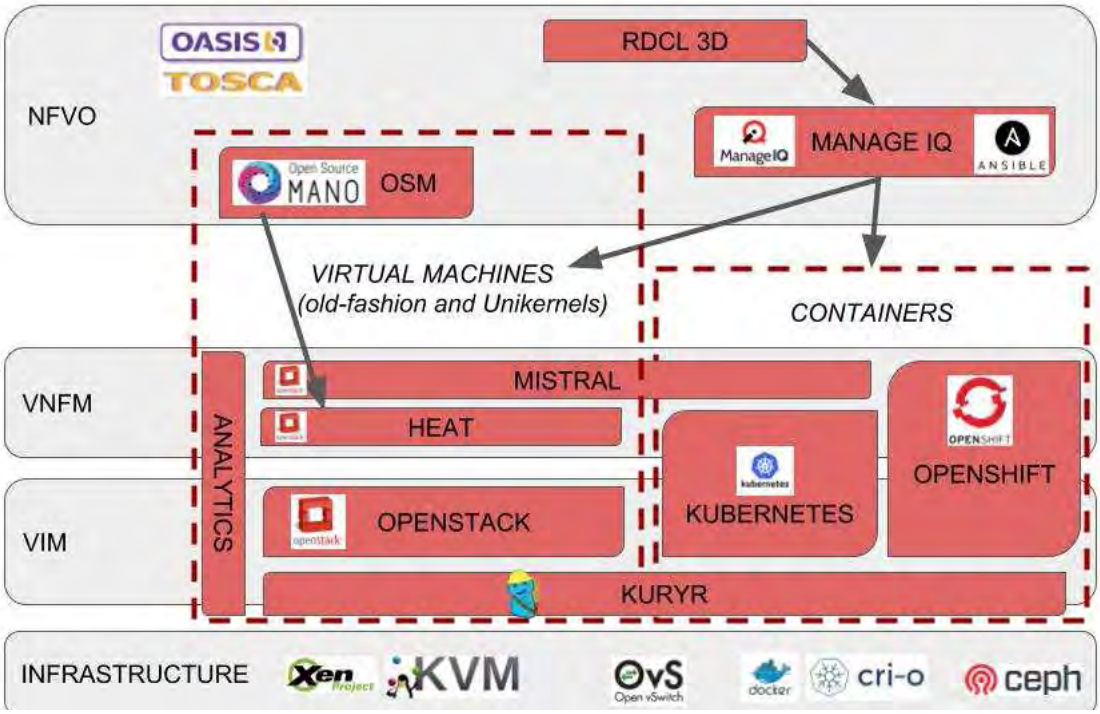


Figure 2: Superfluidity Orchestration Framework

2.1 VIM Options: OpenStack, Kubernetes and Kuryr

Both OpenStack and Kubernetes are the most commonly used VIMs for VMs and Containers, respectively. In addition, Openshift is another well-known framework for container management, leveraging kubernetes power to provide extra dev-ops functionality.

To provide a common infrastructure for both VMs and containers, the problem is not just how to create computational resources, be it VMs or containers, but also how to connect these computational resources among themselves and to the users, in other words, networking. Regarding the VMs in OpenStack, the Neutron project already has a very rich ecosystem of plug-ins and drivers which provide the networking solutions and services, like load-balancing-as-a-service (LBaaS/Octavia), virtual-private-network-as-a-service (VPNaaS) and firewall-as-a-service (FWaaS). By contrast, in container networking there is no standard networking API and implementation. So each solution tries to reinvent the wheel — overlapping with other existing solutions. This is especially true in hybrid environments including blends of containers and VMs. As an example, OpenStack Magnum had to introduce abstraction layers for different libnetwork drivers depending on the Container Orchestration Engine (COE).

Therefore, there is a need to further advance in the container networking and its integration in the OpenStack environment. To accomplish this, we have used and worked on a recent project in OpenStack named ***Kuryr***, which tries to leverage the abstraction and all the hard work previously done in Neutron, and its plug-ins and services. In a nutshell, Kuryr aims to be the "integration bridge" between the two communities, containers and VMs networking, avoiding that each Neutron plug-in or solution needs to find and close the gaps independently. Kuryr allows to map the container networking abstraction to the Neutron API, enabling the consumers to choose the vendor and keep one high quality API free of vendor lock-in, which in turn allows to bring container and VM networking together under one API. So all in all, it allows:

- A single community sourced networking whether you run containers, VMs or both
- Leveraging vendor OpenStack support experience in the container space
- A quicker path to Kubernetes & OpenShift for users of Neutron networking
- Ability to transition workloads to containers/microservices at your own pace.

2.2 VNFMs

To manage VM-base VNFs, we used and extended two already available OpenStack components. We use HEAT to make the deployment actions through templates. And these templates are managed by another OpenStack component designed for that end, named Mistral, to be able to adapt to the given workflows. Besides this, an analytics module is developed as part of Superfluidity to gather information about the VMs performance and build models based on that. This models can later be used to generate optimized versions of the HEAT templates that will better maintain their QoS needs.

As for container based VNFs, there are different options. Kubernetes itself already provides certain VNF management functionality. On top of that, for kubernetes deployments on top of OpenStack VMs, we could make use of the Magnum OpenStack component, that provides extra capabilities for container management.

On the other hand, OpenShift already has other extra container management functionalities on top of kubernetes that can be used through its API, such as Source-to-Image (S2I) that allows users to build their containers/applications directly from git repositories, and even push new containers everytime a new commit gets merge on the git repository -- also enabling easy rollback in case something went wrong. Moreover, as OpenShift leverages Kubernetes functionality, the management capabilities available in kubernetes can be used on Openshift deployments directly.

2.3 NFVO

Finally, in the upper level, we may make use of different NFV orchestrators. There is no complete solution yet, and, as detailed before in this document, different options have been studied. Among them, we decided to make Superfluidity project to target the 2 most promising ones. On the one hand, OSM seems to have some momentum and has a large support by the NFV community. On the other hand, we decided to explore/extend ManageIQ as it is the only one capable of dealing with different VM and Container providers, which, as mentioned before, is needed for the 5G deployments. In addition, ManageIQ allows to easily handle in a single point multiple deployments, as it is the target of Superfluidity, where we can have different cloud deployments at the edges, and the core. Moreover, it provides enough flexibility to include new orchestration actions by relying on ansible playbooks to trigger/execute new required actions. Finally, there is a component named RDCL 3D at the top. This component provides an UI to create the RFB and translate them from (Superfluidity) TOSCA templates to ansible playbooks. These playbooks are uploaded to git repositories consumed by ManageIQ, and they contains the Heat templates needed to handle the OpenStack VMs (and other OpenStack resources) and kubernetes/openshift templates needed for the containers (e.g., pods, replica controller, services, ...). After they are uploaded to the selected git repository, they can be processed by the NFVOs and pushed to the lower layers in their respective template formats. Note that OSM will be deployed through ManageIQ too as a service running in one of the managed clouds.

2.4 Deployment

Once we have reviewed the components at each hierarchy level (VIM, VNFM, and NFVO), as well as the ‘glue’ between VMs and containers (Kuryr), it is important to highlight the different deployment

options. As highlighted in https://ltomasbo.wordpress.com/2017/01/24/superfluidity-containers-and-vms-deployment-for-the-mobile-network-part-2 , the VM and Container deployments can be done in a side-by-side or in a nested way (or a mix of them). Some applications (MEC Apps) or Virtual Network Functions (VNFs) may need really fast scaling or spawn responses and require therefore to be run directly on bare metal deployments. In this case, they will run inside containers to take the advantage of their easy portability and the life cycle management, unlike the old-fashioned bare metal installations and configurations. On the other hand, there are other applications and VNFs that do not require such fast scaling or spawn times. On the contrary, they may require higher network performance (latency, throughput) but still retain the flexibility given by containers or VMs, thus requiring a VM with SRIOV or DPDK. Finally, there may be other applications or VNFs that benefit from extra manageability, consequently taking advantage of running in nested containers, with stronger isolation (and thus improved security), and where some extra information about the status of the applications is known (both the hosting VM and the nested containers). This approach also allows other types of orchestration actions over the applications. One example being the functionality provided by Magnum OpenStack project which allows to install Kubernetes on top of the OpenStack VMs, as well as some extra orchestration actions over the containers deployed through the virtualized infrastructure.

A multi-side deployment example fitting the Superfluidity use cases is presented in the next figure.

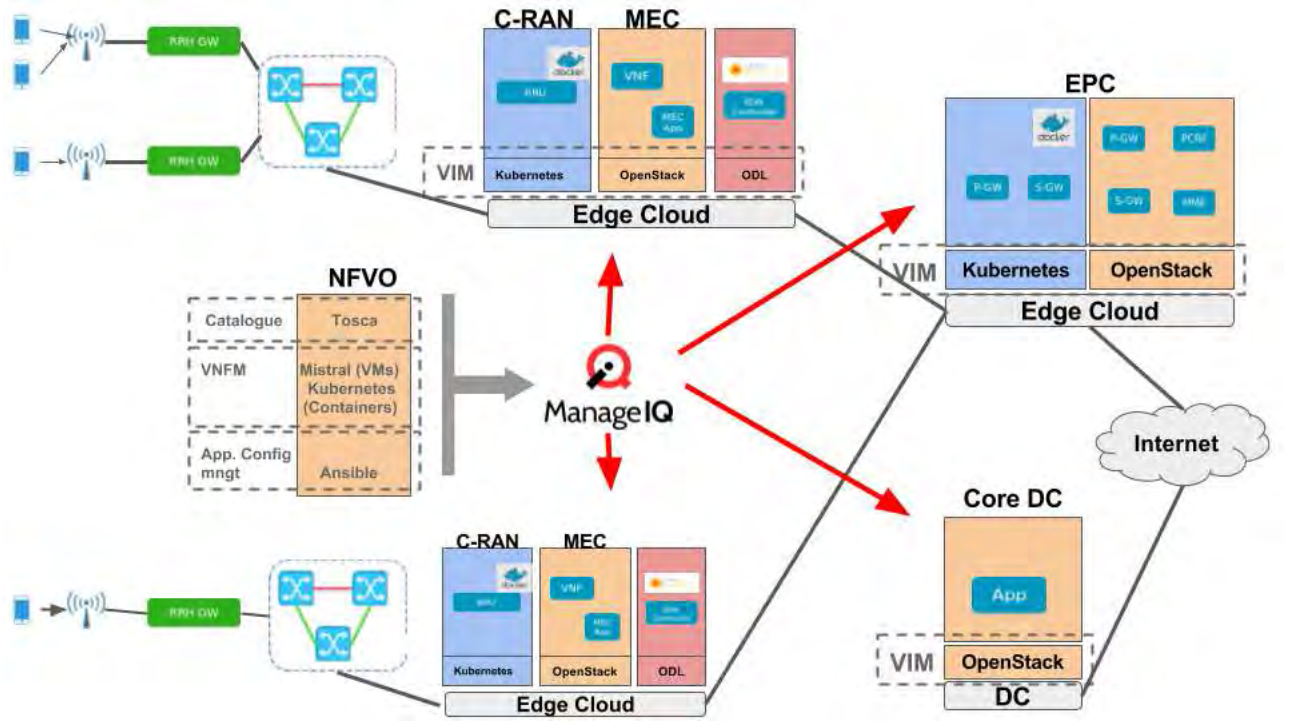


Figure 3: Multi-site Deployment

As it can be seen, there may be several edge clouds in the mobile network, with different roles, such as C-RAN and MEC (co-located in the same edge cloud), or EPC (at the edge of the mobile network). Moreover, outside of the mobile network, there will be other(s) clouds -- known as data centers. In that example, ManageIQ is the NVF orchestration (using Mistral/Heat and kubernetes as VNFMs for VMs and Containers, respectively), and it has a global view of the entire system. Note other local ManageIQ (or OSMs) could also be deployed locally or at some point of the network.

There is a VIM in all of them, but it could be different in different clouds. For instance, it is more common to have OpenStack as a VIM in the big data centers, as the virtualization overhead is not a big concern at that scale. By contrast, at the edge side, the amount of resources is more limited, but the responsiveness need to be higher. Therefore, containers are needed, but so VMs are. In such case the VIM is composed of both OpenStack and Kubernetes/OpenShift, and even SDN controllers (which may also run inside VMs, containers, or baremetal).

Next figure zooms in into one of the C-RAN/MEC edge clouds. There are some servers with Kubernetes and others with OpenStack, and yet use kuryr so that all of them can make use of the Neutron functionality. Thanks to that, some components of the Network Service (NS) or VNF can be running on containers, while others on VMs, and still have layer-2 direct connectivity. For example, this may be required by the firewall load balancer, where one part may need to be on the C-RAN side, and the other on the MEC. There are other use cases where this could also be an advantage, such as the vBRAS, where the routing component could still be a VM with some networking acceleration (DPDK, SR-IOV) for the data plane management, and have a container connected to it in charge of the control plane, i.e., changing the rules applied at the VM when performing the routing actions. Similarly, some MEC Apps, or VNFs may require to be run in containers, due to scaling requirements, or capacity limitations of the edge, while others may need to run on VMs, for instance due to not being yet containerized. This in fact is one of the main advantages of supporting both VMs and Containers: not all the MEC App/VNFs will be containerized at the same time, and some of them may not even ever be.

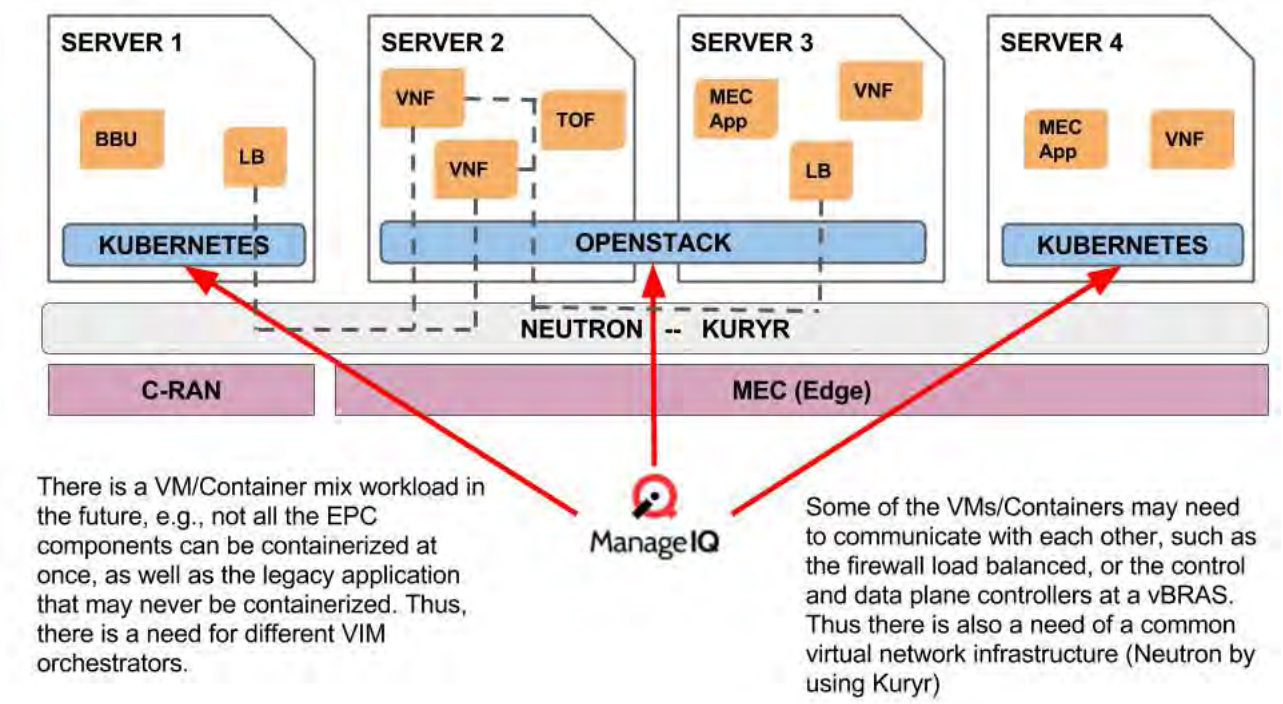


Figure 4: VMs and Containers mix example

3 Task 6.1: Provisioning and Control Framework

The main target of this task is to design and develop a provisioning and control framework and evolve its different components so that:

- it is capable of dynamically scaling network functions as well as enabling their quick instantiation,
- it can perform resource allocation across different sites,
- it enables OpenStack as a suitable VIM for NVF architectures,
- It provides high performance and low latency,
- It can create models of service behavior that can be use as a feedback for the system to improve placement/scheduling/scaling decisions

Having the above points in mind, the Superfluidity efforts have focused on different (correlated) aspects. The main achievements are highlighted next, but more details can be found on Annex A.

3.1 Requirements analysis and state of the art

We have gathered and analyzed the requirements for both NFV Orchestration and MEC engine as an initial step for the design of Superfluidity control and provisioning framework. After that we have research and analyzed different existing tools to decide on top of which one of them to build upon. We compared different orchestrators and tools such as OpenBaton, ManageIQ, OSM, Cloudity or Tacker. The outcome of this evaluation was to select OSM as MEC Manager and ManageIQ as NFVO as our target tools for the Superfluidity control and provisioning framework. More details about the requirements can be found at Annex A, section A1. The details about the state of the art research and orchestration tools comparison can be found at Annex A, section A2.

3.2 Provision and Control Architecture Design

After gathering the requirements and the state of the art, we iteratively designed the architecture of the Superfluidity framework, deciding on the different components that should be part of the ecosystem, as well as the the points where they can be improved to overcome some of their current limitations. The resulting architecture is presented in Figure 2.

The main points to take from this figure are the next:

- We forecast a mixed environment of VMs and containers, as VMs may be more suitable for the Core, while containers are a must at the edge due to its quicker deployment times and higher density. Yet, VMs may also be needed for isolation or supporting legacy applications. What is more, even if containers are more successful in the near future, not all the

applications will be containerized at the same time, therefore even more strong support for the mixed environment assumption.

- The previous assumption lead us to focus on the need for a mixed environment, therefore the focus on Kuryr to enable VMs and containers coexistence at the lower level (VIM level), as well as on ManageIQ at the highest orchestration level (NFVO level) for the multi site environments.
- Moreover, we have worked on enabling 2 type of deployments: containers and VMs deployment side by side (i.e., VMs running on OpenStack, and containers on Kubernetes installed on baremetal server); and nested deployments where containers run inside VMs (i.e., Kubernetes deployments on top of OpenStack VMs).
- Ultimately we have also worked on the option of having a mixed environment where containers can run both on baremetal nodes or on top of OpenStack VMs, and where the infrastructure (VIM level) is provisioned with OpenStack, i.e, OpenStack is deploying the baremetal nodes that will contain the Kubernetes/OpenShift baremetal deployment. More information about this can be found at Annex A, section A3.7. Note this work was presented at DevConf 2018 as a keynote talk.

3.3 Kuryr

Considering that Superfluidity project targets quick provisioning at 5G deployments we decided on the need of containers. However, as there is still need for VMs, we worked on a recent project in OpenStack named Kuryr, which provides production grade networking for container base use cases and that allows the mixed VMs and Containers environment described above. To make it more suitable for the Superfluidity purposes, the next work has been pushed upstream:

3.3.1 Nested containers on VMs without double encapsulation

In order to enable the nested container use case new bindings and controller actions were added to Kuryr. The main idea behind this is to avoid double encapsulation due to the nested environment, with a three-fold objective: 1) reduce overhead due to the double encapsulation; 2) enable an easier debugging in case of networking problems; 3) enable the option of nested containers being on the same Neutron network than other VMs, and even other containers running on baremetal. To enable this behavior we leverage on the Neutron trunk ports feature and have developed new kuryr nested drivers as well as the cni plugins (i.e., Container Networking Interface) to enable them. More information at Annex A, section A3.1.2.

3.3.2 Ports Pool Optimisation

Every time a container is created or deleted Kuryr makes a call to Neutron to create or remove the port used by the container (even some extra calls for the nested case). Although the time needed for this (up to a couple of seconds) is reasonable for VMs, it is not for containers. Consequently, to minimize the impact of Neutron interactions during container lifecycle management (for example containers boot up), we proposed a blueprint and the follow up development implementation to maintain a pool of pre-created Neutron resources that are ready to be used by the containers. Thus avoiding the Neutron interactions during the container boot up process (as pool repopulation is performed as a background activity), with the consequent speed up. In turns it reduces the load on the Neutron server when many containers are booted up at once, also helping to indirectly speed up other actions such as creating the load balancer needed for the kubernetes/openshift services. More information about the pools implementation can be found at Annex A, section A3.1.3.

3.3.2.1 Performance Evaluation

We have also evaluated the performance benefit of the pool implementation at scale, and with different SDN backends, in this case OVN and ODL. For both of them the results were similar and led to the conclusion that pools are a must for medium/big environments, achieving improvements. For more detailed information see Annex A, section A3.1.3.1.

3.3.3 Load Balancer Integration

In addition to the effort for supporting nested containers as well as for speeding up container booting time on OpenStack-based deployments, we have also worked on making available the OpenStack LoadBalancer components to Kubernetes/OpenShift. This means that whenever a Kubernetes/OpenShift service is created, Kuryr will create an OpenStack load balancer for it and add the pods (i.e., their associated neutron ports) as members of it. We have integrated in Kuryr both Neutron LBaaSV2 as well as the new Octavia LBaaS OpenStack project (including their 2 modes, for layer-2 and layer-3 load balancing). For more information on this check Annex A, section A3.1.4.

3.3.4 Integration with Exemplar SDNs: OVN, ODL, DragonFlow

In addition to the previous kuryr extensions, we also focused on supporting different SDN backends so that we can leverage kuryr functionality on OpenStack clouds regardless of their SDN selection. In addition to the standard ML2/OVS driver that comes with OpenStack out of the box, we have tested and make the needed changes to also support OVN, ODL and DragonFlow SDN controllers. In addition, we even perform our ports pool performance testing at scale with two of those SDNs: OVN and ODL. For more information see Annex A, section A3.1.5.

3.3.5 CNI-Split for performance improvement

Originally, Kuryr's CNI (Container Network Interface) consisted on an executable entry point that performed several functions, such as start a watcher on the pod CNI requests or handle the events until it sees the vif annotation. This makes it scale in a bad way as it would require one new https connections to the K8s API per pod. In order to solve this we have split Kuryr CNI into two components:

- CNI driver: Kuryr kubernetes integration takes advantage of the kubernetes CNI plugin and introduces Kuryr-K8s CNI Driver. Based on design decision, kuryr-kubernetes CNI Driver should get all information required to plug and bind Pod via kubernetes control plane and should not depend on Neutron.
- CNI daemon: It runs on every Kubernetes node. It is responsible for watching pod events on the node it's running on, answering calls from CNI Driver and attaching VIFs when they are ready.

As the driver would communicate via socket with the daemon and it's much more lightweight, the scaling issue gets solved by this split. More information at Annex A, section A3.1.6.

3.3.6 Containerized Components (controller and CNI)

It is possible to install kuryr-controller and kuryr-cni on Kubernetes as pods. This follows the approach from Kubeadm linux installation guide.

This functionality allows Kuryr to be installed as a Kubernetes component:

- Kuryr CNI as daemon set
- Kuryr Controller as a pod

This would allow to get all the benefits from a container lifecycle in Kuryr and deeper integration within Kubernetes. Also, it would allow you to test any new code change just by rebuilding the images. Further information can be seen at Annex A, section A3.1.7.

3.3.7 RDO-packaging + Upstream CI

In order to integrate Kuryr with RDO OpenStack distribution (to make it fully/easy available for everyone as well as facilitate testing), we have produced rpm packages to ease up the installation and testing. In addition, we have triggered upstream CI to ensure quality testing of the produced code leveraging on the OpenStack functional testing framework, Tempest.

In addition to this, we've been setting our own CI/CD system, based on what the OpenStack community has been setting up and we've developed a testing framework plugin for the upstream testing system, Tempest. More information about this is detailed at Annex A, section A3.1.8.

3.4 Load Balancing and Service Function Chaining (SFC)

Based on the requirements analysis of the use cases, we identified Load Balancing and Service Function Chaining (SFC) as important capabilities of the Superfluidity platform. After outlining the requirements and specifications of these two areas (table in Annex A, section A1.1), we have:

- Analyzed the state-of-the-art of both Load Balancing (Annex A, section A3.5) and SFC (Annex A, section A3.6), in terms of adoption by the open source projects (OpenStack, OPNFV and OVS).
- On the Load Balancing topic, invested in the Load Balancing as a Service (LBaaS) framework of the OpenStack VIM. Complementing the Kuryr container networking integration (see section 4.3.3), we integrated a commercial Application Delivery Controller (ADC), Citrix NetScaler, using an open source OpenStack LBaaS plugin, which we certified against all OpenStack versions that were released over the course of the project. As part of WP7 activities, we have compared the NetScaler Load Balancing backend against the open source ones that are part of OpenStack. Finally, we implemented open source Ansible modules for NetScaler ADC, which automate the post-deployment configuration of the NetScaler servers and their services, achieving perfect alignment with the relevant support that was introduced to the ManageIQ NFVO (see section 3.6). Annex A, section A3.5 provides more details on these activities.
- On the SFC topic, we have introduced support for Network Service Header (NSH) in NetScaler. As a result, NetScaler ADC can play the role of the Service Function in the SFC architecture. The NetScaler instance receives packets with Network Service headers and, upon performing the service, modifies the NSH bits in the response packet to indicate that the service has been performed. In that role, the NetScaler appliance supports symmetric service chaining with specific features, for example, INAT, TCP and UDP load balancing services, and routing. Considering the complexity of the current SFC solutions and the initial maturity level of their implementations, we have also investigated a promising innovative approach to support SFC, based on IPv6 Segment Routing (SRv6). Please refer to Annex A, section A3.6 for more information on these activities.

3.5 OSM Evaluation and Integration

Open Source MANO (OSM) is a project supported by the ETSI standardization body, aiming to develop open source software that implements the main MANO components of the ETSI NFV framework: the VNF Manager (VNFM) and the NFV Orchestration (NFVO). Despite the OSM relation with the ETSI NFV framework, this tool has been used by the Superfluidity project basically to implement the components of the ETSI MEC (Multi-access Edge Computing). In particular, to implement the Mobile/Multi-access Edge Orchestrator (MEO) and the Mobile/Multi-access Edge Platform Manager (MEPM), which have clear similarities with the NFVO and VNFM, respectively. The work performed by the Superfluidity project in this scope was related to the customization of the OSM tool to the

specificities of the ETSI MEC framework, as well as the differences between the Applications and the VNF/NS concepts.

The status of OSM project has been reviewed, leading to new requirements and a more fine grain information about the strong points as well as the missing features. In addition, integration actions into the Superfluidity framework has taken place, being provisioned from the ManageIQ, and being in charge of the MEC orchestration. For more information, see **Annex A, section A2.2.4, Section A2.3 and Section A3.3.**

3.6 ManageIQ as a NVFO

ManageIQ is a management project that enables managing containers, virtual machines, networks and storage from a single platform, connecting and managing different existing clouds: OpenStack, Amazon EC2, Azure, Google Compute Engine, VMware, Kubernetes, OpenShift, etc.

The main reason for choosing ManageIQ as an NFVO is due to being able to work with both VMs and Container providers. However, after feature analysis, we discover a few gaps that needed to be address for the Superfluidity purposes.

3.6.1 Ansible execution support

The main concern about the existing NFVO tools was the lack of applications life-cycle management actions for container. Therefore we worked on an extension to fix this gap on ManageIQ. The proposed extension is based on supporting ansible playbook execution within ManageIQ. Therefore, any action (as ansible is agent-less) can be triggered in any of the providers, for example, deployment of a HEAT template for the OpenStack/VMs case, or deployment of kubernetes/Openshift templates for the Kubernetes/Containers case. The user then only need to push their desired playbooks to their associated git repositories, and then select them from the ManageIQ UI to execute them in the proper cloud/provider. More details available at **Annex A, section A3.4.1.**

3.6.2 Multi-site support

As a follow up extension for the previous point, we see the need of synchronizing actions across different sides. Hence we worked at supporting multi-site deployment from ManageIQ. Up to now, with ManageIQ you can manage different providers (i.e., different sites), but not execute a set of actions that involves several of them at the same time. We have added to the ansible support at ManageIQ the option to include a host file where you can specify the different sites where the playbook need to be executed -- just by also including into the ansible playbook tasks the info about where (of those sites) they need to be executed. For more information check **Annex A, section A3.4.2.**

3.7 OpenShift-Ansible

In order to easily consume all the above features, as well as to better integrate with current cloud environments, we have contributed to a set of playbooks that allow you to install OpenShift on top of different cloud infrastructures: OpenStack, Amazon, Azure, and Google Compute Engine.

Our efforts have focused on implementing kuryr roles inside the openshift-ansible playbooks so that kuryr components can be deployed on the OpenShift installation on OpenStack, in a containerized way. Thanks to this, it is possible to install OpenShift in an existing OpenStack deployment with kuryr configured by just executing one playbook. More details at Annex A, section A3.7.

3.7.1 Integration with ManageIQ

In addition to the openshift-ansible contributions, we have also integrated it into ManageIQ for an even more simple user experience. Thanks to the integration, a tenant can simply ask for an OpenShift deployment from ManageIQ UI (e.g., ask for a deployment with 1 master node, 1 infra node and 5 worker nodes) and it will be automatically deployed (by executing the openshift-ansible playbook with the configured parameters) on the selected OpenStack provider, on top of OpenStack VMs , giving the user a functionality similar to have: OpenShift as a Service. More details at Annex A, section A3.7.1.

3.7.2 Baremetal containers support

Finally, we extended the openshift-ansible playbooks to also support provisioning baremetal nodes. This adds the flexibility of having an OpenShift/Kubernetes installation that runs both on top of OpenStack VMs as well as on baremetal nodes. This is specially relevant for NVF deployments where some components may require to run (containerized) on baremetal nodes for increase performance, but still being on OpenStack neutron networks so that they can talk to other VMs or nested containers transparently. This work also enables the previous ManageIQ integration to ask for the number of VMs and baremetal nodes that will be used for deploying the OpenShift components. This contribution was presented at a keynote talk at DevConf 2018. More details are provided at Annex A, section A3.7.2.

3.8 RDCL 3D

As shown in Figure 2, the RDCL 3D tool is a part of the NFVO layer in the Superfluidity provisioning and control architecture. The RDCL 3D tool offers a GUI to the user (i.e. the service designer / network operator). Its role is to manipulate the information models of network services and service components (VNFs and more in general RFBs) and to interact with Orchestrators. The RDCL 3D tool

per se is agnostic to the information model, and can be specialized to support different ones. In particular, following the Superfluidity architecture based on the concept of RFBs and of multiple RFB Execution Environments, we have considered information models that are more complex than the state-of-the-art models considered for NFV. The new features that we have considered include the "nested" decomposition of VNFs into more granular RFBs, and the support of traditional VMs, containers, Unikernel VMs.

In the context of WP6, The work on RDCL 3D tool performed in other WPs has been extended to be integrated with the WP6 reference architecture, in particular for the support of -- mainly regarding the integration with ManageIQ and Ansible.

The functionality developed in RDCL 3D allows has included a script translating the Superfluidity data model descriptors into Ansible playbooks that are uploaded to a Git repository where ManageIQ will consume them. More details about this integration effort can be found at Annex A, section A3.8.

3.9 Service Characterisation Framework and Deployment Template Optimisation

Resource allocation is important in the context of "virtualisation" of network services as it plays an important role in both service assurance and Total Cost of Ownership (TCO). In order to provide scalable rationalisation of service resources allocations and TOC we developed a framework that provides orchestration of an experimental lifecycle, management of data collection and analysis of collected data. The output of the framework are optimised deployment templates to deliver specific levels of performance in compliance with required service level objectives (SLOs).

The service characterisation framework was applied to the optimisation of a deployment template for the Unified Origin video transmuxing workload using the WP7 Superfluidity demonstrator implementation. Resource and configuration options focused on vCPU's, RAM, vNIC and Memory Page Size. The results obtained show that varying the combination of the four parameters it is possible to identify three unique performance classes:

- Class A: Throughput > 180 Mbps and Latency < 0.92 ms
- Class B: Throughput > 150 Mbps and Latency < 1.05 ms
- Class C: Best effort (no specific requirements) for both throughput and latency

Analysis of the results in relation to the configuration parameters identified that using SR-IOV can positively impact both throughput and latency and this configuration on its own offers guarantees of good level of performance, placing the service performance within SLA Class A range. At the same time, this represents the most expensive solution from an infrastructure perspective, which might

lead the service provider to choose an option corresponding to a lower cost in case the user requirements fall into SLA Classes B.

In summary, the work carried out was as follows:

- Defined a high-level architecture of a Deployment Template Optimisation Framework.
- Refactored the framework developed in Task 4.1 in order to support different application plugins such as template optimisation.
- Implemented support for Open Source MANO Orchestrator API, provided by Riftware (rift.io) via the framework in order to perform automated deployment and termination of VMs.
- Conducted experimental campaign to identify an optimised template for the Unified Origin workload based on different allocations for resources and configuration options.

More details on this work can be found in Annex A, section A3.9.

4 Task 6.2: Access-Agnostic SLA-Based Network Service Deployment

The work of task 6.2 targeted 3 objectives of work package 6, namely,

- OBJ1: design of SLA based network function descriptors,
- OBJ2: design of cross management domains resource allocation and placement algorithms,
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks and entity in the overall system,

We further structure this section that summarizes the work in Task 6.2 according to the above objectives. While the following sections provide overview of the work and its results, more detailed information is available in **ANNEX B: Access-Agnostic SLA-Based Network Service Deployment**.

4.1 SLA-Based descriptors

4.1.1 NEMO modeling language

Superfluidity has considered NEMO as a modelling language and promoted its ideas via extensions. NEMO [1] is a human-readable command language used for Network Modelling. It is placed by the authors in the scope of Intent-Based Networking (IBN), since it is more descriptive and prescriptive. The NEMO project has launched a series of efforts to get the language standardised. As such, the IBNEMO project within the OpenDaylight (ODL) community is classified in the Intent-Based northbound interfaces (NBIs) group.

NEMO provides basic network commands (Node, Link, Flow, Policy) to describe the infrastructure and controller communication commands to interact with the controller.

We introduced extensions to the Node definition command to import TOSCA or OSM based descriptors as Node definitions. Since Node models can make use of previously defined node models, the resulting language would be recursive and therefore support our notion of (recursive) reusable function blocks.

This concept has been proposed as an Internet draft at the NFV research group (NFV-RG) of the IRTF [2]. In addition to the import process proper, two additional features are defined in NeMo: 1.- the ConnectionPoint to map the VNF's interfaces that are significant in the function description and the connections between them, and 2.- the CONNECTION as a way to express the relations between the enhanced VNFCs (here NodeModels) in a service graph.

Importing OSM VNF Descriptors is proposed in the draft as a two-step process, where the descriptor is first imported and then used to provide the pointers to the connection points. The proposed syntax to import VNFDs is:

```
CREATE NodeModel sample_vnf
    VNFD  https://github.com/nfvlabs/openmano.git
/openmano/vnfs/examples/dataplaneVNF1.yaml;
    ConnectionPoint data_inside at VNFD: ge0;
    ConnectionPoint data_outside at VNFD: ge;
```

The proposed way to define service graphs in NeMo is:

```
CREATE NodeModel          complex_node
    Node      input_vnf     Type  sample_vnf;
    Node      output_vnf    Type   shaper_vnf;
        ConnectionPoint input;
        ConnectionPoint output;
       Connection input_connection Type p2p EndNodes input,   input_vnf.data_inside;
Connection output_connection     Type   p2p   EndNodes   output,   output_vnf.wa   ;
Connection internal Type p2p EndNodes input_vnf.data_outside, output_vnf.lan;
```

The implementation details are available in Annex B.

4.1.2 Nested execution environments

In this work we extended the ETSI NFV ISG specification [3][4] to support nested VDUs using heterogeneous technologies.

Specifically, we have extended the VDU information element contained in the VNF Descriptors to reference different types of Execution Environments that can be instantiated within a VDU and described by means of some descriptor.

Further details are available in Annex B, section B2.

4.2 Cross management domains resource allocation and placement

4.2.1 Data center resource Allocation and Placement

Despite the ever-growing popularity of Network Function Virtualization (NFV), we are still far away from having large scale fully operational NFV networks. One of the main obstacles on this path is the performance of the network functions in the virtual environment. The hardware middleboxes that are in use by network operators today are specifically designed to provide the needed high performance (and high reliability), but getting the same level of performance from commercially off-the-shelf hardware is much more challenging. Hardware accelerators (such as DPDK and SRIOV) were

developed specifically for this purpose, yet the deployment of high performance service chains in a virtual environment remains a complex handcrafted process (e.g., as indicated by Intel's performance reports).

Service chain management in such scenarios is mostly static and operators lose one of the main attractive features of NFV -- the ability to dynamically allocate resources according to the current need. Such a dynamic mechanism would allow a much more efficient utilization of resources, since the same physical resource can be used by different VNFs when needed. Thus, achieving both high performance and agility, by being able to dynamically change the resource allocation of service chains, remains a great challenge.

Identifying near optimal deployment mechanisms for NFV service chains has recently received significant attention from both academia and industry. However, to the best of our knowledge, existing studies do not consider the cost of resources required to steer the traffic within a chain, which is non-negligible for deployment of packet intensive chains such as in the domain of NFV. Therefore, typical models (e.g., in NFV orchestrators) might either lead to infeasible solutions (e.g., in terms of CPU requirements) or suffer high penalties on the expected performance.

In our previous contribution, we focused on evaluating and modeling the virtual switching cost in NFV-based infrastructure. Virtual switching is an essential building block that enables flexible communication between VNFs but it also comes with an extra cost in terms of computing resources that are allocated specifically to software switching to steer the traffic through running services (in addition to computing resources required by the VNFs). This cost depends primarily on the way the VNFs are internally chained, packet processing requirements, and accelerating technologies (such as DPDK).

Example

Figure 5 illustrates a possible deployment of four service chains on three identical physical servers (A, B and C). As one can see, service chain φ^1 *is composed of three VNFs -*$\varphi^1 = \langle \varphi_1^1, \varphi_2^1, \varphi_3^1 \rangle$*-,* φ^2 *is composed of four VNFs,* φ^3 *is composed of five, and* φ^4 *is composed of two VNFs. In the depicted deployment (shown in Figure 5), servers A and C have the same number of deployed VNFs and thus may have the same computing resource requirement for processing. However, determining the amount of processing resources (CPU) needed for switching inside these server is far from being straightforward. In some cases, the deployment can be infeasible due to lack of sufficient computing resources for the switching task. The amount of computing resources needed for the internal switching depends on the structure of the chaining in the server, and the amount of traffic associated with each chain.*

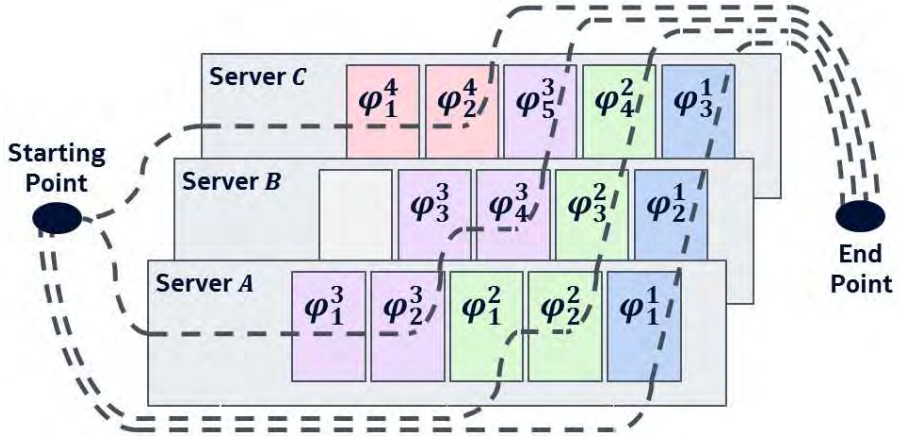


Figure 5: Example of possible deployment of four service chains on top of three

Existing work that measure and evaluate the performance of the internal switching mainly focus on two types of deployment strategies: a ***distribute*** strategy where each VNF in the chain resides on a different server (for example, service chain φ^1 in Figure), and a ***gather*** strategy where the entire chain is grouped on the same server (for example, service chain φ^4 in Figure). Some works refer to these two deployment strategies are respectively referred to as "north/south" and "east/west" deployments. We have analysed these two placement strategies, and developed a cost function that predicts the amount of computing resources needed for the internal switching. Interestingly, OpenStack/Nova global policies also follows these two deployment strategies. Namely, after a basic filtering step that clears resources according to local-server constraints (e.g., CPU cores, memory, affinity, etc.), it implements two types of global decisions that boil down to: ***load balancing*** of a certain metric (i.e., spread VNFs across servers), and; ***energy saving*** of a certain metric (i.e., stack VNFs up to the limit).

In the following, we extend this work and describe a novel SLA based resource allocation scheme for network services.

We consider arbitrary deployments (like the one described in Figure 5) and develop a general service chain resource allocation strategy that considers both the actual performance of the service chains as well as the needed internal switching resource. This is done by decomposing service chains into sub-chains and deploying each such sub-chain on a (possibly different) physical server, in a way that minimizes the total switching overhead cost. Of course, there are exponentially many ways to map the sub-chains to servers. We introduce a novel algorithm based on an extension of the well-known reduction from the weighted matching to the min-cost flow problem, and show that it gives a near optimal solution with a better running time than exhaustive search.

We evaluate the performance of our algorithm against the fully distribute or fully gather solutions, which are very similar to the placement of the de-facto standard mechanism commonly utilized on

cloud schedulers (e.g., OpenStack/Nova with load balancing or energy conserving weights) and show that our algorithm significantly outperforms these heuristics (up to a factor of 4 in some cases) with respect to operational cost and the ability to support additional network functions.

Our main contributions to this work are:

- NFV deployment cost model. We develop a general switching cost model that predicts the switching related CPU cost for arbitrary deployments and evaluate its accuracy over a real NFV environment.
- Optimal deployment mechanism. We develop an efficient online placement algorithm (OCM -- Operational Cost Minimization) that uses this new cost model to minimize the switching cost of service chain requests, thus allowing more network functions to run on the same NFV infrastructure in a more efficient way. We evaluate the expected performance of this novel algorithm and show that it can significantly increase utilization.

These contributions enable NFV providers to design a new pricing models for service chaining, that quantifies the actual consumed resources by the infrastructure (in addition to the resources consumed directly by the VNFs).

4.2.2 The Operational Cost of Switching

We first define a model that captures the operational cost of virtual switching for a given server. Namely, given a placement function P that deploys all service chains in $\Phi = \langle \varphi^1, \varphi^2, \dots, \varphi^m \rangle$ on the set of servers $S = \{S_1, S_2, \dots, S_k\}$, we can infer the set of sub-chains (originating from possibly different service chains) to deploy on server S_j. Our goal is to develop a function that predicts the CPU cost of server S_j.

4.2.2.1 Virtual Switching

In virtualization-intense environments, virtual switching is an essential functionality that provides isolation, scalability, and mainly flexibility. However, the functionality provided by software switching also introduces a non-negligible operational cost making it much harder to guarantee a reasonable level of network performance, which is a key requirement for the success of the NFV paradigm. Assessing and understanding this operational cost and particularly the cost associated with virtual switching is a crucial step towards driving cost-efficient service deployments.

Several recent publications addressed the performance of intensive traffic applications in NFV chaining settings. In most industry, related works the goal is to define the setting that provides the best performance on a specific hardware, and not on the switching cost of a given service chain under

a certain setting. Our previous contribution is different as it does try to evaluate the switching cost but it does so only for the special cases of gather and distribute.

Yet the results of these recent studies indicate that the operational cost of deploying service chains depends on the installed OvS (either kernel OvS or DPDK-OvS), the required amount of traffic to process, the length of the service chain, and the placement strategy. Moreover, it appears that there is no single strategy that is always superior, with respect to the operational cost, and that the best strategy depends on the system parameters and the characterization of the deployed service chains.

4.2.2.2 The Cost of Virtual Switching

Let $\{\varphi^1_{s_1 \twoheadrightarrow e_1}, \dots, \varphi^r_{s_r \twoheadrightarrow e_r}\}$ be a set of r sub-chains, each is part of a decomposition from possibly different service chain. Each sub-chain $\varphi^w_{s_w \twoheadrightarrow e_w}$ (for $1 \leq w \leq r$) might carry different traffic requirements, that are defined by service chain $\varphi^w \in \Phi$. Figure 6 illustrates such a deployment on server S_j. The total cpu-cost consumed by the guests (denoted by $C^v_j(P)$) is just the sum of the required CPU for each VNF, but calculating the hypervisor switching cpu-cost $C^h_j(P)$ is much more involved.

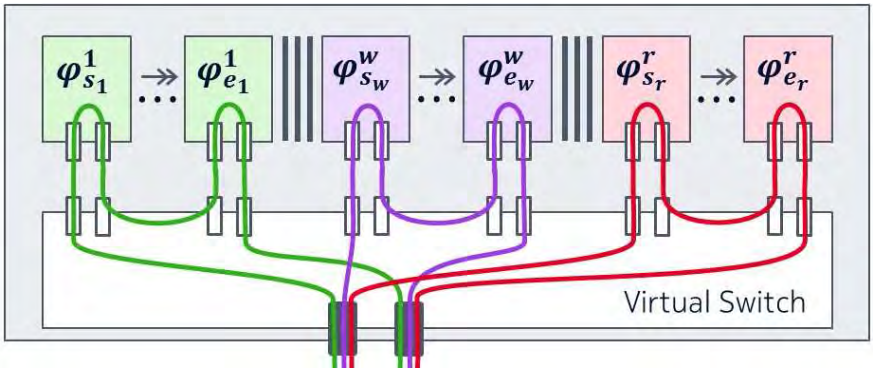


Figure 6: Server Sj deployed with r sub-chains from possibly different service chains

$$\{\varphi^1_{s_1 \to e_1}, \dots, \varphi^w_{s_w \to e_w}, \dots, \varphi^r_{s_r \to e_r}\}.$$

For a single sub-chain, the switching cost is exactly the gather cost of a chain deployed on server S_j and can be obtained directly from function F defined in our previous contribution. However, when we deploy more than one sub-chain, the overall cost is not the sum of the separate deployments. The cost of deploying two sub-chains is much smaller than twice the cost of deploying one sub-chain. Thus, in concurrent chain deployments the switching cost is amortized and the total cost is smaller than the sum of two separate costs.

To quantify this, we define $\|\varphi_{s \to e}\|^r$ to be the ***concurrent deployment*** of r copies of subchain $\varphi_{s \to e}$, and $\Delta(\varphi_{s \to e}, r)$ to be the ***switching cost delta*** between r times deploying a single sub-chain $\varphi_{s \to e}$ and the concurrent deployment of r copies of $\varphi_{s \to e}$, i.e.,

$$\Delta(\varphi_{s\to e}, r) = \left(r \cdot F(\varphi_{s\to e}) - F(\|\varphi_{s\to e}\|^r)\right)$$

Given a set of r sub-chains $\left\{\varphi^1_{s_1\to e_1}, \ldots, \varphi^r_{s_r\to e_r}\right\}$ that are deployed on server S_j (as illustrated in Figure 6), we can now compute the cpu-cost as follows:

$$C_j^h = \sum_{w=1}^{r} F\left(\varphi^w_{s_w\to e_w}\right) - (r-1)\cdot\left(\sum_{w=1}^{r}\Delta\left(\varphi^w_{s_w\to e_w}, r\right)\right)/r$$

The left-hand side of C_j^h is the sum of the cpu-cost associated with the gather deployment of each sub-chain, and the right-hand side reflects the saving due to the concurrent deployment of r chains which is $(r-1)$ times the average switching cost delta. Note that for $r = 1$ we do not have any savings and the cost is just the gather cost of deploying a single sub-chain.

4.2.3 Optimized Operation Cost Placement

We are now ready to present our next contribution, that is a placement algorithm for Operational (switching) Cost Minimization (OCM). Our algorithm entails a strategy to deploy the service chains onto the set of physical servers. Per each service chain in the sequence, we find a partition of the chain into sub-chains and an allocation of a server to each sub-chain in a way that minimizes the total switching cost. This on-line handling does not guarantee global minimum switching cost but as we show in this section, it does reduce the switching CPU cost and allow deploying significantly more services.

4.2.3.1 Operational Cost Minimization Algorithm

The ***Operational Cost Minimization*** (OCM) algorithm receives as an input a set of servers $S = \{S_1, S_2, \ldots, S_k\}$, and a sequence of service chains $\Phi = \langle\varphi^1, \varphi^2, \ldots, \varphi^m\rangle$. For each service chains $\varphi \in \Phi$, the OCM invokes the optimal service chain placement step shown in Figure 7.

Algorithm optimal service chain placement step

Input: $S \leftarrow \{S_1, S_2, \ldots, S_k\}$: set of servers
$\varphi \leftarrow \langle \varphi_1 \rightarrow \varphi_2 \ldots \rightarrow \varphi_n \rangle$: service chain
1: *min-cost* $\leftarrow \infty$: minimum cost found so far
2: *deploy-map* $\leftarrow$ NIL: maps all VNFs to servers in S
3: $\mathcal{A} \leftarrow$ all possible set partitioning (or decompositions)
4: **for** every set partition $a \in \mathcal{A}$ **do**
5: **for** every partition $p \in a$, and server $S_j \in \mathcal{S}$ **do**
6: $\mathcal{C}_j^h, \mathcal{C}_j^v \leftarrow$ compute the cost of deploying p on S_j
7: $G \leftarrow$ reduce to minimum cost flow in a graph
8: **if** $min(G) \leq$ *min-cost* **then**
9: *min-cost* $\leftarrow min(G)$
10: *deploy-map* $\leftarrow$ extract solution from G
11: **return** *deploy-map*

Figure 7: Optimal service chain placement step.

Each placement step is made of three building blocks:

i. List all set partitions of the VNFs in φ. In a relaxed version of the algorithm, we consider all decompositions in which every service chain goes through a server at most once.
ii. Given a set partition build the objective function, namely a cost function that predicts the operational cost of network traffic switching per each server;
iii. Given an objective function build a reduction to minimum-weight matching in bipartite graphs between partitions and servers, where the weights are given by the objective function.

We denote by P_o (and P_r) the ***optimal*** placement function (and the ***relaxed*** placement function) that implements the optimal OCM algorithm (and respectively the relaxed version of the OCM algorithm).

To conclude, we introduced the Operational Cost Minimization (OCM) placement algorithm, a performance-oriented deployment mechanism that minimizes internal switching CPU overhead and improves network utilization. This algorithm uses a novel cost model that captures the operational cost of the internal virtual switching for a given server. We provided empirical evidence, using a real NFV-based environment, indicating that our cost model is accurate comparing to actual deployment measurements (lower than 5%). Using this cost model, we introduced an efficient online placement algorithm that minimizes the switching cost of service chain requests. We show that OCM significantly reduces the operational costs and increases utilization, when compared to commonly used deployment strategies (up to a factor of 4 in the extreme case and 20% - 40% in typical cases). Our work opens opportunities to design new pricing models for service chaining by NFV providers.

More details are available at the attached INFOCOM paper in Annex B.

4.2.4 Resource allocation in Mobile Edge Computing

In a Mobile Edge Computing (MEC) System, applications will run at ME (Mobile Edge) Hosts in a virtualized environment, as VDUs (*Virtualization Deployment Units*), e.g. VM or Container. For that purpose, and in line with ETSI NFV, management and orchestration components have been added to the MEC system, composed by "Mobile Edge Host Level" and "Mobile Edge System Level". That is, while the latter has a global view of the entire MEC System, including all ME Hosts, the former acts at the ME Hosts level, with the functions: Element Manager for the ME Host, Rules and Requirements management for MEC applications, and MEC Applications LCM operations (similar to VNFM for NFV). At each ME Host, the ME Platform component may or may not be deployed over the virtualization infrastructure, possibly using specific HW and SW. Figure 8 depicts the full ETSI MEC architecture.

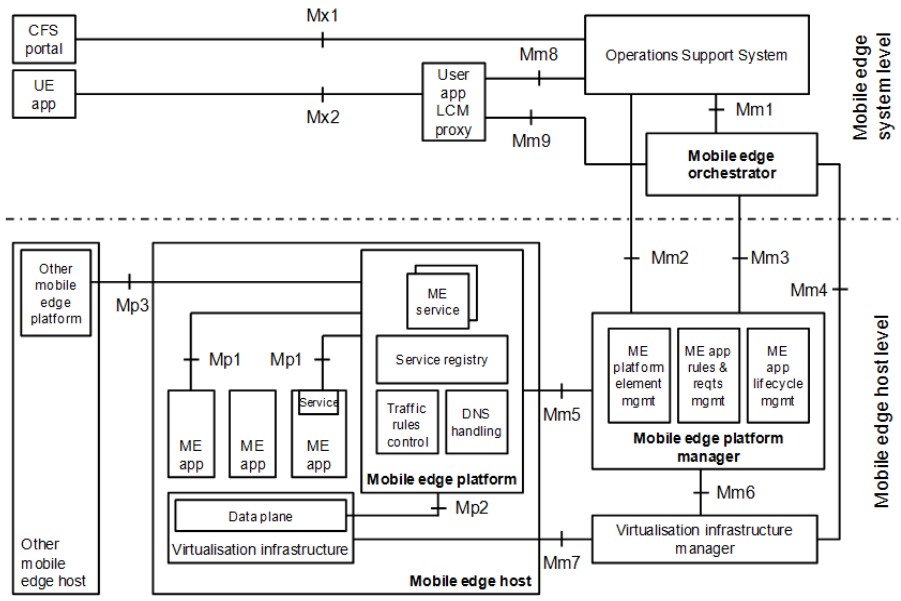


Figure 8 - ETSI MEC Architecture

In this context, resources allocation to MEC Applications will occur on the virtualization infrastructure, existing at each ME Host. This will be controlled by the ME Orchestrator, as part of ME Applications Life-Cycle Management (LCM) operations. SUPERFLUIDITY internal document I6.1 "Initial Design for Control Framework" documents MEC management and orchestration requirements as well as respective flows.

The following entities, hosted at the MEO, are required for ME Applications resources allocation:

- MEC App Catalogue


    - Stores the catalogue of deployable MEC Applications, with associated images and MEC Descriptor (where applicable rules and requirements are specified)
- MEC App Descriptor
    - Even if not yet defined by the ETSI MEC ISG, MEC Applications will have a descriptor associated to them. This will be in line with the VNF Descriptor, including specific information like, delay budget, required MEC Services to run and required virtualization resources.
- MEC Infrastructure
    - Keeps track of available, reserved and in use resources at all ME Hosts, for usage by MEC Applications
- MEC Hosts Inventory
    - Lists and describes the ME Hosts under MEO control. This shall include, per ME Host, a mapping with served eNB and available services (LOC, RNIS, DNS, etc.)

MEC Apps instantiation may occur:

1. Triggered by MEO
    - Selection of ME Hosts to deploy MEC Apps shall be automatic and based on existing information at the MEO and associated MEC Applications' Descriptors. Thus, from no need for allocate resources (only on-boarding), to allocate resources at all ME Hosts, all scenarios are possible.
2. Triggered by management (OSS)
    - Selection of ME Hosts for Applications instantiation will be the decided by a third-party entity, eventually including the analysis of the information contained in MEC Application Descriptors.
3. Triggered by UE
    - Via the MEC "User App LCM Proxy" component, entities at the UE may request the instantiation of specific MEC Apps. Application requirements (e.g. latency, compute resources, storage resources, location, network capability, security condition etc.) will be analysed in order to select a host fulfilling all the requirements.

In any of the previous situations, upon identification of the ME Hosts for Application deployment, resources at the corresponding virtualized infrastructures, must be reserved, involving the defined ME Platform Manager and the VIM, in a similar way to what happens with VNFs.

Even if current ETSI ISG MEC work does not address MEC Applications scale in/out or up/down, there is no reason to exclude that as part of LCM operations, in the context of resources allocation. In the same sense, ETSI ISG MEC considers that individual MEC Applications will be made of single VDUs. Similarly to VNF, there is no strong reason for not considering that MEC Applications may be the result of the composition of several VMs.

Resources allocation for MEC Applications will be managed by the interactions between MEO, MEPM and VIM.

More information on the placement, service migration and migration in MEC is detailed in **Annex B, section B4**.

4.3 Dynamic scaling, resource allocation and load balancing

4.3.1 VM Scaling and Scheduling via Cost Optimal MDP solution

4.3.1.1 Modelling y flexible queuing system

We address cost optimization problem of VM scaling and scheduling which considers both the exogenous and internal cost constraints. The existing VM scaling tools as e.g., in AWS, provide only limited instrumentation for cost optimization and are not designed to account for any additional exogenous constraints. To account for this, we model the problem as cost-optimal task

scheduling to a queuing system with flexible number of queues, where a queue stands for a VM, hence it can be deployed, can have a task scheduled to and can be terminated. We aim to capture the impact of a variety of constraints which includes decision making on queue deployment and termination, considers delay cost, cost associated with having deployed VM (even if idle), task processing incentives which are modeled as scheduling reward and rejection fine.

4.3.1.2 Solution by Markov Decision Process

We analyse the system by Markov decision process (MDP) and numerically solve it to find the optimal policy. The solution we provide captures the aforementioned constraints. We show that the optimal policy possesses decision thresholds, which depend on the system parameters. In particular, distinctive impacts of VM deployment cost, cost of having an idle VM, and the cost associated with a delay are observed, graphically presented, and the corresponding insights are analysed. In addition, to be practically sound, we investigate the impact of average VM deployment time.

4.3.1.3 Validation of the policy found by MDP through AWS setting

We validate policies found by MDP, through directing an exogenous computational tasks flow with known statistics to a set-up implemented on AWS. The policy is implemented by setting accordingly the AWS Elastic Load Balancer (ELB) thresholds for VM deployment and termination. The results clearly confirm the superiority of the optimal policy over any other heuristically applied ELB management. Note that the policy which we find by the presented here method can be adopted by any cloud infrastructure.

More details are found in Annex B: section B5, where we also provide a machine learning based technique for the Life Cycle Management of containerized workloads.

4.3.2 Load Balancing as a Service

Elaborating on the activities covered in Section 3.4, particularly to focus on the Load Balancing capabilities, we proceeded to integrate NetScaler ADC, a commercial-grade load balancer, with OpenStack and the respective Load Balancing as a Service (LBaaS) extensible framework. For that purpose, and in compliance with the ETSI NFV reference architecture, we utilized the respective EMS element, NetScaler Management and Analytics System (MAS). To prepare for the integration and validation activities (WP7), we deployed and verified the concept in the Superfluidity staging environment. For a detailed description of the procedures, please refer to Annex B, Section B6.

4.4 Optimal design and management of RFBs over a Superfluid 5G network

We have first focused on the problem of designing an RFB-based 5G networks, by targeting the reduction of the installation costs for the physical nodes. In [5] we have provided an optimization model that allows the minimization of the total costs, under the RFBs placement on the physical 5G nodes. The presented formulation is able to take into account multiple RFBs types, ranging from the low-level ones, devoted to the setting of the communication channel to the users, to the high level ones, which are able to provide application layer features. The goal is then to optimally select the set of installed 5G nodes, as well to properly dimension each of them in terms of commodity and dedicated HardWare (HW). Results, obtained over a representative 5G case study, demonstrate the efficacy of the proposed approach.

Then, we have faced the problem of the optimized management of a set of RFBs over a 5G network. In this case, the problem is related to the dynamic allocation of the RFB components, which has to be performed in real time, in order to match the required levels of flexibility and agility. Initially, we have focused on the problem modelling by means of optimization tools [6]. The goal is to target a given KPI (e.g., maximization of the user traffic, or minimization of the number of powered on 5G nodes), while ensuring users traffic, RFB placement, and availability of physical resources on the 5G nodes. In the following, we have provided an efficient algorithm, based on bio-inspired techniques [P5G-Globecom2017], which is able to sub-optimally solve the problem in a reasonable amount of time. In both cases, our results clearly indicate that the management of a RFB-based network is feasible, and that the performance achieved by users depends on the specific KPI chosen by the operator. Currently, an extended version of [6] is under revision for a journal (Wiley International Journal of Network Management).

Another important problem that we have considered is the reduction of energy consumption for the 5G nodes, while ensuring Quality of Service constraints to users. To this aim, in [7] we have considered a special case of RFB, where this component is realized by means of a VNF. We have then focused on the problem of managing set of VNF chains, that have to be deployed on physical nodes, in order to provide the service to users. The goal is then to reduce the total energy consumed by the 5G nodes. To this aim, different formulations of the problem are provided, together with a set of fast algorithms. Results show that the energy consumption can be wisely preserved, while still ensuring the service to users.

Finally, we have also provided economic indications about the RFB-based solution at a global level. [8] presents a stakeholder analysis, as well as a Strengths Weaknesses Opportunities Threats (SWOT) analysis. In addition, in [9] we focus on the profitability of a RFB-based 5G network in two cities, i.e., Bologna (Italy) and San Francisco (CA). Results show that the initial investment incurred by the operator can be compensated by properly setting the users' monthly subscription fee. Overall, the RFB-based 5G deployment appears to be profitable for the operator from an economic point of view.

5 Task 6.3: Automated Security Verification Framework

Network security and its correct operation are two sides of the same coin. Misconfigured routers allow attacks to hosts or routers in the network, thus lack of correctness reduces security. The converse also holds true: a network cannot behave correctly, as specified by its operator, if there are low level attacks on the software on any of its components, for instance routers or network functions; such attacks allow the attacker to inject arbitrary traffic, thus breaking the correctness of the network.

5G networks enable operators to quickly deploy new functionality in the form of network functions, and this will allow networks to keep up with the development pace of applications. However, dynamically instantiated network functions make it significantly more difficult to ensure that the network is behaving correctly and that it is secure.

In Superfluidity, we propose an integrated approach that ensures both network security and correctness, at the same time. Our approach leverages network dataplane verification, in particular the Symnet symbolic execution tool and the SEFL language developed in a prior project (Trilogy II) and the initial part of the Superfluidity. The solution has three distinct parts that are used at different times in the service deployment life-cycle:

1) *[before deployment]* Verifying that the high level service configuration meets the correctness requirements set out by the operator.
2) *[before, after deployment]* Verifying that the low-level implementation is bug-free.
3) *[at runtime]* Detecting in real time possible attacks using anomaly detection.

We now provide an overview of these three components; for details please refer to Annex C.

5.1 Verifying high-level service configurations

To change network functionality, the network operator or third-parties will provide a service configuration. These configurations could be written in NEMO, RDCL3D or other similar languages, and they are, essentially, directed graphs connecting various network functions (vertices in the graph). The network functions are, in Superfluidity terminology, Reusable Functional Blocks (RFBs) and their processing can be specified either by selecting from a catalogue of functions (e.g. Click modular router elements or Openstack Neutron primitives) or by providing a program that specifies the processing to be done by the RFB and can be directly run in the dataplane. Such programs could be expressed in programming languages such as P4 or Openstate (an output of the Superfluidity and Beba H2020 projects, from CNIT/Uni Roma Tor Vergata); such programs can be run directly in the dataplane, at very high processing speeds. Recently Barefoot Tofino, the first commercial P4-enabled switch, has started shipping.

Quick service instantiation and reconfiguration is the cornerstone of 5G, but it fosters many risks; in particular, how can one ensure that the new service about to be deployed will not harm the operator's network, i.e. already running network services? Another question is how can we ensure that the new service behaves as its user expects.

To answer such questions, one possible approach is to use active testing but this approach scales poorly and thus has limited coverage. Another possible approach is to use formal verification techniques, which is the approach we take. In particular, before instantiation, we need to ensure that service configurations conform to the policy of the network operator and/or the party deploying the configuration. The policy can be seen as a correctness specification and can include reachability requirements, the way packets will be changed, isolation requirements, and so on.

We use exhaustive symbolic execution of dataplane code to test such policies. In our approach, the network dataplane is a directed graph connecting network functions expressed in the SEFL language. We the inject *symbolic packets* at selected vantage points in the network and track how they are processed by the network. A symbolic packet is a packet whose header fields can take any possible value.

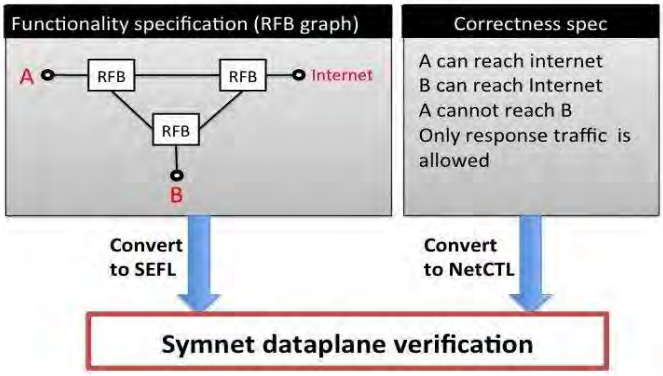


Figure 9: Symnet dataplane verification

To verify that a high-level RFB graph corresponds to the policy specified by the operator we start by translating RFB processing to SEFL and using the Symnet symbolic execution tool to verify that the policy holds. This step is captured in Figure shown top the right: the inputs to our verification tools includes the RFB graph, as well as high-level policies we want the resulting network configuration to obey. We discuss these in greater detail next.

5.1.1 Translating RFBs to SEFL

We have implemented translators for a many types of RFBs, shown below.

- Click modular router elements - over 50 basic Click elements can be translated to SEFL. The elements we do not translate are the ones relating to performance, not functionality (i.e. traffic shaper, rate limiter, etc.)
- Router FIBs - given a snapshot of a router FIB, we can generate SEFL code that performs the same functionality and is optimized for verification.
- *Openstack Neutron network primitives* - including firewall, router, load balancer and VPN.
- Openflow rule snapshots - given a dump of Openflow rules taken from a software or hardware Openflow-enabled switch, our code outputs equivalent SEFL code.
- Iptables rules - given iptables rules, we generate equivalent SEFL code. Our code also models stateful firewall processing and NATs.
- P4 programs - we have developed the Vera verification tool which includes a translator that parses the P4.14 version of P4 and automatically generates equivalent SEFL code. See Annex C for a detailed description.

In summary, given an RFB graph using RFB described in the list above, we can automatically generate equivalent SEFL code. Note that, at this point, the data acquisition must be performed offline; our translators expect the data in the required format, do not contact the switches themselves. In our future work we automatically collect data from a production environment (such as OpenMano or Openstack).

5.1.2 Verifying policy compliance of SEFL dataplanes using Symnet and NetCTL

Symbolic execution simulates how all these packets may be processed, without iteratively testing each concrete packet in isolation because this approach does not scale. Symbolic execution for networks tracks classes of packets that are treated in the same way. When symbolic execution finishes, the result is a list of symbolic packets with constraints or concrete values on their header fields, along with the set of boxes and instructions executed by that packet.

To verify whether a policy holds, we use basic symbolic execution offered by Symnet as follows:

- We convert the policy to NetCTL and then decide which packets to inject and where, as well as to decide which header fields should be made symbolic.
- We use the policy to guide symbolic execution, checking on each path that the policy holds. If it does not, we stop exploration and offer a counterexample that violates the policy.

The full description of NetCTL is included in the internal deliverable I6.3b which is attached to this document. Hereafter we provide a brief overview of our approach.

For any given SEFL program, symbolic execution will explore a large number of paths, many of which are successful. In our evaluation, we typically see hundreds such paths. Examining them manually to decide whether the behavior is correct is time consuming and error-prone. We wish to specify desirable properties and check them automatically.

The specification must combine packet constraints at specific ports of the network (or state properties) with constraints over the possible paths which the packets may take between ports (or path properties). We can already express state properties via SEFL instructions. For instance, the property 'destination IP is always X at port out' can be verified by placing the SEFL instruction Dest-IP != X at port out and observing the successful paths from port out.

In order to express path properties, we have considered a wide range of SDN policy languages, e.g. the Kinetic family, FatTire, NetPlumber, as well as approaches relying on logic programming (e.g. Shenker's FML). We have found that all such languages are limited in their ability to express compositional constraints.

We have thus turned to Computation Tree Logic (CTL). In CTL, temporal operators such as F (i.e. sometime in the future) and G (i.e. always in the future) are combined with path quantifiers: ∃ (on some path) and ∀ (on all paths). For instance, the policy: ∀FdestTCP == 80 evaluated at some port P of a box, expresses that on all possible packet paths from P, destTCP will eventually become 80.

The syntax of NetCTL is given below:

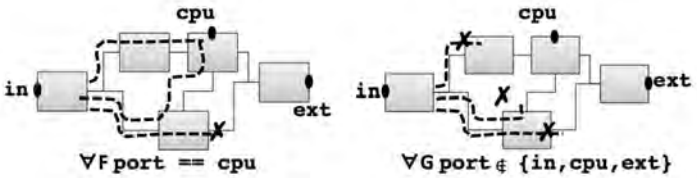
$$\phi ::= \text{SEFL} \mid \neg\phi \mid \phi \wedge \phi \mid XY\phi$$

where whereX ∈ {∃,∀},Y ∈ {F,G}.

Unlike Merlin, FatTree or NetPlumber, NetCTL is compositional: starting from simple properties, we can construct more complex ones. For instance, we can express that ''whenever the IP destination of a packet becomes a public address, port P is reachable" via the formula: ∀G(ip != 192.168.0.0/16 → ∃F port == Internet). CTL can express many other properties including TCP connectivity and invariance across tunnels.

In principle, checking NetCTL formulae can easily be implemented after exhaustive symbolic execution by checking the outputs. However, this approach is also inefficient because in many cases we can check a property without exploring all possible paths.

That is why we integrate NetCTL verification with symbolic execution. In our implementation, NetCTL verification is performed as added checks on the packets after each SEFL instruction is executed; the overhead of these checks is very small in practice. After every check we can decide to prioritize a certain path or stop execution altogether. Because of this, in most cases, our approach checks NetCTL properties faster than exhaustive symbolic execution. In Annex C.1 we provide a correctness verification of a NAT implemented in P4, showing how this approach can reduce the amount of work performed by symbolic execution by a factor of two.



In the figure to the right, we briefly illustrate NetCTL verification. The figure describes two symbolic execution traces performed on the same topology — a simplistic illustration of the P4 NAT model, described in more detail in the subsequent sections. Boxes represent SEFL code blocks and solid lines — links between boxes. Dashed lines describe the paths explored by our verifier. The formula $\phi 1 = \forall F(\text{port} == \text{cpu})$ (left) expresses that all paths eventually reach port cpu. In order to evaluate it, our checker performs symbolic execution starting at port in. The checker will explore each encountered path until port == cpu is satisfied, the path ends, or it becomes unsatisfiable. Suppose the checker explores three paths, as shown in the figure. Since the formula F(port == cpu) is true on the first two paths only, $\phi 1$ is false. The formula $\phi 2 = \forall G(\text{port} \notin \{\text{in,cpu,ext}\})$ (right) ex- presses that all packets are dropped by the NAT. To verify it, our checker will determine if port $\notin$ {in,cpu,ext} is true on each execution path, and after each SEFL code-block. In our example, this is indeed the case, thus the policy is true.

5.2 Verifying low-level implementations

In most cases, the SEFL model is very close in functionality to the actual processing of the box: this is the case for router FIBs and Openflow rules. This is because the functionality of the switch dataplane is simple enough that after passing basic tests, we can trust that both the hardware and the model to be correct. The same is not true for a range of RFBs, including:

- P4 programs which are surprisingly complex to develop and debug.
- Openstack dataplane rules which are created by the very complex Neutron software starting from tenant configurations.
- IPtables rules which are implemented in C and offer a wide range of functionality, beyond basic filtering.
- Click element implementations (written in C++).

To find bugs in P4 programs, our approach relies on the strength of exhaustive symbolic execution to catch a variety of bugs, which are frequent in practice. To find problems with the other RFB implementations, we rely on the novel notion of ***symbolic execution equivalence***. We discuss these in turn.

5.2.1 Finding bugs in P4 programs

We have developed a P4 verification tool that translates P4 to SEFL and integrates NetCTL verification with Symnet. Even without a specification, by default, Vera inserts checks that capture a wide range bugs in P4 programs and flags such bugs to the user as failed paths. For each failed path, Vera also generates a concrete packet that matches the path constraints, which can be used to test the bug in P4 switches, be they software or hardware. Below are the types of errors that Vera catches automatically:

- Implicit drops are flagged when a packet reaches the buffering mechanism without having an egress_spec set. Vera catches this by adding an assertion that the egress_spec must be non-zero when it reaches the buffer.in port.
- Table rules that match dropped packets are flagged as errors by adding an assertion that egress_spec != 511 in the preamble of all actions.
- Invalid memory accesses are frequent P4 mistakes,when users do not test the validity of a header before using its fields. Vera relies on Symnet's memory safety guarantees to capture these errors; when accessing an unallocated field, Symnet will fail the current path.
- Header errors Malformed headers are captured during parsing by using the exists SEFL instruction. Adding an existing header or removing an inexistent one are also caught automatically as deparsing errors.
- Scoping and unallowed writes Certain metadata values are read-only in P4, yet the P4 compiler allows the program to write them (e.g. the egress_port metadata). Further, static registers can only be read from one table according to the spec, yet the compiler allows such reads. Vera catches such errors during the translation process and reports them to the user.
- Out-of-bounds array accesses are caught automatically by Vera by adding, before each array access, an out-of- bounds check for the index. At runtime, the solver will check if the constraint is satisfiable and if it is the user will get a failed path providing an example packet that triggers a possible out-of-bounds access.
- Field overflows/underflows are the only arithmetic exceptions possible in P4 (because division is not supported) and Vera catches them by adding a check before each addition/subtraction operation.

Loops are also caught automatically. The loop detector runs by default on the parser input port and on the egress input port which are the two places where packets can be redirected backwards in the P4 pipeline. Whenever a packet enters visits one of these ports, Vera remembers the entire memory state (i.e. the values and constraints or all the meta- data and header fields). When a packet revisits the same port, its memory state is compared to all the previous saved memory states. Two memory states are different if and only if at least one symbol has a different value in the two states. Note that we compare not only concrete values, but also symbolic ones: if a metadata is bound to the same symbolic value in both states, it is deemed to be equal. Whenever Vera discovers two memory states that are equal, it fails the current path with the "loop detected" message.

Vera and its evaluation is provided in detail in Annex C.1, as part of a paper under submission. Here we simply list the set of bugs Vera has caught in public P4 programs.

Program	Size (LOC)	Verification time (sec)	Implicit drop	Parsing	Deparsing	Header ops.	Invalid access	Underflow / overflow	Loop	Processing dropped packets
copy-to-cpu	70	0.1		•		•				
resubmit	70	0.4							•	
encap	130	0.45	•		•	•				
simple router	145	0.55	•							
simple NAT	290	1.25	•			•				
simple router + ACL	200	0.8	•							•
Axon	100	14		•			•			
Switch	6000	5-15/sym.pkt.				•	•			
Beamer mux[25]	340	1.4	•		•		•			
NDP switch[12]	210	0.8					•			
P4xos[7]	650	13.4	•				•			

Table 2: Bugs caught by Vera

5.2.2 Symbolic execution equivalence and its applications

Automatically finding bugs in C or C++ implementations of RFBs is much more tricky because exhaustive symbolic execution is typically not feasible. Nor is verifying that Neutron correctly implements tenant configurations into the dataplane (e.g. as Openflow and iptables rules).

In the case of Neutron, we do however know that the tenant configuration - the abstract specification of what the network should do - must be equivalent to the resulting dataplane rules. So instead of verifying Neutron per se, we verify that its output is equivalent to the input.

Checking equivalence is a very useful primitive to have in the context of network verification. If we had such an algorithm, we could check whether Neutron is indeed behaving correctly for any given input (a tenant configuration). This would still not constitute a proof of correctness of Neutron for all inputs; it merely validates that for a given input Neutron behaves correctly. Intriguingly, we could apply the same approach in the case of C implementations of RFBs. Given an instantiated RFB (e.g. a firewall together with its rules) in SEFL and in C, we could check they behave similarly, i.e. their output is the same when presented the same input.

Unfortunately, checking equivalence is undecidable for general programs because it can be reduced to the halting problem. However, equivalence is decidable for programs that always terminate and have bounded inputs: to check it, enumerate all possible inputs, run both programs and check that the outputs match. Network dataplanes are such programs: they have bounded inputs (packets) and rarely loop (P4 does not include traditional loops). Loops can appear, especially network-wide, but they can be caught automatically.

We show that it is possible to quickly and automatically decide if two programs describing network dataplanes are equivalent. To make our approach scalable, we rely on exhaustive symbolic execution instead of testing all possible inputs.

Symbolic execution outputs can be used to check that two programs are equivalent but the algorithm to do so is far from trivial. We discuss here informally three basic approaches, which we call input, output and functional equivalence, and explain why they all need to be implemented simultaneously

to ensure correctness. For a description of our algorithm and a formal proof of its correctness, please refer to Annex C, section C2.

The simplest way to check equivalence is to compare which packets can exit any given port by examining the feasible values for each header field—we call this co- domain or output equivalence. Say ports p1 and p2 belong to two different programs, but they should be equivalent. The algorithm to check for output equivalence is simple: compute the reunion of possible values for each header field for each of these ports, and then check whether the resulting sets of packets are identical (this can easily be achieved using set operations).

Consider now two programs where one constraints packets to have TTL>0 and then sets TTL to zero, while the other program simply sets TTL=0. The two models are not equivalent, but checking just output equivalence is not enough to capture this problem.

The next step is to also check the constraints applied on the original (input) values of the header fields, before any modifications are made; when combined with output equivalence checking, we are now checking input/output equivalence. With input/output equivalence, we will find that the first model only allows packets to pass when TTL≥ 1 while the optimized model allows packets when TTL≥ 0; the two ranges are not the same, so the two models are not equivalent.

Checking for input/output equivalence is necessary to find bugs, but on its own it is still not sufficient. To see why this is the case, consider two trivial programs where one leaves the TTL field unchanged, while the other executes the instruction TTL = 255 − TTL. Both the input values (0-255) and the possible output values (0-255) of the two models are the same, yet they are obviously not equivalent. What we also need is functional equivalence: regardless of the initial value of the TTL, the two values of the TTL after executing the two programs should always be equal. In our example, functional equivalence is not true because the condition TTL = 255 − TTL does not always hold. In fact, all these three checks are simultaneously needed to ensure equivalence: removing a single check leads to wrong results.

We have implemented this algorithm we call EQ and have applied it to find Openstack Neutron bugs. In our testbed, we installed a minimal Openstack deployment that included two compute nodes and one network node. In each experiment, we had one or multiple tenants deploy VMs and network configurations using Openstack, and then ran sdiff on another machine that had snapshots of the tenant configuration and a snapshot of the dataplane rules installed by Neutron (including openvswitch rules and iptables rules); acquisition was performed with scripts that dumped and copied the relevant data. The largest Neutron configuration we analyzed with sdiff had 16VMs belonging to six tenants, and each tenant had configured three VLANs and one router to connect them. Equivalence testing for this deployment took around 200s.

EQ is able to detect issues in the implementation of Neutron on the underlying compute and network nodes, and it also points out the dataplane rule which breaks the expected tenant-level behaviour.

We caught 5 software bugs introduced by Neutron's middle-ware implementation, 2 configuration bugs introduced by a misconfiguration on machines in the deployment - and 1 tenant-level misconfiguration. We do describe these bugs in detail in Annex C, section C2.

In our future work, we will apply EQ to check equivalence between SEFL and C code via symbolic execution (with Symnet and Klee respectively).

5.3 Anomaly detection

To complement cases where verification cannot offer guarantees, such as legacy, deployed boxes that cannot be verified, Superfluidity also includes an online anomaly detection component that aims to identify suspicious behaviour. This work was completed and reported in the two internal documents I6.3 and I6.3b, which are attached to this document; we only give a brief outline here.

In short, we proposed a novel universal anomaly detection algorithm, which was able to learn the normal behaviour of systems and alert for abnormalities, without any prior knowledge on the system model, nor any knowledge on the characteristics of the attack. The suggested method utilized the Lempel-Ziv universal compression algorithm in order to optimally give probability assignments for normal behaviour (during learning), then estimate the likelihood of new data (during operation) and classify it accordingly.

We have also evaluated the algorithm on real-world data and network traces, showing how a universal, low complexity identification system can be built, with high detection rates and low false-alarm probabilities. We have applied the detection algorithms to the problems of malicious tools detection via system calls monitoring and data leakage identification. We also provided results for detecting anomalous HTTP traffic, e.g., Command and Control channels of Botnets.

6 Collaboration with 5G-PPP

Superfluidity project is contributing mainly in 5G Architecture Working Group (WG) and Software Network WG, besides to coordination with 5G-PPP Steering Board and Technology Board.

This deliverable introduces a set of innovations that the project built in orchestration, provisioning and control frameworks. The advances made by Superfluidity in ETSI MANO, NFVO and VNFM are described in the recent version of the 5G Architecture white paper produced by the 5G Architecture WG and released in the Mobile Wireless Congress 2018, in which Superfluidity is very active. Mainly, Section 5 of the white paper referring to the softwarization and 5G service management captures the work of the Superfluidity project in TOSCA, OpenStack and VNF on-boarding. Some of the content of this document is also introduced in the Software Network WG and part of the white paper published in January 2017 entitled: '5G-PPP Vision on Software Networks' in which Superfluidity actors took the lead of editing section 4.

It is important to highlight the fact that most of phase 1 projects architecture are based on ETSI MANO and VM technology, which explains that many Superfluidity contributions are not yet captured in 5G-PPP eco-system. However, as some of the partners of Superfluidity are still active in phase 2, e.g. Nokia Bell-Labs France and Nokia IL, we expect that more and more the results of Superfluidity will be disseminated and leveraged even after the completion of the project which demonstrates the high impact that Superfluidity made in Network softwarization and cloudification areas. To corroborate this fact, we can cite that many contributions made by Superfluidity in Container, Kuryr, Ansible, RDCL/RFB etc areas is leveraged in NGPaaS project (http://ngpaas.eu) which is a phase 2 project. To facilitate this technology transfer, the Software Network WG, chaired now by Nokia Bell-Labs France, traces 5G-PPP phase 1 projects outcomes and their reuse in phase 2 projects.

ANNEX A: Provision and Control Framework - TASK 6.1

WP6 focuses on the orchestration and management actions at a distributed system level, building upon WP5 advances. The provision and control framework main objectives are resource provisioning, and control and management of network functions as well as applications located at the edge (in this case MEC). To achieve that this framework targets dynamic scaling and resource allocation, traffic load balancing between virtual functions, or automatic recovery upon hardware failure, among others.

The main objectives from the DoW that task T6.1 targets are:

- OBJ2: design of cross management domains resource allocation and placement algorithms
- OBJ3: design the control framework for dynamic scaling, resource allocation and load balancing of tasks in the overall system
- OBJ4: development of platform middleboxes and services

There are several points where Superfluidity have contributed/focused to achieve these goals:

- Service behaviour models to support autonomous policy management reacting to current status of the system. For instance, detecting an application having interference problems and reacting to it by either performing (QoS) bandwidth limitation, migration or load balancing actions.
- Make OpenStack suitable for C-RAN/MEC components by improving network performance as well as allowing mixed VM and Container environments.
- Placement in a distributed environment with a system-wide overview

The following sections capture the requirement analysis performed for the different scenarios, i.e., NFV, MEC and C-RAN; presents an overview of state of the art solutions at different levels of the controller/orchestration hierarchy; review different orchestration options stating the preferences; and presents the proposed framework and components used at Superfluidity, followed by a list of contributions achieved at Superfluidity to support the proposed framework.

A1. Requirements analysis

In order to tackle the challenge, our approach was split into several steps. As a first step we started by analysing the use cases from WP2 as our input. The objective was the identification of shared attributes and the identification of common requirements that the use cases shared. After doing so, we had the next step ready – investigation of the aforementioned requirements' support in existing orchestration solutions. As a last step we identify the gaps between the requirements and each solution capabilities.

A1.1 NFV Technical Requirements

The following two figures depict the relevant ETSI NFV architectures: the main ETSI NFV and the MANO (Management AND Orchestration). This MANO architecture highlights the management and orchestration components (dashed box), identifying in more detail the management and orchestration interfaces, and other sub-components, like Catalogues and Services/Resources Repositories.

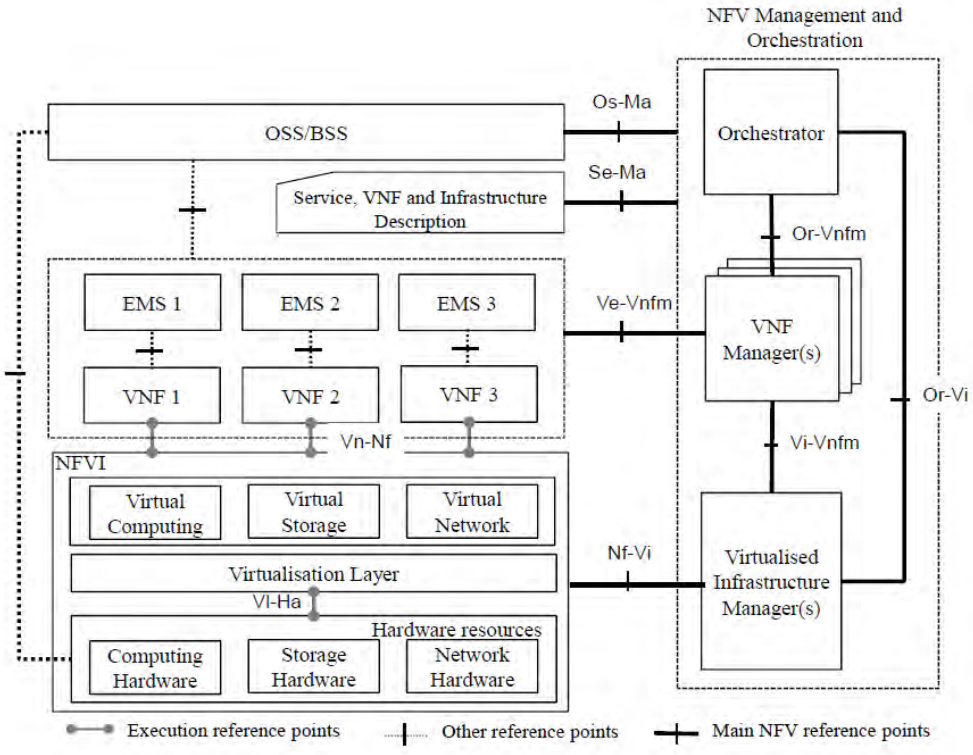


Figure 10: ETSI NFV reference architecture.

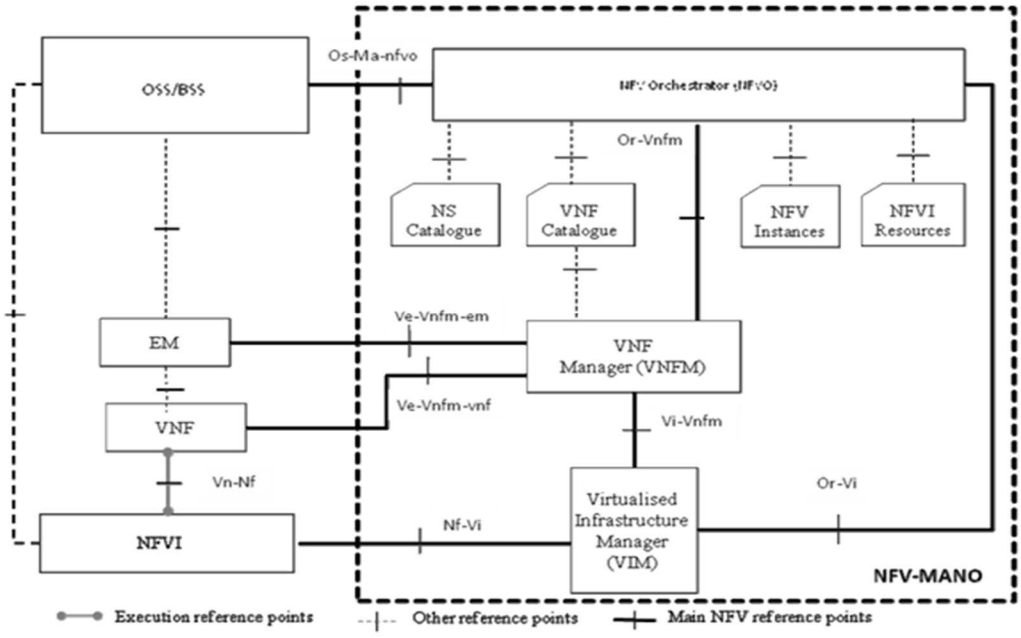


Figure 11: ETSI NFV MANO reference architecture.

Next table describes high-level technical requirements for a NFV management and orchestration system.

Type	Description
Onboarding	[Onboarding-01] The MANO framework MUST support the on-boarding of VNFs and NSs, respectively into the NFV Catalogue and NS Catalogues, making them available for instantiation
	[Onboarding-02] The MANO framework SHOULD perform other actions than on-boarding regarding VNF and NS packages: Disable, Enable, Update, Query and Delete.
Application lifecycle	[Lifecycle-01] The MANO framework MUST support the following VNF and NS lifecycle management (LCM) operations: ● Instantiation ● Scaling ● Modification ● Termination
	[Lifecycle-02] The MANO framework MUST provide API to the OSS or a UE application to process application LCM requests

	[Lifecycle-03] The MANO framework MUST be able to identify the VNF/NS running requirements. This will be the input for the decision on which location VNFs/NSs shall be provisioned.
Application scheduling and instantiation	[Instantiation-01] The MANO framework MUST support the indication of the following virtualized resources: ● Compute (cpu and memory) ● Storage ● Network resources ● Specific hardware support
	[Instantiation-02] The MANO framework MAY support the indication of the following requirements, such as: ● Latency ● Jitter ● Bandwidth
	[Instantiation-03] The MANO framework MUST support the indication of physical location (PoP-DC).
	[Instantiation-04] The MANO framework MAY consider cost requirements, which can be a translation of the operator's estimation for the deployment costs.
KPI's support	[Monitoring-01] The MANO framework MUST be able to collect infrastructure and service monitoring information, in order to feed KPI-based automated management and orchestration features.
Application Scaling	[Scaling-01] The MANO framework MUST be able to scale a VNF and/or NS, on OSS request or automatically based on KPIs, in order to increase/decrease the capacity.
Load Balancing	[LB-01] The MANO framework SHOULD support load balancing function as part of the NFVI/VIM infrastructure. This requires integration with the application lifecycle and scaling functions.
	[LB-02] The MANO framework SHOULD support standard load balancing features. OpenStack LBaaS captures these requirements at https://wiki.openstack.org/wiki/Neutron/LBaaS/requirements.

	[LB-03] The MANO framework SHOULD ideally support firewall load balancing mode. However, this MAY require addressing gaps in NFVI/VIM (OpenStack LBaaS doesn't appear to support this case).
Service Function Chaining	[SFC-01] The MANO framework MUST support the creation of Service Function Chains (SFCs), consisting of an ordered sequence of Service Functions (SFs). SFs are virtual machines, or even physical devices, that perform a network function such as firewall, content filter, content cache, or any other function that requires processing of packets in a flow.
	[SFC-02] The MANO framework MUST support SFCs with both simple (i.e. single SF) and complex (i.e. sequence of multiple SFs) Service Functions Paths (SFPs). Materialisation of SFCs requires the cooperation of the NFV Orchestrator, VIM and SDN controller. The NFV-O provides the VNFFG definition (please refer to relevant requirements in this document), the VIM creates the SFC by attaching the SF VM instances to network ports and the SDN controller configures the network overlay fabric that interconnects these network attachment points. According to the OPNFV SFC project (https://wiki.opnfv.org/display/sfc), SFC also depends on the VNF Manager: http://artifacts.opnfv.org/sfc/docs/design/architecture.html#vnf-manager
	[SFC-03] The MANO VIM MUST support the attachment of SF VM instances to network ports to construct SFPs (for more details on how OpenStack aims to implement this capability, please refer to https://docs.openstack.org/networking-sfc/latest/contributor/system_design_and_workflow.html and https://docs.openstack.org/networking-sfc/latest/contributor/api.html).
	[SFC-04] A Service Function (SF) MAY consist of a cluster of VM instances, which can be used for load balancing (please also see 2.2.2.5). The load balancing function MUST have the option to be sticky (i.e. sessions in progress must be sent through the same SF VM instance). The load balancing function MUST also have the option to ensure symmetric return traffic.
	[SFC-05] The MANO VIM MUST be extensible to support the creation ("rendering") of SFPs in conjunction with different SDN controllers and renderers (e.g. OpenFlow, NETCONF, etc.).

	The support of SFC-related requirements by the OpenDaylight SDN controller is described below: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main
	[SFC-06] The MANO VIM MAY support a network overlay function that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch). For a complete implementation of SFC, the MANO framework would need to also support orchestration of the SFC Classifier, Service Function Forwarder (SFF) and SFC Proxy building blocks. For more information on how OpenStack aims to support these SFC functions, please refer to https://docs.openstack.org/networking-sfc/latest/contributor/ovs_driver_and_agent_workflow.html).
	[SFC-07] The MANO VIM SHOULD support orchestration of SFC Classifiers. The MANO VIM MAY offer an implementation of an SFC Classifier that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-08] The MANO VIM SHOULD support the orchestration of Service Function Forwarder (SFF). The MANO VIM MAY also offer an implementation that is part of NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-09] The MANO VIM SHOULD support orchestration of SFC Proxies. The MANO VIM MAY offer an implementation of an SFC Proxy that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-10] The reference implementation of the SFF, SFC Classifier and SFC Proxy (if available) SHOULD support the preferred SFC encapsulation scheme, NSH (please see IETF draft-ietf-sfc-nsh). Please note that an initial implementation of a subset of the SFC requirements above was made available in OPNFV Brahmaputra, as a combination of OpenDaylight, OpenStack and Open vSwitch: https://wiki.opnfv.org/display/PROJ/Project+Proposals+Service+Function+Chaining An overview of how OPNFV Brahmaputra puts all the pieces together:

	http://artifacts.opnfv.org/sfc/brahmaputra/docs/design/index.html Further progress was achieved, as part of the OPNFV Colorado release: http://artifacts.opnfv.org/sfc/colorado/docs/design/index.html Finally, the requirements for supporting VNF Forwarding Graphs are outlined below: https://wiki.opnfv.org/display/VFG/Openstack+Based+VNF+Forwarding+Graph

Table 3: NFV requirements

A1.2 MEC Technical requirements

The following Figure depicts the relevant ETSI MEC architecture. This architecture describes how a MEC environment should be organised, namely regarding the deployment of MEC App on top of a cloud environment, as well as the whole management and orchestration functions to support this operation.

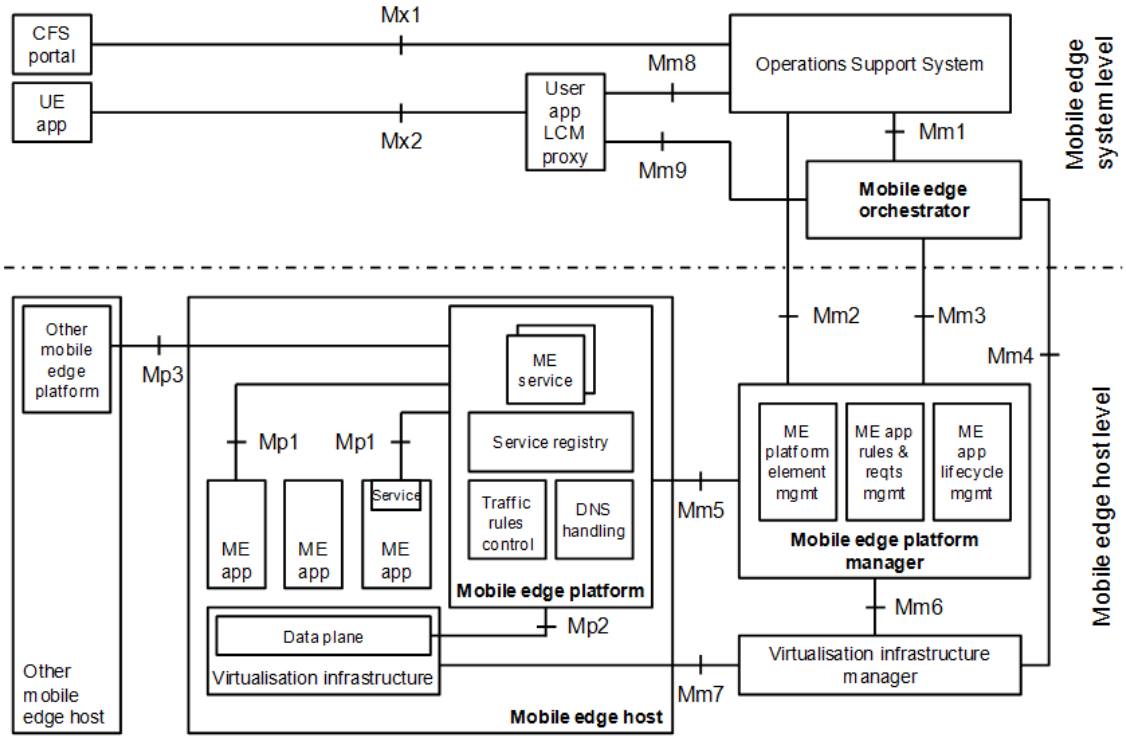


Figure 12: ETSI MEC reference architecture

The next table describes high-level technical requirements for a MEC management and orchestration system.

Type	Requirements description

Application lifecycle	[Lifecycle-01] The management system MUST support the following application lifecycle management (LCM) operations: ● Instantiation ● Scaling ● Relocation ● Modification ● Termination
	[Lifecycle-02] The management system MUST be able to receive and process application LCM requests: ● From the OSS, a third-party, or a UE application ● Based on LCM rules.
	[Lifecycle-03] The management system MUST be able to identify the mobile edge features and services an application requires to run. This will be the input for the decision on which mobile edge host to provision the application.
	[Lifecycle-04] The management system shall support the instantiation and termination of a running application when required by the operator.
Application scheduling and instantiation	[Instantiation-01] The management system MUST be able to deploy the application on mobile edge hosts in various locations, both in a central data center and at the edge of the Core Network.
	[Instantiation-02] The management system MUST support the following deployment application models: ● One App instance per MEC Host, serving multiple users ● Multiple App instances per MEC Host, each serving a single user
	[Instantiation-03] The management system MUST support the indication of the following virtualized resources: ● Compute ● Storage ● Network resources ● Specific hardware support
	[Instantiation-04] The management system MUST support the indication of the following network connectivity resources:

	• Connectivity to local networks • External connectivity to the Internet • Access to user traffic
	[Instantiation-05] The management system MUST support the indication of the following latency requirements: • Maximum • Expected
	[Instantiation-06] The management system MUST support the indication of physical location (edge).
	[Instantiation-07] The management system MUST support the indication of service requirements: • Mandatory - for MEC Apps to be able to operate. • Optional - for MEC Apps can benefit from, if available.
	[Instantiation-8] The management system MUST consider cost requirements, which can be a translation of the operator's estimation for the deployment costs.
Mobility support	[Mobility-01] The management system MUST support multiple MEC Hosts in different locations, including radio sites, aggregation points, or at the edge of the Core Network.
	[Mobility-02] The MEC system MUST guarantee service continuity while the UE moves across the network (between different edges and cells).
	[Mobility-03] The MEC system MUST be able to perform application instance relocation for MEC Apps dedicated to a single user.
	[Mobility-04] The MEC system MUST be able to perform application state relocation for MEC Apps serving multiple users.
KPI's support	[KpiTemplate-01] – The system MUST be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA's and SLO's.

	[KpiScaling- 01] – The system MUST be able to determine the number and types of resources to support workload scaling in order to maintain KPI's and SLO's.
	[Monitoring-01] – The MEC system MUST be able to collect infrastructure and service monitoring information, in order to feed KPI-based automated management and orchestration features.
Network traffic control	[TControl-01] The management system must be able to provide provisioned MEC platforms with guaranteed network bandwidth.
	[TControl-02] The management system must be able to rate limit the provisioned MEC platforms traffic flows.
	[TControl-03] The management system must have the ability to selectively apply the traffic control on different types of traffic, and have the ability of traffic classification.
	[TControl-04] Within the constraints set by the orchestration and management, an authorized mobile edge application shall be able to request the activation, update and deactivation of the mobile edge application traffic rules dynamically.
Scaling	[Scaling-01] – The MEC system MUST be able to scale a MEC App, on OSS request or automatically based on KPIs, in order to increase/decrease the capacity.
	[Scaling-02] The MEC system MUST be able to terminate a MEC App whenever it is no longer required to serve users.

Table 4: MEC Requirements

A1.3 C-Ran Technical Requirements

Delivery D2.3 decomposes C-RAN into RFBs and further discuss the affinity of those RFBs. For completeness, the affinity graph between the different proposed RFBs is presented next.

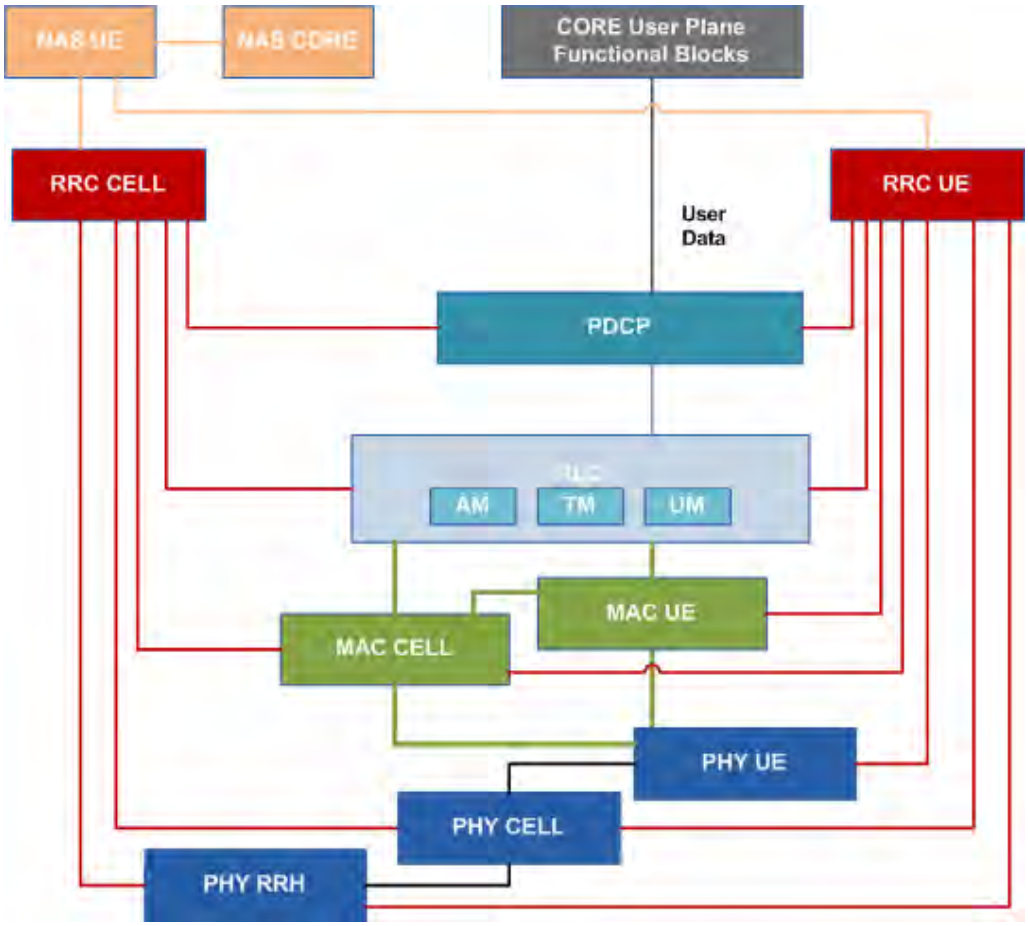


Figure 13: Affinity graph between different C-RAN functional blocks

Here, in the next table, we further analyse the location, event handling capacity and scaling requirements from those functional blocks.

FUNCTIONAL BLOCK (FB)	EXAMPLES OF FB DECOMPOSITION	FB DEPLOYMENT LOCATION	EVENT HANDLING CAPACITY ()	APPLICATION SCALING REQUIREMENT
PHY RRH	Physical NF – not virtualized	Antenna site		Not Scalable as application
PHY Cell	all the processes executed for one cell, e.g. FFT/iFFT, Modulation, Cyclic prefix	Antenna site or Front-End Cloud	every 10 ms (LTE radio frame length)	scaling decision may be reactive (based on computational latency of previous frame). Less than 10 ms requirement.
	Joint Multiuser Detection – jointly process the received		New UE could arrive or leave	Scale in/out may be dependent on UE mobility.

	signals from multiple UE from more than one RAP (MTPD, INS)		asynchronously. Scaling decision should be based on current computational latency and next state prediction	About 5-10 seconds worst case (bus, or train travelling between RAPs)
PHY User (UE)	HARQ must be sent 3 ms after receiving the frame Convolution coding	Front-End Cloud or EDGE cloud	new frame every 10 ms (LTE radio frame length) , but events that results capacity dependent on UE mobility.	Scale in/out may be dependent on UE mobility. About 5-10 seconds worst case (bus, or train travelling between RAPs)
MAC Cell/Scheduling Real Time	ICIC (Intercell Interference Coordination)	Front-End Cloud or EDGE cloud	every 10 ms (LTE radio frame length), Works with a cluster of RAP's, scaling events not coming in peaks.	number of minutes in most cases, dependent on UE mobility. About 5-10 sec
	link adaptive part	antenna site	Dependent on current antenna measurements, need to be executed locally on antenna site, latency sensitive	10 ms If implemented in proactive fashion could be less time sensitive
MAC User (UE)	UE Power control	EDGE cloud	LTE case it can happen maximum 1000 times within a second per ue. capacity is Number of users in 1ms	not coming in peaks, 5-10 seconds worst case
RLC	It includes processes related to segmentation/concatenation of PDCP PDUs based on information exchange with MAC and PDCP. Several	EDGE cloud	*depends on the mobility and traffic intensity of UE. For the EDGE cloud slow change in*	number of minutes to scale

	modes are supported: Transparent, Acknowledged and Unacknowledged. Each case could be a separate FB		*number of ue associated with it.*	
PDCP Packet Data Convergence Protocol	transfer of user plane data, transfer of control plane data, header compression, ciphering, integrity protection.	EDGE cloud or Central cloud	Depends on ue activity levels, would change through the day in predictable manner (peak in the morning, less activity in the night, etc)	scaling not strict time constrained, and predictable. number of times in a day
RRC Cell		EDGE cloud or Central cloud		
RRC User (UE)	Handover UE measurements reporting, QoS management, paging	EDGE cloud or Central cloud	about 30% of UE are in the handover state, so with central deployment number of scaling events in a day	scaling not strict time constrained, number of times in a day
NAS User (UE)	It refers to the user procedures related to signaling between the UE and MME	EDGE cloud or Central cloud	Asynchronous, depends on user mobility. Because of deployment on central cloud slow change in number of the users in the whole network	scaling not strict time constrained, number of times in a day
NAS Core	MMEs load balancing, MME overload control, GTP-C signaling load control…	EDGE cloud or Central cloud		

Table 5: C-RAN RFB requirements

A1.4 NFV vs. MEC Comparison

NFV	MEC

NFVO only orchestrates Network Services (NS), not VNFs (for those are VNFMs)	MEO orchestrates MEC Apps (MEC has no combination of MEC Apps as NSs combine VNFs)
NFV has no services platform to provide services	MEC has a service platform to provide services to Apps, which must be managed (access, auth, etc.)
The deployment details of NSs (e.g. location) can be decided by the NFVO, but also by the VNFM	The deployment details of a MEC App is only determined by the MEO
Mobility issues are not very relevant (although in some cases may arise)	Mobility issues (state movement) are relevant
Location issues are not always relevant (although in some cases may happen)	Location issues are always relevant

Table 6: NFV vs MEC comparison

A2. State of the art

A2.1 VIM OpenStack Virtual Infrastructure Management

This section provides a summary of the capabilities exposed by the virtual infrastructure, which are relevant to the orchestration layer.

Network Traffic Control

Neutron has become OpenStack's 'networking as a service' de facto project, and provides multiple networking services, QoS is being one of the key features provided. The supported traffic control requirements in Mitaka release are rate limiting answering [TControl-02], and the dynamic activation/deactivation upon request [TControl-04]. However, on the downside the missing features are bandwidth guarantee [TControl-01] and having a more mature traffic classification capability [TControl-03] (e.g. layer 7), with the latter becoming an active discussion at last OpenStack summits.

Scheduling parameters

In order for the orchestration layer to be able of making a 'smart' scheduling decision, the VIM has to expose the required set of parameters for the orchestrator to take into account. However, at this point in time, most of the aren't supported. On the upper side - requirements [AppSched-05] (description of the virtualized resources) can be satisfied by the usage of templates provided by such projects as Heat and Tacker as well as [AppSched-06] (Required network connectivity description). However, on the downside requirement [AppSched-08] (Physical location requirements) is hardly fulfilled. The possible solutions to accomplish that can be by made by the usage of OpenStack's Nova (compute project) regions and cells. For the containers scheduling on Kubernetes/OpenShift, more options are provided and more fine grain scheduling (through labels) can be done.

Mobility support

While the OpenStack Nova (compute) project provides support for a subset of functionality for migrating VM instances from one physical host to another, it lacks some of the properties required for full mobility support: [Mobility-01], [Mobility-02]. The user of the migration feature in its current form cannot specify the physical host on which the VM will be migrated, as this decision is left out to the scheduler. For the container COE, the idea is to quickly recreate the container rather than migrate it.

KPI Support

A KPI is a metric used to evaluate factors that are crucial to the performance of a workload or service. Operationally KPIs act as a simple set of indicators to measure data against -- a sort-of service success gauge. In order to appropriately monitor and measure KPIs requires quantitative and qualitative metrics. These metrics are typically captured through the use of telemetry providing both platform and service level data.

Current service orchestration approaches are based on the use of predefined configurations for the node(s) hosting the workloads. The Orchestrator then requests instantiation of the predefined configuration to bring the workload into service on specific hardware platform, for instance through usage of pre-compiled deployment templates (i.e. OpenStack Heat Orchestration Templates (HOT), TOSCA descriptors, etc.). These templates are managed by orchestration platforms through the use of catalogues, (for instance, OpenStack Murano project can be used to store and manage HOT templates for OpenStack Heat). However, this approach does not scale efficiently. As the number of different services to be supported by the platform increases as well as the granularity of service specific KPIs (Key Platform Indicators) and SLOs (Service Level Objectives) it results in a huge number of deployment templates to supported deployment of services. A more effect approach may be based around the use of dynamic template definitions at deployment time to meet specified KPI's as described in Section A3.9.3.

A2.2 VNFM/NFVO

A2.2.1 Cloudband

Cloudband management system is based on two main components, VNFM (VNF management) and NFVO (NFV orchestrator). In the following we would focus on the VNFM.

Cloudband VNF management system is mostly based on OpenStack and open source services. Specifically, on top of OpenStack main projects (NOVA, Neutron, Cinder and Glance) Heat is utilized for VNF deployment and resource allocation. To further allow VNF lifecycle management we utilized Mistral workflow engine that operates in conjunction with Heat. We note that the selection of a workflow engine for a generic VNF management has been identified as an efficient approach in terms of providing a quite broad generic management capabilities and with relatively low complexity (Odini, Marie-Paule. "Short Paper: Lightweight VNF manager solution for virtual functions." Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on. IEEE, 2015).

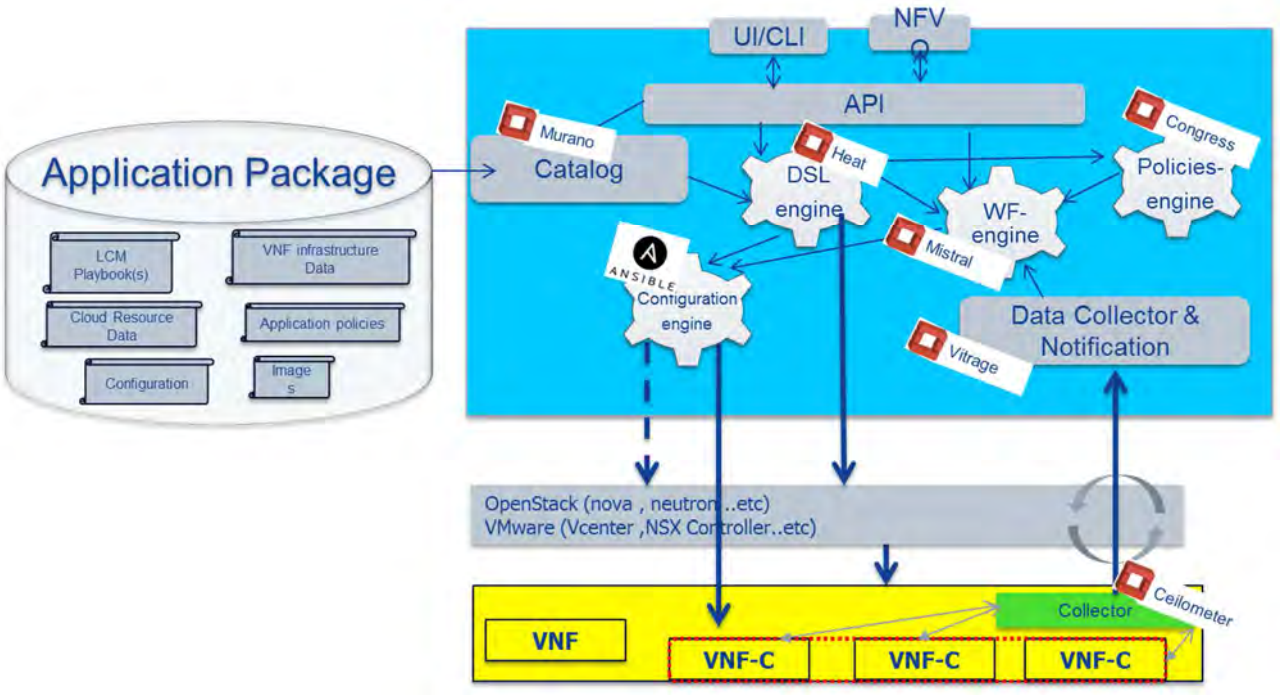


Figure 14: Openstack based generic VNF management system

Figure 14 depicts the architecture for the VNF management system. As indicated, the architecture is based on OpenStack services, such as: Heat, Mistral, Murano, Ceilometer, Vitrage and possibly Congress. In addition, it utilizes Ansible as an open source configuration management. This architecture can support all of the operations that are required for a VNF lifecycle management, including deployment, monitoring, scaling healing and termination (as depicted in Figure 15).

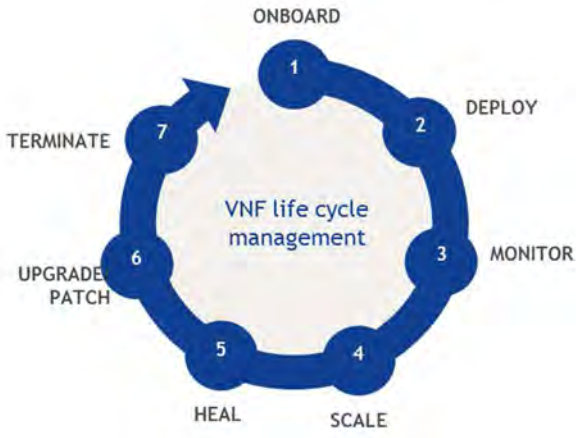


Figure 15: VNF lifecycle operation

For example, Deployment takes place once the onboarding process is complete. Deployment entails ensuring that the newly-introduced application is deployed with its name and the correct environment, on the correct VMs, with the right IPs, etc.

After the onboarding process is complete, the second LCM stage—Deployment takes place (Figure 16).

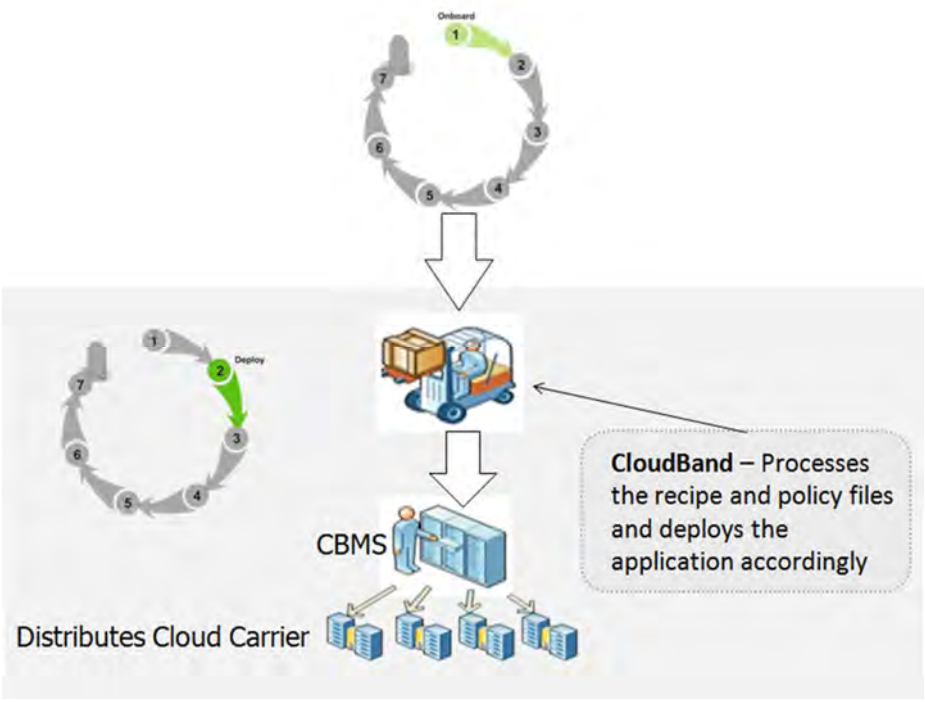


Figure 16: The deployment workflow

Only the customer user can deploy applications. There are two ways to deploy:

- From the Catalog (add application blueprint to the Catalog specified in onboarding)
- Direct deployment of Deploy Stack Directly on OpenStack Node

The HOT template is validated by OpenStack during deployment. No validation is performed when the HOT template is onboarded.

After an application is deployed, a service will be created in the MY CLOUD > DEPLOYMENTS. Under the service the customer user can see the stacks of the application.

For each deployment, a job will be created.

For the deployment to succeed, one should ensure that the Hot is valid and that all the required resources for the stack are on the node (for example, the image).

A2.2.2 OpenMano

OpenMANO implements components from the ETSI NFV MANO stack. Currently, the situation with regards to the requirements outlined in this Annex is the following:

Network Traffic Control

OpenMANO supports the definition of link parameters in the VNFD descriptor as well as in the Network Scenario Descriptors (NSDs). They include the type of link (point-to-point, LAN-type, etc.) as well as quality of service parameters

Scheduling parameters

Currently, OpenMANO does not support scheduling internally. However, the OpenMANO component in the OpenMANO project controls a VIM where NFV services are offered including the creation and deletion of VNF templates, VNF instances, network service templates and network service instances using the openmano API. This can be used by other components to implement scheduling.

Mobility Support

Currently, OpenMANO concentrates on creating NFV-based scenarios. As such, the VNFDs are static and do not provide hooks to define mobility for the virtual machines (VMs) that are included in a VNFD.

KPI Support

OpenMANO offers a northbound interface, based on REST (openvim API), where enhanced cloud services are offered including the creation, deletion and management of images, flavours, instances and networks. The implementation follows the recommendations in NFV-PER001.

A2.2.3 Open Baton

Open Baton (http://openbaton.github.io) is an ETSI NFV compliant MANO framework. It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures. In the release 3, Open Baton significantly increased the number of available components that are part of the ecosystem and included new functionalities for simplifying the way Network Service developers deploy their services.

Open Baton is easily extensible. It integrates with Openstack infrastructure and provides a plugin-based mechanism for supporting additional VIM types. It supports Network Service orchestration either using a generic VNFM or interoperating with VNF-specific VNFM. It uses different mechanisms (REST or PUB/SUB) for interoperating with the VNFMs. It integrates with additional components for the runtime management of a Network Service. For instance, it provides auto-scaling and fault management based on monitoring information coming from the monitoring system available at the NFVI level.

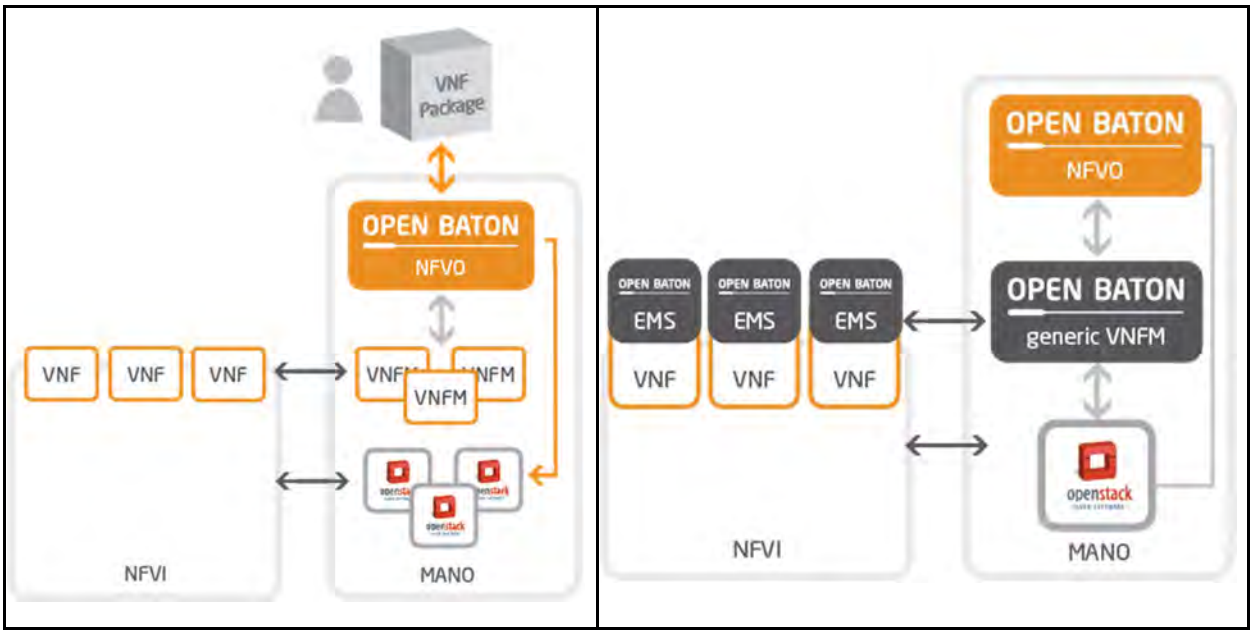


Figure 17: OpenBaton integration into OpenStack

The Figure below depicts the Open Baton architecture.

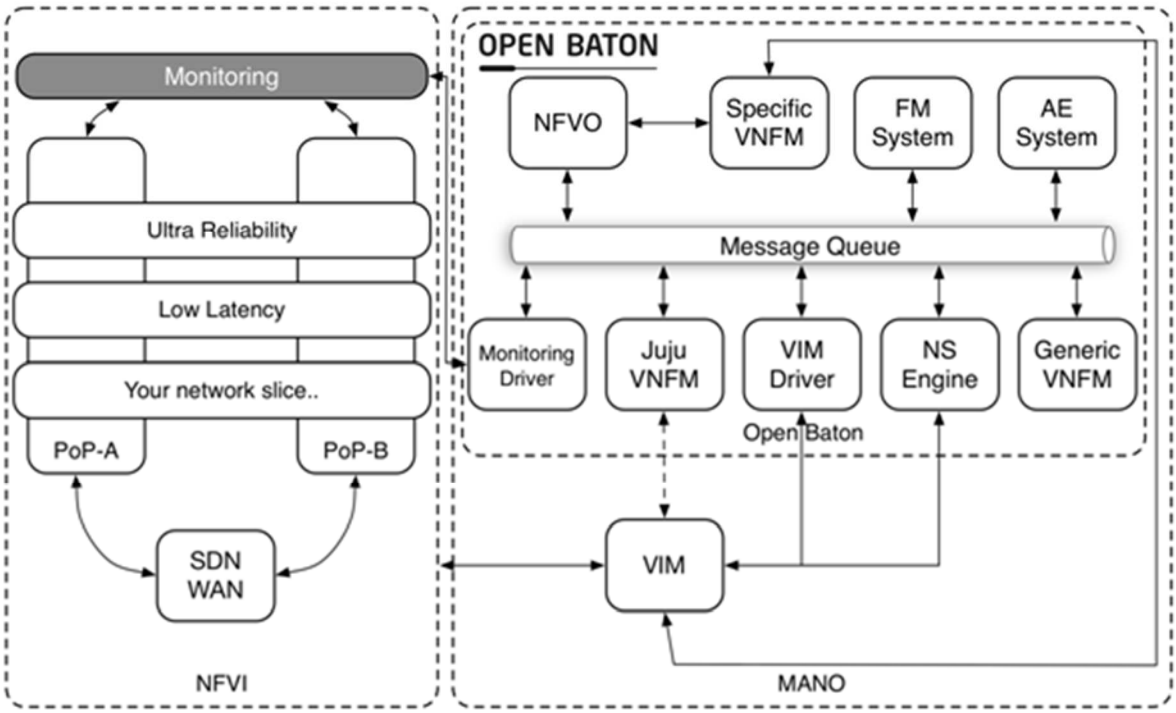


Figure 18: Open Baton architecture

A2.2.3.1 Features

The Open Baton implementation includes the following list of features:

- A Network Function Virtualization Orchestrator (NFVO) completely designed and implemented following the ETSI MANO specification.
- A generic Virtual Network Function Manager extendable (VNFM) able to manage the lifecycle of VNFs based on their descriptors. The Generic VNFM can execute the following operations:
    - Request to the NFVO the allocation of specific resources for the virtual network instance (using a granting mechanism)
    - Can request from the NFVO the instantiation, modification, starting and stopping of the virtual services (or directly to the VIM)
    - Instructs the generic Open Baton EMS to save and to execute specific configuration scripts within the virtual machine instances
- A Juju VNFM Adapter in order to deploy Juju Charms or Open Baton VNF Packages using the Juju VNFM.
- A driver mechanism for adding and removing different type of VIMs without having to re-write anything in your orchestration logic.
- A powerful event engine useful based on a pub/sub mechanism for the dispatching of lifecycle events execution.
- An auto-scaling engine which can be used for automatic runtime management of the scaling operation operations of your VNFs.
- A fault management system, which can be used for automatic runtime management of faults which may occur at any level.
- It integrates with the Zabbix monitoring system.
- A set of libraries (the openbaton-libs) which could be used for building your own VNFM.
  A Marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFMs.
- A user-user friendly dashboard, which enables the management of the complete environment.
- It provides also a set of mechanisms, which enable the support of external VNFMs. This can be done in the following ways:
    - Publish/Subscribe mechanism using a message queue based on AMQP.


    - REST APIs.
- Open Baton integrates via Plugins to different VIM. By default an OpenStack plugin is provided.
- Docker-based Element Management System
    - It is possible to instantiate a VNF on top of a docker container using the GenericVNFM and VNF Package approach;
    - The compute node need to use the nova-docker driver.
- Identity Management:
    - Possibility of defining different projects;
    - Possibility of registering users and assign them different roles;
    - Possibility of registering users and assign them to different projects.
- Network Slicing Engine (NSE)
    - The NSE instantiates rules on physical networks for allocating dedicated bandwidth as per Network Service specific requirements;
    - The NSE provides an abstracted view of the inter-datacenter networking topology allowing the instantiation of guaranteed bandwidth levels on top of the physical network elements.

A2.2.3.2 Evaluation

Strongest Points:

- Aligned with NFV;
- TOSCA aligned with ETSI NFV;
- Ability of auto-healing ;
- Ability of auto-scaling with configuration;
- Use the QoS capabilities of Mitaka to make network slicing;
- Work with docker or VMs, depending on openstack configuration;
- Extendable in the most of components.

Weakest Points:

- Does not work with the more recent release of openstack, it follow the openstack releases with delay of one or two (now Mitaka with release 3);
- The documentation is not sufficient for all the features;
- The Open Baton EMS needs to have inside the VM, and act as an agent.
- It isn't stable (tested on release 2);
- Have performance issues in auto-scale mechanism (tested on release 2).

A2.2.4 OSM

Open Source MANO (OSM) (https://osm.etsi.org) is the ETSI open source community which aims to deliver a production-quality MANO stack for NFV, capable of consuming openly published information models, available to everyone, suitable for all VNFs, operationally significant and VIM-independent. OSM is aligned to NFV information models, while providing first-hand feedback based on its implementation experience.

The OSM implementation is built on top of 3 main components:

1. RIFT.ware, a service orchestrator (NSO).
2. OpenMANO, a resource orchestrator (RO).
3. Juju, a configuration manager (CM).

The integration of those components is the main job of the OSM developers. The Figure below depicts the basics of this integration.

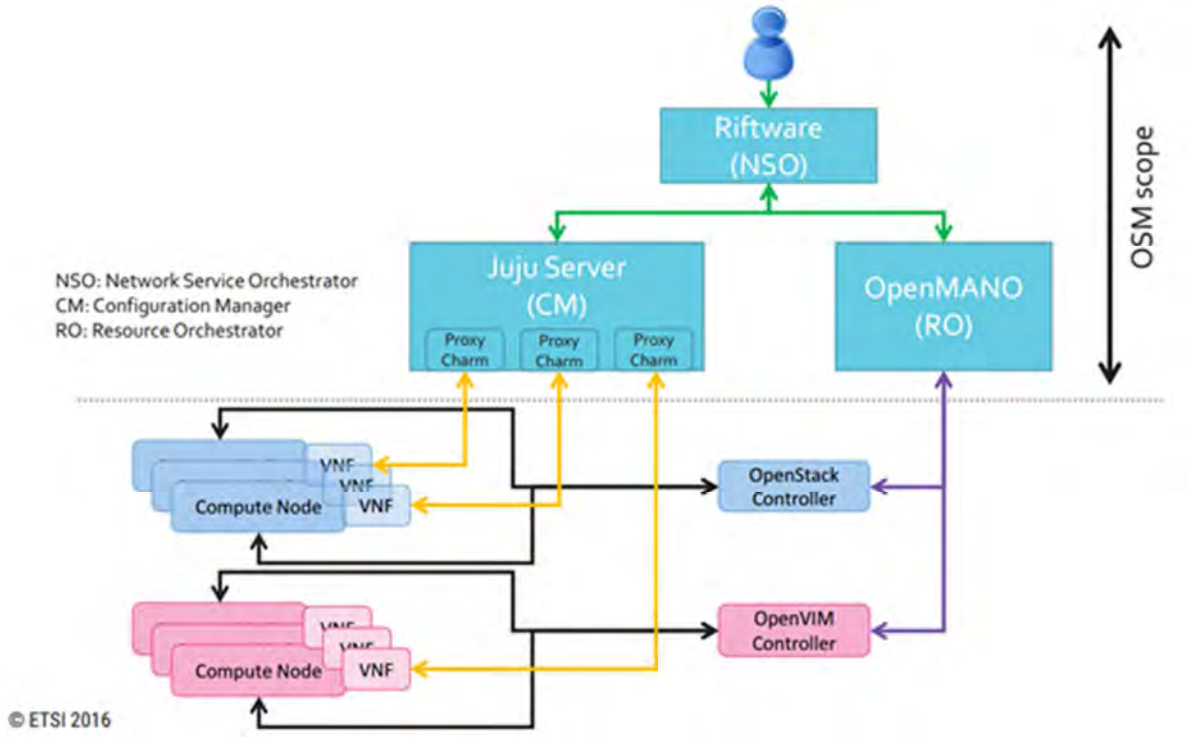


Figure 19: OSM Architecture

A2.2.4.1 Features

The OSM implementation includes the following list of features (based on R2):

- On-boarding experience & VNF Packaging;
    - Allow cloud-init configuration;
    - Create networks at VIM;
    - Remove NSD datacenter network reference;
    - Detailed feedback when deploying and configuring & Error Message from RO & VCA;
    - Distinction between template, particularization and instance for NS;
    - Composer should display descriptors in YAML format;
    - Enhance Visual Differentiation Between NS Catalog and VNF Catalog;
    - Restructure layout of service primitive page;
    - Package creation command line utility;
- EPA based resource allocation;
    - Not explicitly captured;


    - May be included as part of an upgrade to OpenStack Mitaka, likely push to R2 timeframe;
- (Networking) Service Modelling;
    - Juju-2.x;
    - Network types in RO;
    - Allow IP parameters for networks;
    - Configuration/Service Primitive model enhancements;
- Multi-VIM;
    - New VIM connector for VMware vCloud Director;
    - Openvim as reference VIM with EPA capabilities;
    - Datacenter capabilities;
    - Support for VIM Accounts;
- Multi-Site;
    - Multi-site NS.

A2.2.4.2 Evaluation

Strongest Points (OSM R2):

- Is in active development by ETSI group;
- ETSI NFV aligned implementation;
- Roadmap well defined and going in the right direction;
- It works with the more recent release of RIFT.ware (version 4.3.3);
- Open Source community behind;
- Large documentation available (although sometimes not enough);
- Integration with (external) monitoring.

Weakest Points (OSM R2):

- Limited TOSCA adoption, resorting to YAML (NSD and VNFD);
- The documentation is not complete, missing complex examples;
- Limited set of functionalities (even less than RiftWare alone, one of the pieces);
- Some limitations identified, e.g.:
    - Missing an integrated monitoring;
    - Not supporting scaling operations;
    - Not supporting VIM images onboarding;
- Evolutions and corrections come slowly sometimes;
- Not very stable and reliable yet.

A2.2.5 Cloudify

Cloudify (http://getcloudify.org) is an open source cloud orchestration framework that allows users to model applications and services and automate their entire life cycle, including deployment on any cloud or data center environment, monitoring all aspects of the deployed application, detecting issues and failure, manually or automatically remediating them and handle ongoing maintenance tasks. The Figure below depicts the implementation architecture.

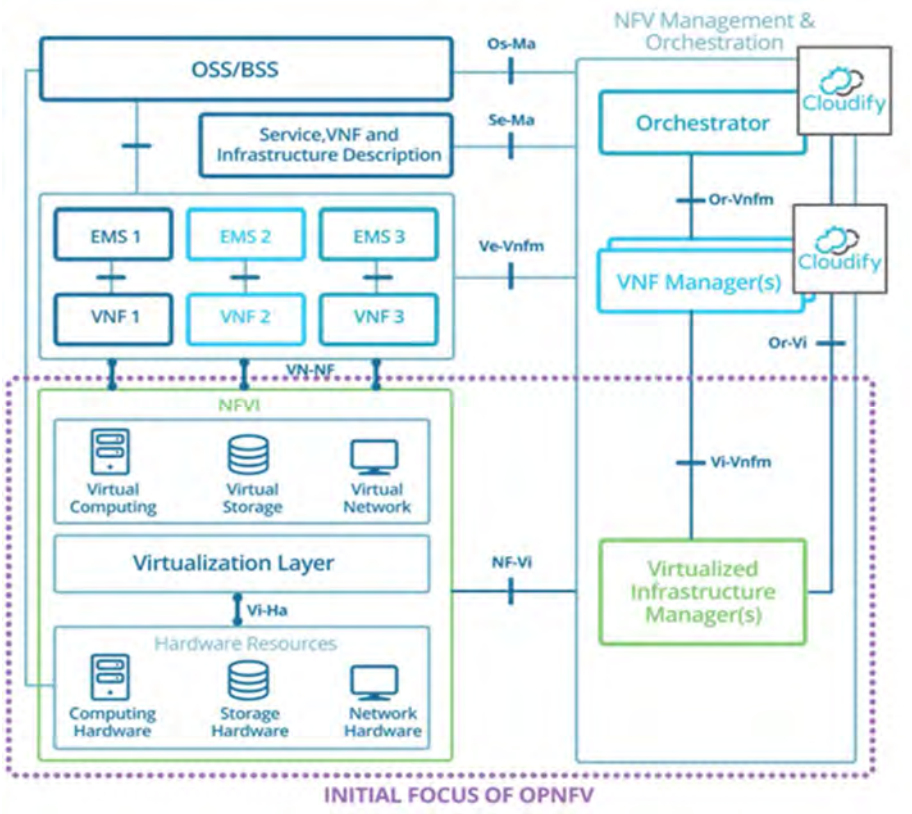


Figure 20: Cloudify architecture.

The VNFM (Virtual Network Function Manager) is responsible for the VNF lifecycle management - e.g. it takes action on instantiation, termination, failover, scaling in and out, and more. Cloudify serves as a generic VNF manager (G-VNFM) and enables full automation of all lifecycle stages for any network function.

The NFVO (NFV Orchestrator), as its name implies, basically serves the purposes of orchestrating and managing end to end network services, through the complex NFV architecture, including integration with SDN controllers, OSS/BSS systems, and more. Cloudify provides a fully open NFVO.

The Cloudify solution is based on the integration of many basic components for data storing, messaging, logging, monitoring and many others. The Figure below depicts this ecosystem.

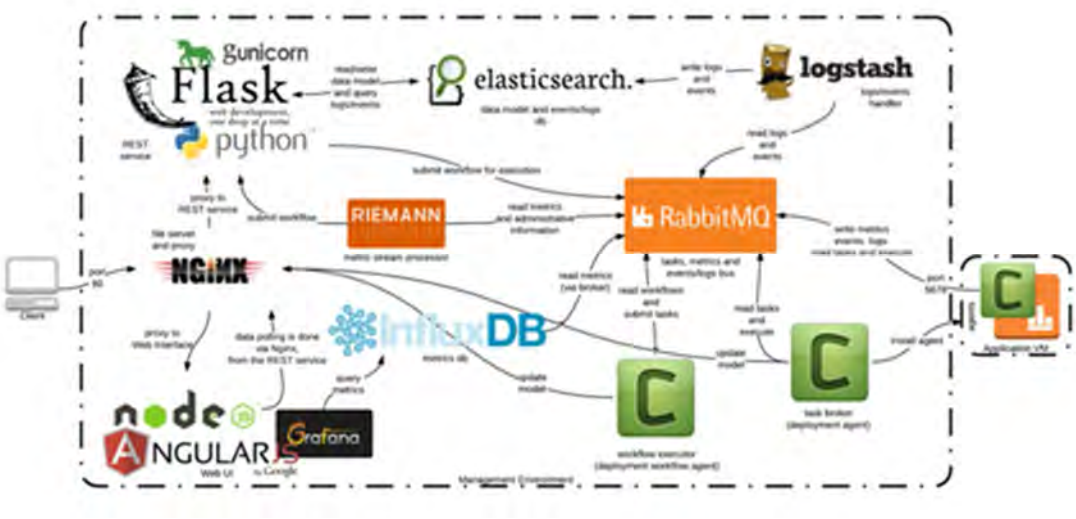


Figure 21: Cloudify components integration

A2.2.5.1 Features

The Cloudify implementation includes the following list of features:

- Full Application Lifecycle Orchestration with Cloudify:
- Pure-play orchestration of applications on the cloud:
- Infrastructure & Tooling Agnostic:
- NFV Orchestration: The only pure-play technology agnostic MANO implementation:
- Hybrid Cloud & Workloads with Cloudify:
- Cloudify Simplifies Cloud Migration & Portability for Enterprises:
- Blueprint Composition:
- Cloudify Telecom Edition: The Telecom Edition is an NFV-specific offering of Cloudify's TOSCA-based orchestration framework. This is the most advanced open source NFV MANO offering and includes built-in multi-VIM support with added extensibility to hybrid workloads and micro services. Included in this release:
    - Multi-VIM - Now fully open source vCloud and vSphere plugins, alongside OpenStack native support;
    - Overlay Service Chaining;
    - Netconf plugin;
    - TOSCA/YANG data modelling interoperability;
    - Network service management;
    - Clearwater vIMS blueprints and plugins;
    - F5 BigIP plugin;


- VNF updates for running VNFs and services;
- NIC ordering;
- Enhanced Platform Awareness coupled with Data Plane Acceleration through integration with Intel;
- Drag and drop Cloudify Composer with VNF-specific components;
- Using ARIA as the kernel for TOSCA Orchestration;
- Support installation within environments with no internet access;
- Support for Cloud Native services through Kubernetes plugin.

A2.2.5.2 Evaluation

Strongest Points:

- Active development by GigaSpaces and will be upgrade in future releases, with public roadmap;
- Well-defined roadmap;
- Overall mature solution;
- TOSCA-based (although non-NFV compliant);
- Multi-Vim;
- Support for containerized and non-containerized VNFs;
- Overlay Service Chaining;
- Built-in auto-healing and auto-scaling policies;
- VNF updates for running VNFs and services;
- Metrics Queuing, Aggregation and Analysis.

Weakest Points:

- Roadmap defined by a single organization;
- It not work with the more recent release of openstack, it follow the openstack releases with delay of one or two (now Liberty with version 3.4);
- It's not multi-tenancy;
- Composer and Web UI are premium (not included in the free distribution).

A2.2.6 Tacker

Tacker is the Openstack project devoted to cover the Management and Orchestration functions of the ETSI NFV architecture (MANO). The main purpose is to manage the lifecycle of VNFs and orchestrate NSs. The Tacker basic architecture is depicted in the following Figure.

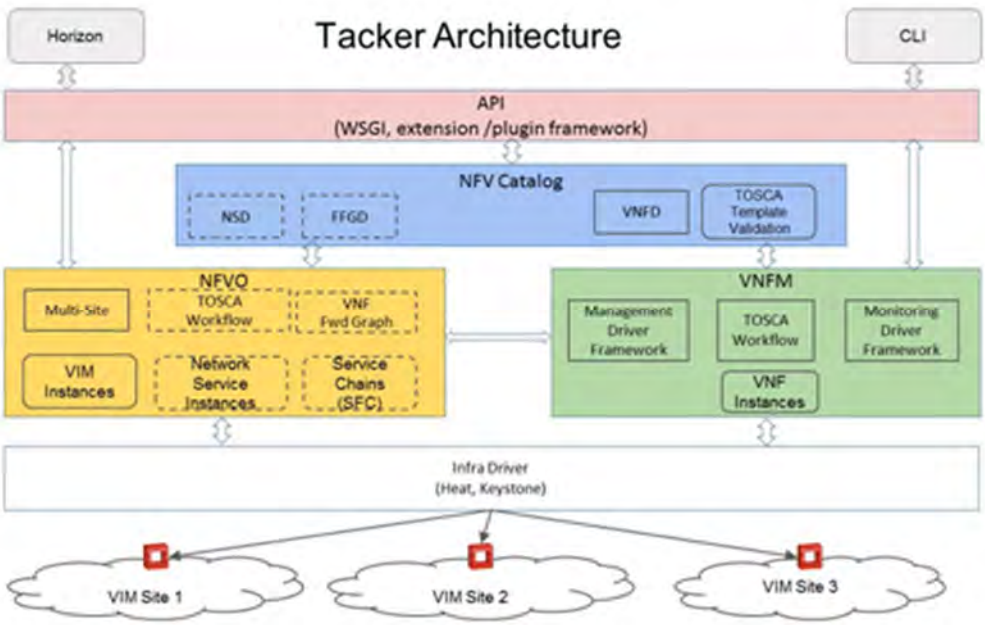


Figure 22: Tacker Architecture

A2.2.6.1 Features

The Tacker implementation includes the following list of features:

- Tacker VNF Catalog
    - Repository of VNF Descriptors (VNFD)
    - VNF definition using TOSCA templates
    - Support for multiple VMs per VNF (VDUs)
    - Tacker APIs to on-board and maintain VNF Catalog
    - VNFDs are stored in Tacker DB
- VNFD using TOSCA
    - Describes the VNF attributes
        - Glance image IDs
        - Nova properties
        - Security Groups


        - Performance Monitoring Policy
        - Auto-Healing Policy
        - Auto-Scaling Policy
    - Working with Heat-Translator

- VNF Auto Configuration
    - Tacker provides a Management Driver Framework
    - Facilitates VNF configuration based on Service selection
    - Inject initial configuration using:
        - Config-drive
        - custom mgmt-driver (connect using ssh / RESTapi and apply configuration)
    - Update configuration in active state
    - Extendable

- VNF Self-Healing
    - Tacker health check starts as VNF becomes ready
    - Ongoing network connectivity check
    - Auto-restart on failure (based on VNFD policy)
    - Extendable Vendor and Service specific Health Monitoring Driver framework

- VNF Auto-Scaling
    - Auto-Scale VNF based on policy
    - Continuous performance monitoring according to KPI described in VNFD
    - Basic Auto-Scaling using common VM metric
        - CPU threshold
        - Custom Monitoring Metric
    - Alarm-based monitoring driver using Ceilometer
    - Manual-scaling option to scale in/out the VDUs

- Multisite VIM Usage
    - Manage multiple Openstack sites
    - Deploy VNFs in multiple OpenStack sites

- VNF Forwarding Graph
    - TOSCA NFV Profile based FG Descriptor can be uploaded to VNF-FGD Catalog


- VNF-FFGD template describes both Classifier and Forwarding Path across a collection of Connection Points described in VNFDs

- Recently NS support was introduced
  - NSD Catalogue
  - NS instantiation

A2.2.6.2 Evaluation

Strongest Points:

- Active development in openstack environment and will be upgrade in every openstack version;
- Roadmap well defined and large community involved;
- Aligned with Openstack releases;
- Good documentation;
- Aligned with NFV;
- Uses TOSCA;
- Integration with other tools (ManageIQ);
- Auto-healing and auto-scaling capabilities;

Weakest Points:

- Still immature and unstable;
- VNFFG and NS only released very recently;
- Basic scaling features;
- Only support Openstack as VIM;
- A few examples available;

A2.2.7 ManageIQ

ManageIQ is an open-source project that allows administrators to control and manage today's diverse, heterogeneous environments that have many different cloud and container providers and/or instances spread out all over the world. Thus it is a higher layer that build upon the VIM management layer, making it a candidate for the NVFO scope. Its main advantage is that it provides a single pane of glass for all the deployments, simplifying management as well as helping to have a global view of the multi-site system.

A2.2.7.1 Features

- Continuous Discovery: ManageIQ is able to connect to virtualization, container, network and storage management systems and discover their inventory, map relationships, and listen to changes.
- Self-Service: ManageIQ defines bundles of resources and publish them in a service catalog. Users can order them from there and manage the full life cycle of a service, including policy, compliance, chargeback/showback and retirement.
- Compliance: ManageIQ may scan the contents of your VMs, hosts and containers to create advanced security and compliance policies.
- Optimization: ManageIQ captures metrics from the different providers, allowing to better understand the current utilization and normal operating ranges. This data may be used to find unused or overbooked systems, get right-sizing recommendations, do capacity planning, or run what-if scenarios.

A2.2.8 Evaluation

Strongest Points:

- Support for several VIM types and instances;
- Good documentation;
- Integration with other tools;
- Supports both VMs and containers;
- Self-service capabilities;

Weakest Points:

- Not aligned with NFV;
- It doesn't make direct use of TOSCA descriptors;

A2.3 Comparison between Orchestrators

Out of all the options covered above, in this section we compare the two main orchestrators that are being considered for the Superfluidity framework: *OSM* and *ManageIQ*. Because ManageIQ is not compliant with the NFV architecture, it's been analysed in conjunction with other tools such as OpenStack Tacker. In both cases OpenStack has been used as a VIM, though ManageIQ also supports Container Orchestrator Engines, such as Kubernetes or Openshift.

Both orchestrators fulfil the NFV requisites previously mentioned with the exception of scaling already deployed VNFs, specific hardware support and in the case of OSM, there is no consideration of costs previous to deployment (while the deployment in ManageIQ has to be approved by an administrator). Also, in both cases, the creation of Service Function Chains is under recent development.

In the context of Superfluidity there is however another requirement, the use of containers as Virtual Deployment Units. While the ETSI NFV architecture does not require VNFs to be deployed only as virtual machines, most NFVOs such as OSM only contemplate this possibility and are not suited for Superfluidity scenes where the orchestration of containers is required. ManageIQ on the other hand, does allow deploying containers, but Tacker does not. Which means that it is possible to deploy containers using ManageIQ but not using TOSCA descriptors or following the NFV philosophy.

The solution to this problem involves a Superfluidity contribution to ManageIQ and is approached in a later section of this document.

Regarding OSM, in the context of the Superfluidity project, containers support (e.g. containers creation and removal) is achieved by the definition and execution of Juju charms by the Juju Server.

A2.4 Management and Orchestration Design

This section intends to identify and describe the different available options regarding cloud infrastructure, cloud infrastructure management and orchestration. We also discuss the pros and cons and the best approaches to be followed by the project.

A2.4.1 Cloud Infrastructure

The cloud infrastructure is the basis of the emerging cloud technology. It allows to create isolated virtual entities, with compute, storage and networking capabilities, appearing as if they were physical machines. The use of hypervisors (e.g. KVM, ESX) is still the most common virtualization technology. However, container-based technologies (e.g. Kubernetes, Dockers*) are getting momentum. ETSI NFV refers to this as NFV Infrastructure (NFVI); we will use this term from now on.

Independently of the virtualization technology in use, some architectural aspects need to be discussed and decided, in the context of the project, in order to find the best approach that fits with our requirements. Superfluidity shall support two different types of services: network functions (e.g. eNB, EPC) and applications (e.g. MEC).

Option 1: One NFVI per Service

The easiest way to support different services is to use a separated cloud infrastructure (i.e. servers, storage, network) (see Figure 23). However, this leads to an inefficient use of resources, as there are

no synergies between similar infrastructures. Furthermore, for an operator, the management effort is considerably larger, as isolated silos need to be built.

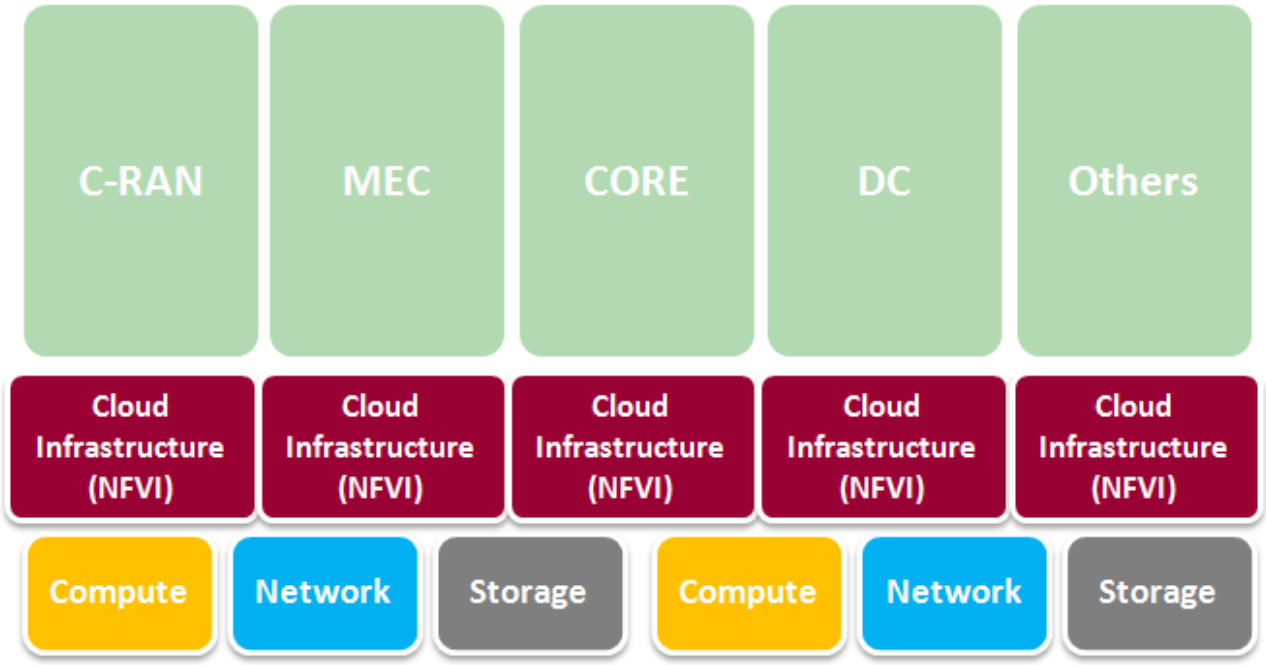


Figure 23: One NFVI per service

Conclusion: Inefficient and complex

Option 2: Common NFVI for all Services eventually locations

To increase efficiency and reduce complexity, it is preferable to have a common infrastructure, which can be used to hold all kinds of services, eventually even in multiple locations (see sections below). For this to be possible, it is required to ensure that all services can rely on similar infrastructure standards. After some discussions among service specialists, we were not able to identify any service specificities that prevent this approach. For this reason, it seems that the best strategy is to have a common cloud infrastructure (NFVI) for all services. This model increases efficiency and simplifies management. The next Figure depicts this view.

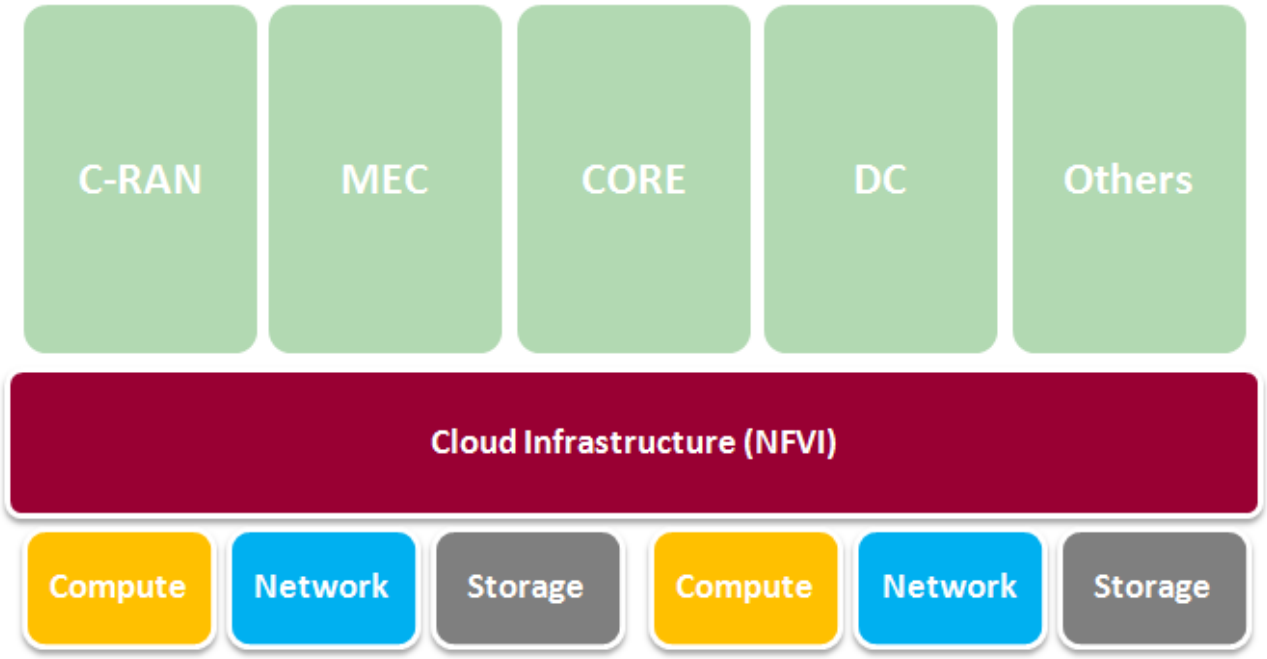


Figure 24: Common NFVI from all services

Conclusion: Preferred

A2.4.2 Cloud Infrastructure Management

To manage a cloud infrastructure (NFVI) a controller/manager is required. This manager is responsible to interact with the hypervisors and provide users with the capacity to manage (create, remove, update and delete virtual resources (compute, storage, network). ETSI NFV refers to this as Virtual Infrastructure Management (VIM); we will use this term from now on. Today, the reference for this component is the open source OpenStack solution. Although there are others like OpenVIM, OpenStack is clearly a de facto standard.

Assuming that a common NFVI can support all services (see section above), it is important to define the strategy to efficiently support the management of resources spread across a large number of datacenters (core and, especially, edges). As described below, there are several options, each with pros and cons.

Option 1: One local VIM per NFVI

The simplest and most common approach is to use one VIM per NFVI, i.e. one manager/controller per cloud infrastructure (datacenter). Following this approach, the VIM function is deployed locally on the datacenter (e.g. edge) and manages all NFVI resources located there (see next Figure). This has the advantage of being a well-known and resilient approach, as inter-datacenter connectivity is not required. However, it has two main disadvantages. Firstly, this may lead to a large number of

VIMs, making the life of the upper Orchestration layer more complex, as it needs to interact with multiple VIMs endpoints. Secondly, the use of multiple VIMs may prevent the use of some capabilities like “VM live migration” among different locations, which may be an important feature. Up to now, it is not clear whether this feature is required and has advantages when compared to other models (e.g. service migration at Orchestration level). Some work still needs to be done to evaluate this.

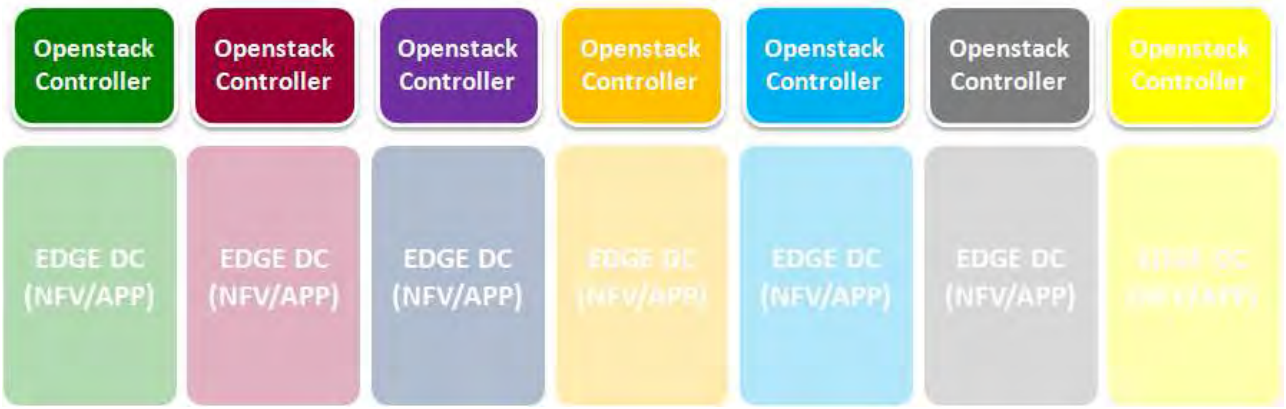


Figure 25: One local VIM per NFVI

Conclusion: Acceptable

Option 2: Single centralized VIM for all NFVIs

The use of a single VIM for all NFVIs located in multiple datacenters (core and edges) is another option to consider. In this case, a single centralized VIM is able to manage all resources located in different locations, providing an external view of a single and federated large datacenter (see next Figure). The different locations can be identified, when needed, based on regions. This approach has the advantage of simplifying the life for the Orchestration layer, as it has a single VIM as endpoint, where all resources can be requested. On the other hand, it allows the use of features like “live migration”, only possible within the same VIM domain, as referred above. However, it has also some disadvantages. From one side, it makes the VIM operation more complex, as it needs to manage a large amount of resources and locations. Furthermore, there may exists some limitations on the number of managed resources. Finally, the manager/controller is no longer local to the NFVI, resulting in traffic increase and delay for the actions to be taken, making also appropriate connectivity a requirement. Anyway, today this seems to not be a hard limitation, as services today already are highly connectivity dependent.

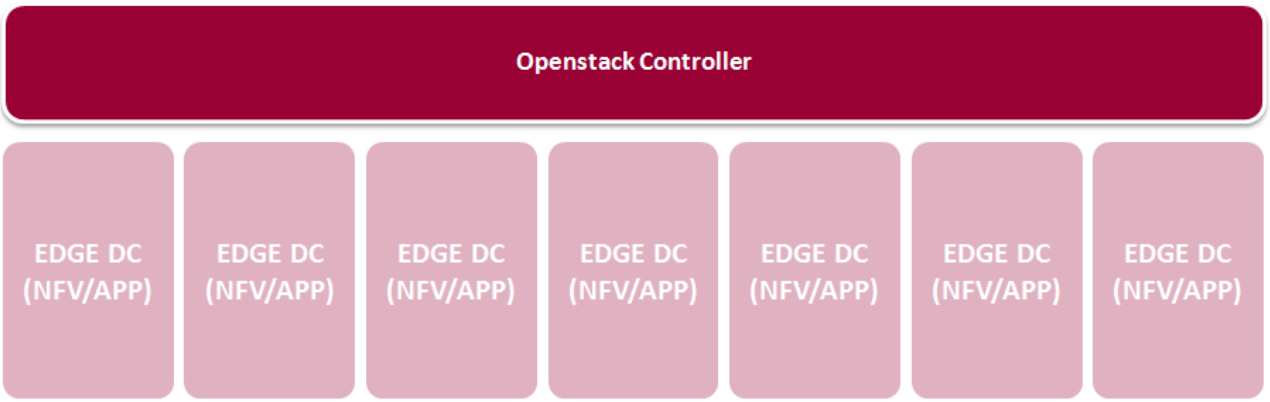


Figure 26: Centralized VIM for all NFVIs

Conclusion: Acceptable

Option 3: Hybrid Option 1 and Option 2

There is still a compromise approach between the two options referred above. In this hybrid approach, multiple datacenters (NFVIs) are grouped into zones and managed by a single VIM (see Figure 27). This option intends to take advantage of the best of both worlds, overtaking some limitations. The group sizing needs still to be defined, but it may depend on a case by case. Compared to option 1, it reduces the number VIM endpoints to a more reasonable number, making the Orchestrator's task easier. On the other hand, it allows users to take advantage of features like "live migration" within the same zone; if groups are properly defined, it can lead to a good tradeoff. Compared to option 2, it can reduce overall complexity and overtake any resource management limitations. In this option, the manager is also no longer local to the NFVI; however, this seems not today a hard limitation as stated above.

Note that in the two extreme cases, this solution is similar to the previous options. If groups are very small, we may lead to groups of a single NFVI, meaning Option 1. On the other extreme, large groups may lead to a single group, meaning Option 2. With this flexibility, it is reasonable to consider this the best option.

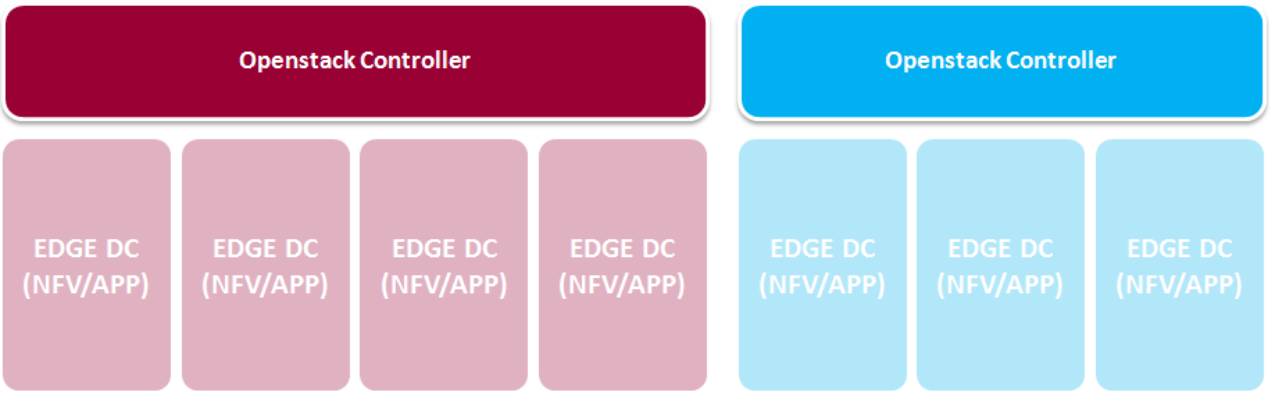


Figure 27: Hybrid Option

Conclusion: Preferred

A2.4.3 Cloud Management and Orchestration

Running on top of Cloud Management, the Orchestration layer is responsible to build complex services by combining and interconnecting the required pieces, on the right locations. Among other things, the Orchestration is able to select the appropriate resources in the right place, based on predetermined constraints. For this, it requires interaction with VIMs. However, as Orchestration can be a very complex task, it will not be simply a single piece, but a set of them, dealing partially with the Orchestration tasks. This section discusses some Orchestration strategies and how do they map to the Infrastructure Management (VIMs).

Option 1: One Orchestrator for all Services and locations

A simple approach to orchestrate all Services in all locations is to use a single Orchestrator. One multi-purpose Orchestrator can deal with all resources and has the advantage of having an overall view of all services, taking eventually advantage of some synergies from that. This model is depicted in next Figure. However, the Orchestrator needs to deal with Service specificities and it may be hard to have a common Orchestrator to handle all that. On the other hand, in real world, different vendors provide different Services, and it is very likely each one brings its own Orchestration for his particular Service. In that case, this solution can be hard to achieve, both technically and commercially.

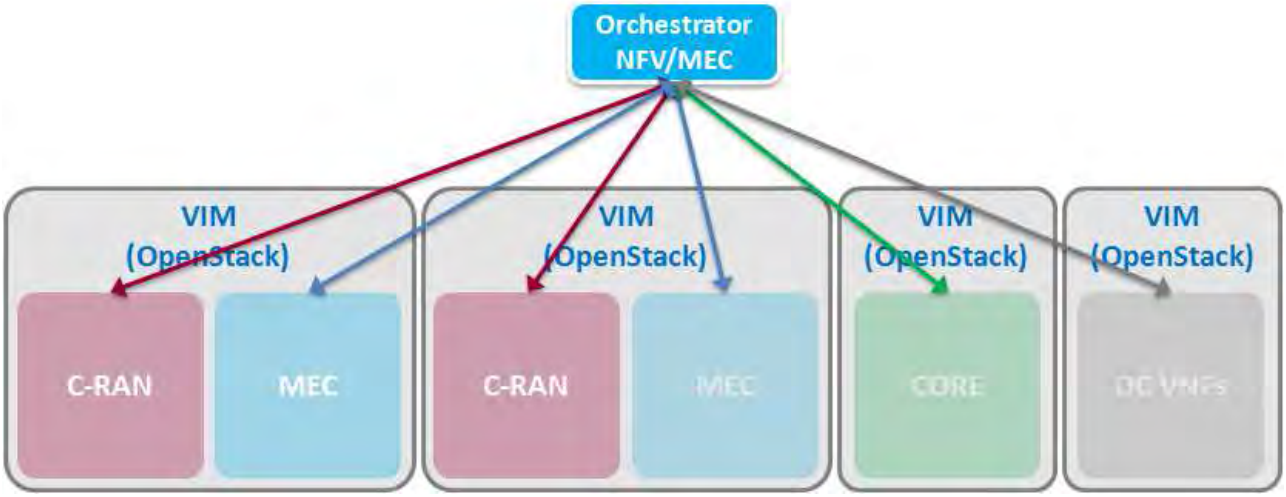


Figure 28: Single orchestrator

Conclusion: Non-realistic

Option 2: One Orchestrator per Service

Another approach is to use different Orchestrators to comprise the overall Orchestration layer. In this case, each Orchestrator is in charge of part of the overall Orchestration tasks (see next Figure). As state above, a dedicated Orchestrator per Service seems a realistic approach; however, other options may also be reasonable. For example, if a vendor provides the C-RAN and the Core, maybe a single Orchestrator can take care of both. Similarly, if an operator has multiple C-RAN vendors, which is common, different Orchestrators may be needed for the same service, one for each particular vendor.

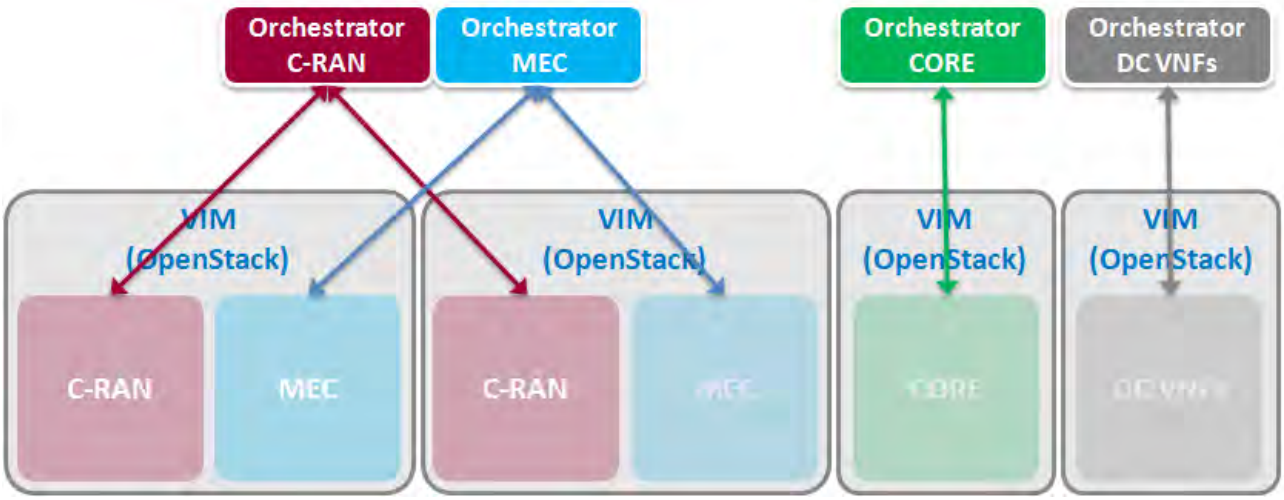


Figure 29: One orchestrator per service

Conclusion: Preferred

A2.4.4 Orchestration Layer

As described in the section above, the Orchestration layer may be composed by multiple Orchestrators, each of them devoted to a particular part of the Orchestration tasks/domains, namely to a particular Service (and from a particular vendor). In this situation, it is relevant to discuss how these Orchestrators can talk to each other and how an operator can have a global view and control about the Services. This section discusses the available options and interfacing models that can be used for this purpose.

Option 1: Northbound and Southbound Interfaces

One possible option leads to the creation of a Top Orchestrator, which integrates all the Service Orchestrators. In this case, Service Orchestrators interact with the Top Orchestration using a Northbound interface (Southbound interface from the Top Orchestrator perspective). For this option, the interaction between Service Orchestrators is not required, as everything is coordinated via the Top Orchestrator. Here, the Operator will own the Top Orchestrator and must integrate it with all Service Orchestrators. The next Figure depicts this hierarchical model.

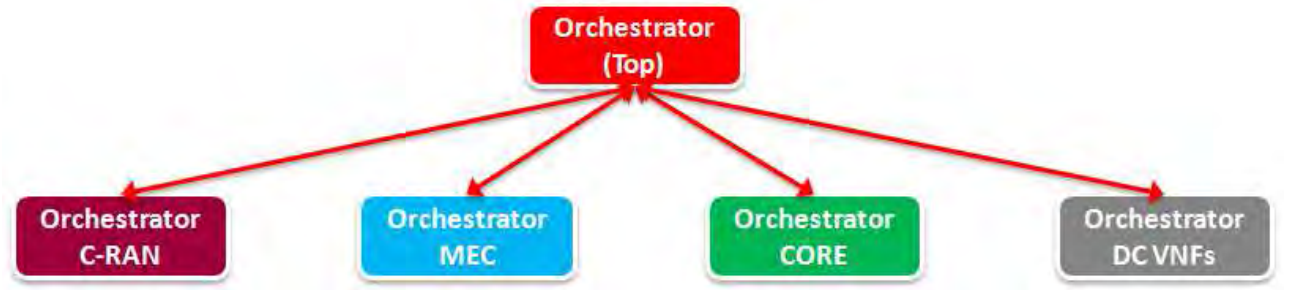


Figure 30: Top orchestrator

Conclusion: Acceptable

Option 2: Eastbound and Westbound Interfaces

Another option is to make Service Orchestrators to integrate with each other's, using East and Westbound interfaces, in order to build an overall service. In this case, Service Orchestrators need to potentially integrate with all (or at least some) of the other Service Orchestrators, making things apparently more difficult and complex (more integrations required – partial/full mesh). On the other hand, the operator does not have any central Orchestration point where he can control the whole

system, but instead multiple Orchestrations, one per Service, which in some cases, may difficult obtaining a global orchestration view. Next Figure depicts this peer-to-peer model.

Figure 31: east-west orchestrator

Conclusion: Difficult

Option 3: Hybrid Option 1 and Option 2

There is still a compromise approach between the two options referred above. In this approach, a Top Orchestrator integrates all Service Orchestrators (interfaces Northbound and Southbound) in a central Orchestration point. This approach reduces the number of integrations required and provides to the operator an overall Orchestration view. Additionally, Eastbound and Westbound interfaces can be used in order to improve the efficiency of the system, in cases where the integration is preferable between Service Orchestrators. The number of interactions among Service Orchestrators and between Service Orchestrators and the Top Orchestrator will depend on the particular cases. The next Figure depicts this hybrid model.

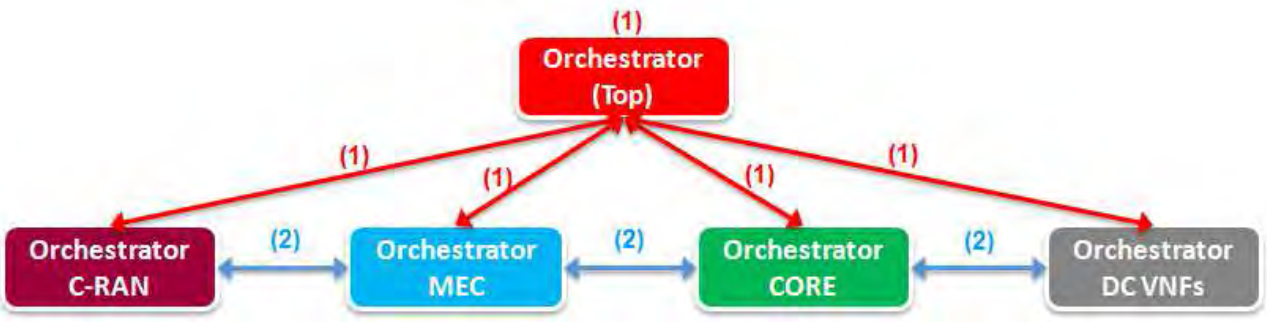


Figure 32: Hybrid orchestrator

Conclusion: Preferred

A3. Superfluidity Contributions

This section covers the contributions within the Superfluidity project that enables the above mentioned deployments and orchestration actions required to enable fluid 5G deployments.

A3.1 Kuryr

Considering that Superfluidity project targets quick provisioning at 5G deployments, there is a need to further advance in the container networking and its integration in OpenStack environment. To accomplish this, we worked on a recent project in OpenStack named Kuryr, which tries to leverage the abstraction and all the hard work previously done in Neutron, and its plugins and services, and use that to provide production grade networking for containers use cases. Hence with a two fold objective: a) make use of neutron functionality in containers deployments; and b) being able to connect both VMs and Containers in hybrid deployments.

In order to map Docker libnetwork to Neutron API, Kuryr is in charge of creating the appropriate objects in Neutron, so that every solution that implements Neutron API can be used for container networking. In this way, all the additional Neutron features can be applied directly to containers ports, such as security groups or floating IPs. To do this, the kuryr service works as an intermediary between the Docker network service (or Kubernetes) and the Neutron server, as shown in next figures, which also include the nested container use case.

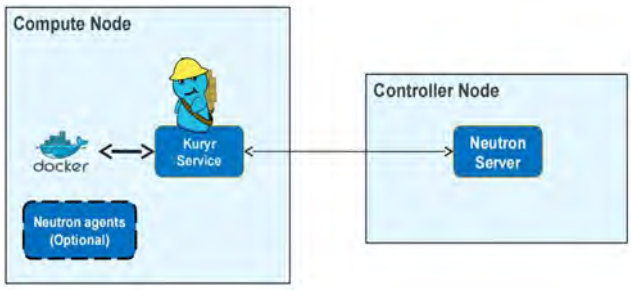


Figure 33: Kuryr Baremetal

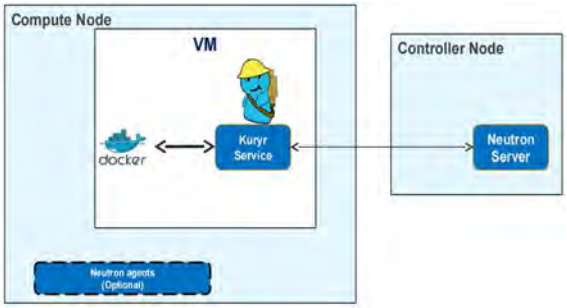


Figure 34: Kuryr Nested

Note that the potential of Kuryr does not need to stop at core API or basic extensions, but also to more advanced networking services such as enabling Load Balancing as a Service for Kubernetes services. What is more, it drives changes to Neutron community, such as the 'tags' addition to Neutron resources in order to allow API clients (like Kuryr) to store mapping data and port forwarding. Kuryr uses this to store the mapping between the Docker and Neutron networks. This information is used, among others, to know if a network must be removed from Neutron when it is removed from Docker (depending on who created the network or if it is being used).

Besides the interaction with the Neutron API, it is needed to provide binding actions for the containers so that they can be linked to the network. This is one of the common problems for Neutron solutions supporting containers networking as there is a lack of nova port binding infrastructure and no libvirt support. To address this, Kuryr provides a generic VIF binding mechanisms that takes the port types received from Docker namespace end and attach it to the networking solution infrastructure as highlighted in next figure.

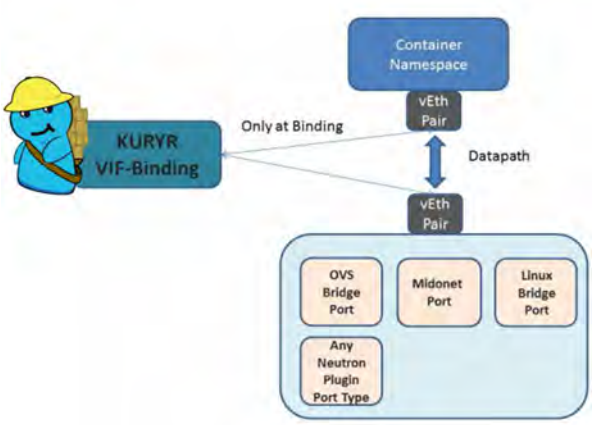


Figure 35: Kuryr VIF-Binding

Additionally, Kuryr provides a way to avoid double encapsulation as is the case in current nested deployments, for example when the containers are running inside VMs deployed on OpenStack. As we can see in next figure, when using docker inside the OpenStack VMs, there is a double encapsulation: one for the Neutron overlay network and another one on top of that for the containers network (e.g., flannel overlay). This creates an overhead that needs to be removed for the 5G scenario target by Superfluidity.

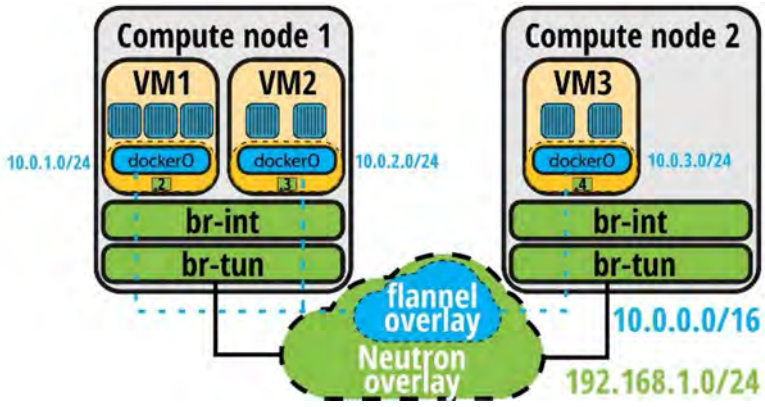


Figure 36: Double encapsulation problem

A3.1.1 Side by side OpenStack and OpenShift/Kubernetes deployment

To enable side by side deployments through Kuryr, a few components had to be added to handle the OpenShift (and similarly the Kubernetes) container creation and networking. An overview of the components is presented in the next image

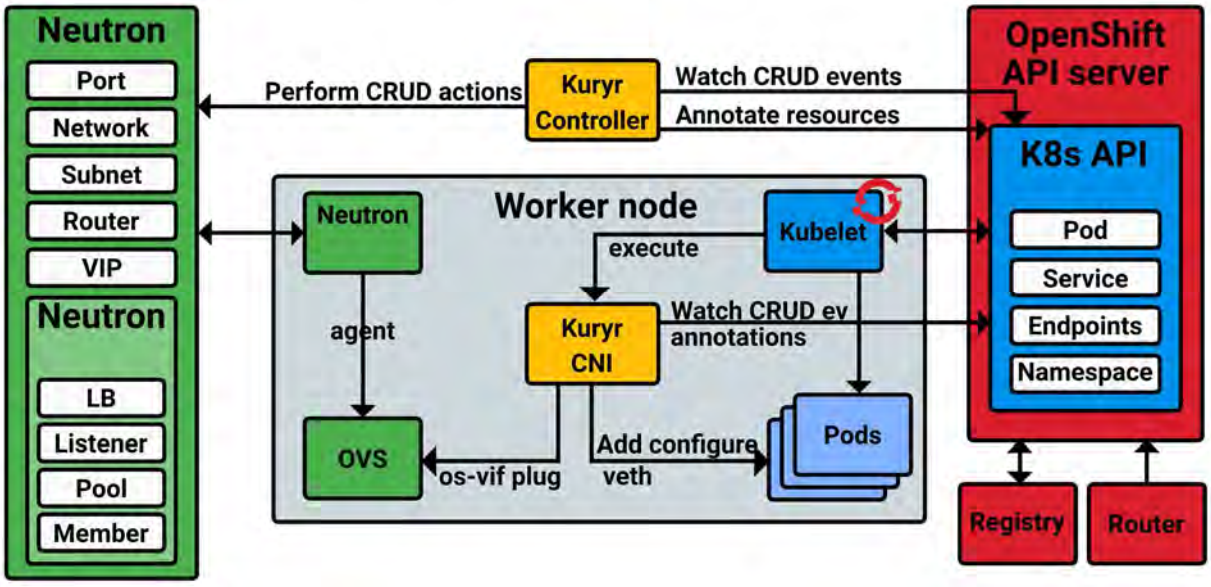


Figure 37: Kuryr components integration

The main Kuryr components are highlighted in yellow. The Kuryr-Controllers is a service in charge of the interactions with the OpenShift (and similarly Kubernetes) API server, as well as the Neutron one. By contrast, the Kuryr CNI is in charge of the networking binding for the containers and pods at each worker node, therefore, there will be one kuryr CNI instance in each one of them.

The interaction process between these components, i.e., the Kubernetes, OpenShift and Neutron components, is depicted in the following sequence diagram.

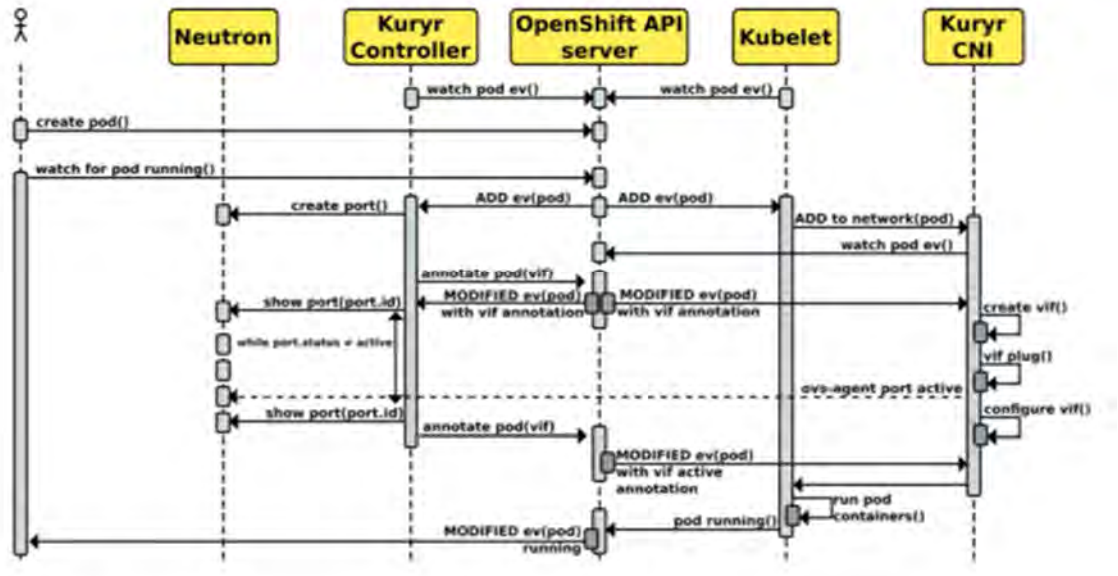


Figure 38: Sequence diagram: Pod creation

Similarly to Kubelet, the Kuryr-Controller is watching over the OpenShift API server (or Kubernetes API server). When a user request to create a pod reaches the API server, a notifications is sent to both, Kubelet and the Kuryr-Controller. The Kuryr-Controller then interacts with Neutron to create a Neutron port that will be used by the container later. It calls Neutron to create the port, and notifies the API server with the information about the created port (pod(vif)), while it is waiting for the Neutron server to notify it about the status of the port becoming active. Finally, when that happens, it notifies the API server about it. On the other hand, when Kubelet receives the notification about the pod creation request, it calls the Kuryr-CNI to handle the local bindings between the container and the network. The Kuryr-CNI waits for the notification with the information about the port and then starts the necessary steps to attach the container to the Neutron subnet. These consist of creating a veth device and attaching one of its ends to the OVS bridge (br-int) while leaving the other end for the pod. Once the notifications about the port being active arrives, the Kuryr-CNI finishes its task and the Kubelet component creates a container with the provided veth device end, and connects it to the Neutron network.

A3.1.2 Nested deployment: OpenShift/Kubernetes on top of OpenStack

There were still some gaps regarding nested containers that we are addressing at the Superfluidity project. Note these VMs are not managed directly by Nova and hence do not have an OpenStack agent inside them. This means we need a mechanism to perform the VIF binding inside the VM and

it needs to be initiated by the local Docker remote driver or Kuryr-Kubernetes CNI for the kubernetes case.

We have extended kuryr to leverage on the new *TrunkPort* functionality provided by Neutron (also known as VLAN-Aware-VMs) to be able to attach subports that are later bound to the containers inside the VMs, running a Kuryr Controller to interact with the Neutron server. This enables better isolation between the containers co-located in the same VM, even if they belong to the same subnet as the network traffic will belong to different (local) VLANs.

To make Kuryr working in nested environment, a few modifications and extensions were needed. These modifications have been contributed to the Kuryr upstream branch, both for Docker and Kubernetes/OpenShift support:

- (Docker) https://review.openstack.org/#/c/402462/
- (Kubernetes) https://review.openstack.org/#/c/410578/

The way the containers are connected to the outside Neutron subnets is by using a new feature included in Neutron, named Trunk Ports (https://wiki.openstack.org/wiki/Neutron/TrunkPort). The VM, where the containers are deployed, is booted with a Trunk Port, and then, for each container created inside the VM, a new subport is attached to that VM, therefore having a different encapsulation (VLAN) for different containers running inside the VM. They also differ from the own VM traffic, which leaves the VM untagged. Note that the subports do not have to be on the same subnet as the host VM. This thus allows containers both in the same and in different Neutron subnets to be created in the same VM.

To continue the previous example based on Kubernetes/OpenShift, a few changes were to be made to the two main components described above, Kuryr-Controller and Kuryr-CNI. As for the Kuryr-Controller, one of the main changes is regarding how the ports, which will be used by the containers, are created. Instead of just asking Neutron for a new port, there are two more steps to be performed once the port is created:

- Obtaining a VLAN ID to be used for encapsulating containers traffic inside the VM.
- Calling neutron to attach the created port to the VM's trunk port by using VLAN as a segmentation type, and the previously obtained VLAN ID. This way, the port will be attached as a subport to the VM, and can be later used by the container.

Furthermore, the modifications at the Kuryr-CNI (and kuryr-libnetwork for the docker case) are targeting the new way to bind the containers to the network, as in this case, instead of being added to the OVS (br-int) bridge, they are connected to the VM's vNIC in the specific vlan provided by the Kuryr-Controller (subport).

For the nested deployment with Kubernetes/OpenShift the interactions as well as the components are mainly the same. The main difference is how the components are distributed. Now, as the OpenShift/Kubernetes environment is installed inside VMs, the Kuryr-Controller also needs to run on a VM so that it is reachable from the OpenShift/Kubernetes nodes running in other VMs on the same Neutron network. With regards to the Kuryr-CNI, instead of being located on the servers, they need to be located inside the VMs actuating as worker nodes, so that they can plug the container to the vNIC on the VM on which they are running.

A3.1.3 Ports Pool Optimization

Every time a container is created or deleted, Kuryr makes a call to Neutron to create or remove the port used by the container. Interactions between Kuryr and Neutron may take more time than it is desired from the container management perspective, and specially from the speed needed by the target superfluidity scenarios.

Fortunately, some of these interactions can be optimized or even avoided. For instance, by maintaining a pre-created pool of Neutron resources instead of asking for their creation during pod lifecycle pipeline. As an example, every time a container is created or deleted, there is a call from Kuryr to Neutron to create/remove the port used by the container. To optimize this interaction and speed up both container creation and deletion, we propose a new Pool management driver at kuryr-kubernetes that takes care of both: Neutron ports creation beforehand, and Neutron ports deletion afterwards. This will consequently remove the waiting time for:

- Creating ports and waiting for them to become active when booting containers
- Deleting ports when removing containers

The main idea behind the proposed pool management driver resides on when and how the Neutron resources are managed, i.e., handling the Neutron resource creation, deletion and updates outside the container lifecycle pipeline -- when possible.

The proposed pool management driver handles different pools of Neutron ports:

- Available pools: There will be a pool of ports for each tenant, host (or trunk port for the nested case), and security group, ready to be used by the new pods being created. Note at the beginning there is no pools, and once a pod is created at a given host/VM by a tenant, with a specific security group, a corresponding pool gets created, and populated with the desired minimum amount of ports.
- Recyclable pool: Instead of deleting the port during pods removal, the port will be included into this pool. The ports in this pool will be later recycled by this driver and put them back into the corresponding available pool, after reapplying security groups to avoid any security breach.

The logic behind this pool driver ensures that at least X ports are ready to be used at each pool, i.e., for each security group and tenant. To provide this functionality, a new VIF Pool driver has been designed (one for the baremetal and one for the nested deployment types) that manages the ports pools upon pods creation and deletion events. It makes sure that at least a certain number of available ports exist in each pool (i.e., for each security group, host or trunk, and tenant) which already has a pod on it. The ports in each Available_pool are created in batches, i.e., instead of creating one port at a time, a configurable amount of them are created at once through Neutron bulk calls. The pool management driver checks for each pod creation that the remaining number of ports in the specific pool is above X. Otherwise it creates Y extra ports for that pool (with the specific tenant and security group). Note both X and Y are configurable and need to consider neutron quotas.

Having the ports ready at the Available_pools during the container creation process will speed up the process. Instead of calling Neutron port_create and then waiting for the activation of the port, it will be taken from the *available_pool* (hence, no need to call Neutron) and only the port info will be updated later with the proper container name (i.e., call Neutron port_update). Consequently, at least two calls to Neutron can be skipped (to create a port and wait for port to become ACTIVE), in favour of one extra step (the port name update), that is faster than the others. On the other hand, if the corresponding pool is empty, a *ResourceNotReady* exception is triggered and the pool is repopulated. After that, a port can be taken from that pool and used for another pod.

Similarly, the pool driver ensures that ports are regularly recycled after pods are deleted and put back in the corresponding *available_pool* pool to be reused. Therefore, Neutron calls are skipped as there is no need to delete and create another port for a future pod. The port cleanup actions return ports to the corresponding available_pool after re-applying security groups -- only if they have been changed during its attachment, otherwise there is no need to call Neutron at all during te deletion. A

maximum limit for the pool can be specified to ensure that once the corresponding available_pool reach a certain size, the ports above this number get deleted instead of recycled. This upper limit can be disabled by setting it to 0.

More information about the upstream design and implementation of these capabilities can be found at the next document:

- Blueprint: https://blueprints.launchpad.net/kuryr-kubernetes/+spec/ports-pool
- DevRef: https://review.openstack.org/#/c/427681/

More instructions about how to enable it and use it are available at:

https://ltomasbo.wordpress.com/2017/05/09/kuryr-ports-pool-speeding-up-containers-booting-time-on-neutron-networks/

A3.1.3.1 Ports Pool Performance Evaluation

Thanks to the above mentioned effort on pool management driver, the time needed to create containers, both in baremetal and in nested deployment when using kuryr is remarkably decreased as it can be seen in the figure below. In Figure 39, Upstream Baremetal and Upstream nested means the current available version of Kuryr for the generic and nested cases, respectively. On the other hand, the Pool Driver ones, represent the optimizations carried out as part of Superfluidity efforts.

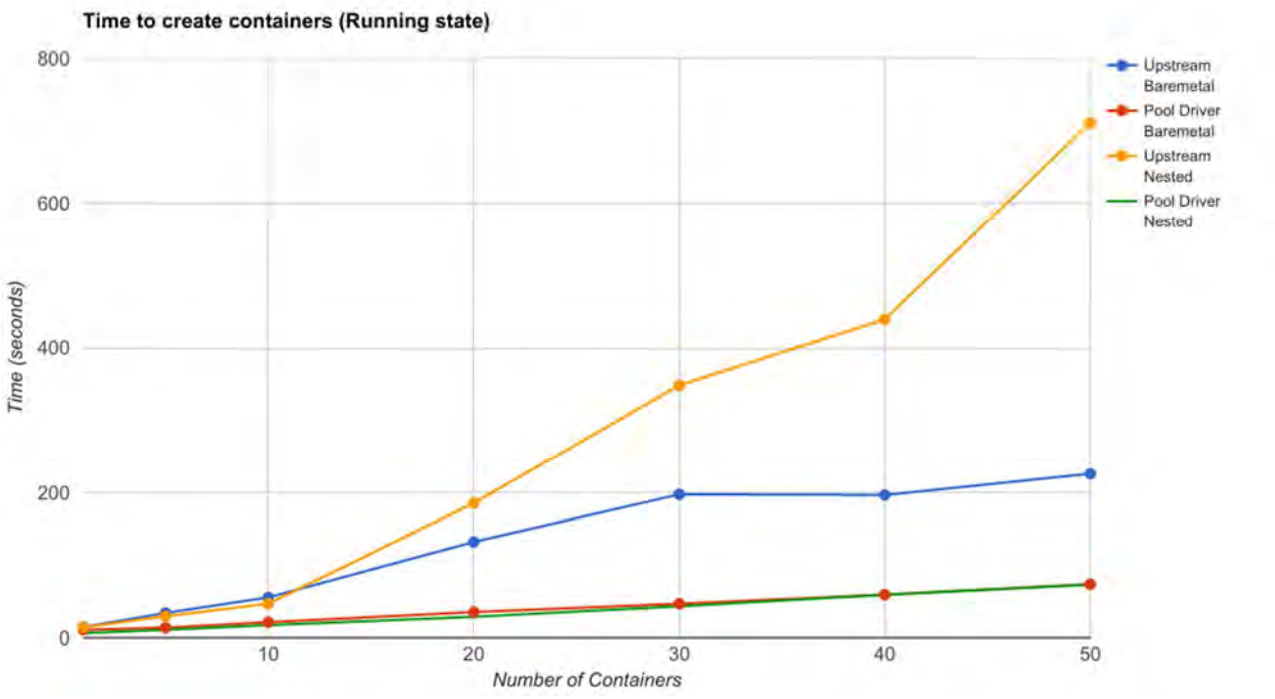


Figure 39: Time from pod creation to running status

As it can be seen, thanks to that, the creation times remain fairly constants, and slowly increase as the number of containers increased, but just do to the actual time to create the containers regardless of the network. Another important point to highlight is that there is no actual difference on performance of creating the containers in baremetal or inside VMs, from the booting time perspective.

In addition to that, thanks to reducing calls to Neutron, the load impose by container networking management on Neutron server is reduced too. Consequently, the ports pool feature helps to improve the scale at what Neutron can work, as well as speeds up other neutron actions unrelated to pod networking.

More details about scale testing of ports pool feature with different SDNs is presented at D7.3.

A3.1.4 Load Balancer as a Service (LBaaS) Integration

A Kubernetes Service (and consequently an OpenShift Service) is an abstraction which defines a logical set of Pods and a policy by which to access them. Whenever this set of Pods is updated, the Service gets updated too. Kubernetes service in its essence is a Load Balancer across Pods that fit the service selection. Kuryr's choice is to support Kubernetes services by using Neutron LBaaS service. The initial implementation is based on the OpenStack LBaaSv2 API, so compatible with any LBaaSv2 API provider. In order to be compatible with Kubernetes networking, Kuryr-Kubernetes makes sure that services Load Balancers have access to Pods Neutron ports.

More specifically, Kubernetes service is mapped to the LBaaSv2 Load Balancer with associated Listeners and Pools based on the protocol and specified exposed ports. Service endpoints are then mapped to Load Balancer Pool members.

As regards to the implementation details, two different kubernetes event handlers has been added to the kuryr controller pipeline to listen to the proper kubernetes events and trigger the related Neutron LBaaS operations:

- LBaaSSpecHandler manages Kubernetes Service creation and modification events. Based on the service spec and metadata details, it annotates the service endpoints entity with details to be used for translation to LBaaSv2 model, such as tenant-id, subnet-id, ip address and security groups.
- LoadBalancerHandler manages Kubernetes endpoints events. It manages LoadBalancer, LoadBalancerListener, LoadBalancerPool and LoadBalancerPool members to reflect and keep in sync with the K8s service. It keeps details of Neutron resources by annotating the Kubernetes Endpoints object.

Both Handlers use Project, Subnet and SecurityGroup service drivers to get details for service mapping. LBaaS Driver is added to manage service translation to the LBaaSv2-like API.

Next figures show the sequence diagram for the pod creation and the load balancer creation, respectively. In the first diagram the interactions between Kuryr, Kubernetes and Neutron components are detailed, for the nested case.

Similarly, in the second diagram, the interactions with the Neutron LBaaSv2 are presented. Note there is no interaction with the CNI part as there is no modifications to be made to how the containers are plugged into the network, just about how they are exposed at the load balancer level. Thus, they just need to be included as members at the pool associated to the corresponding load balancer.

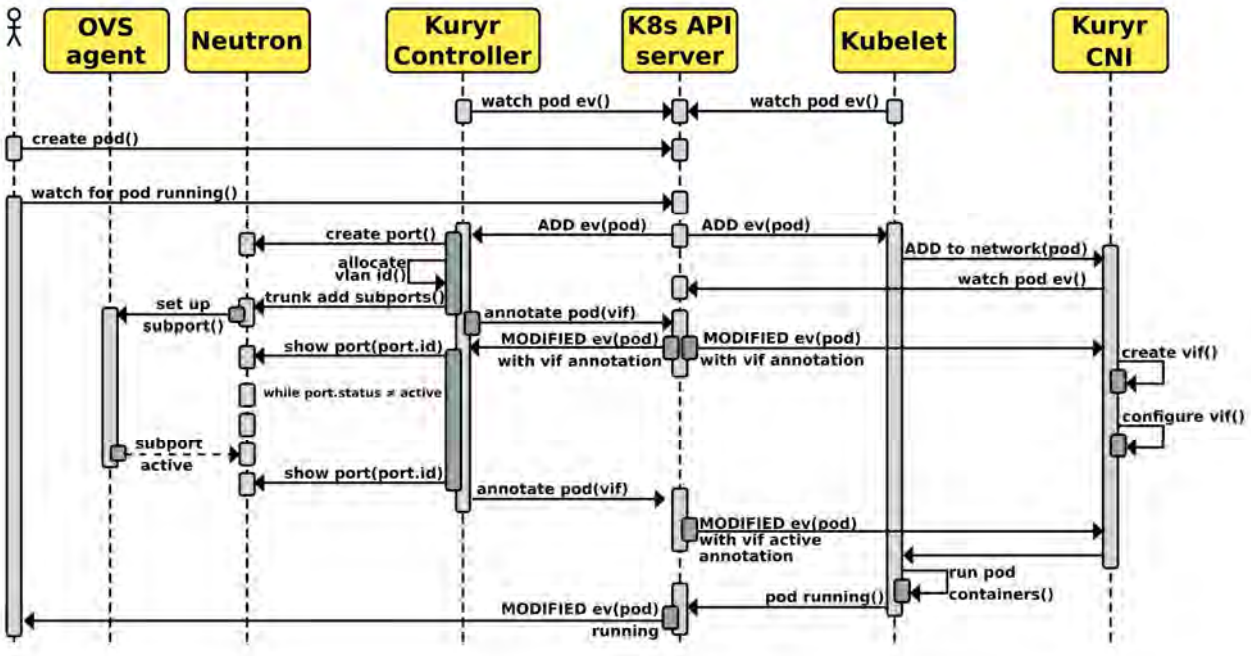


Figure 40: Sequence diagram: nested pod creation

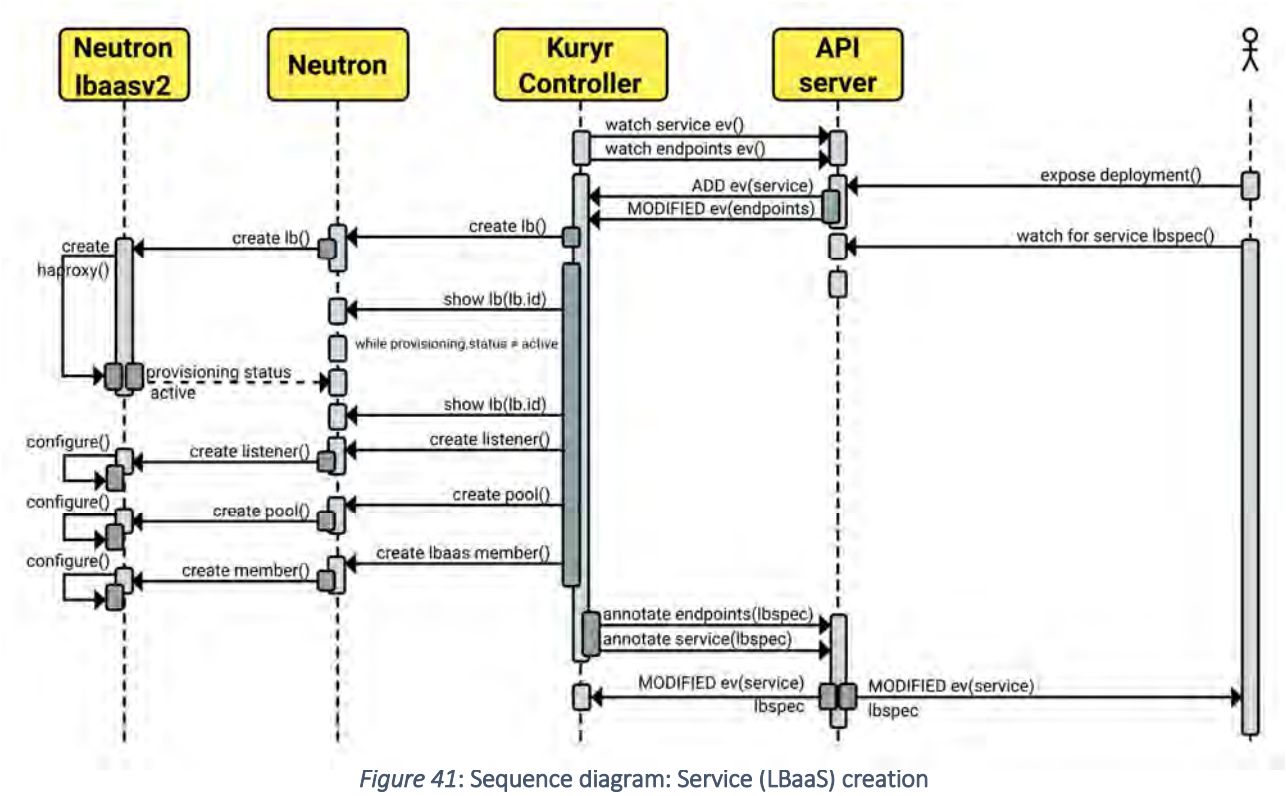


Figure 41: Sequence diagram: Service (LBaaS) creation

In addition to the integration between Kuryr and LBaaSV2, we have also work on extending the support to also integrate the new OpenStack load balancer project, named Octavia. A few modifications were needed to fully support Octavia load balancer modes, as for example it also supports L2 mode:

- https://review.openstack.org/#/c/499103/

Note just internals were modified as the process as well as the interaction are the same described above.

A3.1.5 Support for different Software Defined Networks (SDNs)

In addition to the previous kuryr extensions, we also focused on supporting different SDN backends so that we can leverage kuryr functionality on OpenStack clouds regardless of their SDN selection. The default OpenStack Neutron ML2 driver is OVS and that is supported (and tested) by Kuryr. However, in order to ensure other SDNs, we need to ensure:

- The Kuryr code works with them
- There is enough documentation to be able to use it
- There is enough testing to ensure the support is not lost at some point

We have focused on adding support for another 3 well-known/common SDNs, in this case OpenDaylight (ODL), Open Virtual Network (OVN), and DragonFlow (DF).

In order to cover the above mentioned points, we have tested and make the needed changes to support the three of them on Kuryr: OVN, ODL and DF. It must be highlighted that, thanks to the Neutron API abstraction, the support from the Kuryr side was mostly working out of the box, but thanks to the integration we helped discovering and fixing different problems on the SDNs side. As an example, we discovered and fixed problems related to the Trunks port support in kuryr. These functionality was tested with default ML2/OVS drivers, but both ODL, OVN, and DF recently added the functionality and had some gaps into their implementation, e.g., they were not setting the subports status to ACTIVE:

- ODL: bug (https://bugs.launchpad.net/networking-odl/+bug/1707008) and fix (https://review.openstack.org/#/c/489517/)
- OVN: bug (https://bugs.launchpad.net/networking-ovn/+bug/1707141) and fix (https://review.openstack.org/#/c/488354/)

In addition to ensure proper support for the different SDNs, and specially related to Superfluidity as different SDN solutions may be used at different points on the network, we worked on enhance documentation about how to use the different SDNs with Kuryr, as this will both help adoption as well as finding possible problems that may need fixing:

- https://review.openstack.org/#/c/487885/
- https://review.openstack.org/#/c/497151/
- https://review.openstack.org/#/c/487906/
- https://review.openstack.org/#/c/543556/

Finally, in order to ensure that the current support is not lost, we are also working on CI and the creation of new gates upstream that will ensure new Kuryr additions will not break the support for those SDNs.

A3.1.6 CNI Split

CNI Daemon is an optional service that should run on every Kubernetes node. It is responsible for watching pod events on the node it's running on, answering calls from CNI Driver and attaching VIFs when they are ready. In the future it will also keep information about pooled ports in memory. This helps to limit the number of processes spawned when creating multiple Pods, as a single Watcher is enough for each node and CNI Driver will only wait on local network socket for response from the Daemon.

Currently CNI Daemon consists of two processes i.e. Watcher and Server. Processes communicate between each other using Python's multiprocessing.Manager and a shared dictionary object. Watcher is responsible for extracting VIF annotations from Pod events and putting them into the shared dictionary. Server is a regular WSGI server that will answer CNI Driver calls. When a CNI request comes, Server is waiting for VIF object to appear in the shared dictionary. As annotations are read from kubernetes API and added to the registry by Watcher thread, Server will eventually get VIF it needs to connect for a given pod. Then it waits for the VIF to become active before returning to the CNI Driver.

Communication

CNI Daemon Server is starting an HTTP server on a local network socket (127.0.0.1:50036 by default). Currently server is listening for 2 API calls. Both calls load the CNIParameters from the body of the call (it is expected to be JSON).

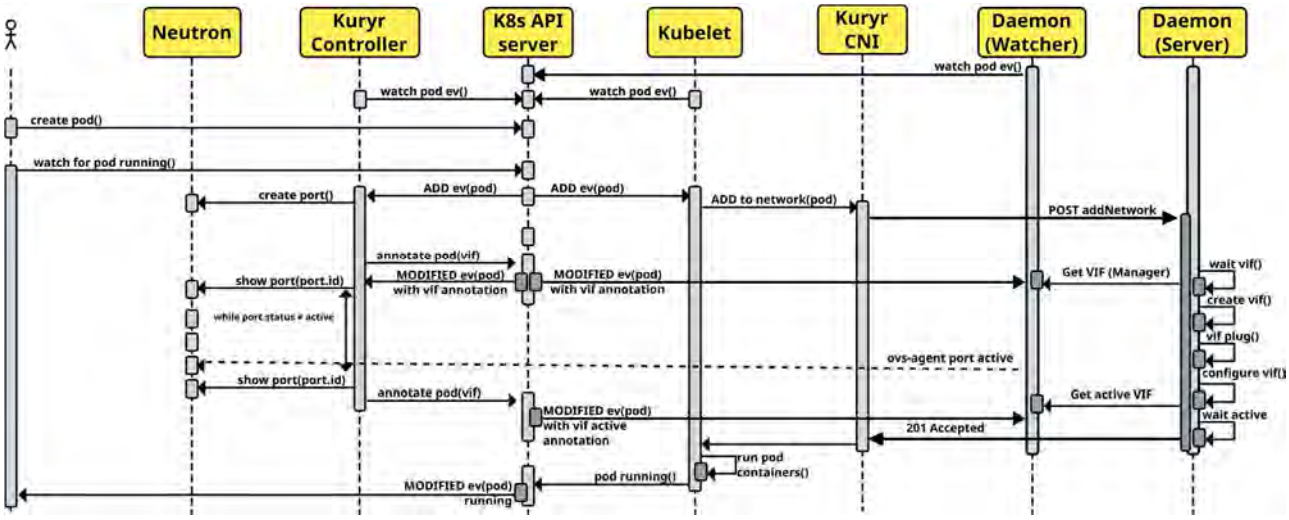


Figure 42: Sequence Diagram: Pod creation with CNI Daemon (Split)

A3.1.7 Containerized Kuryr Components

Following what was already explained before. The main idea would be to follow Kubernetes Kubeadm linux installation guide and install pod networking like this:

kubectl apply -f kuryr.yaml

This simplifies greatly the deployment of Kuryr-Kubernetes from the end-user side and also creates a easy way to perform upgrades. Instead of redeploying from scratch, a user or operator would only have to launch a new container.

Kuryr-Kubernetes gets split into:

- Kuryr CNI as daemon sets
- Kuryr Controller as a Pod
- Service accounts

The end integration in terms of networking is:

Kubelet (host space) -> kuryr-cni exec (host space) -> socket file -> kuryr-cni daemon (container space) -> netlink (container space) + k8s api

While this is just a simple task of further integration with kubernetes, it also had to be integrated with DevStack (Developer-oriented installation of OpenStack for the upstream testing needs). We have developed a way to generate both images and resource definitions for Kuryr-Kubernetes on devstack:

It includes a tool that lets you generate resource definitions that can be used to Deploy Kuryr on Kubernetes. The script is placed in tools/generate_k8s_resource_definitions.sh and takes up to 3 arguments:

$ ./tools/generate_k8s_resource_definitions <output_dir> [<controller_conf_path>] [<cni_conf_path>]

- output_dir - directory where to put yaml files with definitions.
- controller_conf_path - path to custom kuryr-controller configuration file.
- cni_conf_path - path to custom kuryr-cni configuration file (defaults to controller_conf_path).

This should generate 4 files in your <output_dir>:

- config_map.yml
- service_account.yml
- controller_deployment.yml
- cni_ds.yml

Deploying Kuryr resources on Kubernetes

To deploy the files on your Kubernetes cluster run:

$kubectl apply -f config_map.yml -n kube-system
$kubectl apply -f service_account.yml -n kube-system
$kubectl apply -f controller_deployment.yml -n kube-system
$kubectl apply -f cni_ds.yml -n kube-system

After successful completion:

- kuryr-controller Deployment object, with single replica count, will get created in kube-system namespace.
- kuryr-cni gets installed as a daemonset object on all the nodes in kube-system namespace

A3.1.8 RDO Package and CI

RDO is a community of people using and deploying OpenStack on CentOS, Fedora, and Red Hat Enterprise Linux. You can say that is the community version of OSP (Red Hat OpenStack Platform). In order to make the consumption of Kuryr easier for end-users, we've gone through the steps of packaging both Kuryr and its tests (Kuryr Tempest Plugin).

RDO produces two set of packages repositories:

- RDO CloudSIG repositories provide packages of upstream point releases created through a controlled process using CentOS Community Build System. This is kind of "stable RDO".
- RDO Trunk repositories provide packages of latest upstream code without any additional patches. New packages are created on each commit merged on upstream OpenStack projects.

Following diagram shows the global packaging process in RDO.

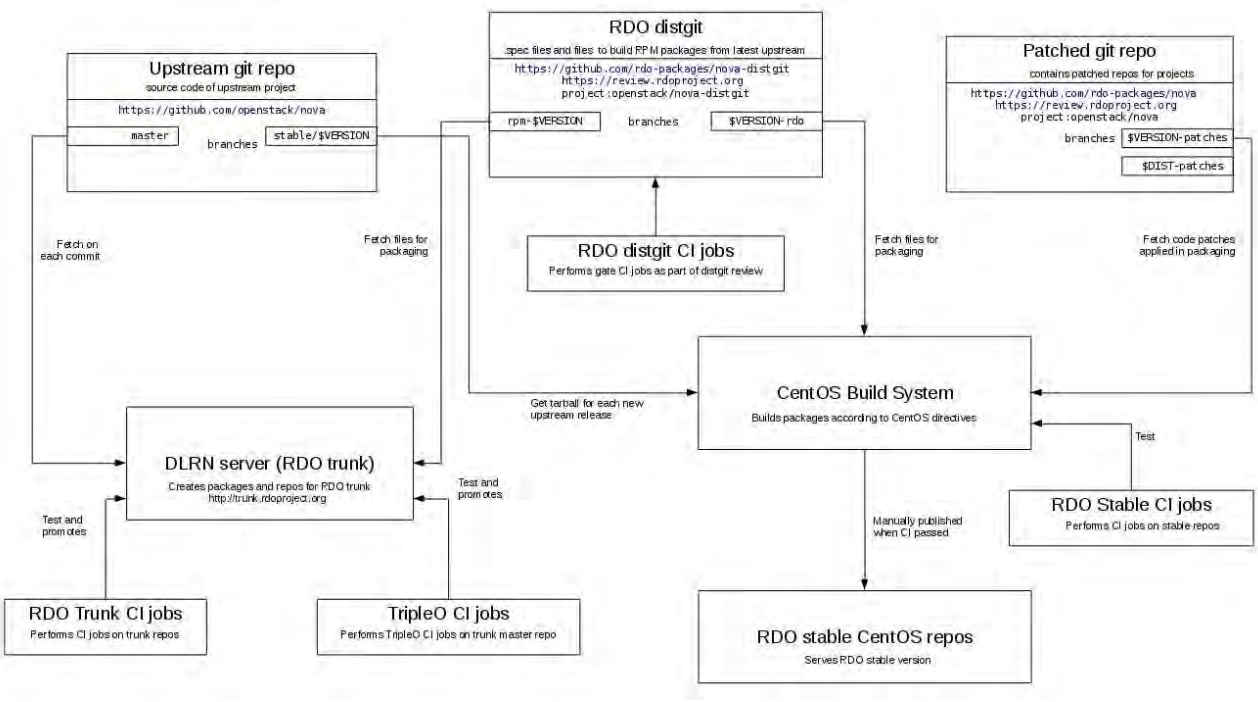


Figure 43: RDO packaging process

While this section does not intend to provide a comprehensive walk-through over the packaging process, we'd like to highlight two main different sections.

distgit - where the .spec file lives

distgit is a git repository which contains .spec file used for building a RPM package. It also contains other files needed for building source RPM such as patches to apply, init scripts etc.

RDO packages' distgit repos are hosted on review.rdoproject.org and follow $PROJECT-distgit naming.

DLRN

DLRN is a tool used to build RPM packages on each commit merged in a set of configurable git repositories. DLRN uses rdoinfo to retrieve the metadata and repositories associated with each project in RDO (code and distgit) and mock to carry out the actual build in an isolated environment.

DLRN is used to build the packages in RDO Trunk repositories that are available from http://trunk.rdoproject.org.

NVR for packages generated by DLRN follows some rules:

- Version is set to MAJOR.MINOR.PATCH of the next upstream version.
- Release is 0.<timestamp>.<short commit hash>

For example openstack-neutron-8.1.1-0.20160531171125.ddfe09c.el7.centos.noarch.rpm.

With the usage of this tools, we make sure that there are always updated packages available for consumption coming from Kuryr side.

Upstream CI

From the upstream side of things, we do not use packages but source code. The workflow used follows this diagram:

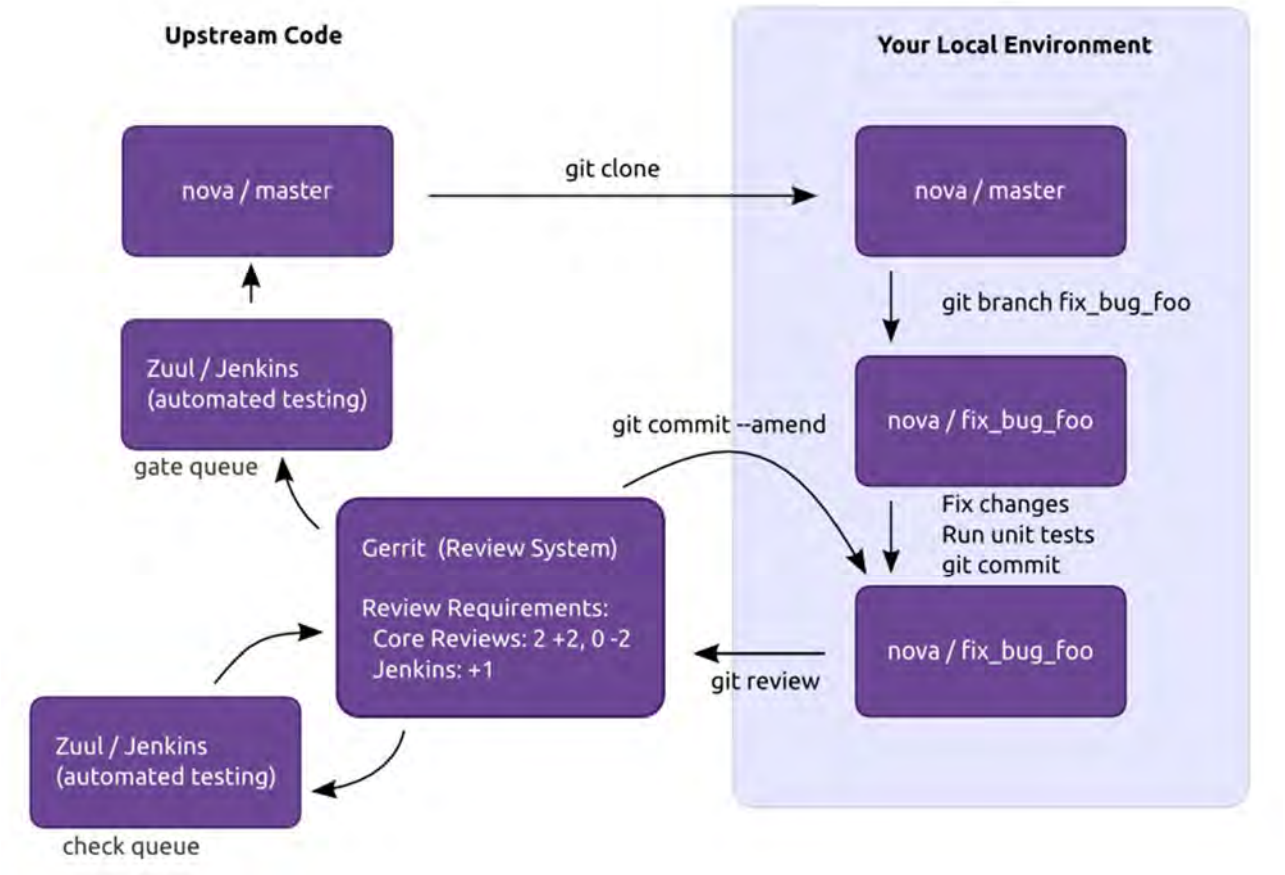


Figure 44: Upstream CI workflow

Once that you have cloned the repo and worked on your patch, it goes through an extensive testing system called Zuul.

Zuul is a program that drives continuous integration, delivery, and deployment systems with a focus on project gating and interrelated projects.

Zuul's configuration is organized around the concept of a *pipeline*. In Zuul, a pipeline encompasses a workflow process which can be applied to one or more projects. For instance, a "check" pipeline might describe the actions which should cause newly proposed changes to projects to be tested. A "gate" pipeline might implement the automation to merge changes to projects only if their tests pass. A "post" pipeline might update published documentation for a project when changes land.

While Zuul focuses on the infrastructure, the tests that are being run into OpenStack's pipelines for Kuryr are based into two different projects.

- Tempest: Is the functional testing framework for OpenStack.
- Kuryr-Tempest-Plugin https://github.com/openstack/kuryr-tempest-plugin:
    - We have developed a new upstream depository dedicated to the Integration of kuryr testing into upstream OpenStack's Tempest framework. For that, we have been focusing on pod to vm networking tests, which cover the functionality needed for all the Superfluidity's scenes.

A3.2 Mistral Orchestration

There is no doubt that workflows are a key ingredient for cloud automation, and automation is a key element in orchestration. However, when looking at an end-to-end, Telco grade, full NFV deployment over multiple distributed sites, orchestration starts becoming trickier.

Using TOSCA as the main DSL for the Network Service Descriptor provides a hierarchical view of services, components and their relationships, which is decoupled from the underlying VIM/NFVI. When combining TOSCA interfaces and a Mistral workflow, a full network service lifecycle can be achieved.

Mistral is an OpenStack workflow service. Most life cycle management (LCM) processes consist of multiple distinct interconnected steps that need to be executed in a particular order in a distributed environment. One can describe such LCM process as a set of tasks and task relations and upload such description to Mistral so that it takes care of state management, correct execution order, parallelism, synchronization and high availability. Mistral also provides flexible task scheduling so that we can run a process according to a specified schedule (i.e. every Sunday at 4.00pm) instead of running it immediately.

Although Mistral is quite generic it is built to become a natural part of OpenStack ecosystem. Out of the box Mistral provides "openstack" action pack for using functionality provided by other OpenStack services like Nova, Neutron or Heat. Mistral workflows can also be run from Murano PL.

To fulfil the vision of Superfluidity, and develop a telco grade orchestration based on Mistral, we identified several gaps that we addressed and submitted to the OpenStack Newton and Ocata releases. Specifically, (i) we addressed the performance and stability of Mistral, making it ~100 times faster, (ii) we extended the scope of Mistral to support multi VIM as envisioned in Superfluidity architecture, (iii) improved its reporting and event notification engine, and (iv) improved its usability.

In more details, to improve Mistral performance and reduce the LCM operation time, we shortened the database transactions, optimized the databased queries, and applied caching techniques.

Mistral, originally, was configured to execute workflows on a specific cloud VIM. In order to execute workflows on a different VIM, one needed to start a new Mistral instance. To allow a more fluent support for the distributed architecture of Superfluidity, we extended Mistral's workflow execution

parameters to make it possible to target a specific cloud without modifying the configuration of the Mistral service.

Additionally, we improved the reporting and tracking mechanisms in Mistral, e.g., adding an endpoint to track action/workflow executions belong to a certain task.

Finally, to improve usability we added a Mistral dashboard in OpenStack Horizon, as well as developed UI for better experience with writing custom actions.

A3.3 OSM

After an evaluation process of different MANO tools (see section 3.2), the OSM was selected to implement the MEC Orchestrator (MEO) component. Although OSM is built for NFV scenarios (to materialize the NFVO and VNFM components), we performed a work to adapt it to the MEC components.

There are a few differences among NFV and MEC. In the NFV world there are mainly two layers of entities: VNFs and NSs. In the MEC world there is only a single entity: the MEC Applications (MEC Apps). For this reason, we decided to implement MEC Apps as VNFs, creating additionally a NS with a single VNF inside. The reason for doing that is because the OSM does not allow the deployment of VNFs, but only NSs. So this is a simple workaround we found to overtake this difference. Considering this, in order to onboard an MEC Apps, we basically need to create 2 separated descriptors: a VNFD and a NSD.

Using OSM, the on-boarding process is dealt by the Riftware (NSO) component. The MEC App (as NS) is stored at the NS Catalogue as depicted in the Figure below.

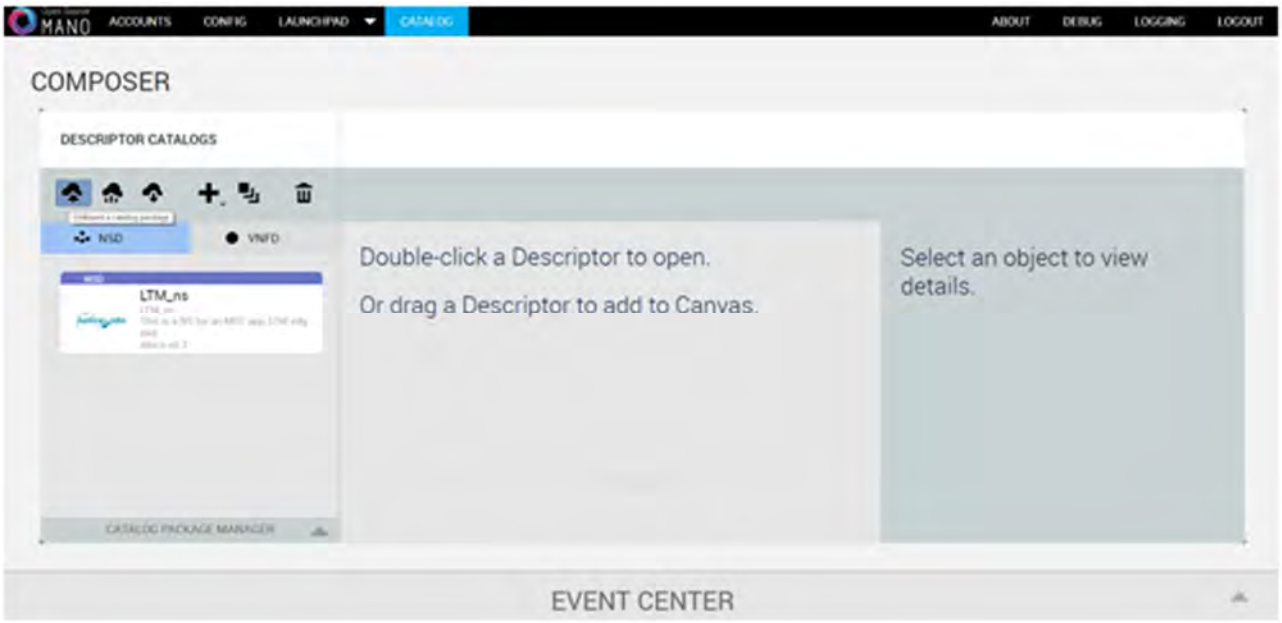


Figure 45: Onboarding of MEC Apps on OSM.

The VNF Catalogue stores the embedded MEC App VNF. Although the MEC App Descriptors and the NFV Descriptors contain different directives, they are very similar, allowing us to reuse the NFV ones. However, in the future we will need to extend it with MEC specific directives, in order to accommodate the requirements defined by MEC. The on-boarding process is performed through a file with the format described below.

The on-boarding process is very simple. It basically requires the selection of the type of descriptor to on-board (VNFD/NSD), and upload a tar.gz file containing the descriptor and other related data. The CSAR (Cloud Service Archive) is a format defined by TOSCA/NFV to upload all the material related to a VNF and NS, e.g. descriptors, lifecycle scripts, etc.

The OSM OpenMano (RO) component is responsible to interact with the VIM and can be mapped directly to MEC environments. As OSM is multi-VIM, it allows us to select the VIM where the MEC App is deployed. In the instantiate process the user can choose one of the available VIMs, previously configured. In MEC environments, assuming a VIM per edge, this permits a single orchestration tool to manage the multiple edges.

The OSM Juju component is used as configuration tool, in order to run all the scripting required for the life cycle management. Juju uses the charm concept to store and run sets of scripts associated to the configurations required for each of the lifecycle stage. Scripts can be written in any language. Juju allows the use of a large number of public charms available (more than 300) for some services. It also allows the user to create new charms, which can also use internally other charms. For example, in our case, the charm we created is composed by the charm ‘basic’, ‘sshproxy’ (ssh capabilities) and ‘vnfproxy’ (common VNF functionality).

The MEC Apps lifecycle management can be performed using the so-called charms hooks associated with events. There are several predefined events that each application goes through: install, configure, start, upgrade and stop. To perform configurations, when those events occur, Juju triggers a charm that can respond by pointing to executable files (hooks); then, Juju executes the specific hook for the appropriate event. For example, in our case, we use the events ‘start’ and ‘stop’ to interact with our MEC App. In the start event we provision the MEC App and the TOF piece with traffic offloading rules; and in the stop event we remove the TOF rules.

Charm can receive inputs from the VNFD, if they are configured in the VNFD and in Juju. For example, we use this mechanism to pass to Juju credentials for ssh and some ip/port data.

Juju charms need to be compiled before they are uploaded to Juju. To create a charm, you just need to install the Juju package, and execute one command to create the basic charm layer. Charms are built using the following directory tree.

```
├── config.yaml
├── icon.svg
├── layer.yaml
├── metadata.yaml
├── reactive
│   └── ltm.py
├── README.ex
└── tests
    ├── 00-setup
    └── 10-deploy
```

The *layer.yaml* file specifies the charms available to use; the *metadata.yaml* file describes what the charm does; the *config.yaml* file specifies the inputs received from VNFD. Inside the *reactive* folder are located all the scripts that need to be invoked from the main file (*ltm.py* in the example above), in the correspondent hook event, or putting the code directly within the file (in case of python language). If you need other hooks than the defaults explained before, just create a folder *actions* with the file who get the script from reactive folder, and create an *action* file to specify the action and the parameters required.

The charm is then built and included in the CSAR file. The OSM has a particular project called 'descriptor-packages' to help on that. Using a simple command, it produces the CSAR file (tar.gz) with the following structure. This file is used to on-board the MEC App as described above.

```
ltm_vnf
├── charms
│   └── ltm
├── checksums.txt
├── icons
├── images
├── ltm_vnfd.yaml
├── README
└── scripts
```

A3.4 ManageIQ

ManageIQ is a management project that enables managing containers, virtual machines, networks and storage from a single platform, connecting and managing different existing clouds: OpenStack, Amazon EC2, Azure, Google Compute Engine, VMware, Kubernetes, OpenShift, …

The main reason for choosing ManageIQ as an NFVO is due to being able to work with both VMs and Container providers. However, after feature analysis, we discover a few gaps that needed to be address for the Superfluidity purposes

A3.4.1 Ansible execution support

The main concern about the existing NFVO tools was the lack of applications life-cycle management actions for container. Therefore we worked on an extension to fix this gap on ManageIQ. The proposed extension is based on supporting ansible playbook execution within ManageIQ. Therefore, any action (as ansible is agent-less) can be triggered in any of the providers.

One of the more powerful capabilities of ManageIQ is the self-service. It allows an administrator to maintain a catalog of requests that can be ordered by regular users, for example, to provision a single VM, a container or an application stack. It starts with an administrator creating a "service bundle," which is a collection of "service items". Each service item is an action that ManageIQ knows how to create/handle. The order in which items in a bundle are provisioned is specified by the administrator, in what is known as the state machine. Services typically require some amount of input. For example, if the request is to provision a VM, then a typical question would be the memory and disk size. This information can be requested from the user through a dialog, which can be created using ManageIQ's built-in dialog editor.

Once the service bundle and the dialog are created, they need to be associated with an "entry point" in the ManageIQ workflow engine (called "Automate"). The entry point defines the process to provision the bundle. With the bundle definition, dialog, and entry point, the request can be published in a service catalog, which then enables users to order the service.

Given the above, Self-service is good for both the administrator and the end user, and it is the capability we have used to add support to run Ansible playbook at Containers deployments, in our case Kubernetes and/or OpenShift -- note it also enables running them at both baremetal or OpenStack deployments. By enabling the execution of playbooks located at git repositories, we provide an easy way to onboard new lifecycle management actions. It is really easy to update, include, or extend different playbooks that take care of different VMs/Containers/Apps lifecycle actions, such as creation, termination, scaling or any other configuration actions. It is as simple as using the common *git commit* and *git push*.

In our ManageIQ catalog item, we have defined a dialog that takes, on the one hand, the ansible playbook to execute (usually a site.yml file), and on the other hand the provider where you want to execute it -- being the only requirement to have ansible installed at the ManageIQ appliance and ssh-passwordless towards the providers master nodes.

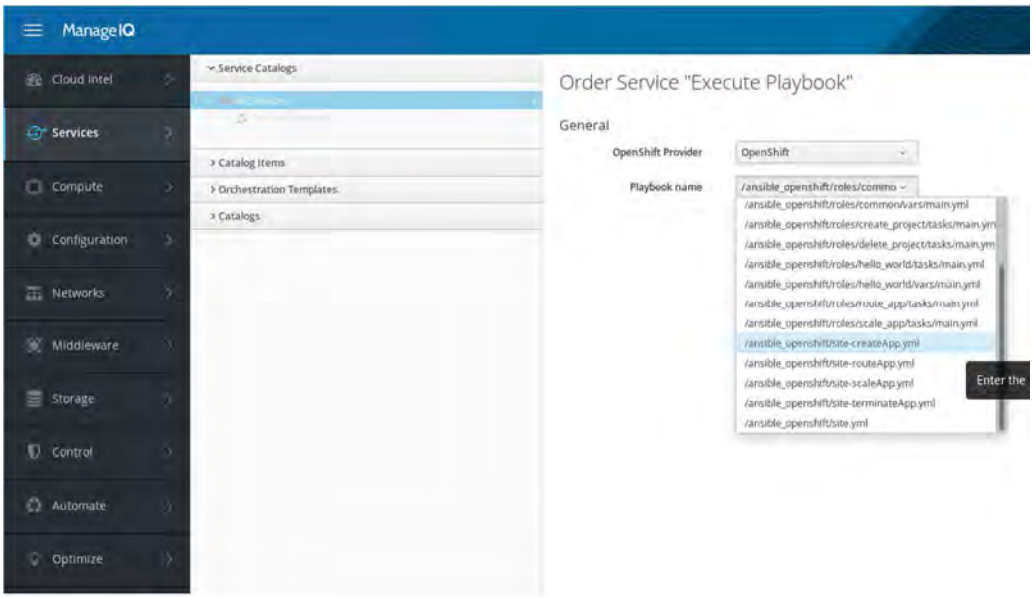


Figure 46: Ansible playbook execution

In the figure above, we can see there are different playbooks to create, delete, scale and scale the Application for a given github repository. Note there may be several of them, as many as cloned in the specified directory in the ManageIQ appliance.

As regards to the service bundle, it contains a series of methods that are organized in a state machine that performs an initial action of reading, checking and processing the input from the user dialogs (named parse_dialog). Then, there is a second step where playbook execution is actually triggered (run_job).

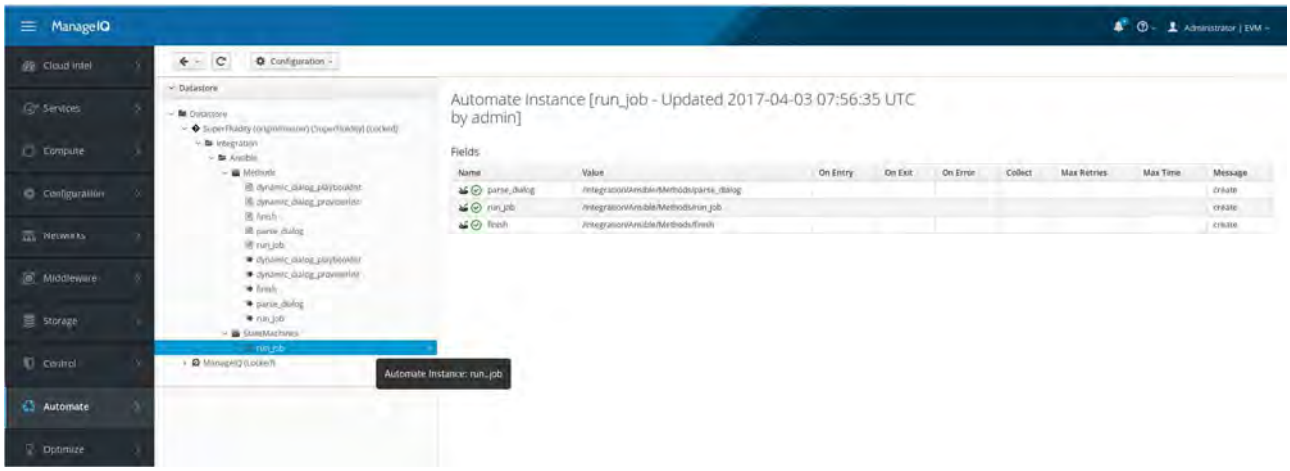


Figure 47: ManageIQ State Machine

A3.4.2 Multi-site support

Building upon the previous point, we added extra support to execute those playbooks across different sites as there was a need for synchronization about tasks being performed at different sites. As an example, when deploying the CRAN, MEC, EPC components, they can share some information or even must know about each other. Thus, it may be convenient to have then in the same playbook and control when the steps take place, e.g., deploying the EPC before the CRAN (or the other way around)

Consequently, we have worked at adding this support for multi-site deployment from ManageIQ. Up to now, with ManageIQ you can manage different providers (i.e., different sites), but not execute a set of actions that involves several of them at the same time. We have added to the ansible support at ManageIQ the option to include a host file where you can specify the different sites where the playbook need to be executed. This way, you can directly specify on the playbooks where each task is meant to be executed: e.g., the EPC related tasks in the EPC cloud (which for instance can be a VM running on an OpenStack deployment), and the CRAN tasks at the CRAN server (which can be a kubernetes cluster or even just a linux server).

Therefore, to make use of this new functionality the only need from the user point of view is to include a host file defining all the hosts (be it OpenStack endpoints, kubernetes master APIs, or single servers) into their git repository and annotate the tasks with the host where it need to be executed. Then, at the ManageIQ UI, the multisite deployment flag needs to be enabled, and the proper host file selected - beside the desired playbook to be executed.

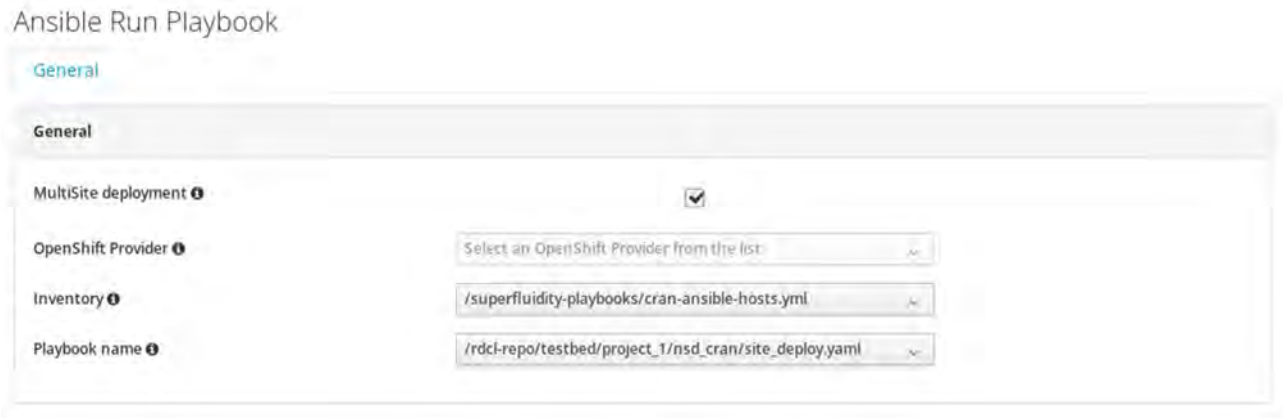


Figure 48: Muti-Site playbook execution

A3.5 Load Balancing as a Service

As identified in previous deliverables (Deliverable D2.1), many of the use cases of interest depend on common load balancing capabilities, to fulfil the scalability and high-availability requirements. The requirements of Load Balancer, as a functional block, are analyzed further in Deliverable D2.2.

OpenStack offers a few open source options for implementing the infrastructure load balancing capability, most notably HAProxy (upstream open source project) and Octavia (OpenStack project). Deliverable D5.3 provides more information on these options.

Something we wanted to evaluate, as part of Superfluidity, is how they compare with commercial, carrier-grade, options. Citrix NetScaler ADC [20] is such an option. The figure below illustrates how NetScaler ADC fits into the NFV architecture and how it integrates with the MANO elements, namely the VIM (OpenStack) and the SDN Controller.

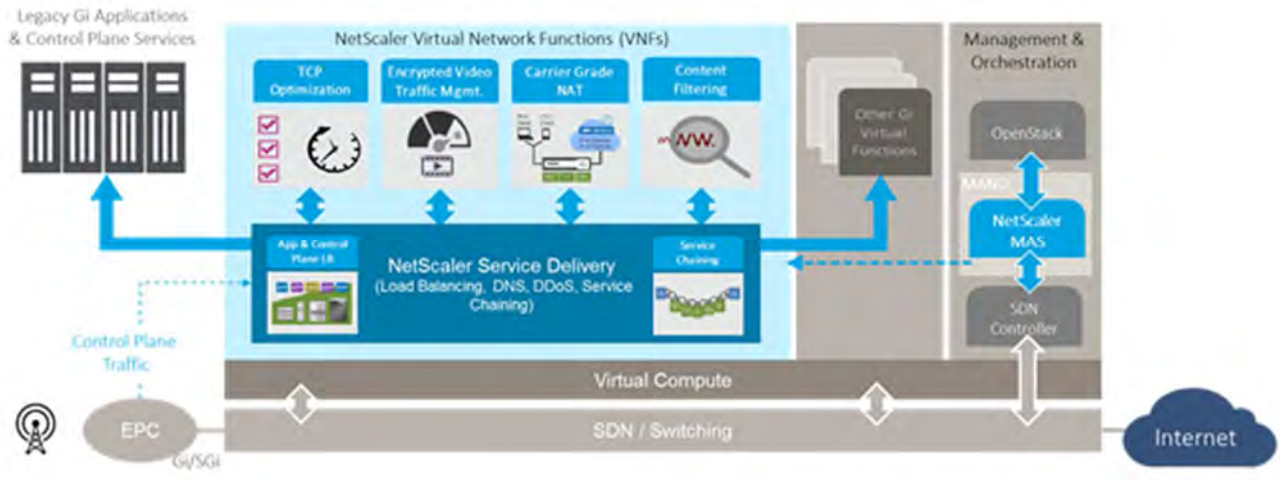


Figure 49: NetScaler ADC at NFV architecture

NetScaler MAS [21] acts as an Element Manager for NetScaler ADC. It integrates using standard APIs with OpenStack and supported SDN Controllers, translating them to the RESTful APIs supported by NetScaler ADC [22]. More information on the integration between NetScaler MAS and OpenStack can be found at [23]

The evaluation we have performed as part of Superfluidity consists of two parts:

- Comparing the capabilities of open source options (initially HAProxy) versus NetScaler ADC
- Validating the integration of NetScaler ADC, NetScaler MAS and OpenStack in the Reference NFV Lab we have deployed to support the Task 5.3 activities.

From a load balancing capabilities standpoint, the key difference we identified is the availability of a far broader range of:

- Load Balancing Algorithms: [24]
- Persistence Types: [25]

From an experimentation standpoint, we have successfully deployed and validated the above integration against the versions of OpenStack that were released during the lifetime of the Superfluidity project so far, namely Liberty, Mitaka, Newton, Ocata and Pike.

As part of this activity, we have leveraged enhancements in both the NetScaler LBaaS driver (OpenStack Neutron/LBaaS project: [26], [27]), as well as NetScaler MAS and ADC (commercial software). The VPX (virtual) edition of NetScaler ADC was used for that purpose, specifically the 12.0 release, and the respective NetScaler MAS release. The above outcomes were integrated with the Superfluidity platform as part of WP7 activities.

Finally, for perfect alignment with the support for the execution of Ansible playbooks by the NFVO (ManageIQ, see A3.4), we have implemented two extensive Ansible modules:

- netscaler_server: automates and manages the configuration of NetScaler servers (http://docs.ansible.com/ansible/latest/netscaler_server_module.html)
- netscaler_service: automates creation and manages the configuration of NetScaler services (http://docs.ansible.com/ansible/latest/netscaler_service_module.html

The implementation of the above Ansible modules for NetScaler was contributed as open source (https://github.com/citrix/netscaler-ansible-modules) and relevant developer documentation was published (https://developer-docs.citrix.com/projects/netscaler-ansible-modules/en/latest/).

A3.6 Service Function Chaining

As identified in the requirements analysis (Deliverable D2.1), many complex use cases consist of compositions of in-network services, which require Service Function Chaining (SFC) to materialize. The components of the IETF SFC architecture, namely the Service Functions (SF), SF Forwarder, Network Overlay, SFC Proxy and SFC Classifier are analysed further in Deliverable D2.2.

Based on our monitoring, the open source projects above have been making more progress since:

- OpenStack SFC support (https://docs.openstack.org/ocata/networking-guide/config-sfc.html) is maturing, adding new capabilities (https://docs.openstack.org/networking-sfc/latest/): New SFC Driver for OVN (https://docs.openstack.org/networking-sfc/latest/contributor/sfc_ovn_driver.html, http://openvswitch.org/support/dist-docs/ovn-architecture.7.html), support for Symmetric Port Chains, Service Function Tap for Port Chains, etc.
- The OPNFV OVS project introduced “NSH for VXLAN” support in the OPNFV Colorado release. However, this required a special release of OVS (which included Yi Yang’s OVS NSH patches: https://github.com/yyang13/ovs_nsh_patches). Actually, it seems this is still the case in the OPNFV latest (Euphrates) release.

- The OPNFV Colorado, Danube and Euphrates versions support the SFC os-odl_l2-sfc-noha, os-odl_l2-sfc-ha scenarios, i.e. OpenStack + OVS + OpenDaylight + SFC, at least for the Apex and Fuel installers. The version of OpenDaylight OPNFV Euphrates is dependent on is Nitrogen SR1 (SR2 is the latest). The latest documentation of the OPNFV SFC submodule: http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/index.html
- As foreseen in section 3.1.1, the VNFM has to be involved as well. The scenarios implemented by OPNFV SFC utilize OpenStack Tacker for that purpose (see http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/architecture.html#vnf-manager). More information on how this is implemented: https://specs.openstack.org/openstack/tacker-specs/specs/newton/tacker-networking-sfc.html
- Lastly, when using NSH with VXLAN tunnels, it is important that the VXLAN tunnel is terminated in the SF VM. This allows the SF to see the NSH header, allowing it to decrement the NSI and also to use NSH metadata. When using VXLAN with OpenStack, the tunnels are not terminated in the SF VM, but in the OVS bridge. In the OPNFV Danube release, this required a workaround (http://docs.opnfv.org/en/stable-danube/submodules/sfc/docs/development/design/architecture.html#ovs-nsh-patch-workaround), but this is no longer the case in the Euphrates release (http://docs.opnfv.org/en/stable-euphrates/submodules/sfc/docs/development/design/architecture.html).

As clear from the above, the current situation with SFC is unnecessarily complex, due to unfulfilled upstream project dependencies that require special patches and workarounds. In anticipation of improvements, we invested in implementing NSH in NetScaler. This is in NetScaler 11.1 build 47.14 (June 2016), or later releases: https://docs.citrix.com/content/dam/docs/en-us/netscaler/11-1/release-notes/NS_11_1_53_11.html (#593459).

As a result, a NetScaler ADC (virtual) appliance can now play the role of the Service Function in the SFC architecture. The NetScaler instance receives packets with Network Service headers and, upon performing the service, modifies the NSH bits in the response packet to indicate that the service has been performed. In that role, the appliance supports symmetric service chaining with specific features, for example, INAT, TCP and UDP load balancing services, and routing. Restrictions of the initial release: Only VXLAN-GPE is supported as the tunneling protocol. IPv6 is not yet supported.

A3.6.1 Service Function Chaining with IPv6 Segment Routing (SRv6)

The SFC based on NSH still suffers from limited maturity of the implementations, as outlined in the previous section. Hopefully, this will improve in the future. Anyway, there is also an intrinsic shortcoming of the traditional SFC/NSH approach, because it typically requires stateful operations in the SFC proxies and SFC classifiers along the path of the Service Chains. Handling thousands of Service Chains with this stateful approach could lead to scalability concerns. In addition to these potential scalability issues, the need of setting up the required state information in the intermediate nodes is not optimal considering the requirement to setup and configure Service Chains in very short time.

For the above reasons, we have also considered a "disruptive" solution based on IPv6 Segment Routing (SRv6). The SRv6 technology works by carrying a *segment list* (a list of IPv6 addresses) in the IPv6 header. SRv6 can be used as underlay or overlay technology and it extends the capability of the networking layer, allowing to put "network programs" in the packet headers [28]. SRv6 can become a key enabler for the Future Internet Architecture, where a high level of flexibility is a required. SRv6 has been recently introduced in Linux Networking (with 4.10 kernel) and its standardization is progressing in IETF [29] [34], where it has been recently proposed for the user plane of mobile networks [30]. Consequently, the 3GPP has recently started a study item that will consider SRv6 as a candidate for user plane in 5G [31].

In the context of Superfluidity, we have explored this technology because it offers the possibility to configure a Service Chain only operating at the edge of the network, with no need to interact with the nodes in the middle of the chain. We have contributed to an Internet draft on using Segment Routing for Service Function Chaining [33] and we have participated to the realization of a proof of concept using the features of the Linux OS [32]. It is evident that, although being a very promising technology, there is still work to be done integrate it in production scenarios using the reference NFV technologies (e.g. OpenStack at the VIM level and the various NFV Orchestrators). For this reason, the said SFC/SRv6 proof of concept [32] has not been integrated in the overall Superfluidity demonstrator.

A3.7 OpenShift-Ansible

To provide the flexibility required by Superfluidity as well as future 5G deployments, we have worked on a set of ansible playbooks that allows to easily consume all the above features, as well as to better integrate with current cloud environments. This set of playbooks (https://github.com/openshift/openshift-ansible) allows to install OpenShift on top of different cloud infrastructures: OpenStack, Amazon, Azure, and Google Compute Engine.

Our efforts within the Superfluidity project has focused on the OpenStack provider as this is the one we suggested for our architecture:

https://github.com/openshift/openshift-ansible/tree/master/playbooks/openstack

Our main contributions are related to the implementation of kuryr roles inside the openshift-ansible playbooks so that kuryr components can be deployed on the OpenShift installation on OpenStack,

using the above mentioned features (ports-pool, containerized and cni-split). These new roles enable the installation of OpenShift in an existing OpenStack deployment with kuryr configured by just executing one playbook.

A3.7.1 ManageIQ integration

It must be noted that by combining openshift-ansible playbooks with the support at ManageIQ to execute ansible playbooks, we are now able to also easily deploy OpenShift clusters in existing OpenStack deployments (on top of VMs) with kuryr support from ManageIQ.

More work has been done at the integration between openshift-ansible and ManageIQ for an even more simple user experience. Thanks to this integration a tenant can simply ask for an OpenShift deployment from ManageIQ UI (e.g., ask for a deployment with 1 master node, 1 infra node and 5 worker nodes) and it will be automatically deployed by executing the openshift-ansible playbook with the configured parameters on the selected OpenStack provider, i.e., on top of OpenStack VMs , giving the user a functionality similar to have: OpenShift as a Service.

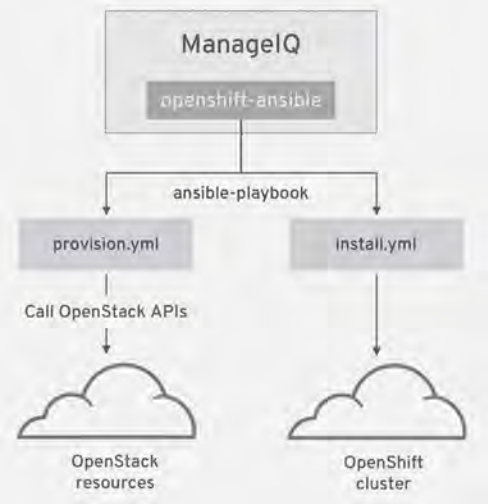


Figure 50: OpenShift-Ansible integration

A3.7.2 Baremetal Support

Finally, in addition to the previous support, we have also extended the openshift-ansible playbooks to also support provisioning baremetal nodes. This adds the flexibility of having an OpenShift/Kubernetes installation that runs both on top of OpenStack VMs as well as on baremetal nodes. This is specially relevant for NVF deployments where some components may require to run (containerized) on baremetal nodes for increase performance, but still being on OpenStack neutron networks so that they can talk to other VMs or nested containers transparently.

As presented in next figure, these modifications involve creating different type of application nodes (where the pods run) where they can be either Virtual Machines created on OpenStack by using the Nova component, or physical servers (a.k.a. baremetal node) created on OpenStack by using the Ironic component.

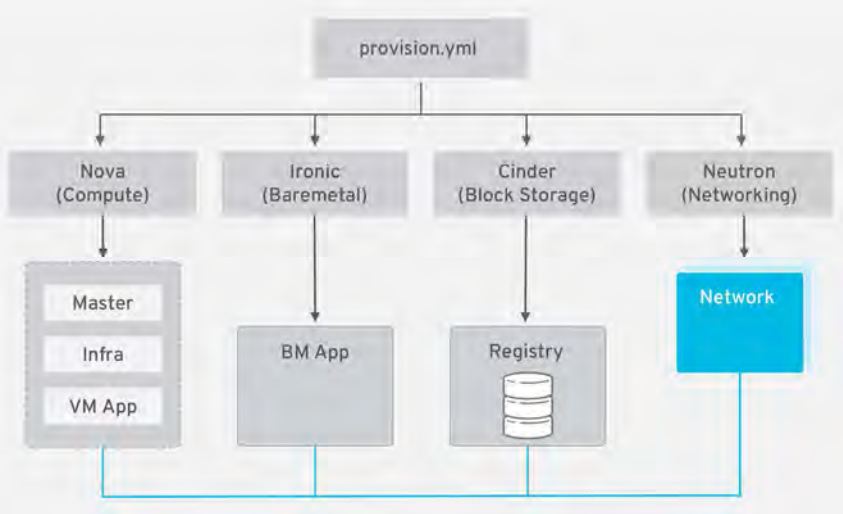


Figure 51: Provision Interactions Overview

To fully enable this work, changes were not only needed at the openshift-ansible side, but also at Kuryr. We are working on adding support for multi-pools:

- https://bugs.launchpad.net/kuryr-kubernetes/+bug/1747406
- https://review.openstack.org/#/c/528345

The main reason behind this proposal is the existence of different type of nodes that will need to use different type of neutron drivers (i.e., baremetal and nested-vlan pod vif drivers). This forces us to make the kuryr-controller able to handle different type of pod vif drivers, and in turn having different pool drivers to also speed up the creation of pods regardless of the pod vif used. To do that, the multi-pool support enables different pool drivers (and pod vifs) at the same time, and it reads the driver that it needs to use depending on the node where the pod is being deployed.

In addition to this modification, we extended the ManageIQ support for openshift-ansible playbooks to also ask for the number of VMs and baremetal that the user would like to have for its OpenShift deployment. A Proof-of-Concept of these modifications was presented at DevConf 2018 in Brno as a keynote:

- https://www.youtube.com/watch?v=BloeOXBsETo

A3.8 RDCL 3D

The vision of the Superfluidity project is to orchestrate functions dynamically over and across heterogeneous environments. It is not possible to consider a single language and tool to cover the diversity of the heterogeneous environments. In the Superfluidity architecture, the different languages for the description, composition and orchestration of services and service components are referred to as RDCL (RFB Description and Composition Languages). The project has designed and developed a tool called RDCL 3D (Design, Deploy and Direct) to assist in the editing of the descriptors of services / service components and in the interaction with different types of orchestration tools. The RDCL 3D tool is not focused on a specific data model / description language but it is meant as a framework that can be specialized for any data model / description language.

RDCL 3D offers a web GUI that allows visualizing and editing the descriptors of components and network services both textually and graphically. A visualized network service designer can create new descriptors or upload existing ones for visualization, editing conversion or validation. The created descriptors can be stored online, shared with other users or downloaded in textual format to be used with other tools. In particular, these descriptors can be used for the deployment and operational management of NFV services and components.

Figure presents a screenshot of the RDCL 3D web GUI, where users can perform intuitive drag-and-drop operations on the graph representation of RDCLs. The source code of RDCL 3D [10] is released under the Apache 2.0 Open Source license, to facilitate its uptake by research and industrial communities [11]. In addition, a live deployment of the system can be found online at: http://rdcl-demo.netgroup.uniroma2.it, where users can login using a guest account and explore all RDCL 3D capabilities. It is also possible to request an account that allows saving and retrieving projects and related descriptor files across different sessions.

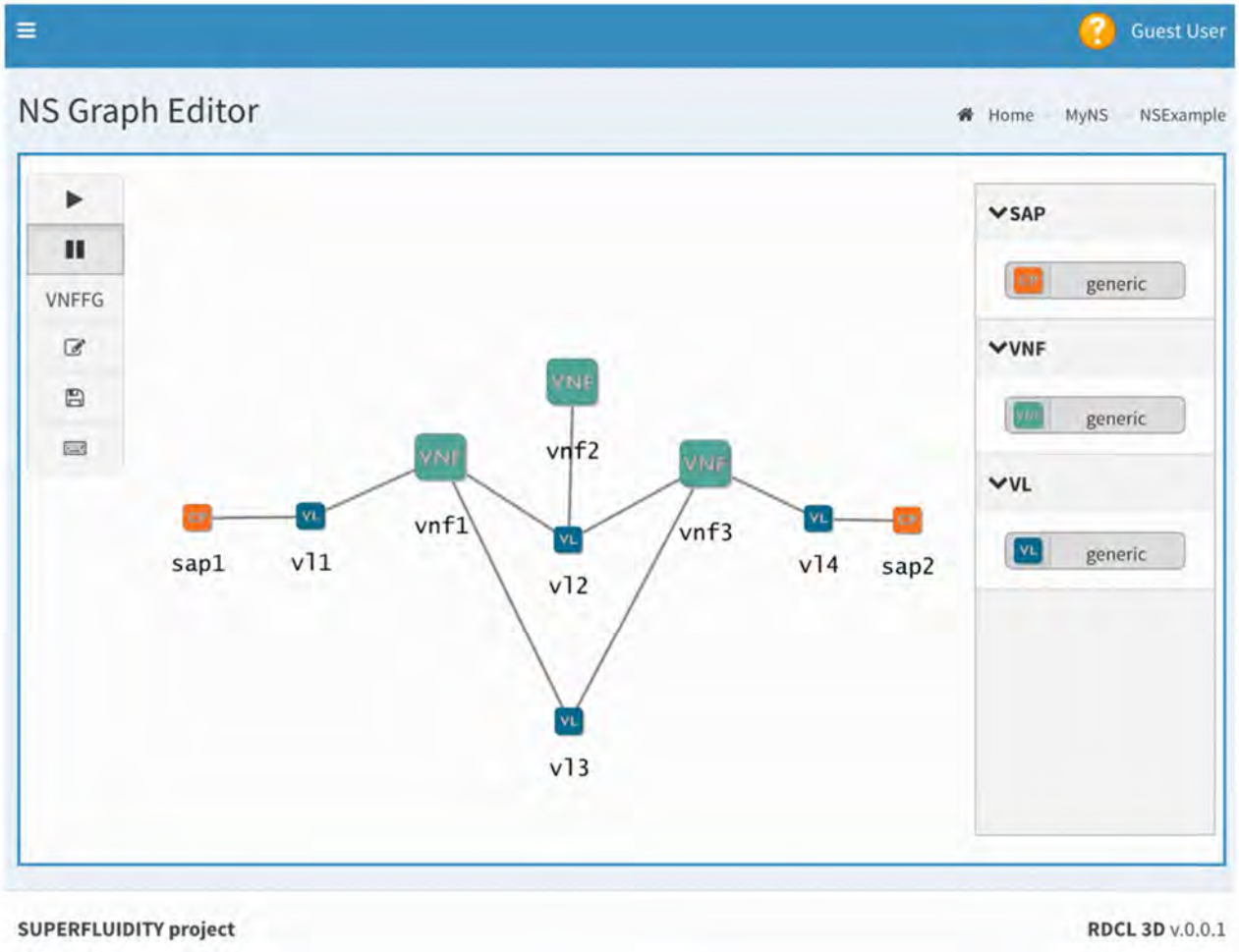


Figure 52: Network Service design using the RDCL 3D GUI

Currently the RDCL 3D framework supports the models defined by the latest ETSI NFV ISG specifications [12] [13], the TOSCA Simple Profile for NFV [14], the TOSCA Simple Profile in YAML [15], the network elements described in the Click configuration language [16] and the Open Source Mano information model [17]. However, RDCL 3D is designed for extensibility in order to support new models or combine existing ones.

With reference to the ETSI MANO architecture, RDCL 3D can play different roles, which correspond to different usage scenarios:

1. RDCL 3D can be used as a standalone tool to edit the NS and VNF descriptors, as shown in <1> in Figure 53. This approach is currently implemented in the demo for the ETSI and TOSCA models. The produced descriptor files can be manually retrieved and provided to an Orchestrator (NFVO).
2. RDCL 3D can support the direct interaction with programmatic APIs of external Orchestrators (<2> in Figure 53) by including the logic for such interaction in an Agent module. In this case,

the Agent module receives instructions from the RDCL 3D web GUI, hands the descriptor files to the external Orchestrator, and provides feedback to the user on the GUI. Note that this scenario can be extended when there is the need to combine different orchestration platforms with different description languages, so that RDCL 3D can become a *meta-orchestrator*.

3. Another usage scenario that we have considered is to use the platform to build Orchestrator prototypes, so that RDCL 3D plays the role of the NFVO (<3>in Figure 53Figure 52). This is especially useful when one needs to explore new functionalities and it is easier to have a small stand-alone proof-of-concept implementation rather than integrating the new functionality into a fully-fledged NFVO.
4. A different usage of the proposed network is to be integrated as a library within Orchestrators that do not yet support a GUI as shown in <4> in Figure 53.

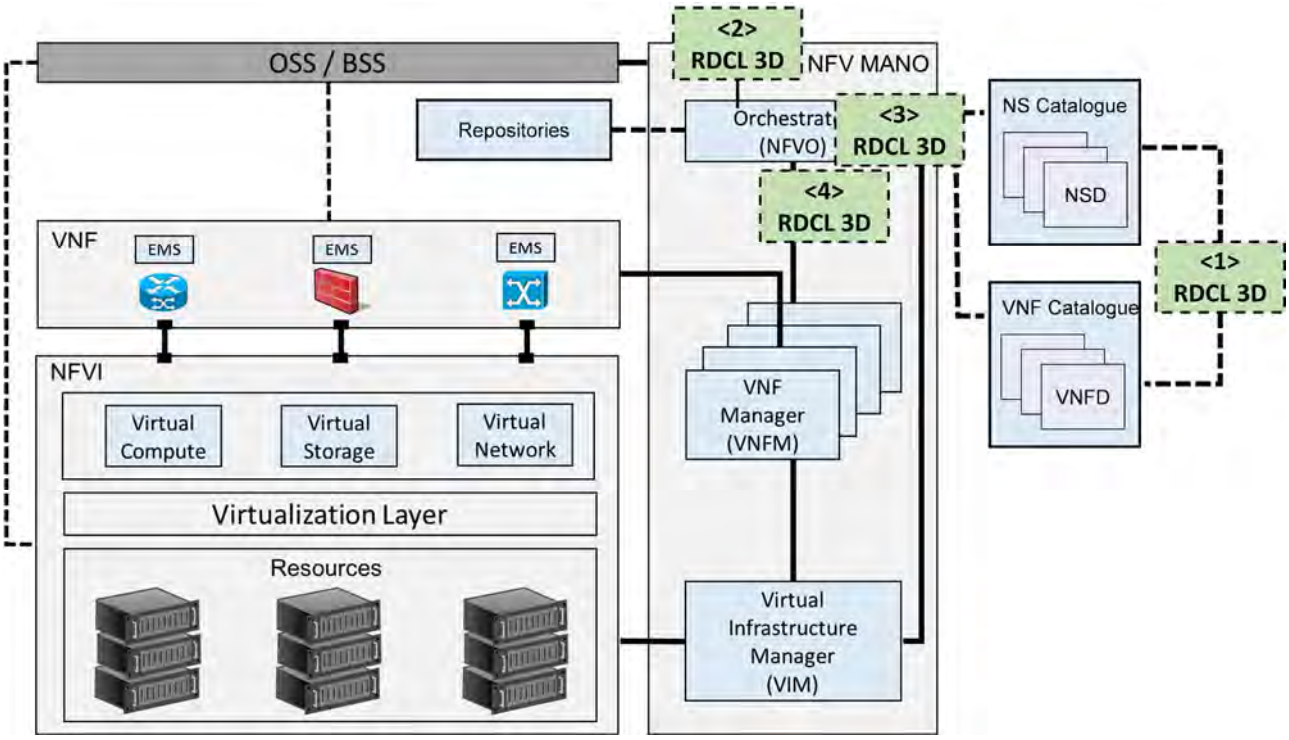


Figure 53: Positioning RDCL 3D in the ETSI MANO architecture

A3.8.1 Integration between RDCL 3D and ManageIQ

The integration between RDCL 3D and ManageIQ is implemented through the following steps:


- The user/network designer employs the RDCL 3D GUI to design or load a network service using the Superfluidity/ETSI language
- The network service is translated by an RDCL 3D agent into HEAT or Kubernetes templates, which are included in an Ansible playbook. The Ansible playbook contains the network service templates along with the commands (for OpenStack or OpenShift) to deploy them
- The RDCL 3D agent commits the Ansible playbook in a local git repository and pushes the changes to a remote git repository to which ManageIQ has access
- ManageIQ pulls the Ansible playbook from the git repository and executes the Ansible playbook.

A3.8.2 Software Architecture

RDCL 3D is a web application with backend and frontend components, as depicted in Figure The backend component running on a web server is based on the Python Django framework. It includes the persistence layer (database). The frontend component running in the web browser is developed in JavaScript and exploits the D3.js library. The platform is designed with a modular approach both in the backend and in the frontend, so that it can be easily extended to support new project types. Each project type can be seen as a "plugin" for the RDCL 3D framework, composed by a backend plugin (in Python) and a frontend plugin (in JavaScript).

Each user (e.g. a service designer or network administrator) can instantiate projects of the supported project types. The descriptor files for the projects are stored in the persistence layer in the backend. Predefined descriptor files are available for each project type (i.e. they represent examples of services or existing components that can be reused). A Data model for a project is created in the backend by parsing and validating the descriptor files (<1> in Figure 54). This process is specific for the project type, therefore it is performed by the specific plugin for the project type. The instance of the Data model contains all the information of a project instance i.e. all the information contained in the project descriptor files. The Data model is send to the frontend (<2> in Figure 54), where is it processed and filtered to produce the different graphical views on the browser GUI (<3> in Figure 54). The project descriptor files are also sent to the frontend. The operations on the GUI (e.g. adding or removing nodes and links, editing of the local descriptor files) are reflected on the local version of the Data model and sent to the backend when it is needed to update the information stored in the persistence layer (e.g. the descriptor files). When interacting with orchestrators which provide their own catalogue service (e.g. Open Source MANO), the backend is responsible for retrieving and then uploading descriptors to the orchestrators.

The backend can also optionally deploy the designed network services through an *RDCL 3D Agent*. An RDCL 3D Agent can deploy the designed network services by either interacting directly with the APIs

of an orchestrator or a VIM or by pushing the network service descriptors to a git repository to which the Orchestrator/VIM has access. Moreover, to allow the interaction with specific VIMs, the RDCL 3D Agent can also perform online translation between descriptor formats. The RDCL 3D Agent exposes REST APIs, which are invoked by the Deployment Handler plugin. Our RDCL 3D Agent implementations are based on node.js, but any technology which allows to expose HTTP REST APIs can be employed.

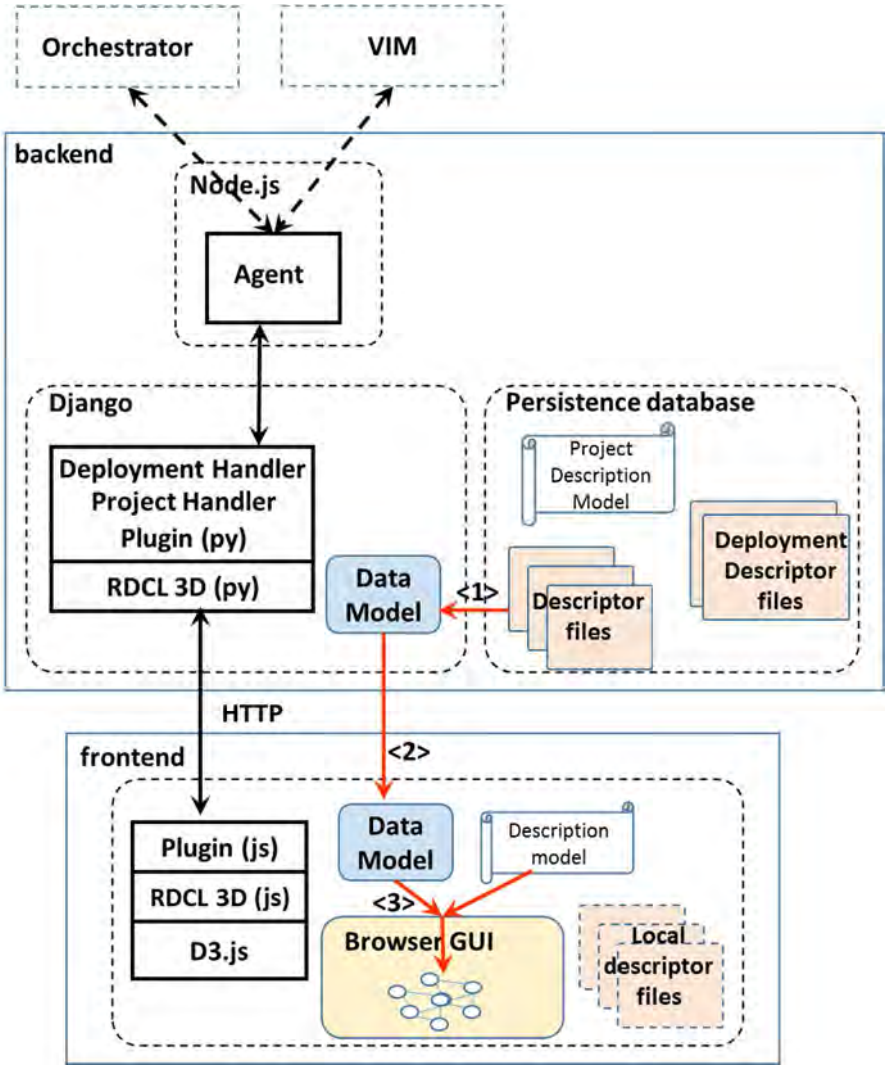


Figure 54: RDCL 3D Software Architecture

The operations performed by the frontend, like visualizing a view of the graph, adding/removing nodes and links, are dependent on the project type, so that they should be handled by the JavaScript plugins. We are able to minimize the code that needs to be developed in a plugin to support a project type by introducing a *description model* for a project type. The description model includes the types of nodes and links that are supported, their relationships, the constraints in their composition,

describes what are the different views of the projects and which nodes and links belongs to which view. The description model is expressed as a YAML file. By parsing it, the JavaScript frontend is able to perform most of the operations without the need of specific code for the project type. In order to handle the operations specific to the project type, the description model includes the possibility to associate operations to functions that are defined in the plugin. The definition of the structure of the description model and some examples are included in the *docs* folder in [5].

To simplify the integration of new project types, the framework includes a script that creates the skeletons of the Python and JavaScript plugins (respectively for the backend and the frontend) and of the description model. Starting from these skeletons, a developer adding the support of a new project will:

1. include the parser to translate the descriptor files into the Data model representation in the Python plugin (backend);
2. customize the description model, capturing all the relevant properties of the project type and identifying the operations that need to be processes in a specific way for the project type by the JavaScript plugin;
3. develop the project type specific processing operation in the JavaScript plugin (frontend);
4. optionally, develop a deployment handler and an RDCL 3D Agent to allow the direct deployment of network services.

Summarizing, RDCL 3D is a web framework for visualizing and editing services and components in NFV scenarios. RDCL 3D is not focused on a specific data model / description language. It is designed to facilitate the support and the integration of any model and language: i) it has a modular architecture in which a new project type can be added as a plugin; ii) a description model allows describing the structural properties of the project type, minimizing the need to develop code; iii) a script is used to generate the skeletons of the plugin and of the description model, to reduce the development effort.

A3.9 Optimization of Service Deployment Templates using a Service Characterisation Framework

A key focus area in Task 6.1 is the control framework which addresses resource allocation for virtualised network functions and services. Resource allocation is important in the context of “virtualisation” of network services as it plays a very important role in both service assurance and Total Cost of Ownership (TCO). It is important to allocate the right quantity and type of resources for service execution in order to support a required service level, which is usually measured against Service Level Objectives (SLOs). It is also equally important to prevent overprovisioning of resources


as this leads to higher TCO, due to unused resources and, in some cases, can result in service performance degradation.

In Cloud Computing environments, allocating resources to enable execution of a service is performed by an orchestrator (such as OpenStack Heat, ETSI Open Source Mano (OSM), etc.). In order to fulfil user requirements, the orchestrator takes as input a service descriptor, which specifies types and quantity of the resources for each service component (for instance, the number of vCPUs, the number and type of vNICs, the preference for enabling enhanced platform features, etc.). The requested resources are then allocated at deployment time through a virtual infrastructure manager (VIM).

The presence of enhanced platform features and their specific configuration options, such as core pinning, Linux kernel hugepages, SR-IOV enabled Network Card Interfaces (NICs), and so forth, can be exploited to further improve service performance, if required by the user. However this strictly depends on the availability of those resources within the infrastructure landscape and their relevance to a given service. The current approach for resource allocation is based on pre-defined deployment descriptors which are defined by the service vendors: the orchestrator takes them as an input and requires the instantiation of such a pre-defined configuration to deploy the service on the specific hardware platform at deployment time. This service descriptor is defined by the vendor on the basis of the specific infrastructure available during on-boarding tests and does not take into account all potential infrastructural differences (related to hardware and software configurations) within the service provider's production environment.

A3.9.1 Service On-Boarding Characterisation

The main focus of the service characterisation activities is to support automated characterisation of the relationship between service performance and resource allocation cost on a Network Function Virtualised Infrastructure (NFVI). Characterisation is carried out prior to initial placement of a new service into production, resulting in performance optimisation of the services based on the available hardware ingredients. For this reason, this approach is part of the pre-deployment characterisation of a service. In the context of Task 6.1, the design and implementation of an automation framework is envisioned in order to automate some aspects related to the generation of placement insights for an orchestrator. The primary goal therefore is to automatically define a set of rules that can be interpreted by an orchestrator in order to make intelligent decisions on the quantity and type of resources to be allocated to a service by a VIM.

Automation is key to determining the best composition of quantity and types of resources to be allocated to a service according to its required KPIs and SLOs and considering changes in operational conditions such as variations in user load. Making automated and performant deployments decisions

enables support for performance requirements in a scalable manner, which results in increased efficiency in the management of features exposed by the platform and the infrastructure resources.

The results of the characterisation procedure in this context are optimised service descriptors which contain the values for each configuration parameter of interest corresponding to the most efficient resource allocation option in order to satisfy the customer specified SLOs. This supports the [KpiTemplate-01] requirement: "The system must be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA's and SLO's", as outlined earlier in section A3.1.1

A3.9.2 Characterisation Lifecycle

There are three phases necessary in order to characterise a service and optimise its deployment template:

- Experiment Execution: automatic deployment of the service and execution of stress tests.
- Data Collection: collection of metrics from workload generators and telemetry agents to collect metrics related to the service's behaviour under representative operation scenarios.
- Data Analysis: extraction of information from the data collected and generation of orchestration insights.

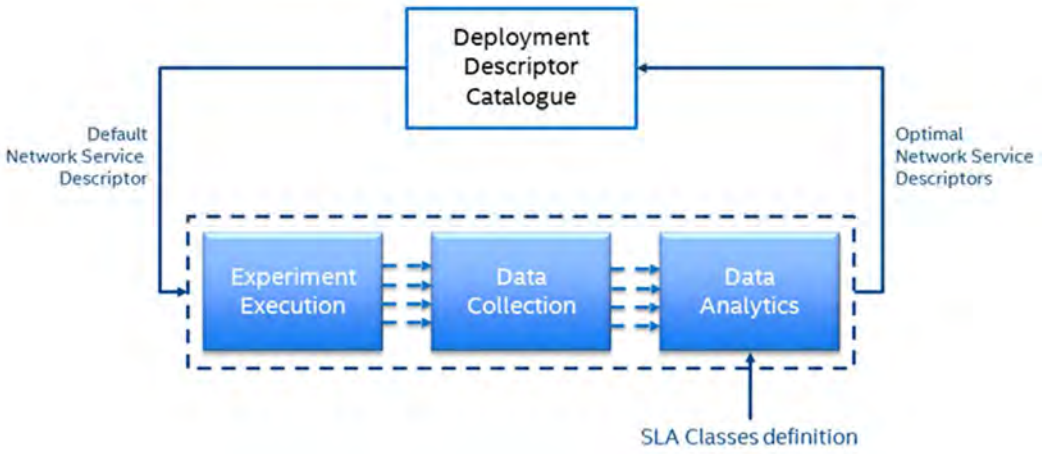


Figure 55: Characterisation Phases

Characterisation of a service starts with the Service Descriptor taken from a given descriptor catalogue. Examples of catalogues include OpenStack's Murano project and OSM's Network Service Descriptor (NSD) and Virtual Network Function Descriptor (VNFD) catalogues.

The following sections outline the details of each phase shown in Figure and highlight some important requirements.

A3.9.2.1 Experiment Execution

The framework automates the execution of experiments analysing different configurations in order to determine the relationship between service performance and the deployment configuration parameters. There are a multitude of possible parameters of interest for a given configuration selection. Parameters of more potential significant in the context of Superfluidity are:

- The *flavour* of the virtual components, as per the number of cores and the amount of RAM to be used by each component of a service;
- The *vNIC type*, which could be based on either virtual switch technology (such as Open vSwitch) or pass-through (such as SR-IOV);
- The *core pinning policy*, which may or may not allow core isolation on a given compute node;
- The *memory page size*, which could allow usage of either small or large memory pages to improve the efficiency of memory management.

In order to identify the specific configuration to be used, the characterisation methodology has to stress test the deployment of the service changing specified configuration values and measuring the impact that the changes have on performance.

For the configurations of interests, a single experiment can be defined as per the following sequence of actions:

1. Deployment of the service according to the predefined configurations options;
2. Deployment of any noisy neighbours or concurrent workloads, (if required);
3. Execution of the stress test (in the form of a packet/traffic generation tool);
4. Termination of the service.

Actions 1, 2 and 4 can be implemented by exploiting standard orchestrator functionalities. Therefore, to automatically execute experiments, integration with an orchestrator API is necessary. Action 3 requires the execution of a stress test, such as the application of packet traffic via packet generator, and therefore requires the integration of the framework with appropriate open source or commercial packet generators.

If the configuration space of interest is reasonably small, a brute force approach is plausible, where all possible permutations are repeatedly tested and measured to generate statistically validate data sets. For instance, analysing four parameters and assuming each of them can assume two possible values, if the service is composed of one single component, the total number of possible permutations is 16 and assuming the experiments will be repeated three times in order to validate statistical consistency, this will result in 48 unique experiments, which can be executed in reasonable

time period by an automated engine provided the total deployment time per unique configuration is in the order of minutes.

A3.9.2.2 Data Collection

The collection of data is a very important aspect of the characterisation process. There are two types of data that are of relevance to workload/service characterisation, namely:

- Stress test metrics: metrics collected by the packet generator, such as throughput, latency, jitter, etc. It is likely that a subset of these metrics will be considered as SLOs, and represent the target for analysis (for instance, the user could require a throughput higher than a given value, or a latency lower than a given value).
- Telemetry metrics: metrics that reflect the effect of the service execution on specific infrastructure ingredients. They are usually represented by hardware counters or ingredient specific measurements (such as CPU, NIC and disk utilisation, etc.).

The main focus of this characterisation activity is on the first category (stress test metrics), since they are part of the SLOs that will represent the constraints of the problem.

This requires integration of the characterisation methodology with the packet generator used for the experiment execution in order to trigger and control the execution of the stress tests as stated before and to collect the results of the test and format/clean the data in a manner suitable for the data analysis pipeline to process them.

A3.9.2.3 Data Analysis

The main scope of the data analysis is focused on the optimisation of the deployment template in order to determine the rules that will be used to eventually generate the configuration.

Those configurations can be seen as a formal representation of the trade-off between the user requirements, expressed in the form of SLOs (and a probability to satisfy them over time) versus the service provider requirements, expressed in the form of a cost.

The constraints taken into account are represented by the SLOs defined in the user requirements. They need to be treated as thresholds (for instance the desired throughput has to be higher than 5 Gbps) and the value of those SLOs have to be within the desired range with a given probability (for instance the throughput is higher than 5 Gbps for 99% of the execution time).

For each configuration option, it is necessary to calculate the contribution to the total cost. For instance, selecting SR-IOV instead of an OvS network connection will increase the cost due to the fact that the availability of SR-IOV channels is finite within a given NFVI. The cost does not necessarily need to be expressed in an actual monetary terms, but it needs to be proportional, such that the

comparison between the different configuration options is meaningful (for instance, it could be based on availability of resources).

Selecting the best combination of configuration options requires the analysis of the data collected during previous phases in order to associate the configuration options to the impact that they have on performance. From this perspective, it is important to identify which are the specific configuration parameters that provide a key contribution to the performance of the service under test as well as the impact on cost. Application of an appropriate data analytic technique or pipeline of techniques is necessary to identify the configuration that satisfies the SLO constraints while providing the lowest cost.

A3.9.3 Exploration of the Deployment Configuration Space for a Virtualised Media Processing Function

A3.9.3.1 Experimental Setup

The testbed used for investigating deployment configurations comprised of a dual socket E5 2620 v3 (15M Cache, 2.40 GHz) based server running an OpenStack Kilo environment as shown in Figure 56. The Hammer workload generator was deployed on a dedicated single socket E5620 (12M Cache, 2.40 GHz) based server and connected to an OpenStack controller node via a 10 Gbps switch. A Heat template was defined to deploy the virtualised media processing function (Unified Origin). The template was manually updated with required configuration settings prior to each deployment. At the end of each of experiment, a python script was used to parse the Hammer results output file and to insert the extracted data into an InfluxDB database. The open source data visualisation platform Grafana was used for inspection of the experimental results.

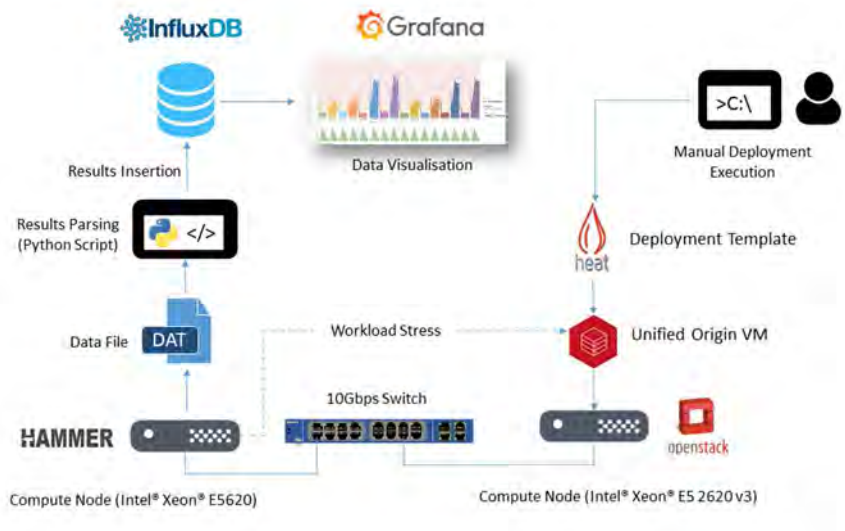


Figure 56: Experimental Configuration

A3.9.3.2 Experimental Results

The definition of a Heat template to be used for the deployment of a virtualised media processing function (Unified Origin) was successfully implemented, as well as integration with Citrix Hammer, enabling the automated deployment in conjunction with stress test execution in an OpenStack cloud environment. An experimental campaign was carried out to identify the performance ranges and the impact that the configuration parameters have on Unified Origin.

The results obtained from 16 different configurations are shown in next Table and were collected using the testbed configuration as shown in Figure.

	vCPUs	RAM	vNIC Type	Memory Page size
Configuration 1	4	4	SR-IOV	Large
Configuration 2	4	4	OvS	Large
Configuration 3	4	4	SR-IOV	Small
Configuration 4	4	4	OvS	Small
Configuration 5	2	4	SR-IOV	Large
Configuration 6	2	4	OvS	Large
Configuration 7	2	4	SR-IOV	Small
Configuration 8	2	4	OvS	Small
Configuration 9	4	2	SR-IOV	Large
Configuration 10	4	2	OvS	Large
Configuration 11	4	2	SR-IOV	Small
Configuration 12	4	2	OvS	Small

Configuration 13	2	2	SR-IOV	Large
Configuration 14	2	2	OvS	Large
Configuration 15	2	2	SR-IOV	Small
Configuration 16	2	2	OvS	Small

Table 7: Configuration settings

The results obtained are shown in Figure 57 and Figure 58, which are snapshots of the data in InfluxDB.

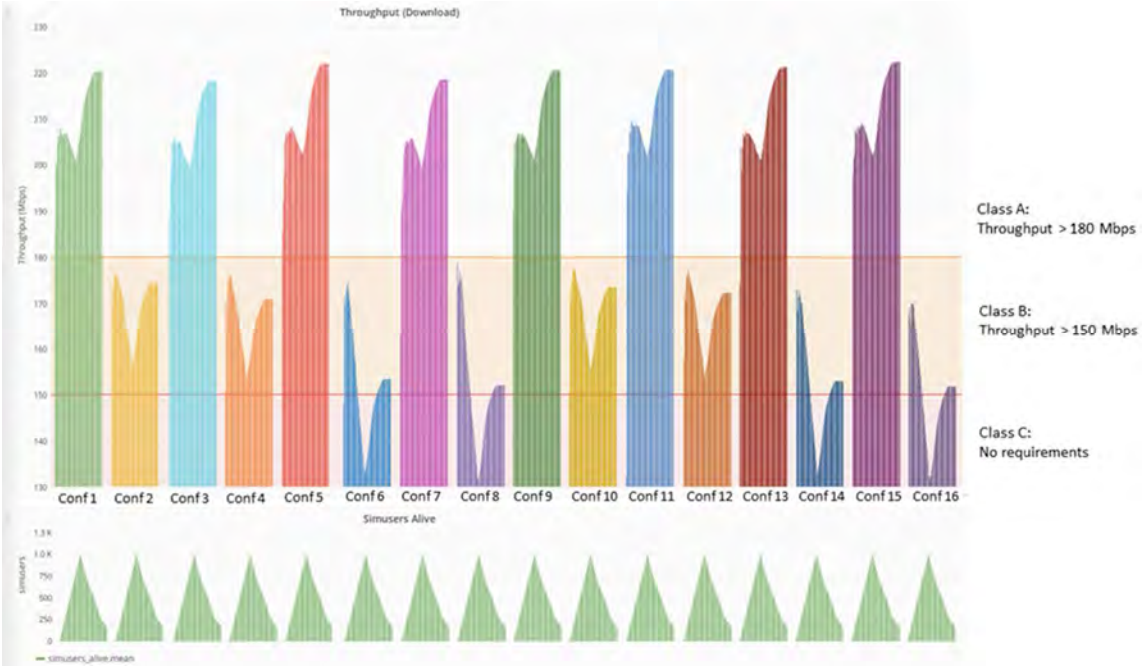


Figure 57: Throughput results for different configurations.

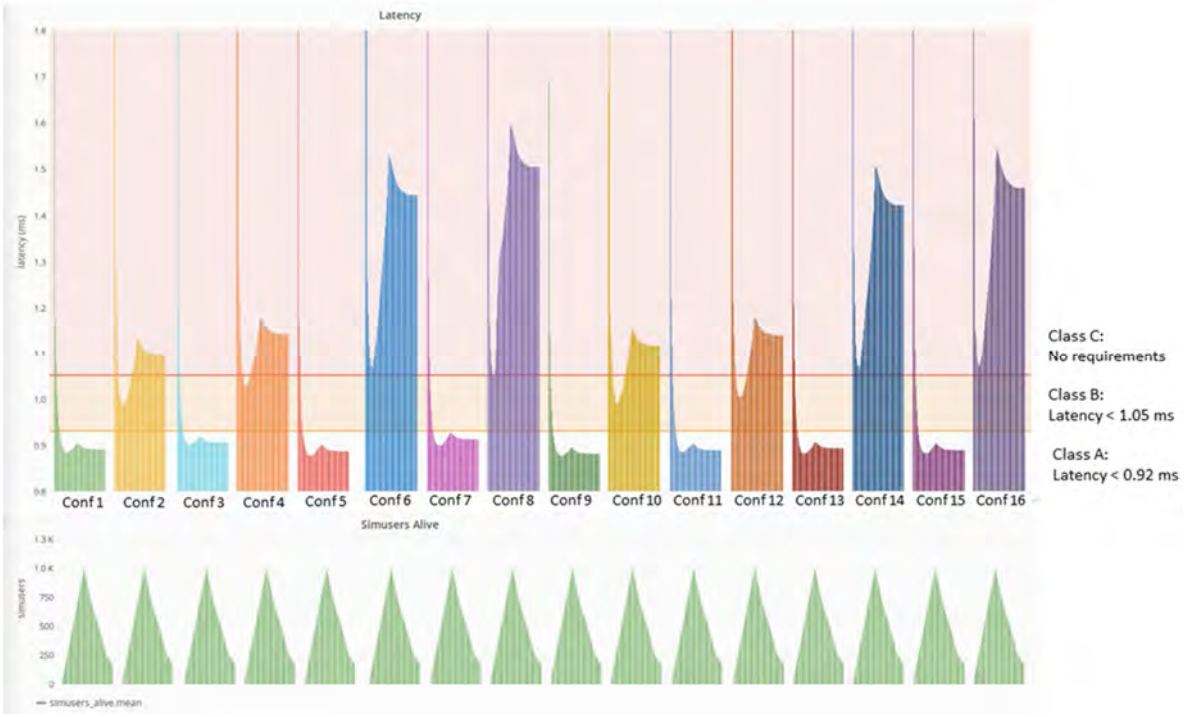


Figure 58: Latency results for different configurations

These results show that varying the combination of the four parameters it is possible to classify the configurations in to 3 unique performance classes:

- Class A: Throughput > 180 Mbps and Latency < 0.92 ms
- Class B: Throughput > 150 Mbps and Latency < 1.05 ms
- Class C: Best effort (no specific requirements) for both throughput and latency

In Figure 57 and Figure 58, the simuser profile used for the experiment is shown. The simuser ramp scales from 0 to 1000 users in the first phase of the experiments and then ramps back down to 0 in the second phase of the experiment. The results also show the impact of increasing the number of users on the KPIs, throughput increases and the latency decreases when the number of user increases and vice versa.

An initial analysis of the results in relation to the configuration parameters highlights how the usage of SR-IOV has positive impact on both throughput and latency and this specific configuration on its own seems to be guarantee of good level of performance, placing the service performance within SLA Class A ranges. At the same time, this represents the most expensive solution from an infrastructure perspective, which might lead the service provider to select an option corresponding to a lower cost in the case the user requirements fall into SLA Classes B.

The results obtained for configurations 6, 8, 14 and 16 highlight that when only 2 vCPUs and OvS based configurations are used both latency and throughput performance is degraded significantly, placing the service performance within SLA Class C.

These results show that the resource allocation policy has a huge impact on the performance of the service. Therefore it is important to identify which hardware features influence a service in order to deterministically achieve target performance levels and to avoid the allocation of resources which have no meaningful impact on performance, therefore avoiding wasteful allocation of resources and reducing service execution costs.

Collection of characterisation results requires human involvement in the execution loop, which introduces a time penalty (for experimental execution and results generation). The automated methodology is designed to minimise human involvement in the execution loop. The results obtained using the manual approach act as a reference or ground truth to sanity check the results obtained by the automated methodology described in the following section.

A3.9.4 Generalised and Automated Generation of Optimised Service Deployment Templates

As previously demonstrated, achieving the performance level defined in Classes A, B or C clearly depends on the resources allocated to the service. The set of resources assigned to a service for its execution is defined as the deployment configuration of the service. The deployment configuration of a service can therefore be associated to a cost, which relates to the quantity and type of resources in the configuration. Every deployment configuration can also be associated with a given performance level, expressed using SLOs, such as throughput and latency.

As outlined previously selecting the configuration to use in order to achieve a performance representing class A, B or C can be done by humans, through manual inspection and deciding the best set of resources to be used. In order to optimise the utilisation of infrastructure resources and to reduce the cost of service execution, while achieving a desired level of performance, both the cost and performance of every configuration need to be considered. The number of possible configurations is usually high, which makes investigating all possible configurations very time consuming and impractical operationally. The problem is further exacerbated by the large number of virtual services that need to be characterised by Service Providers for inclusion in their service catalogues. The solution to this problem requires two main elements: a high degree of automation and a generalizable implementation.

In previous sections, the characterisation process was described together with a number of requirements. Among them, a number of similarities have been identified with the framework developed during the course of Task 4.1's automated KPI mapping activities, which has been used for KPI Mapping of Unified Origin. As a consequence, the framework previously developed has been

reengineered and extended to accommodate the new requirements identified in the context of Task 6.1, i.e. automated optimisation of service deployment templates.

The proposed methodology finds a generalizable and automated solution by generating a model (using a machine learning approach) which expresses the allocation of resource in relation to specific levels of performance, from which rules are extracted and used to add intelligence to the decision process, determining which resources to use for the execution of a given service.

A3.9.4.1 Performance Measurements Based on SLOs

When using SLOs to measure performance of a service, it is important to decide how to represent the performance classes (such as Class A, B and C defined in section x.xx) from a statistical perspective (choosing from measuring the average, the median and variance, the minimum/maximum, etc. of a SLO over time) since this choice clearly impacts the meaning and the reliability of the results. To solve this problem in the context of the methodology defined in this section, the throughput measurement methodology as specified in the RFC 2544 (https://www.ietf.org/rfc/rfc2544.txt) was utilised. This applies to the measurement of the performance of a Device under Test (DUT) acting as middle-box in a network. The throughput measurement as defined by the RFC 2544 requires the middle-box to maintain a constant level of throughput for the full duration of the test trial: if the throughput falls below the required threshold during the trial, the current level of throughput is determined not to be supported by the middle-box and it is necessary to use a lower value of throughput and to repeat the test until the DUT supports the current throughput level for the full duration of the trial. The highest throughput which satisfies this condition is taken as the maximum supported throughput for the DUT. Since the methodology defined in this section needs to be applied to generic services and not only to middle-boxes, it requires the definition of measurement criteria which can generalise the process. From this perspective, there are different aspects which need to be considered, some of which are service-specific:

1. Definition of a specific traffic/user profile for the service to model the behaviour of real life traffic/users;
2. Ability to run a required configuration profile against a virtual service instance deployed on a cloud infrastructure using a traffic/packet generator with fully automated control;
3. Implementation of a system to measure the performance level (in terms of SLOs) which expresses a performance measurement that is appropriately contextualised for the service under consideration.

For this reason, some effort was spent to implement new features in the Service Characterization Framework such that it could be extended by adding Use Cases. A use case is intended as a flow of actions designed for a specific service in order to support: the collection of user input for the definition of traffic profiles which might change across different use cases; the configuration and

control of a packet/traffic generation technology to stress test the service; the collection of performance measurements specific for the service under test.

The user is also required to specify the SLOs of interest and if they are required to be minimised or maximised. For example, the user may want throughput to be maximised and latency to be minimised. This information is used by the methodology to decide if the data points in the test trial can be considered part of a given Class (for instance, in Class A the throughput threshold will be considered as the one where the throughput is equal or **greater than** 180 Mbps, whereas for the latency threshold it will be equal or **lower than** 0.92 ms). Following this approach, the minimum value are taken into account for SLOs that need to be maximised (i.e. throughput), whereas the maximum value will be taken into account for SLOs that need to be minimised (i.e. latency).

A3.9.4.2 Automated Methodology for Deployment Rule Extraction

Automating the identification of patterns in the relationship between resource utilisation and performance is necessary to solve the problem in a scalable manner. However, the implementation of an intelligent methodology that can reason over the appropriate decision is non-trivial. To solve this problem, a fully automated pipeline was defined as part of Task 6.1 activities which is composed of data collection, data manipulation and machine learning steps. This ultimately has been integrated into the Service Characterisation Framework and allows the collection of data relating to the behaviour of a service. It also allows the identification of patterns across different configurations of the same service and to determine a set of rules to combine into a service-specific policy in a fully automated manner, without any prior -knowledge of the service.

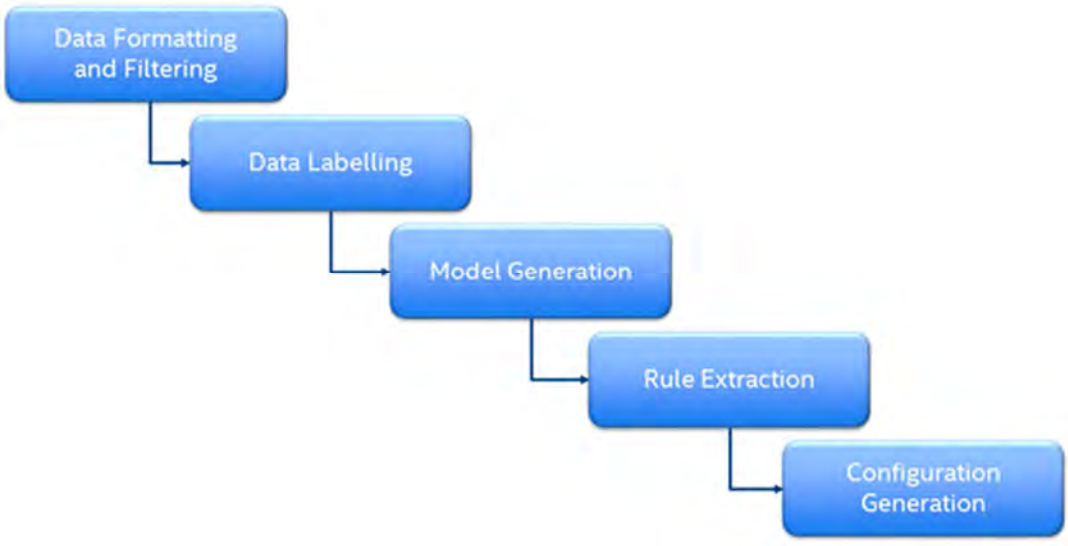


Figure 59: Template Optimisation Methodology Pipeline

Once the data related to the experiments is collected, the following steps are performed by the framework:

1. Data formatting and filtering: Data for all the experiments is merged in a single data frame, taking all the available samples. The number of samples depends on the duration of the experiments (which is defined by the user when defining the Hammer profile and ramp) and the sampling rate used by Hammer (the default value is 1 sample every 3 seconds). After the data merge, all the data points are analysed to identify outliers which are removed from the data set.

2. Data Labelling: The values of the user defined SLOs are analysed and the class of each data point is determined accordingly. Every data point is labelled with the name of its corresponding class, as per the user defined SLA classes (for instance, Class A, B and C).

3. Model Generation: The classification problem was solved using the C4.5 machine learning algorithm [37]. This approach classifies each data point creating a decision tree model that takes into account all available test trials with the same deployment configuration. For this application decision trees provide a stable classification approach particularly when dealing with limited amounts of data and when anomalies are present in the data.

4. Rule Extraction: Once the model has been generated in the form of a decision tree, the rules used by the decision tree are extracted and processed, in order to discover the most important configuration parameters to facilitate the classification. Processing the rules allows the system to discover the key configuration parameters first, which are then coupled with the cost information. This in turn allows the selection of the deployment configuration that provides the lowest cost among the available options.

5. Configuration Generation: Once the configurations are listed and connected to the cost information, the configuration with the lowest cost for each user defined class is selected and included into the results, which are converted into a JSON structure and exposed to an Orchestrator.

A3.9.5 Service Characterisation Framework Implementation

The Service Characterisation Framework designed and developed in the scope of Task 4.1 in WP4 was extended during the Task 6.1 activities with new features in order to accommodate the requirements of methodology developed. For instance, the capability of implementing new Use Cases to support the generalisation of stress testing services. This allows the Service Provider to easily integrate various packet generators (such as IXIA, Spirent, Agilent, etc.) by developing plugins for the Service Characterisation Framework and integrating the plugin into the automated experiment execution

module. In order to implement this feature the "Stress tools" box in the architecture was re-engineered as well as its interface with the core of the framework supporting a plugin system. The Hammer Use case was re-engineered to support this architecture change. Also the capability to collect input from the user was enhanced in order to define different profiles as necessary to stress KPIs differently for the service. In the latest version, the use cases take as its input a portion of a YAML file which defines the use case parameters. For instance, Hammer can be configured using the following YAML syntax:

profile_name: mixed-profile # Name of the profile to use
step_deltas: [20] # Number of Users to add per step
step_periods: [10] # Number of seconds of each step
step_repetitions: [100] # Number of steps

These new features allows the user to define the arbitrary behaviour of the packet generator, as requested by a specific test and represents the basis of automated experimental execution. The current high-level architecture of the framework is shown in Figure 60.

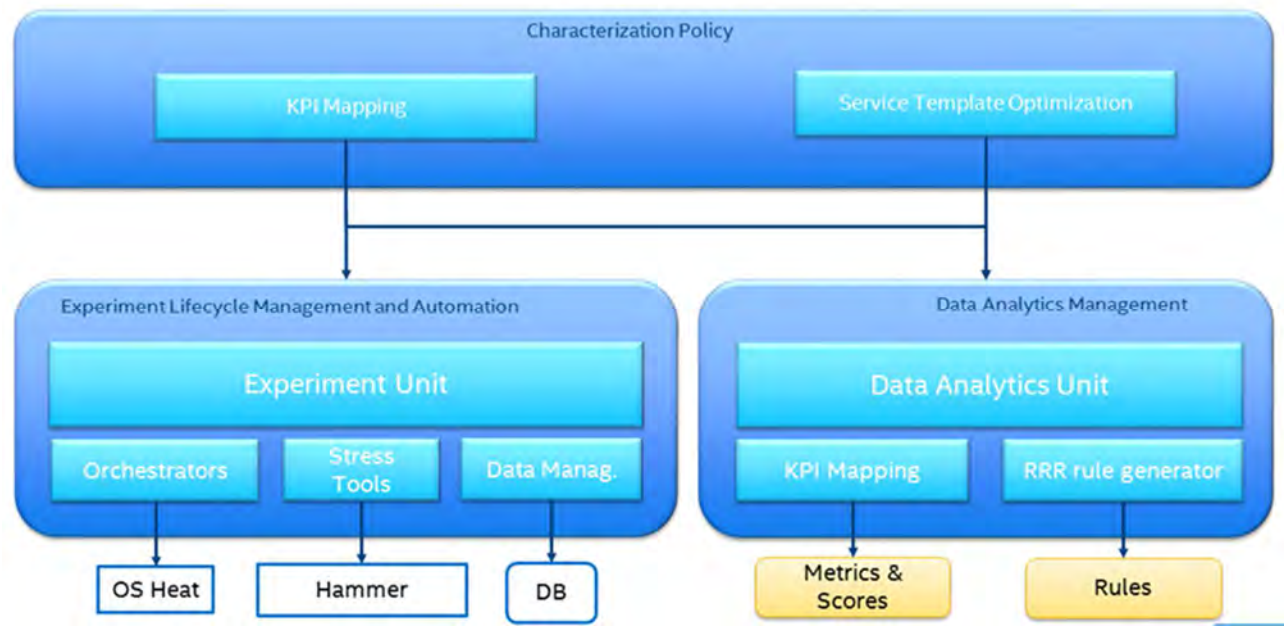


Figure 60: Service Characterisation Framework high-level architecture.

Both the KPI mapping and the Service Template Optimisation policies can be executed by the framework, which provides a common interface to all policies supported by the Experiment and the Data Analytics Units. This allows execution of both the KPI Mapping and Service Template Optimisation policies for Virtualised Video Processing Function (Unified Origin).

The **Experiment Unit** provides automatic management of the experiment lifecycle as described before: supports the deployment of a service, the deployment of any desired noisy neighbour e.g. CPU, network I/O etc., the execution of a stress test and termination of services. In order to

implement this capability, a heat template to deploy Unified Origin was defined that enables the automated deployment of the service. The Experiment Unit manages the generation of all the user required permutations and their execution lifecycle.

To execute the deployment of the services the Experiment Unit is integrated with OpenStack Heat and ETSI OSM. Integration with OpenStack Heat supports the deployment of a service based on a Heat template, whereas with OSM the user is required to specify the name of a service currently available within the NSD catalogue.

To run a stress tests appropriately for Unified Origin, the framework was integrated with Citrix's packet generator Hammer. The framework creates a Hammer profile for a simulated users (simuser) profile in accordance to a specific user definition (e.g. request type, number of users, linear to stepwise user ramp etc.), which is used to stress test different deployment instances of Unified Origin. The user can specify to the framework which kind of simuser profile to use, defining the name and all the necessary details of the user ramp dynamically. Additionally, integration allows at the end of an experiment the collection the metrics stored by Hammer. Those metrics are collected by the framework by automatically establishing an SSH session to the Hammer node and copying the data file (CSV format) for extraction and storage in an InfluxDB database in a time series format. The Data Analytics Unit provides automatic analysis of the experiment data. It loads the experiments data according to the specific type of processing required. The data analytics module which was initially developed support KPI Mapping in Task 4.1 (see D4.1) was extended to support Service Template Optimisation.

A3.9.6 Automated Methodology Results

In order to characterise the workload the Service Characterisation Framework was used to run 3 different iterations for each of the 16 configurations listed above (for a total of 48 experiments). This is supported by a YAML configuration file used by the framework, which is composed of different sections as discussed below.

Service definition Section:

service_id: ./heat_templates/unified_origin.yaml
deployment_parameters:
vm_ram: [2048, 4092]
vm_vcpus: [2, 4]
vnic_type: [normal, direct]
memory_page_size: [small, large]
cpu_policy: dedicated

Experiment section:

instance_name: unified_origin
use_case:
name: hammer
parameters:
profile_name: mixed-profile
step_deltas: [20]
step_periods: [2]
step_repetitions: [100]
nic: eth5
vlan_id_1: 400
vlan_id_2: 401
hammer_ip_address: 10.1.0.26

Characterization Methodology section:

iterations: 3
action:
type: template_optimization
parameters:
sla_classes:
class_A:
latency:
- '{} < 0.92'
curl_bytes_download:
- '{} > 180'
class_B:
latency:
- '{} > 0.92'
- '{} < 1.05'
curl_bytes_download:
- '{} < 180'
- '{} > 150'
costs:
vm_ram:
2048: 1
4092: 2
vm_vcpus:
2: 1
4: 2

vnic_type:
normal: 1
direct: 10
memory_page_size:
small: 1
large: 5

The configuration file firstly defines the service ID (file name of the Heat template (YAML) for service deployment) and the parameters required for the template, including the features of interest to be used during the deployments, (e.g. network connection type, memory page size, etc.). Secondly, the configuration file specifies the characteristics of the stress to be applied using Hammer during each experimental deployment (e.g. step size of simulated user ramp, number of steps, duration of steps, hammer profile for HTTP_GET requests, etc.). Thirdly it provides information about the characterisation methodology to be used, in this case "template optimisation" and the parameters for the characterisation to be used, like the number of iterations of experiments to be executed, the classes of performance to be explored, (expressed on the basis of the KPIs' values) and the costs corresponding to each configuration value.

Collecting the results of the 48 experiments, the Service Characterisation Framework applied the template optimisation methodology and returned the results in the following JSON object:

{
Class_A:
vm_ram: 2048,
vm_vcpus: 2,
vnic_type: direct,
memory_page_size: small
cpu_policy: dedicated
},
{
Class_B:
vm_ram: 2048,
vm_vcpus: 4,
vnic_type: normal,
memory_page_size: small
cpu_policy: dedicated
}
{
None:


vm_ram: 2048,
vm_vcpus: 2,
vnic_type: normal,
memory_page_size: small
cpu_policy: dedicated
}

The class “None” is related to the best effort behaviour, (which corresponds to Class C in previous data set). The methodology provides full execution of the pipeline, from experiment execution to the extraction of the policy into JSON format. The full execution cycle took approximately 10 hours to complete without any human intervention, after the definition of the YAML configuration file for the framework. Examination of the results and comparing them with the explorative data set, the template optimisation methodology was able to successfully extract the patterns from the tested configurations and interpret them correctly, generating the rules for the optimised template. These rules corresponded to the minimum impact on cost, according to the user input. The rules were outputted as a JSON file representing the 3 optimal set of configuration values to use for the 3 deployment templates (one for each class).

For Class A the importance of having SR-IOV (which is indicated with vnic_type = direct) was identified, whereas all the other variables become useless once SR-IOV is enabled, and therefore are set to the lower cost values.

For Class B, in the same manner, the vCPUs were identified as the key configuration parameter in order to achieve the required performance level. For Class C the minimum cost configuration was selected by the methodology.

A3.9.7 Superfluidity System Integration

As described in the previous sections, the focus of this work has been the development of an automated methodology which can define optimised deployment templates to deliver specific levels of performance in compliance with SLOs. This work provides input and is exploited by the following scenes which are being implemented as part of the overall Superfluidity demonstrator in WP7.

- Scene 2a - Offline Workload Characterisation
- Scene 2b - Streaming service deployment
- Scene 3a - Core Service Automatic Scaling

The functionality additions to the Service Characterisation Framework are being used in the workload characterisation activities. The optimised deployment templates generated by the framework are

used in scenes 2b and 3a to deploy the virtualised video processing component (Unified Origin) in an optimised manner.

A3.10 MicroVisor Orchestration

One of the most challenging requirements captured in Section 2 is [AppSched-03], that states that an application must be provisioned in <10ms. For a VIM such as OpenStack, this poses a number of challenges as this Python-based framework was designed for 'traditional' VMs that usually comprise a Linux or Windows-based guest OS. These VMs are heavy-weight and need miniaturization before they could start in the order of seconds. Containers and other light-weight virtualization techniques as those currently investigated in Superfluidity can start up much faster. When looking to approach fast provisioning and orchestration tools it is important to profile all aspects of the virtualization workflow.

In order to support <10ms provisioning times it is important to consider the design of the orchestration platform and to remove overheads. The MicroVisor, Hypervisor platform that OnApp are bringing to Superfluidity is purpose-built, light-weight, distributed and focused on maximising the performance of virtual workloads running on distributed resources. As such, there have been improvements carried out to the MicroVisor orchestration framework that can be used to help decide on decisions for the rest of the Superfluidity orchestration tools. The MicroVisor UI is based on OpenStack and has had various improvements to be able to manage the expected workloads of Superfluidity.

A3.10.1 UI design for managing a large collection of resources

Virtual workloads that are going to load in <10ms are potentially going to be far more numerous than standard visualization approaches currently account for. Horizon, which is the OpenStack Dashboard can handle the scale of Virtual Machines that currently are used by large enterprises, but will likely have some scalability issues when faced with orders of magnitudes more VMs than are currently used. A rethink of the UI is therefore needed for it to display the information available to administrators and end-users in a useful manner.

Computer assisted workload placement will therefore move from being just an optimization effort, to being a tool to help manage the workloads at the scales that are expected. A mockup diagram showing a possible visualization of the physical to virtual workloads is shown in *Figure 61*. This Figure captures the physical, network overlay and virtual resources and how they relate to each other. The work is ongoing to determine which visualization mechanisms users and administrators will find useful.

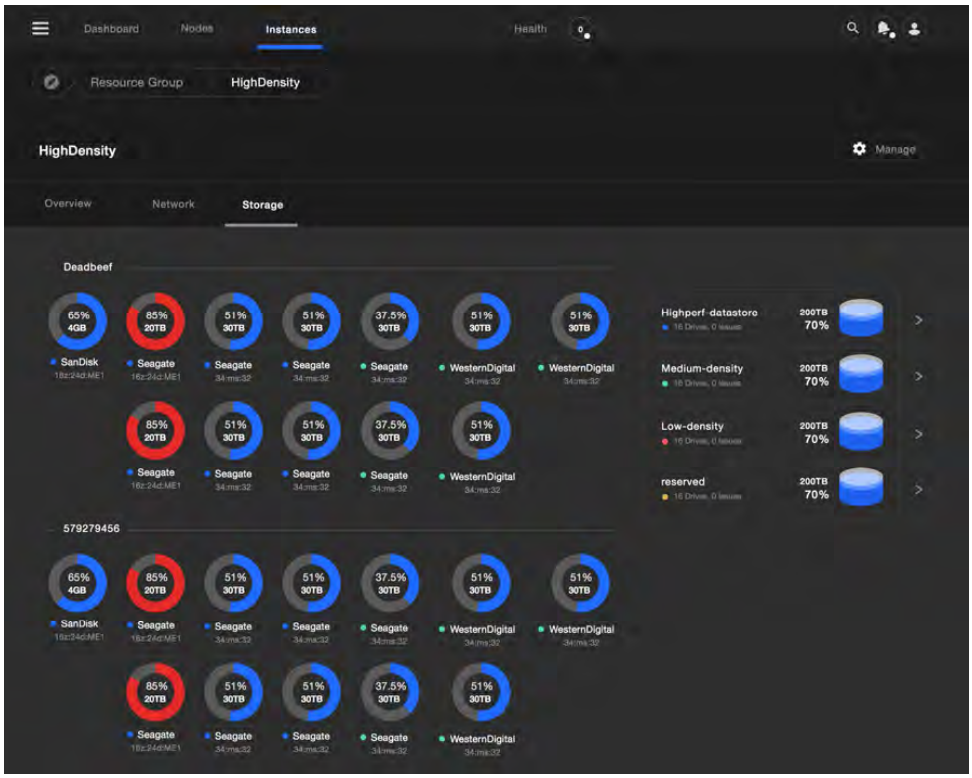


Figure 61: Visualization UI

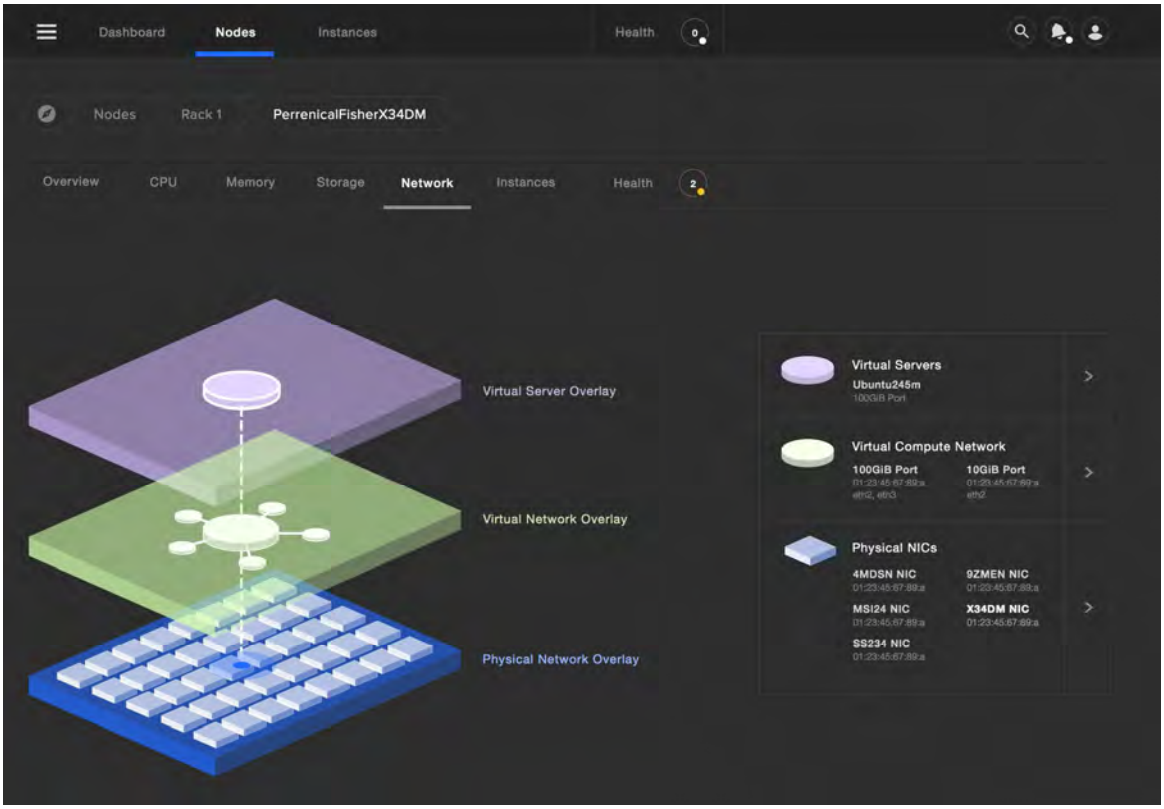


Figure 62: Mock-up diagram showing a UI that relates virtual to physical resources

In *Figure 62* a visualization mock-up of the administration panel is shown. In this visualisation, the physical racks have a number of rack servers that are numbered and can be probed for more information. Each rack then has a number of compute nodes that can be contained within a single physical server. The CPU load of each compute unit is then visualised, with standard traffic light colouring used to indicate low-utilisation (green), through to heavily loaded compute units (red). This gives an administrator a powerful tool to quickly identify if there are any servers that are struggling and to indicate issues that potentially need to be resolved either through computer assisted orchestration, or manual intervention.

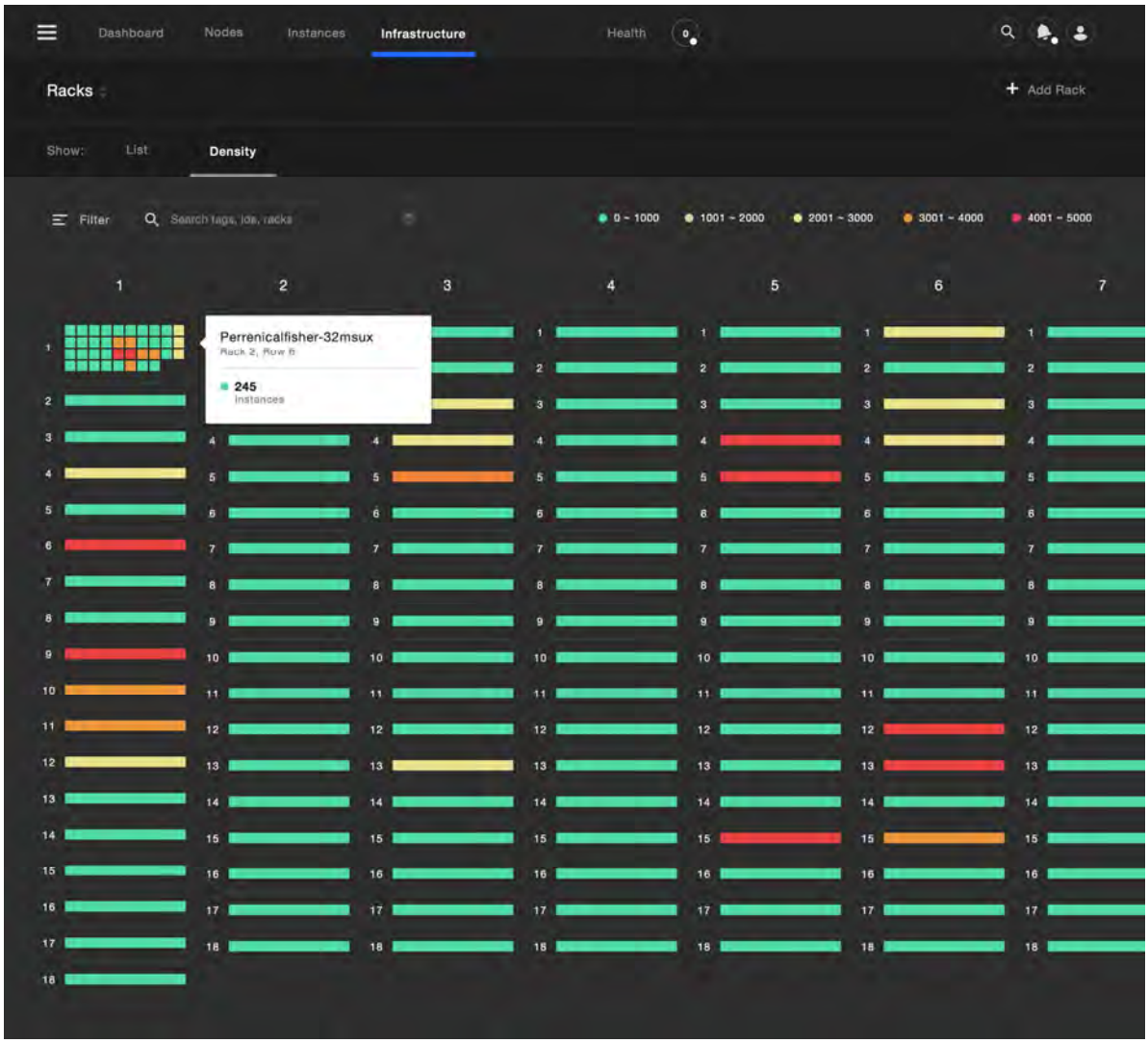


Figure 63: Mock-up diagram showing the rack utilization

Aside from the physical to virtual mapping and CPU load that have been shown in the previous Figures, it is also important to show the utilization of the storage resources. A mock-up Figure showing the utilization of the storage can be seen in Figure 63. Disks that are close to being full are shown in red with the less utilized disks being coloured in blue. All of the storage resources are

associated with particular racks and are separated accordingly. Also shown in the diagram is the notion of tiered storage performance levels. Given that certain virtualized workloads may have different I/O requirements it is important for the system to indicate different performance levels. This information can be captured in the data models in T4.1 and then analysed by the algorithms and heuristics in T5.1 to decide on where to place the workloads.

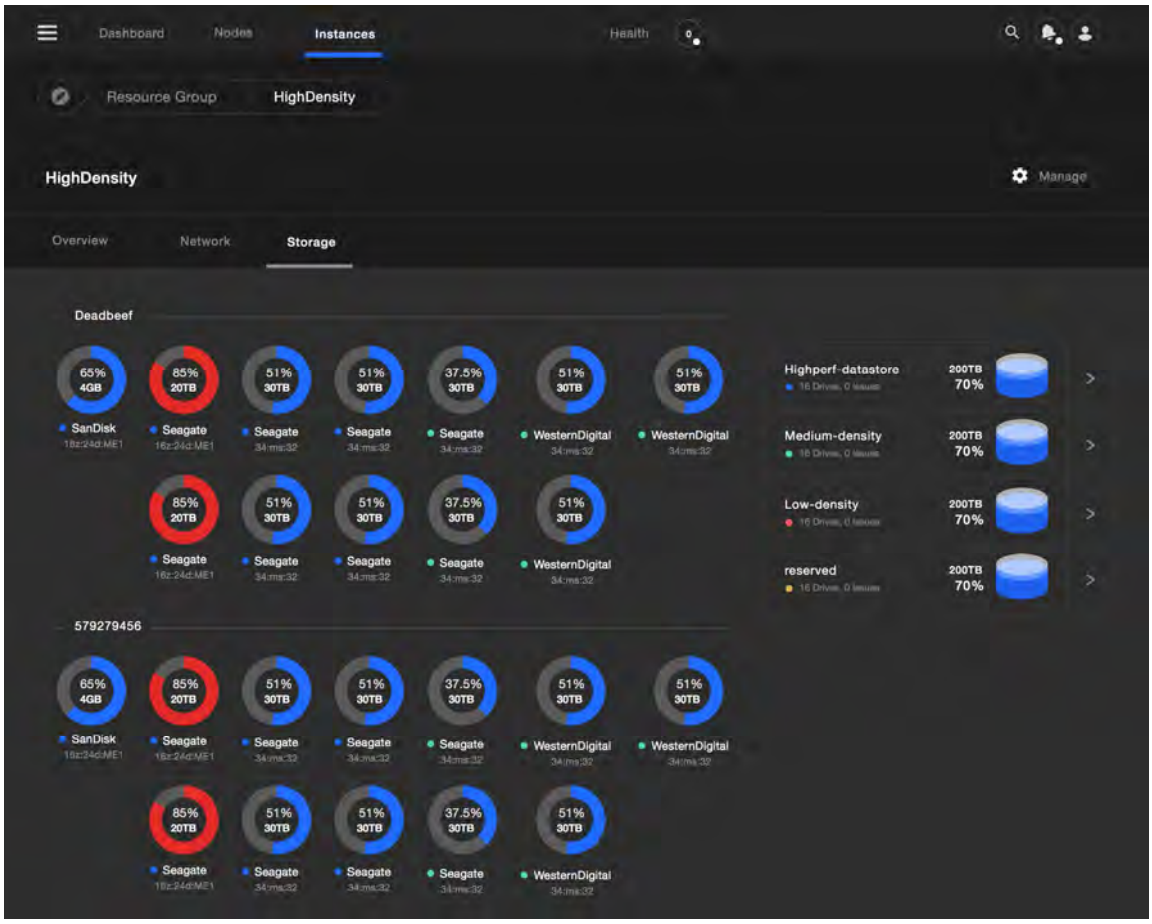


Figure 64: Mock-up diagram showing the storage utilization in the management UI

Given that SDN networking will also allow reconfiguration of a network, it is important for both the management platform and the orchestration system to be able to capture and possibly modify the network topology. This will allow maximization of the performance for a given set of workloads and configurations decided by the administrator. In Figure 64, a mock-up of the network mapping UI is shown. This can be used to visualize the current network topology and also could be used to capture modifications required of the network that could be then mapped to the network routers and hypervisors through tools such as OpenDaylight or others.

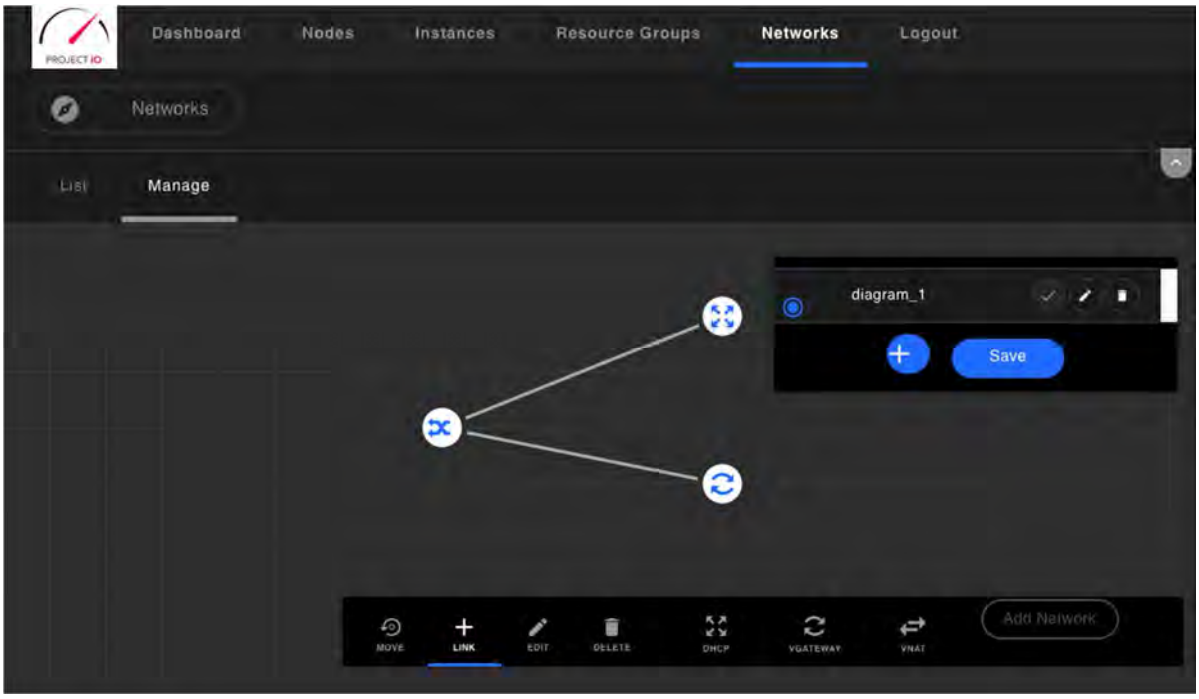


Figure 65: Mock-up showing the network planner UI

ANNEX B: Access-Agnostic SLA-Based Network Service Deployment – Task 6.2

B1 NEMO enhancements – implementation details

In order to integrate NFV expressions into NEMO a set of new features has been implemented.

Firstly, there is the need to reference the VNFD file as Universal Resource Identifier (URI). For this purpose, a new parameter has been defined in the NodeModel so that the lexical analyser will recognize a new token corresponding to the VNFD and the parser will add it as part of the NodeModel's parameters.

Secondly, NEMO needs to be aware of the virtual network interfaces defined in the VFND. In order for this to happen, a new network object has been defined, named ConnectionPoint. This new object has different functions: it can record the name of one network interface or it can be just the point of connection for a new NodeModel. Moreover, a new interesting feature has been integrated: we can think about VNFDs with N interfaces that could be set for the ConnectionPoint and we can choose which of these interfaces could be attached to the ConnectionPoint when it is instantiated. This is already possible by using the NodeModel's properties.

Thirdly, NEMO model provided a connection model to express the link between two network nodes. However, this feature has been extended and it now provides the link between either network nodes or connectionPoints. Because of this, it is possible to create the link between two simpler VNFDs.

The implementation of these basic features has enabled the creation of recursive VNFs. This means that it is possible to reuse NodeModels recursively to create complex NodeModel. Each NodeModels will check whether a node type matches the basic definitions (host, l2-group, l3-group, ext-group, chain-group, firewall and loadbalancer) or needs to be instantiated because it has a template definition (so achieving the recursiveness).

Moreover, it is possible to delete every Object created by NEMO. So that if a mistake occurs while writing the intent, it is possible to delete it and rewrite it again.

Finally, the processing of the NodeModels is needed in order to generate a complex YAML file (based on the baseline provided by OSM VNFDs) as outcome.

B2 Support for heterogeneous and nested execution environments

This section proposes extensions to the ETSI NFV ISG specification [4] to support nested VDUs using heterogeneous technologies.

7.1.6.2.2 [Vdu Information Element] Attributes

Clause 7.1.6.2.2 is modified as follows.

The following rows are added to Table 7.1.6.2.2-1.

Attribute	Qualifier	Cardinality	Content	Description
vduNestedDesc	O	0..1	Identifier (Reference to a VduNestedDesc)	This is a reference to the actual descriptor deployed in the VDU (e.g. a Click configuration). The reference can be relative to the root of the VNF Package or can be a URL.
vduNestedDescType	O	0..1	Enum	Identifies the type of descriptor file referred in the vduNestedDesc field (Click configuration, kubernetes template, etc.) and consequently the type of the Execution Environment running in the VDU. This field must be present if vduNestedDesc is present.
vduParent	O	0..1	Identifier (Reference to a VduId)	This is a reference to the parent VDU which contains this VDU, thus this field is needed only for Nested VDUs. The referred VduId must be defined in the current VNFD. Setting this field to the special value “None” specifies that this VDU is set to be deployed on bare metal.
vduParentMandatory	O	0..1	Boolean	This field specifies if the parent VDU must be present or if this VDU can be deployed also without its parent VDU. This

				field can be present only of the vduParent field is present and specified. The absense of this field is equivalent to setting its value to “false”.

Table 8: VDU attributes

We have extended the VDU information element contained in the VNF Descriptors to reference different types of Execution Environments that can be instantiated within a VDU and described by means of some descriptor. So far we considered kubernetes VDUs and Click router configurations VDUs. In particular, we have introduced four new attributes to the VDU information element: namely *vduNestedDescType*, *vduNestedDesc, vduParent* and *vduParentMandatory*.

- The *vduNestedDescType* attribute defines the type of RFB Execution Environment that is running in the VDU (in our case *kubernetes* or *click*).
- The *vduNestedDesc* attribute is an identifier. It provides a reference to the actual descriptor that is deployed in the REE running in the VDU (in our case a Click configuration file or a Kubernetes template). The proposed *vduNestedDesc* attribute uses the same approach of the *swImage* attribute, which provides a reference to the actual software image that is deployed in a “regular” VDU.
- The *vduParent* attribute is also an identifier. In case of a nested VDU, it references the parent VDU. An example of nested VDU could be a kubernetes pod (i.e. a group of containers) *K* inside a Virtual Machine *V*. In this case the VDU associated to K would have its *vduParent* attribute set to the *vduId* of the VDU associated to V. The VDU identifier must belong to the current VNFD, as the *vduId* is unique only in a VNFD scope (see clause 7.1.6.2.2).
- The *vduParentMandatory* attribute applies also to nested VDUs only and specifies if the VDU can be deployed also without its parent VDU. Referring to the above example, it specifies whether the pod K can be deployed also on bare metal (*vduParentMandatory* set to false) or if K must be deployed inside V (*vduParentMandatory* set to true).

7.1.6.4.2 [VduCpd] Attributes

Clause 7.1.6.4.2 is moadified as follows.

The following rows are added to Table 7.1.6.4.2-1.

Attribute	Qualifier	Cardinality	Content	Description
InternalIfRef	O	0..1	String	Identifies the network interface of the VDU which corresponds to the VduCpd. This attribute allows to bind the VduCpd to a specific network interface of a multi-interface VDU.

Table 9: Additional VDU attributes

We have also introduced a new attribute, namely internalIfRef, to the VduCpd information element. The VduCpd information element is referenced by the VDU information element through the intCpd attribute. We add the attribute internalIfRef to the VduCpd element to create a correspondence between a VduCpd element and the network interface of a multi-interface VDU. For example, ClickOS instances internally name their interfaces as numbers starting at "0". A ClickOS-based firewall with two interfaces would thus have an interface named "0" and an interface named "1". The firewall could expect (in its Click configuration file) traffic from an external network A on port "0" and traffic from an internal network B on port "1". The VDU corresponding to this ClickOS-based firewall would thus have two internal VDU connection points, one leading (through other connection points and virtual links) to the network A and one leading to network B. In this case the VduCpd element that would lead to network A would have its internalIfRef attribute set to "0", while the VduCpd element that would lead to network B would have its internalIfRef attribute set to "1".

B2.1 Notes on Kubernetes Nesting

When specifying Kubernetes VDUs, the *vduNestedDescType* attribute is set to the value "kubernetes" while the *vduNestedDesc* attribute is set to the identifier of a Kubernetes template.

Kubernetes templates can describe a pod, which in general includes several containers (sharing the same IP address). Thus in case the *vduNestedDescType* is set to "kubernetes", the VDU represents a pod (not a single container).

Application containers such as Docker do not expose to users the concept of network interface. Thus in the NFV scenario we should consider a pod as single interface VNF/VDU. This means that for kubernetes VDUs we do not need to specify the *InternalIfRef* VduCpd attribute.

Kubernetes VDUs can use the *vduParent* attribute as specified above and as exemplified below.

Example1: pod to be deployed on VM

In this example we have a kubernetes pod to be deployed inside a VM/kubernetes worker node. We assume that the VDU associated to the VM has the *vduId == "vmid"* and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the attributes of the VDU associated to the pod would be:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: vmid
- vduParentMandatory: true

Example 2: pod to be deployed on bare metal

In this example we have a kubernetes pod to be deployed directly on bare metal. We assume that the pod is described by the kubernetes template *k8stemplate*. In this case the values of the VDU associated to the pod would be:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: none
- vduParentMandatory: true

Example 3: pod should be deployed on VM, but can be deployed also on bare metal

In this example we have a kubernetes pod to be deployed on a VM/kubernetes worker node, but, if needed (e.g. due to resource unavailability), the container can be deployed directly on bare metal. We assume that the VDU associated to the VM has the *vduId == "vmid"* and that the pod is described by the kubernetes template *k8stemplate*. Then the values of the VDU associated to the pod would be:

- vduNestedDescType: kubernetes
- *vduNestedDesc*: k8stemplate
- vduParent: vmid
- vduParentMandatory: false

To actually deploy a kubernetes pod in its parent VM/worker node, the *nodeSelector* field of *PodSpec* can be used:

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

The appropriate *nodeSelector* value should be added during the translation/deployment phase to the kubernetes template referenced by the *vduNestedDesc* attribute.

B3 Core data-center placement

See attached: PAPER – 1: Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching

B4 Mobile Edge Computing

B4.1 Placement

Placement of MEC Applications, meaning defining at which ME Hosts they will be deployed and run, must be determined by their requirements, especially by the maximum allowed delay to the UEs to be served. Considering that MEC addresses the need to have services provided as close as possible to users, it is expected that MEC Applications will be deployed at the closest ME Host to UEs to serve. However, the existing delay budget and resources availability may decide differently. Additionally, availability of certain ME Services, like RNIS (radio interface status) or LOC (location), will also constrain the MEC Applications placement.

In addition, depending on Applications' provided services and resources availability, Applications may not be immediately moved to all the ME Hosts, as part of the on-boarding procedure. Business and licensing conditions may also determine when and where to deploy and run MEC Applications.

ME Hosts will serve a number of eNBs. This number may vary from one ME Host per eNB, possibly running at the eNB, to several eNB being served by a single ME Host. Considering C-RAN (Cloud-RAN) deployments, it makes all sense to consider the deployment of ME Hosts, sharing the same cloud infrastructure with RAN centralized components.

ME Orchestration (MEO) must have a topological view of the entire MEC System, in order to determine where to deploy ME Applications to serve specific geographical areas. Thus, a mapping between eNB and ME Hosts is required. This information was referred before as stored in the MEC Hosts Inventory repository.

In addition, ME Application descriptors must include parameters to help MEO to decide where to deploy each ME Application. This must be complemented by Operators' rules and established business relationships, and translated into MEO understandable policies.

MEC Applications' placement will be decided based on Applications and Operators' requirements and constraints, and not UEs, with the exception of instantiations requested by entities running at UEs (e.g. Applications counterparts running at UEs).

Thus, in general, the following placement factors may apply, determining how many MEC Application instances are required and where to deploy those instances:

- Delay
  MEC Application Descriptors shall indicate desirable and maximum admissible delay towards UEs to be served.
- Geographical scope
  Some ME Applications may provide localized services, like Augmented Reality for a specific building or street. The geographical scope will, most likely, be provided in such a way (e.g.


reference to a monument or specifying a geographical area) that will need to be mapped to eNBs, and from that to ME Hosts, by the MEO.

- Required resources
  Besides specifying computational and networking resources to run, MEC Applications may require specific hardware to run, like video processing.
- Available resources
  Considering that all on-boarded ME Applications cannot be deployed and run at all ME Hosts, their instantiation may depend on the availability and establishment of priorities to access ME Hosts' resources.
- Required MEC Services
  MEC Applications may need to access local MEC Services to run, for instance, with RNIS or LOC.
- Licensing and agreements with service provider
  Business agreements with ME Application providers may also determine where and how many instances to deploy. The limitations may be in any of the parties (Operator only allowed or allowing to simultaneously run a certain number of Application instances).
- UEs' location
  This is a specific scenario that will apply to Applications' instantiation requests generated by the UEs (via the "User App LCM Proxy", e.g. for Application computation off-loading). Besides any of the abovementioned factors, this specific one will be determinant in the placement of the requested MEC Application instance.

MEO shall handle the identification and evaluation of the parameters that apply. This will happen after MEC Applications on-boarding and whenever the MEC System topology change, for instance, by the addition or removal of MEC Hosts or the reorganization of the mapping between eNB and ME Hosts. It will also happen whenever a UE or the OSS requests the instantiation of some MEC Application. No algorithm or parameters evaluation process is proposed for the moment.

B4.2 Service Migration & Mobility

B4.2.1 Mobility in MEC scope

In MEC context, service migration need, or relocation, may happen as a result of UEs' mobility, and applies to MEC Applications. While moving, most likely UEs will attach to different eNB. Depending on the network topology, the same or different MEC Hosts may serve those eNB. While moving between eNB served by the same MEC Host, no relocation shall happen, as UE's mobility will not be noticed by the MEC System. For the other situations, the frequency and the type of relocation to be executed depend on the Application type and mode of operation:

- Generic Applications (not tied to any UE in particular)
  - Stateless applications: no need for any relocation action
    - Application instance is already working at the destination ME Hosts: Service is provided at the edge
    - Application instance is not working at the destination ME Hosts: Service is not provided at the edge
  - Stateful applications: state may be is need
    - Scenario detailed below
- UE specific Applications
  - Service needs to follow the UE, independently from being stateless or stateful, in order to keep proximity, according to the specified requirements

(Generic) MEC Applications will be deployed at certain ME Hosts, as described above. They will have traffic rules associated to them, instructing the ME Host Data Plane on how to identify and handle traffic of interest, to be delivered to each MEC Application. These rules will be defined, in general, based in destination FQDN or IP/Ports. It means that, in general, Applications will be deployed and make their services available in anticipation and independently of the UE which will request them. Thus, MEC Applications' provided services are accessible at the specific ME Hosts on which the MEC System decided to deploy those MEC Applications. Therefore, UE's mobility shall not trigger, in general, MEC Application relocations, except for the ones instantiated under UE's request. However, application state created and associated to the UEs may need to be relocated.

MEC Applications mobility is mentioned in current ETSI MEC ISG documents and its discussion triggered the creation of an Work Item (WI), to be developed till end of MEC Phase 1 (end of 2016). For instance, [ETSI GS MEC 002, Mobile-Edge Computing (MEC); Technical Requirements] states that:

"Other mobile edge applications, notably in the category 'consumer-oriented services', are specifically related to the user activity. Either the whole application is specific to the user, or at least it needs to maintain some application-specific user-related information that needs to be provided to the instance of that application running on another mobile edge host.

As a consequence of UE mobility, the mobile edge system needs to support the following:

- *continuity of the service,*
- *mobility of applications (VM), and*
- *mobility of application-specific user-related information.*

"

The same document identifies three mobility requirements:

1. *"[Mobility-01] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell associated with the same mobile edge host."*

2. *"[Mobility-02] The mobile edge system shall be able to maintain connectivity between a UE and an application instance when the UE performs a handover to another cell not associated with the same mobile edge host."*
3. *"[Mobility-03] The mobile edge platform may use available radio network information to optimize the mobility procedures required to support service continuity."*

As a summary, the MEC System shall guarantee service continuity, by application or "application-specific user-related information" mobility, or maintaining connectivity to the ME Host where the required MEC Application is running. This last option will have as a limitation the maximum delay allowed by the Application.

Whenever relocation is needed, the following types may apply.

B4.2.2 MEC relocation types

Considering the stated requirements, the following application relocations types can be envisioned:

- Application state relocation

  Application state associated to served UEs is transferred between different MEC Applications instances, located at the old and new MEC Hosts, involved in the UE's mobility.

  Even if it is possible to have a MEC System assistance (aspect not yet defined at ETSI ISG MEC), MEC Applications will most likely manage themselves the state transfer via communication between the involved instances.

- Application instance live relocation

  Applications running at a ME Host, as VDU, and providing services to specific UEs, are live migrated from old to the new ME Hosts (between different NFVI), serving the eNBs where the UEs are now connected. Mechanisms for a complete VDU relocation, shall resort to virtualized infrastructure capabilities.

  MEC Management and Orchestration needs to participate in the process, coordinating it.

- Application instance emulated relocation

  Application instances relocation can be emulated by the creation of a new instance and deletion of old instance, at originating and target ME Hosts

  If there is no need for Application migration or state transfer but the provided services are needed at the new MEC Host, relocation can be emulated by starting a new Application instance at the new ME Host, while stopping the previous instance at the originating ME Host. For the UE, this operation shall be transparent and the service works in a continuous way.

  MEC Management and Orchestration needs to deal with this process.

- No relocation, keeping service anchor

In order to keep service continuity and due to required Application state maintenance specificities, the solution may consist in keeping the provided service anchored at the original MEC Application instance, running at the first ME Host the service started being provided.

This implies associated traffic rerouting between ME Hosts.

- No relocation at all

  At target ME Host, either the MEC Application is not running (e.g. because of the defined Application geographical scope) or an already existing instance is able to provide the same service, without need for any communication with previous instance (e.g. stateless MEC Application). There is no state or application relocation.

B4.2.3 UE's mobility detection

In all relocation scenarios, and without considering any interaction with EPC control plane (e.g. S1-MME), UE's mobility is only detected and the new serving MEC Hosts is only known once the UE attaches to the new eNB and its traffic is identified. Even if the UE is aware of eNB change, it is not aware of the possible MEC Host change. This way, any needed relocations can only be triggered after UE's arrival at the new MEC Host.

UE's mobility detection can be done at two levels:

1. By the ME Hosts, by observing traffic send to/from a new IP/TEID (Data Plane level)

   As a result, the detecting ME Host may proactively query neighbouring ME Hosts about source IP (which ME Host was previously handling that IP).

   The previous ME Host handling the UE, may query/notify all running Applications or only the ones that stated the existence of state associated to that IP, about relocation needs.

2. By the MEC Applications themselves, when applicable traffic rules deliver traffic to them

   As a result, and if required (stateful Applications), will require the hosting ME Host to provide its mobility services.

In addition, UE's mobility may be detected by the UE itself. If eNB change is notified to local Application counterpart, this one may notify the MEC System, via the "LCM Proxy", about the possible need for relocation actions.

B4.2.4 Relocation need detection

In parallel to that, the need for relocation (instance or state) must be identified. This can be known, and also considering previous scenarios:

1. Previously by the ME Host as part of the MEC Application description and communicated to the ME Host at instantiation time

2. At Application execution time (state is created and exists, associated to certain UEs) and communicated to the ME Hosts via an appropriate API:
    a. Inform the MEC System, what IP addresses need to be monitored regarding mobility aspects (proactive)
    b. Whenever a new IP is detected by an Application instance, it may ask the hosting ME Host to trigger the required relocation actions (reactive)
3. Unknown by the ME Host

B4.2.5 Proposed processes

Both aspects, UE's mobility detection and need for relocation, must be considered together. One approach consists in delegating to MEC Applications the identification of UE's mobility and the identification of relocation needs. Based on that, the following proposals are made.

A. Proposed process for Generic MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. Configured traffic rules will extract and deliver UE's traffic to applicable MEC Applications
3. Upon detecting traffic from a new IP, stateful MEC Applications will query the hosting ME Host if Application's state exists in another ME Host, expressing relocation actions
4. ME Host queries neighbouring ME Hosts about state related to that IP and MEC Application (IP, AppID)
5. If existing, previous ME Host, notifies, based on AppID, local Application instance about relocation needs
6. As a consequence, that MEC Application instance requests the ME Host to:
    a. Establish a tunnel to the new MEC Host for traffic redirection or
    b. Provide new MEC Application IP address, for managing state relocation
7. Depending on the previous:
    a. Traffic rules at the new ME Hosts are changed accordingly
    b. Application instances exchange state and service for that UE continues at the new Application instance

A similar behaviour may be achieved but with Applications communicating with the corresponding Manager, running at the respective ME Platform Manager. The Application Manager may work as a central point for all instances, coordinating with the MEC System entities, relocation needs and actions. No details for this option are provided.

B. Proposed process for UE's specific MEC Applications:

1. UE handovers to a new eNB, served by a new MEC Host
2. ME Host Data Plane detects a new IP/TEID

3. ME Host queries neighbouring ME Hosts about specific MEC Applications running for that UE related to that IP
4. If existing, previous ME Host, notifies, the local Application instance about UE's mobility
5. As a consequence that MEC Application instance requests the ME Host to:
    a. Establish a tunnel to the new MEC Host for traffic redirection or
    b. Proceed with Application instance relocation to the new ME Host

B4.2.6 Mobility API

Besides the proposed relocation mechanisms, other solutions are possible. The existence of an API for Applications to communicate with ME Hosts is needed and additional options may be provided, giving place to the definition of other solutions. It is therefore proposed the following API features:

1. Apps to notify ME Host about UEs (IP addresses) to be monitored
2. Apps to request ME Host about UE's originating ME Hosts
3. Apps to request ME Host about originating App IP address
4. Apps to request ME Host about another hosting ME Host IP address
5. Apps to request ME Host to store information at local persistent storage
6. Apps to request ME Host to move/copy stored information to another ME Host
7. Apps to request ME Hosts LCM operations (Stop, Create)
8. ME Hosts to notify Apps about UE's mobility (IP address)
9. ME Host to notify an App about the arrival of stored information (AppID)
10. ME Host to query other ME Hosts about UE handling

B5 Optimal scaling and load balancing based on MDP models

B5.1 Introduction

We address the problem in which the enterprise has to find dynamic policy of VM deployment, displacement and scheduling of incoming VNF tasks, as a function of the set of costs and the number of VNF instances currently being served in the already active VMs.

An enterprise, which decides to virtualise any of their network services needs first to deploy ("build") the corresponding service, in order to be able to run VNF instances.

The deployment process involves having leased a VM and loading on it the corresponding software.

Once deployed, the enterprise has to pay per time of usage for the leased VMs, regardless of the load.

The process of deployment can take time, and an additional deployment cost. Hence, having idle resources is undesired. On the other hand, the delay involved with the deployment or lack of space for the new tasks can cause some VNF instances to be rejected from running, thus inflicting a profit loss to the enterprise. Note that in some cases, the displacement ("destroy") operation, i.e., the process of releasing VMs can also incur a cost.

An additional basic demand is to facilitate scaling. For simplicity, consider an enterprise which needs to run networking tasks of only one type. Hence, we assume all VMs are similar and able to run similar VNF tasks (e.g., flows a firewall handles). The total number of VMs allocated is dynamically increased (via a scale out operation) or decreased (scale in operation) according to the demand from this network service. In scale out/in operations, we increase/decrease (i.e., deploy/displace) the number of VMs that are allocated, respectively. Note that we do not consider the scale up and down operation, which are less common in NFV use cases. (In scale up/down operation we can, for example, add/remove CPU cores to a given VM).

Clearly, increasing the number of VMs would disperse the total load and improve the performance of each VM, yet would imply having leased more costly resources. Accordingly, our scaling decision should minimize the number of leased VMs, while keeping with the application required SLA.

Hence, the decision whether to scale out or in remains an important problem that needs to be addressed, especially in dynamic scenarios.

In conjunction with the scaling decisions, we are also facing a load balancing challenge, steering the traffic flows to the different VMs and balancing the load between them. In this study, we tackle both the scaling decision and the load balancing strategy as a single problem.

Going back to the simple single-VNF enterprise scenario, the load balancing will merely amount to having equally loaded VMs. Hence the trade-off in this case means the average load on a VM against

the number of deployed VMs. Yet, having in mind deployment time and cost, this trade-off still represents a significant challenge as the optimal policy derivation is not straightforward.

To this end, we model the problem by queuing system with a dynamic (but limited) number of queues; each queue stands for a VM running VNF instances. We assume that VNF tasks arrive with constant average rate. Upon each arrival event, the Decision Maker (DM) decides to which VM the arriving task should be scheduled, or whether it should be rejected. The VM deployment decisions are made at arrival times, as long as the limit of active VMs is not reached.

At departure from a queue (i.e., running of VNF task ends) which has been left empty, DM decides whether to keep that queue alive or to destroy it.

In order to reflect the load at busy VMs we introduce a delay cost which (not necessarily linearly) increases with the load.

The maximal number of running task on a single VM is limited by borderline number which indicates a performance fall beyond the SLA demands and, hence, should never be exceeded.

Deployment and displacement operations of VM consume additional fixed costs. Once a VM was deployed, the enterprise is charged with fixed per-unit of time reservation and maintenance cost. We term this keep-alive or the holding cost.

The DM aims to find a policy which will maximize the total income in the long run.

While the set of costs described above implies no trivial policy could exist, some of the parameters have contradicting impacts which may dictate certain properties of the policy.

For example, in the case where the delay cost function sharply increases with a load, the DM will attempt to deploy as many VMs as possible.

On the other hand, in the case where keep-alive cost is comparatively high, the DM may prioritize minimization of the active VMs number.

A distinctive impact has a VM deployment time, which we separately explore.

In this work, we treat the dynamic VM deployment and VNF tasks scheduling problem by introducing a control model based on MDP formulation. The solution of the MDP provides the optimal policy.

Once applied, the policy potentially reveals the average number of active VMs and average number of tasks in the task queue of each VM.

This allows the enterprise to assess the demands and to plan ahead.

Moreover, the solution to the MDP is expressed by value function which indicates what are the costs and revenues the enterprise will receive in the long run, for a given scenario along with its set of parameters.

Complete details of this work are available at: [18]. In the following we present some of the numerical results of our work.

B5.1.1 Numerical results

In the following, we present simulation results which both show additional threshold properties of optimal policies, and provide a comprehensive study of the value functions and the corresponding policies. We explored a system with five identical queues.

Figure 66 demonstrates the impact of the keep-alive queue cost. The simulation was run with negligible delay cost, rejecting fine equal to 10, $\lambda = 4$ (VNF tasks arrival rate) and $\mu_i = 1, \forall i$ (servers total processing rate). Buffer limit was 4 tasks. One sees that there is a small region of keep-alive queue cost, where the number of active queues declines. The trade-off in this simulation is the accumulatively paid fine against the accumulatively paid keep-alive queue cost. As long as the queues are identical, there is no preference which queue should be kept idle. That is why the number of active queues declines to zero within a very small interval of keep-alive queue cost. Hence, once it is more affordable to pay the fines by rejecting all incoming tasks rather than maintaining the VMs (i.e., the queues) the queues stay idle at all times.

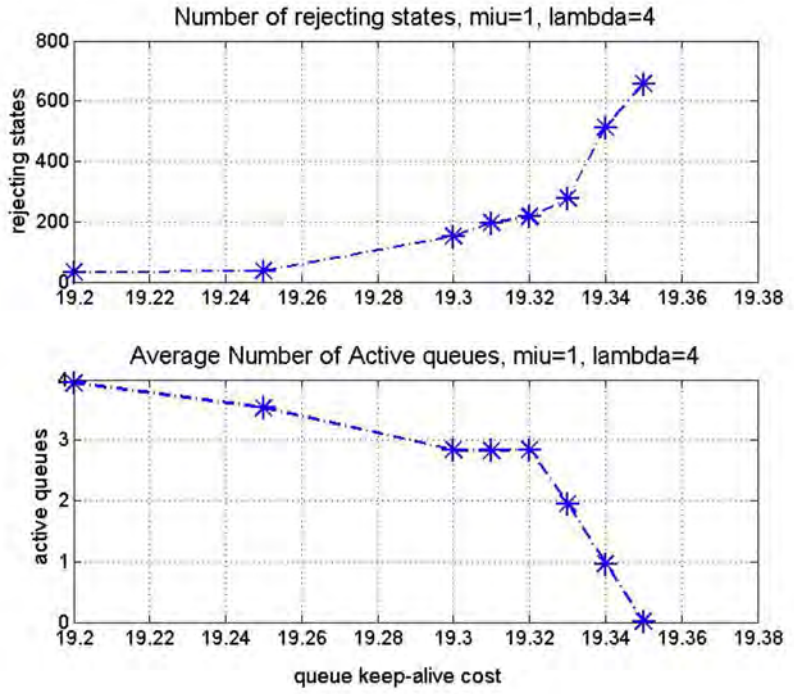


Figure 66: Impact of the allocated VNF cost

Figure 67 shows simulation of the delay cost h, while the keep-alive cost was fixed equal to 1, the buffer size of each queue was 6. The arrival intensity was 4.75 and $\mu_i = 1, \forall i$. Rejecting fine was equal to 10. The delay constants were $\eta_1 = 1, \eta_2 = 1.8, \eta_3 = 2.5, \eta_4 = 3.5, \eta_5 = 4.5, \eta_6 = 5.5$. By the cost model, one expects that the tasks will be balanced in the queues. Observe that in this simulation, the keep-alive cost was low enough to allow that. Indeed, all queues were active while

the total average number of tasks declined with the delay cost. Observe that in this simulation, the interval of the varying cost was significantly larger. This stems primarily from the fact that η_1 is small. Note that the lower graph in both simulations shows the number of rejecting states, out of the states-space.

Additional thresholds can be seen in "build" and "destroy" actions. For example, if The building cost β and/or the destroy cost ψ are high if compared to keep-alive cost, the optimal policy acts to leave all queues active, even if empty.

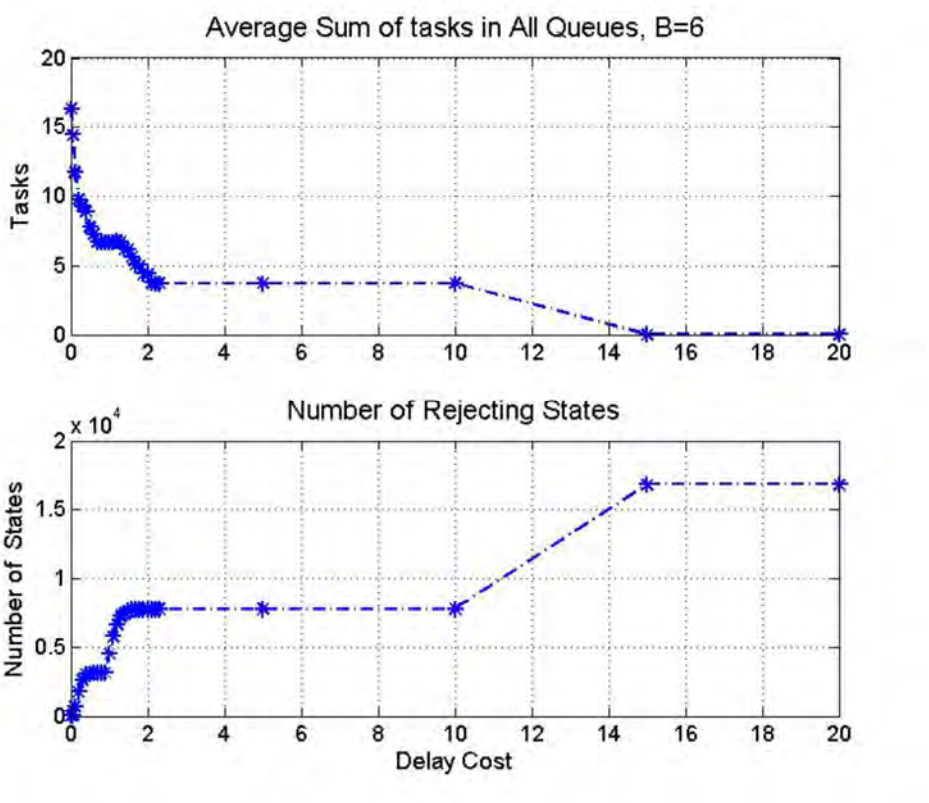


Figure 67: Impact of delay cost

To conclude, the set of system parameters will determine if the optimal policy will prioritize balanced and mostly active queues or a small number of active and comparatively busy queues. The values of β and ψ will determine if keeping alive the empty queues is economically reasonable.

B5.2 Machine learning techniques for the LCM for containerized workload

B5.2.1 Introduction

To allow multi-tenancy of containerized applications, typically, each tenant/application is allocated VMs as the underlay infrastructure. Then, the containerized application scales out and in within that underlay. Accordingly, we are now facing two scaling processes, that is, (i) of the underlay VMs, and (ii) of the containerized applications.

For the first process, namely the VM Scaling, we have our MDP model (see section B5.2) that can be extended to tackle containerized workload. Such extension is pending investigation in the third year.

In this work, we focus on the second process of the scaling of the containers. Here we devise through machine learning techniques, a mechanism that automatically scales the containerized application based on infrastructure metrics. We further demonstrate that our mechanism dramatically outperforms Kubernetes. Specifically, we demonstrated that a containerized video streaming application suffers from high delays (due to lack of resources) while being managed by Kubernetes. On the other hand, our mechanism manages to scale the application in a timely manner while providing the appropriate resources to obtain the required application QoS (i.e., packet latency).

B5.2.2 Kubernetes scaling mechanism

Typically, the LCM operation of containerized applications is handled by the Containers Orchestration Engine (COE). Since Kubernetes is the most dominant COE of today, in this work we adopted it as our COE of choice. Indeed, Kubernetes manages the scaling operation. However, in this work, we demonstrated that Kubernetes suffers from severe drawbacks. Specifically, Kubernetes v1.57 triggers scaling only based on CPU utilization or based on application metrics (via APIs). Indeed, this was handled in Kubernetes v1.6, where other infrastructure metrics may trigger scaling. Yet a function that considers all infrastructure metrics is not defined and it is not clear how to combine those metrics to trigger scaling operation.

Accordingly, in the project's second year we tackle this issue by applying machine learning techniques to control the scaling decisions.

B5.2.3 Machine learning based scaling

Our goal is to devise a machine learning mechanism that receives as an input samples of the resource utilizations by each container and predicts application performance.

As a learning sequence, we assume that our system receives indications on the application performance. Based on this learning metric, we aim at devising a mechanism that learns the optimal

scaling policy based on infrastructure metrics that minimize the consumed resources (minimize the number of deployed containers) while maintaining the application's required performance.

B5.2.4 Offline implementation – video streaming application

Aiming at devising the optimal online machine learning scheme, we first analyse and solve this problem in an offline scenario.

To that end, we consider a containerized video streaming server. Specifically, we containerized this server especially for this work (available at Docker Hub under ruvenmil/mycont2). Next, we ran this server under different deployment configurations, i.e., with multiple number of clients as well as with multiple numbers of deployed pods (Kubernete's minimal deployment unit). For each configuration, we recorded the packet delay. Note that for video streaming the packet delay constitutes a representative indication for the video quality. Our setup topology is given in Figure 68.

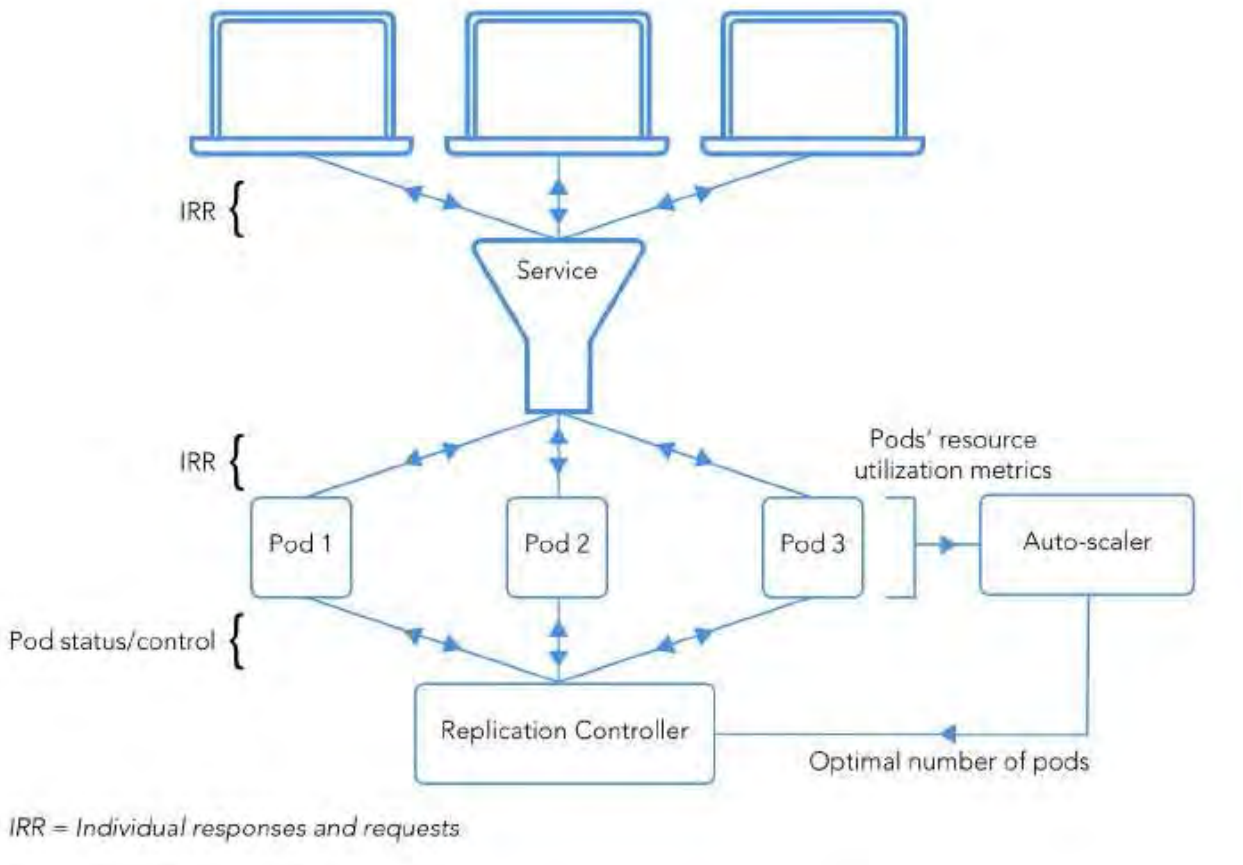


Figure 68: setup topology

In addition to the packet delays, each record also includes the infrastructure utilization, namely, CPU, memory and bandwidth consumption.

With all data at hand, we employed several prediction mechanisms, including: random forest and gradient boosting, and obtained a predictor for the video quality based on the infrastructure resource consumption. Next, we triggered the scaling operation based on this predictive function. Utilizing that function obtained a much better video quality and initiated scaling just on time.

B5.2.5 Implementation results

In the following we compare our auto-scale algorithm with the default scheme of kubernetes.

Packet error measurement:

We can see in Figure 69 that the suggested algorithm has better performance and less error burst (depicted in dark blue).

Kubernetes: suggested algorithm

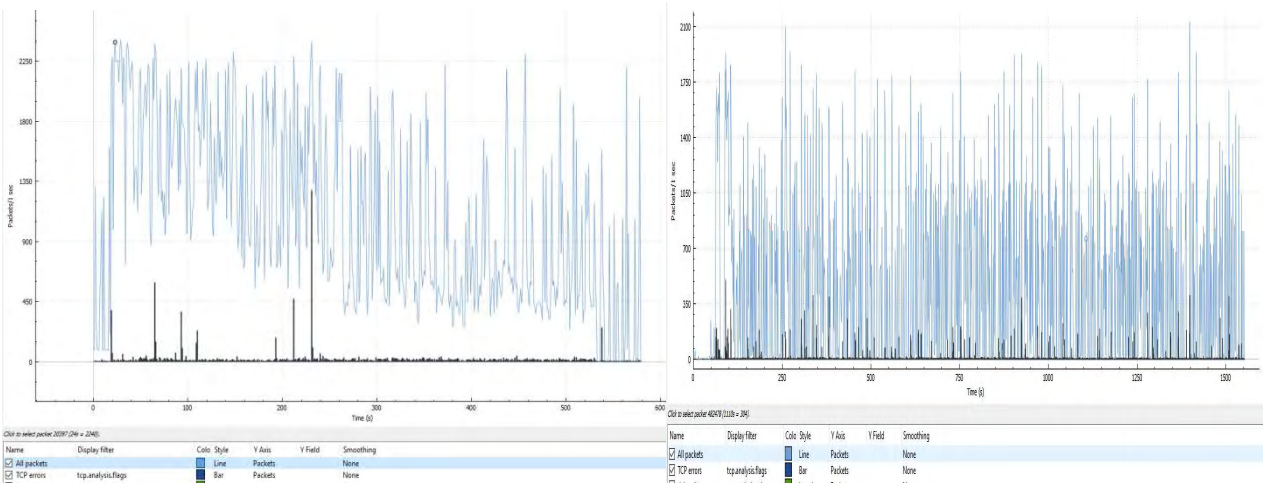


Figure 69: packet errors

Throughput measurement:

Figure 70 depicts the throughput measurement results. One can see that the achieved throughput is higher with our suggested algorithm compared to with the of the shelf Kubernetes mechanism.

Kubernetes: suggested algorithm

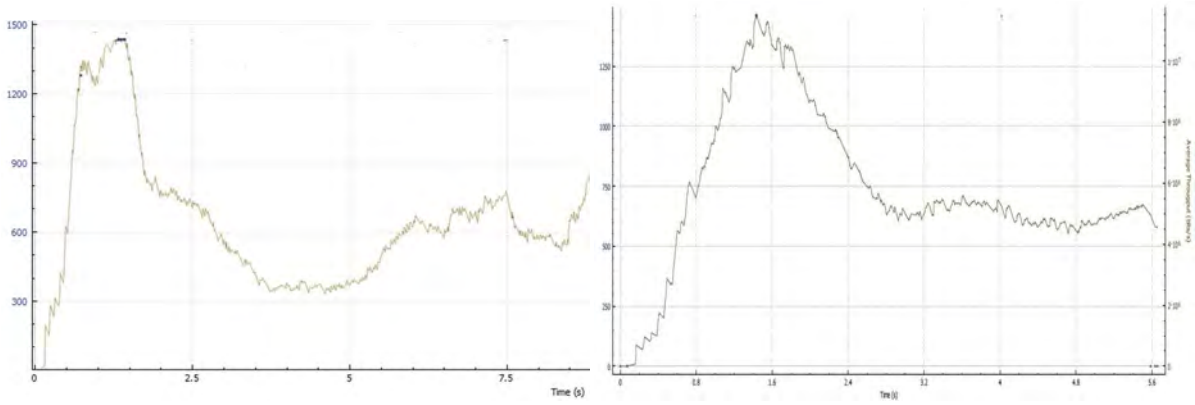


Figure 70: Throughput measurements

B6 Load Balancing as a Service

B6.1 Load Balancing principles

Efficient load balancing is an area of paramount importance for identifying possible flaws and limitations of the SUPERFLUIDITY architectural components, most of which are currently under heavy development, undergoing continuous modifications to improve performance.

The actual definition of load balancing, as the distribution of network or application traffic across a cluster of servers, consequently leading to improved responsiveness and increased service availability, is provided in Deliverable D5.3. In addition, Deliverable D2.1 identified the load balancing-dependent use cases, where specific performance must be established to fulfil the pre-defined scalability and high availability requirements, while the specific requirements of Load Balancer as a functional block were further analyzed in Deliverable D2.2. Last but not least, an additional analysis of certain load balancing approaches can also be found in Annex A, section A3.5 of this deliverable. With load balancing references being omnipresent in the majority of technical deliverables, one may identify the significance of this functionality for the SUPERFLUIDITY platform as a whole.

B6.2 HA Proxy and LBaaS in OpenStack

The project's virtual infrastructure manager (VIM) of choice, OpenStack, offers certain open-source options for implementing the infrastructure load balancing capability, originally through HAProxy, a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler, currently integrated from an upstream open-source project. OpenStack fully supports HAProxy deployment in its controller nodes where each instance of the software configures its front end to accept connections only to the virtual IP (VIP) address, while the backend is a list of all available IP addresses that may need load balancing of their ingress traffic. However, the most effective load balancing service of OpenStack, currently integrated inside Neutron is no other than Load Balancing as a Service (LBaaS), which allows both proprietary and open-source load balancing technologies to handle the excessive traffic load. As stated in [18], LBaaS builds on top of HAProxy by leveraging agents that control HAProxy configuration and manage the HAProxy daemon, is therefore somehow considered as an enabling module that introduces additional functionality to the core load balancing mechanism of OpenStack. OpenStack is also introducing carrier-grade load balancing backend mechanism through Octavia [19].

B6.3 Citrix Netscaler ADC and MAS

The increased demands of contemporary networking environments in terms of traffic, necessitate the evaluation of the proposed SUPERFLUIDITY architecture paired with commercial, carrier-grade

load balancing options. Citrix NetScaler ADC [20] is an application delivery controller that provides flexible delivery services for traditional, containerized and microservice applications from any cloud or private datacenter, and was evaluated as part of SUPERFLUIDITY. As covered in Annex A, section A3.5 of this deliverable, NetScaler ADC can be integrated into the NFV architecture, working in parallel with all existing MANO entities, such as VIM and the SDN Controller.

NetScaler ADC is operated through a dedicated Element Manager, namely the NetScaler Management and Analytics System (MAS) [21] NetScaler MAS integrates using standard APIs with OpenStack, translating necessary messages to the RESTful APIs supported by NetScaler ADC. It facilitates administrators to monitor, automate and manage network services for scale-out application architectures with ease, and provides application-level integration with external orchestration systems.

Integrating MAS with OpenStack

The overall evaluation process conducted as part of SUPERFLUIDITY, involved a direct comparison of the capabilities of open-source load balancing options already integrated in vanilla versions of OpenStack (HAProxy) against NetScaler ADC. This process also required deploying and validating NetScaler ADC and MAS in a dedicated NFV Lab, thus verifying that certain enhancements introduced in these products as well as the NetScaler LBaaS driver [26] [27] are working properly.

As stated in previous paragraphs, for being able to use NetScaler ADC instead of the integrated load-balancing solutions of OpenStack, certain modifications are needed in Neutron LBaaS module. In particular, Citrix engineers developed an LBaaS plugin which can be deployed in Neutron and implements all the LBaaS driver CRUD APIs for operating on OpenStack VIPs, Pools, Pool Members and Health Monitor entities. The integration consists of a driver class configured in the Neutron config file (neutron.conf), and the accompanying unit tests, while its abstract functionality is shown in Figure 71.

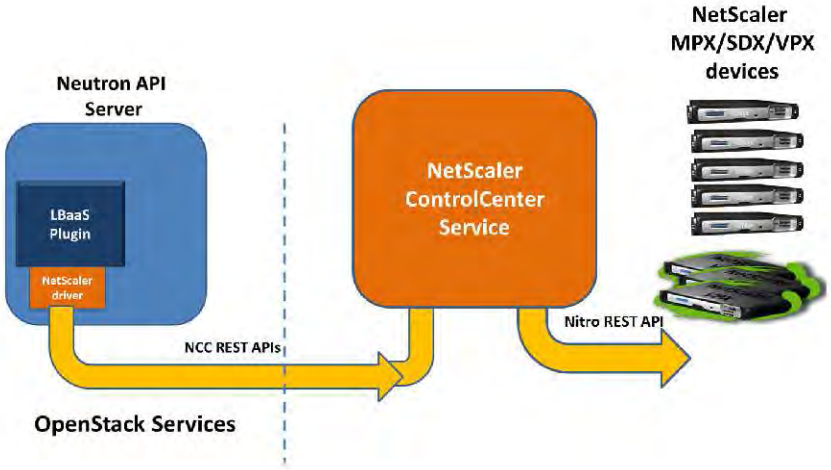


Figure 71: NetScaler LBaaS Integration

The NetScaler LBaaS integration consists of a driver class that implements the Neutron LBaaS driver which calls the NetScaler Control Center (NCC) service using NCC REST APIs. NCC is a separate service that runs outside of OpenStack infrastructure, and is deployed as a "virtual appliance" on supported hypervisor platforms (KVM/ESX/XenServer) for ease of setup. Since Release 11.1, NCC is integrated in NetScaler MAS, which is now in charge with all NetScaler resource and tenancy management tasks, as well as NetScaler device configuration, allowing the driver component in OpenStack to remain lightweight, simpler to maintain and easier to evolve going forward as the Neutron service evolves. During SUPERFLUIDITY validation process, a full-scale NetScaler MAS node was deployed and integrated with the existing OpenStack Platform, as follows.

OpenStack Neutron LBaaS plugin includes a NetScaler driver that enables OpenStack to communicate with the NetScaler MAS. OpenStack uses this driver to forward any load balancing configuration done through LBaaS APIs, to the NetScaler MAS, which creates the load balancer configuration on the desired NetScaler instances. OpenStack also uses the driver to call NetScaler MAS at regular intervals to retrieve the status of different entities (such as VIPs and Pools) of all load balancing configurations from the NetScaler ADCs. NetScaler driver software for OpenStack platform is bundled along with the NetScaler MAS. To download and install the drivers, it is necessary to first install NetScaler MAS and launch the application.

B6.4 NetScaler MAS Installation

For consistency reasons NetScaler MAS was installed on a Linux KVM server, after verifying that all hardware virtualization extensions and ***virsh***, a command line tool for managing virtual machines, were available. As described in [35], after obtaining the necessary image files the installation process includes, navigating to the folder where the compressed file is saved, using the tar command to untar

the NetScaler MAS image, verify that a domain disk image (.qcow2 file format), a domain XML file and the necessary checksum file were present, editing the XML file for specifying the necessary networking attributes, using ***virsh*** to define the VM attributes through the recently added XML file and initiating NetScaler MAS by entering the following command:

virsh start [<DomainName> | <DomainUUID>]

Certain configuration steps for the NetScaler MAS were also needed, after the previous process is concluded and the service became available. In particular it was necessary to:

- Login to the NetScaler MAS node
- Type shell > networkconfig to configure the management IP address
- Complete the initial network configuration by adding information regarding Netmask and GW IP
- Execute the deployment script by typing the following command in the shell prompt # deployment_type.py
- Restart the server and login to the GUI

B6.5 NetScaler Driver Software Installation

It is possible to install NetScaler driver on OpenStack via NetScaler MAS GUI. After logging in, click Downloads and download the latest NetScaler bundle .tar file to a temporary directory of the OpenStack Controller. The bundle includes LBaaS V2 drivers for Openstack Liberty/Mitaka/Newton releases along with the Heat plug-ing. Extract the files from the NetScaler driver tar, navigate in to the OpenStack <Release Name> folder and execute the following command to install the driver and specify the NetScaler MAS IP address:

./install.sh --ip=<NetScaler_MAS_IP> --password=<password> --protocol=<protocol> --neutron-lbaas-path <neutron-lbaas-directory-path>

B6.6 Registering OpenStack with NetScaler MAS

OpenStack information needs to be registered on the NetScaler MAS. The process includes specifying the OpenStack controller IP address and cloud administrative user credentials, and also the OpenStack NetScaler driver user credentials. It is also possible to later specify the same login credentials in the NetScaler_driver section of the Neutron configuration file (neutron.conf) so that NetScaler driver in OpenStack can connect to NetScaler MAS during LB configurations.

After OpenStack and NetScaler MAS are registered with each other, both can talk to each other. Also, OpenStack users can use their existing credentials in OpenStack to log on to the NetScaler MAS user interface to check how their LB configurations are performing in NetScaler [35].

B6.7 Adding OpenStack Tenants in NetScaler MAS

Provided that OpenStack and NetScaler MAS are now interconnected through the previous registration process, it is possible to create a Tenant in OpenStack using the NetScaler MAS. Simply

- Navigate to **Orchestration** > **Cloud Orchestration** > **OpenStack** > **OpenStack Tenants**, and then click **Add**.
- In **Add OpenStack Tenants** page, click **+Add**, and then select the OpenStack tenant.
- Click **OK**.

B6.8 Provisioning NetScaler VPX instance in OpenStack

Download the required NetScaler instance image from the Citrix download page, and upload it on Glance, the OpenStack Imaging service. Having an image available on Glance allows you to configure a NetScaler instance on-demand when assigning the instance to the tenant.

To auto-provision the NetScaler VPX devices on OpenStack

1. In NetScaler MAS, navigate to **Orchestration** > **Cloud Orchestration** > **OpenStack**.
2. Click **Deployment Settings**.
3. Set the necessary parameters:
   - Management Network
   - Profile Name
   - Licences
   - NetScaler VPX image in Glance
   - Proxy settings

The overall process of NetScaler MAS integration with OpenStack Workflow is shown in Figure 72.

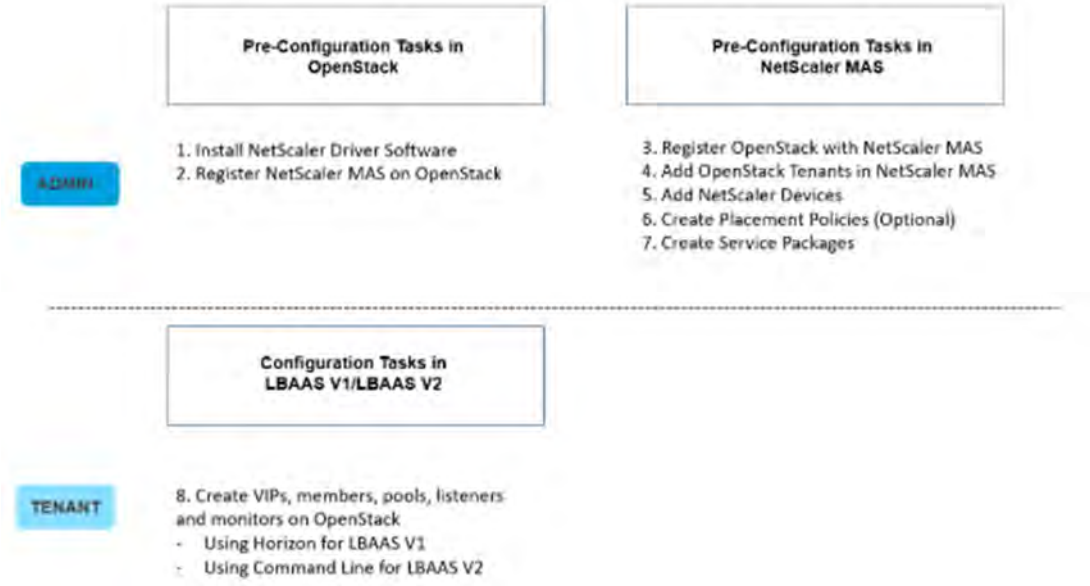


Figure 72: NetScaler MAS - OpenStack Integration Workflow [36]

The above outcomes will be integrated with the SUPERFLUIDITY platform as part of WP7 activities.

ANNEX C: Automated Security Verification Framework – Task 6.3

C1 Debugging P4 programs with Vera

See attached: PAPER – 2: Debugging P4 programs with Vera

C2 Equivalence and its applications to network verification

See attached: PAPER – 3: Equivalence and its applications to network verification

References

[1] "NeMo: An Application's Interface to Intent Based Networks" http://nemo-project.net/

[2] "High-level VNF Descriptors using NEMO" https://datatracker.ietf.org/doc/draft-aranda-nfvrg-recursive-vnf/

[3] ETSI NFV ISG, "Network Functions Virtualisation (NFV); VNF Packaging Specification", ETSI GS NFV-IFA 011 V2.1.1 (2016-10) pdf link

[4] ETSI NFV ISG, "Network Functions Virtualisation (NFV); Network Service Templates Specification", ETSI GS NFV-IFA 014 V2.1.1 (2016-10) pdf link

[5] L. Chiaraviglio, F. D'Andreagiovanni, G. Sidoretti, N. Blefari-Melazzi, S. Salsano, "Optimal Design of 5G Superfluid Networks: Problem Formulation and Solutions", 21st Conference on Innovation in Clouds, Internet and Networks (ICIN 2018), February 20-22, 2018, Paris, France

[6] L. Chiaraviglio, L. Amorosi, S. Cartolano, N. Blefari-Melazzi, P. Dell'Olmo, M. Shojafar, S. Salsano, "Optimal Superfluid Management of 5G Networks", 3rd IEEE Conference on Network Softwarization, NetSoft 2017, 3-7 July 2017, Bologna, Italy

[7] M. M. Tajiki, S. Salsano, M. Shojafar, L. Chiaraviglio, B. Akbari, "Energy-efficient Path Allocation Heuristic for Service Function Chaining", 21st Conference on Innovation in Clouds, Internet and Networks (ICIN 2018), February 20-22, 2018, Paris, France

[8] V. Frascolla, J. Englisch, L. Chiaraviglio, S. Salsano, S. Barberis, V. Palestini, A. De Domenico, E. Calvanese Strinati, K. Takinami, K. Yunoki, K. Sakaguchi, T. Haustein, "Millimeter-waves, MEC, and network softwarization as enablers of new 5G business opportunities", 1st Workshop on Economics and Adoption of Millimeter Wave Technology in Future Networks at IEEE WCNC 2018, 15-18 April 2018, Barcellona, Spain

[9] L. Chiaraviglio, N. Blefari-Melazzi, C.F. Chiasserini, B. Iatco, F. Malandrino, S. Salsano, "An Economic Analysis of 5G Superfluid Networks", 18th IEEE International Conference on High Performance Switching and Routing (IEEE HPSR), 18-21 June 2017, Campinas, Brazil

[10] RDCL 3D Home Page, https://github.com/superfluidity/RDCL3D

[11] Salsano S, et al "RDCL 3D, a Model Agnostic Web Framework for the Design of Superfluid NFV Services and Components", 3rd IEEE International Workshop on Orchestration for Software Defined Infrastructures, O4SDI at IEEE NFV-SDN conference, Berlin, 6-8 November 2017 (arXiv preprint: https://arxiv.org/pdf/1702.08242)

[12] ETSI GR NFV-IFA 015, "NFV MANO Release 2; Report on NFV Information Model" V2.1.1 (2017-01)

[13] ETSI GS NFV-IFA 011: "NFV MANO, VNF Packaging Specification". V2.1.1 (2016-10)

[14] "TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0", Committee Specification Draft 03, OASIS, 2016

[15] “TOSCA Simple Profile in YAML Version 1.0”, OASIS, 2016

[16] J E. Kohler, et al., “The Click modular router,” ACM Transactions on Computer Systems (TOCS), vol. 18, no. 3, 2000

[17] OSM Information Model, https://osm.etsi.org/wikipub/index.php/OSM_Information_Model

[18] M. Shifrin, E. Biton, and O. Gurewitz. Optimal control of VNF deployment and scheduling. In Science of Electrical Engineering (ICSEE), IEEE International Conference on the, 2016.

[19] OpenStack Octavia [Online]. Available: https://wiki.openstack.org/wiki/Octavia [Accessed 28/06/2017]

[20] Citrix NetScaler ADC [Online]. Available: https://www.citrix.com/products/netscaler-adc/

[21] Citrix NetScaler MAS [Online]. Available: https://www.citrix.com/products/netscaler-management-and-analytics-system/

[22] Citrix Docs, “NITRO API”, https://docs.citrix.com/en-us/netscaler/12/nitro-api.html

[23] Citrix Docs, “Integrating NetScaler MAS with OpenStack Platform”, http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform.html

[24] Citrix Docs, “Load balancing algorithms”, https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-customizing-algorithms.html

[25] Citrix Docs, “About Persistence”, https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-persistence/persistence.html

[26] OpenStack Neutron/LBaaS Project [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS/NetScaler

[27] OpenStack LBaaS Netscaler Driver Repository [Online]. Available: https://github.com/openstack/neutron-lbaas/tree/master/neutron_lbaas/drivers/netscaler

[28] “SRv6: Network as a Computer and Deployment use-cases” https://pc.nanog.org/static/published/meetings/NANOG71/1445/20171005_Dawra_Segment_Routing_Ipv6_v1.pdf

[29] http://www.segment-routing.net/ietf/

[30] “Segment Routing IPv6 for Mobile User-Plane”, draft-ietf-dmm-srv6-mobile-uplane-00

[31] “Study on User-plane Protocol in 5GC”, 3GPP #: 29.892

[32] A. AbdelSalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, L. Veltri, “Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-based NFV Infrastructure”, 3rd IEEE Conference on Network Softwarization, NetSoft 2017, 3-7 July 2017, Bologna, Italy

[33] F. Clad, C. Filsfils, P. Camarillo, D. Bernier, B. Decraene, B. Peirens, C. Yadlapalli, X. Xu, S. Salsano, A. AbdelSalam, G. Dawra, “Segment Routing for Service Chaining”, Internet Draft, draft-clad-spring-segment-routing-service-chaining, Work in progress, October 2017

[34] C. Filsfils, et al, “SRv6 Network Programming”, Internet Draft, draft-filsfils-spring-srv6-network-programming, Work in progress, October 2017

[35] Citrix Docs, “Installing NetScaler MAS on KVM” [Online]. Available: http://docs.citrix.com/en-us/netscaler-mas/12/deploy-netscaler-mas/install-mas-on-kvm.html

[36] Citrix Docs, “Integrating NetScaler MAS and OpenStack - Preconfiguration” [Online]. Available: http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform/preconfiguration-tasks-mas-openstack.html

[37] “C4.5 algorithm”, https://en.wikipedia.org/wiki/C4.5_algorithm

PAPER – 1: Optimizing NFV Chain Deployment Through Minimizing the Cost of Virtual Switching

PAPER – 2: Debugging P4 programs with Vera

PAPER – 3: Equivalence and its applications to network verification

ANNEX: Internal deliverable I6.3

ANNEX: Internal deliverable I6.3b