SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE I5.3:

PLATFORM API DESIGN AND IMPLEMENTATION

Deliverable Type:	Report
Dissemination Level:	CO
Contractual Date of Delivery to the EU:	01/05/2017
Actual Date of Delivery to the EU:	01/05/2017
Workpackage Contributing to the Deliverable:	WP5

Editor(s):	Christos Tselios (CITRIX)
Author(s):	Christos Tselios (CITRIX), Luis Tomas Bolivar (REDHAT), Stefano Salsano (CNIT), Claudio Pisa (CNIT), Francesco Lombardo (CNIT), Juan Manuel Sanchez (Telcaria)
Internal Reviewer(s)	George Tsolis (CITRIX), Elisa Rojas (Telcaria)

Abstract:	This deliverable provides an overview of the current status of the project's platform API design and implementation

Keyword List: Unified Platform, Interfaces, API functions

VERSION CONTROL TABLE			
VERSION N.	PURPOSE/CHANGES	AUTHOR	DATE
V1	First Version	Christos Tselios	03/05/2017
V1.1	Updated Version	Christos Tselios	12/05/2017

INDEX

GLOSSARY 6
1 INTRODUCTION 7
1.1 DELIVERABLE RATIONALE 7
1.2 EXECUTIVE SUMMARY 7
2 CURRENT UNIFIED SUPERFLUIDITY PLATFORM 8
2.1 VIRTUALIZATION PLATFORM 8
2.1.1 Kernel-based Virtual Machine 8
2.2 VIRTUAL INFRASTRUCTURE MANAGEMENT 10
2.2.1 OpenStack 10
2.2.2 Compute 11
2.2.3 Networking 11
2.2.4 Load Balancing 17
2.2.5 Heat 19
2.2.6 Mistral 21
2.2.7 Telemetry 22
3 COMMUNICATION INTERFACES AND API DESIGN 25
3.1 ETSI NFV ARCHITECTURE 25
3.2 NETWORK SERVICE INTERFACES 26
3.3 MANAGEMENT AND CONFIGURATION VNFS 27
3.4 VIRTUAL RESOURCE INTERFACES 28
3.5 NFVI FUNCTIONS 29
3.6 EXTERNAL AND INTERNAL INTERFACES OF ETSI NFV ARCHITECTURE 30
3.7 NFV ORCHESTRATOR API/FUNCTIONS 31
3.7.1 Top Level User-Manager API Implementation Challenges 32
3.7.2 Relation to NFV Information Model 32
3.7.3 VNF Packaging Specification 33
4 PLATFORM ORCHESTRATION 34
4.1 OSM RELEASE 0 34
4.2 OSM RELEASE ONE 36
4.3 OSM RELEASE TWO 43
5 SUPERFLUIDITY MECHANISMS, ALGORITHMS AND INNOVATIONS 44
5.1 RDCL 3D 44
5.2 SEFL AND SYMNET 48

5.2.1 OpenStack Neutron Configurations 50
5.3 NETWORK MODELLING LANGUAGE 50
6 CONCLUSION 52
7 REFERENCES 53

List of Figures

Figure 1: KVM Architecture 9
Figure 2: OpenStack Mitaka High Level Architecture 11
Figure 3: QoS Rules in Policies 14
Figure 4: Open vSwitch [6] 16
Figure 5: Load-Balancing-as-a-Service Architecture 19
Figure 6: Heat Orchestration Template 20
Figure 7: Mistral Workflow using YAML 22
Figure 8: OpenStack Ceilometer data collection mechanisms 24
Figure 9: OpenStack Ceilometer architecture and communication interfaces 24
Figure 10: ETSI NFV Reference Architecture diagram 26
Figure 11: ETSI NFV Network Service Reference Point 27
Figure 12: ETSI NFV VNF Reference Points 28
Figure 13: ETSI NFV Virtual Resource Reference Points 29
Figure 14: ETSI NFV NFVI Reference Point 30
Figure 15: ETSI MANO Architecture - Specifications and Reference Point Analysis 31
Figure 16: Open Source MANO - Release Zero 35
Figure 17: Open Source MANO - Release One 36
Figure 18: OSM Release One Login Screen 37
Figure 19: OpenStack Newton Deployment for OSM-VIM testing 38
Figure 20: OpenStack Newton - OSM Release One testbed 39
Figure 21: Validation Network Service deployment 39
Figure 22: VNFD Onboarding 40
Figure 23: NSD Inspection 40
Figure 24: NS Instantiation 41
Figure 25: Successfully Instantiating the NS from the OSM Node perspective 41
Figure 26: OpenStack Graph representing the validation NS deployment 42
Figure 27: OpenStack Network Topology representing the validation NS deployment 42
Figure 28: Network Service design using RDCL 3D GUI 45
Figure 29: Positioning the RDCL 3D in the ETSI MANO architecture 46
Figure 30: RDCL 3D Software Architecture 47
Figure 31: Overview of a NEMO project in the RDCL 3D tool 51
Figure 32: Editing a NEMO intent using the RDCL 3D tool 51

List of Tables

Table 1: SUPERFLUIDITY Dictionary. 6
Table 2: Heat Orchestration Template structure 21
Table 3: Common API Calls 32
Table 4: OSM Release Comparison 37

Table 5: SEFL Syntax 49

Glossary

(TO BE FILLED OUT IN THE FINAL VERSION OF THE DELIVERABLE)

SUPERFLUIDITY DICTIONNARY	
TERM	DEFINITION

Table 1: SUPERFLUIDITY Dictionary.

1 Introduction

1.1 Deliverable Rationale

The scope of this deliverable is to focus on the design and implementation of the platform's API. The API will be used by the architecture defined in WP3 to monitor platforms, and to instantiate, migrate and stop virtualized network services on the platform. The deliverable will further report on the integration of the Superfluidity platform's mechanisms and its optimal algorithm allocation into a unified platform.

1.2 Executive summary

This deliverable is mainly focused on presenting the overall work related to Task 3.3, which merges all different components of Superfluidity under a coherent, uniform platform and exposes an API to it to be used by the project's architecture developed in WP3. The work described in Deliverable 5.3 aims to include the integration of the network service dynamic and allocation algorithms, as well as the security considerations developed in Task 3.3. It will further concentrate on the definition of an API to monitor the performance of the platform and instantiate, stop and migrate virtualized network services. Finally, the task will implement such an API, and if applicable, integrate it into existing management frameworks such as OpenStack or OpenDaylight.

Before any development and integration activity begins, a careful analysis of the state of the art is necessary, in order to properly evaluate available components and tools that may be utilized by Superfluidity. In particular, the latest advances in relevant OpenStack projects are analysed and presented in this deliverable, namely OpenStack Compute and Neutron, HAProxy and Load Balancing-as-a-Service (LBaaS), Heat, Mistral as well as the OpenStack Telemetry. In addition to OpenStack, an analysis of the current status of the ETSI NFV Architecture is presented in Section III, emphasizing on the necessary network interfaces, management and configuration VNFs, the NFVI functions and the virtual resource interfaces that should be implemented towards an end-to-end solution.

Last but not least, this deliverable presents a detailed analysis of the certain Platform Orchestration tools, namely Open Source MANO, along with certain Superfluidity mechanisms, algorithms and innovations that will further enhance Superfluidity's vision of an efficient 5G architecture, such as RDCL 3D, Symnet together with SEFL and also NEtwork MOdelling (NEMO) language.

2 Current Unified Superfluidity Platform

2.1 Virtualization Platform

A hypervisor is a dedicated software or firmware component that can virtualize system resources by utilizing highly efficient and sophisticated algorithms, thus allowing multiple operating systems running on different Virtual Machines (VMs) to share a single hardware host [1]. The hypervisor is actually in charge of all available resources, which are allocated accordingly, making sure that all VMs run independently without disrupting each other. A node on which any hypervisor operates creating one or more virtual machines is often referred as host machine, while each of the virtual machines is called guest machine.

Hypervisors can be categorized as native/bare-metal (Type-1) or hosted (Type-2). Native hypervisors operate directly on the host machine's hardware to deploy and manage guest machines, while hosted hypervisors run on top of conventional operating systems, rendering the guest operating system an actual process of the host. Sometimes the distinction between the two types is not crystal clear; however, there is an apparent performance gain when using bare-metal hypervisor solutions. Specific performance requirements of the Superfluidity Virtualization Platform led towards adopting Kernel-based Virtual Machine (KVM) as the hypervisor of choice for experimenting onto, and evaluating all components and techniques currently under investigation.

2.1.1 Kernel-based Virtual Machine

Kernel-based Virtual Machine (KVM) [2] is a free, open-source virtualization solution, which enables advanced hypervisor attributes on the Linux kernel. It consists of a loadable kernel module, kvm.ko, which facilitates the core virtualization infrastructure and a processor-specific module kvm-intel.ko or kvm-am.ko for Intel and AMD processors, respectively. Upon loading the aforementioned kernel modules, KVM converts the Linux kernel into a bare metal hypervisor and leverages the advanced features of modern hardware, thus delivering unsurpassed performance levels [3].

By itself, KVM does not perform any emulation but exposes the /dev/kvm interface used by the user space host to set up the guest VM's address space and feed the corresponding simulated I/O. Besides performance, the integration with the operating system kernel provides KVM significant security and scalability benefits, especially since industry-standard x86 architecture processors and systems have grown increasingly more powerful in terms of processing, memory and I/O. Physical resources can be easily divided into sufficient pools of virtual resources that may handle larger workloads than before. Figure 1 presents the basic elements of the KVM architecture and their interaction through dedicated interfaces.

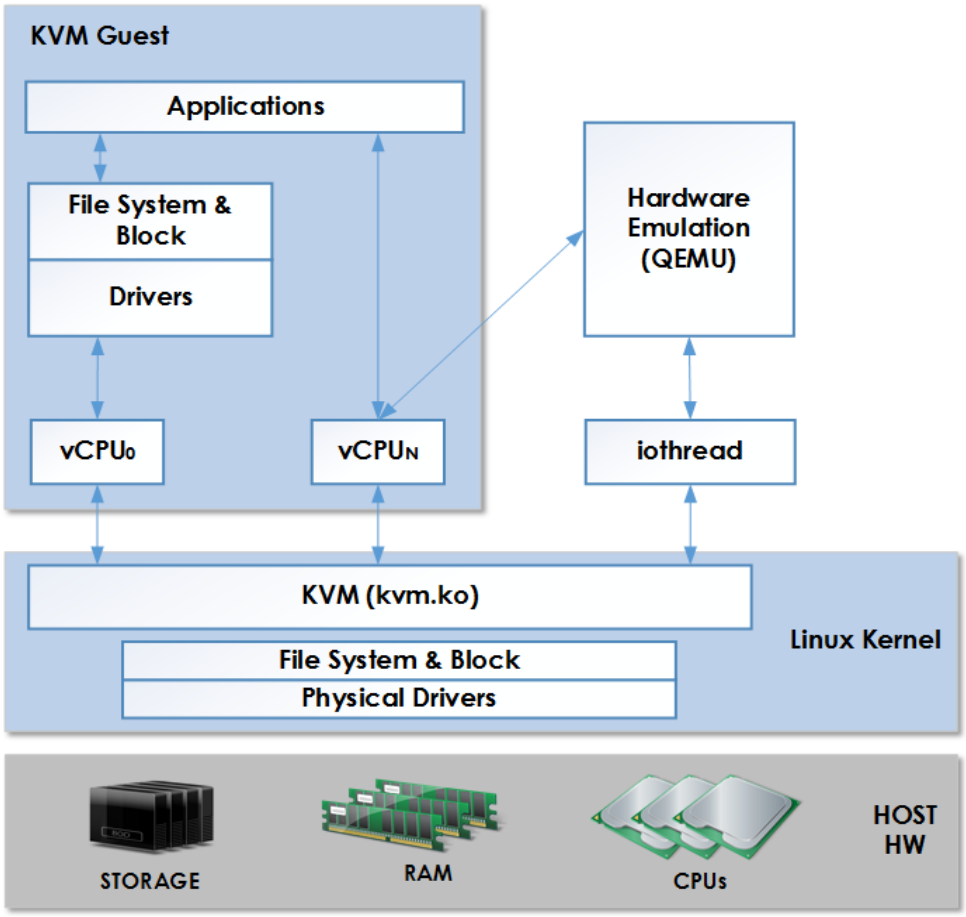

Figure 1: KVM Architecture

One of the main drivers behind the specific KVM model design was harnessing the benefits of existing Linux functionality, where the kernel was responsible for all hardware-related resource management while the KVM acted as a driver handling the newly-introduced virtualization instructions. This allowed KVM to integrate all features deployed in the Linux kernel over time, such as improvements in the CPU scheduler, efficient memory, storage and power management.

Nowadays, there are several projects that use KVM as the default hypervisor, with OpenStack being the most popular one. The use cases under investigation in the auspices of those projects involve large-scale deployments of KVM hosts, often with several VMs per deployment. As guest OSes grow larger in terms of memory and virtual CPU cores, the downtime trade-off for live migration increases. The necessity for low latency network packet processing needed by Telcos for the upcoming 5G era, in conjunction with additional security and hypervisor footprint limitation concerns, shape an ecosystem in which KVM is likely to thrive, therefore Superfluidity project partners will thoroughly inspect its capabilities.

2.2 Virtual Infrastructure Management

A Virtual Infrastructure Manager (VIM) can be defined as the virtualization and management layer for any cloud deployment offering primarily computational, networking and storage services. VIM coordinates physical hardware resources thus simplifying the creation of virtual server instances. It can also be used to manage a range of virtual resources across multiple physical servers and delivers centralized administration of the underlying infrastructure, including creating, storing, monitoring, backing up VMs hosting a variety of applications. For instance, a VIM may create and manage multiple instances of a hypervisor across different physical servers and occasionally facilitate virtual server migration amongst physical nodes.

VIM plays a vital role in the Infrastructure-as-a-Service (IaaS) cloud deployment model, where system administrators can provision processing, storage, network or other fundamental computing resources and seamlessly deploy and run arbitrary software, spanning from simple applications to modern operating systems. VIM software allows pooling and sharing of resources thus providing an enhanced degree of granularity in every service.

2.2.1 OpenStack

OpenStack is a cloud operating system that controls large pools of compute, storage and networking resources throughout a datacenter, all managed over a dashboard that gives administrators total control while empowering all connected users to provision resources through a web interface. One may describe OpenStack as a combination of open source tools called projects that use pooled virtual resources to build and manage private clouds. OpenStack's popularity derives from its inherent support of several proprietary and open source technologies, making it ideal for heterogeneous infrastructure deployments. It offers a highly modular architecture based on distinct elements for Compute, Networking and Storage service support, while each and every function can be managed using command-line tools or a highly efficient RESTful API, besides the previously mentioned web interface. Additional elements also provide Identity and Image services, while other optional projects can be bundled together to create unique deployable clouds. OpenStack modules have certain legacy codenames marking the original project they derived from. The official documentation is often referring to these modules via their codenames, therefore they are included in this section. The Superfluidity testbed is currently based on the OpenStack Mitaka release, the architecture of which is presented in Figure 2.

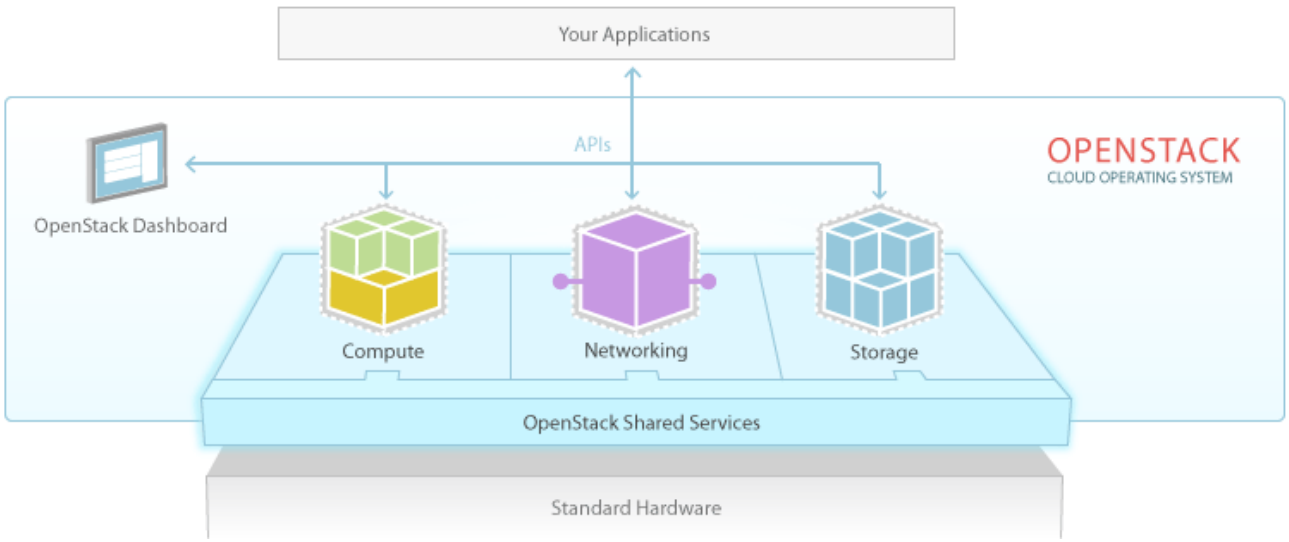

Figure 2: OpenStack Mitaka High Level Architecture

2.2.2 Compute

Compute module (codename Nova) is a highly sophisticated cloud controller designed to provide massively scalable, on demand access to large pools of compute resources. Nova scales horizontally and has the ability to work with all sorts of hardware setups, from bare-metal to high performance computing configurations, as well as the majority of the available virtualization technologies [7]. Its functionality is constantly extended through auxiliary projects and supplementary modules, such as the Ironic Project [8] that is implementing the notion of treating physical machines in the same way to on demand VMs, or the Magnum API service [10] that aims to introduce seamless container functionality thus delivering optimal quality of experience build on top of Nova. However, in the currently deployed version of OpenStack in the Superfluidity testbed, these advanced use cases are not yet utilised.

2.2.3 Networking

2.2.3.1 Neutron

Neutron is an OpenStack Project which provides “networking as a service” between interface devices managed by interconnected OpenStack services. It implements mechanisms for pluggable, scalable, API-driven network and IP management on demand, through technology-agnostic network abstractions. This ensures that networking would never become a limiting factor or bottleneck despite the highly demanding operational environment where node numbers, routing configurations and security rules may quickly escalate to over six figure numbers. In such an environment, traditional network management techniques fall short on providing a truly scalable and automated method of

control, constantly supporting user's ever-growing expectation for flexibility with quicker provisioning. Neutron supports floating IPs that enable dynamic traffic rerouting, load balancing features, software-defined-networking (SDN) technology like OpenFlow [4] and a vast extension framework of different back-ends called "plugins" offering a constantly growing variety of networking technologies, so that third-party network services can be seamlessly deployed and managed.

2.2.3.2 QoS and Role-Based Access Control (RBAC)

Quality of Service (QoS) commonly refers to the minimum service level that an application must provide to the users, for example, in a web server it could mean serving the user requests within a certain time-interval, e.g., less than 1 second. However, with the current trend of moving more and more applications to the cloud, applications are not running on dedicated server anymore, therefore making assumptions about performance more difficult as they may be affected by other co-located applications, an effect known as "noisy neighbour". More specifically this term refers to the fact that one tenant in a cloud computing infrastructure can monopolize resources of a server, such as bandwidth, disk, I/O or CPU and negatively affect other tenants' performance when co-located in the same server.

While for CPU isolation there are already methods in place such as creating VMs with a defined size, or using virtual CPU to physical CPU pinning to avoid that some applications share cores in the same servers, there are still some other shared resources that may lead to interference between co-located VMs, such as L3 caches or the network itself. Due to the networking focus of Superfluidity, we focus on the QoS from the network perspective. In such environment, QoS refers to the resource control system that guarantees certain network requirements such as bandwidth, latency, reliability in order to satisfy a Service Level Agreement (SLA) between an application provider and end users. One example of an application requiring this kind of assurance could be Voice over IP or video streaming, as their traffic needs to be transmitted with some minimal bandwidth constraints. For such a case, on a system without network QoS management, all traffic would be transmitted in a "best-effort" manner making it impossible to guarantee service delivery to customers as co-located applications can impact their performance if they are congesting the network.

In physical networks there were already ways of providing this – although not so flexible – by making network devices such as switches and routers to mark traffic. This enabled the option of prioritizing a flow over another to fulfil the QoS conditions agreed at the SLAs. However, again, with the raise of overlay and software defined networks at cloud platforms, a more flexible way of ensuring QoS was needed, especially at the edge network. There are already different mechanisms, such as OvS (2.2.3.3) min, max or Linux Traffic Control (TC), but there is no industry standard regarding the way of expressing bandwidth guarantees. Consequently, our goal is to enable cloud administrators to better control the network resources by allowing network tuning to specific application types and being able to provide different SLAs.

2.2.3.2.1 Features

We extended Neutron OpenStack to enable QoS related actions, such as limiting or prioritizing traffic. Note this is not yet another traffic shaping policy, but a framework where new policies can be easily included. To accomplish this, the QoS is coded as an advanced service plug-in, decoupled from the rest of the OpenStack Networking code on multiple levels, available through the ml2 extension driver.

The QoS API allows the implementation of policies, which are collections of rules to be applied on Neutron ports or networks. These policies allow the differentiation between different tenants or even applications, as they can be applied either at complete networks or per port basis, respectively. For instance, two policies can be created, one named gold and another one named bronze with different features (e.g., max bandwidth) and that will enforce different QoS needs where they are applied.

As shown in the next figure, policies are made of QoS Rules. These rules can be of different nature, known as QoS Rule Types, and as explained above, new rule types can be easily included.

So far, we have two QoS rule types are already implemented:

- QoS Bandwidth Limit: This ensures that a VM connected to a port will only be allowed to transfer at the specified maximum bandwidth, therefore limiting the impact it has in other co-located VMs, regardless of the tenant it belongs to. Note that, thanks to the design focus on extendibility, we implemented the support for OvS and push it to the OpenStack open source community and other companies already extending the QoS bandwidth limit rule to also cover Linux bridges.
- QoS DSCP Marking: This will mark the outgoing packets from the VM connected to the port with the chosen DSCP mark, so that different network traffic prioritization can be subsequently applied.

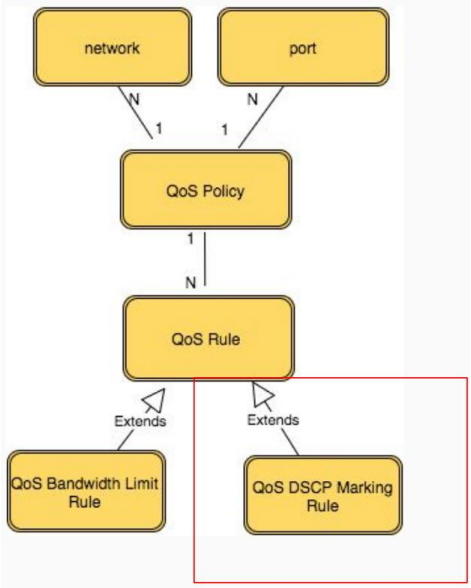

Figure 3: QoS Rules in Policies

On the other hand, besides the specific work done in the QoS API implementation, a parallel effort was made to increase the flexibility in the way the QoS policies could be apply. By default, when a policy is created, and similarly with other OpenStack functionality, such as security groups, it can be used either by the tenant who created it, or make it public, so that all tenants can use it. This does not provide the flexibility needed for applying QoS management in the 5G/Edge cloud environments. To overcome this restriction, we implemented the Role-Based Access Control (RBAC) policy framework that enables both operators and users to grant access to resources for specific projects, i.e., it is not a binary sharing (either all or none), but it can be decided with more granularity which tenants have access to what policies.

2.2.3.2.2 Usage

In order to use the QoS features, the below need to be configured:

- On the server side:
  - Enable qos service in service_plugins (neutron.conf)
  - Set the needed notification_drivers in [qos] section (message_queue is the default)
  - For ml2, add 'qos' to extension_drivers in [ml2] section
- On L2 agent side:
  - Add 'qos' to extensions in [agent] section

Moreover, to enable during QoS devstack installation, update the local.conf to include:

- enable_plugin neutron git://git.openstack.org/openstack/neutron
- enable_serivce q-qos

For more information visit: http://docs.openstack.org/mitaka/networking-guide/config-qos.html


Once it is up and running, a typical workflow is:

- Create a policy
  - neutron qos-policy-create POLICY_NAME
- Add rules to the policy. For instance, to limit max bandwidth:
  - neutron qos-bandwidth-limit-rule-create POLICY_NAME --max-kbps 3000 --max-burst-kbps 300
- Associate the policy to a network or a port
  - neutron net-update NET_NAME --qos-policy POLICY_NAME
  - neutron port-update PORT_ID --qos-policy POLICY_NAME
- [Optional] Change the policy online to immediately propagate it to the ports. This could be either detach the QoS policy from the network/port:
  - neutron net-update PORT_ID --no-qos-policy
  - neutron port-update PORT_ID --no-qos-policy

  Or modify the rules associated to the policies:
  - neutron qos-bandwidth-limit-rule-update RULE_ID POLICY_NAME --max-kbps 2000 --max-burst-kbps 200

Note for DSCP marking the process will be exactly the same, but using the DSCP marking API for creating the rules to be associated to the policies. For instance:

- neutron qos-dscp-marking-rule-create POLICY_NAME --dscp-mark 26

Finally, if the flexibility given by RBAC wants to be used, so that a defined policy may be shared with a specific tenant, the next steps are needed:

- neutron rbac-create --target-tenant TENANT_ID --actions access_as_shared --type qos-policy POLICY_ID

Where the target tenant is the project that requires access to the QoS policy, the action specifies what the project is allowed to do, and the type is the type of resource being shared, in this case the QoS policy.

2.2.3.3 Open vSwitch

Open vSwitch (OvS) is a production quality, multilayer software switch designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols [6]. Licensed under the open source Apache 2.0 license, OvS is designed to support distribution across multiple physical servers, in addition to exposing standard control and

visibility interfaces to the virtual networking layer and operate both as a soft switch within the hypervisor or as the hardware's control stack. Written in platform-independent C, it is easily ported to a variety of environments offering support for advanced feature sets such as: (i) standard 802.1Q VLAN model with trunk and access ports (ii) NIC bonding (iii) Quality of Service (QoS) configuration and policing (iv) automated control through OpenFlow (v) increased visibility and monitoring using NetFlow and sFlow (vi) 802.1ag connectivity fault management (vii) high performance forwarding using a dedicated Linux kernel module.

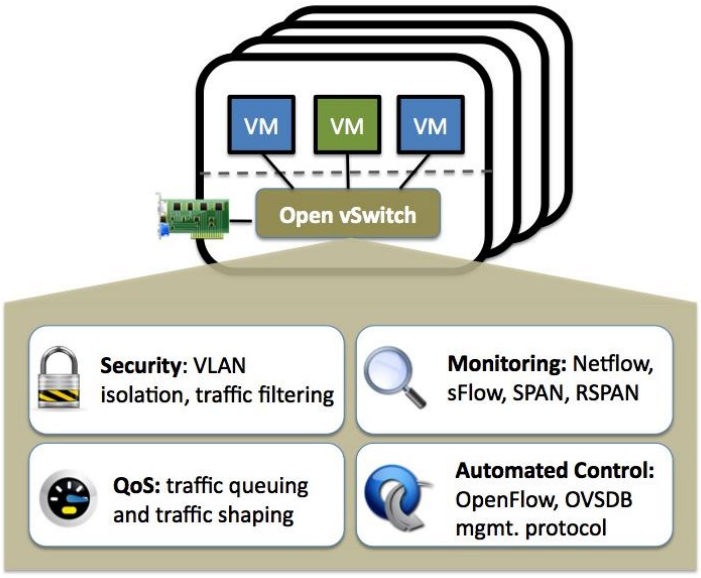

Figure 4: Open vSwitch [6]

Open vSwitch is ideal for multi-server virtualization deployments, a stack where the build-in L2 switch (the Linux Bridge) of Linux-based hypervisors fails to deliver optimal performance. These environments are often characterized by highly dynamic end-points, specific logical abstractions and finally task offloading to special purpose switching hardware. The following characteristics and design considerations help OvS to cope with the exceptional requirements of any networking setup, as described in [12].

The mobility of state

All network state associated with a network entity (i.e. a VM) should be easily transferred between different hosts. The network state may include "soft state" such as L2 learning table entries, L3 forwarding and routing policy states, monitoring configuration and QoS policies. OvS supports configuration and network state migration between instances through a real data-model that records its overall status. This approach also facilitates the development of structured automation systems.

Responding to network dynamics

Virtual environments are often characterized by high change rates. OvS supports several features that allow a network control system to adapt and respond to topology alterations. The dominant one

is a network state database (OVSDB) that records topology changes through remote triggers, along with auxiliary accounting and monitoring protocols such as NetFlow [14] and sFlow [15]. Finally, OvS supports OpenFlow as a method of exporting remote access to control traffic.

Logical tag maintenance

Distributed virtual switches maintain logical context within the network through appending or manipulating tags in network packets. These tags can be used to uniquely identify interconnected nodes thus getting an overview of the actual topology. OvS supports multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration and stored in an optimized manner. This allows dynamic reconfiguration of all tags in case a node migrates from one datacenter to another.

Hardware integration

As stated in [12], Open vSwitch's forwarding path (the in-kernel datapath) is designed to be amenable to "offloading" packet processing to hardware chipsets, whether housed in a classic hardware switch chassis or in an end-host NIC. This allows for the OvS control path to be able to both control a pure software implementation or a hardware switch. The advantage of hardware integration is not limited to performance only. When physical switches follow the same abstractions as OvS, both bare-metal and virtualized hosting environments can be managed using identical mechanisms for automated network control.

OvS can be integrated to OpenStack Networking service using the ML2 plugin. However, since this integration involves several different entities, certain restrictions do apply, mostly related to available virtual resources and pre-deployed OpenStack components, as described in [13]. In particular, one Controller node, one Network node and two Compute nodes are needed. The Controller node requires one dedicated interface for management networking purposes, the Compute nodes need three different interfaces for Management, Tunnelling and VLAN networks respectively, while the Network node must also have an additional interface for the necessary external network.

2.2.4 Load Balancing

Load balancing can be defined as the distribution of network or application traffic across a cluster of servers, which leads to improved responsiveness and increased service availability. A load balancer is the dedicated node located between the client and the server cluster that accepts incoming traffic and distributes it across multiple backend servers using various methods. By balancing application requests across several servers, a load balancer reduces individual load and prevents nodes from becoming a single point of failure due to increased service requests. Load balancing is the most straightforward method of scaling out an application server infrastructure. As application demand increases, new servers can be easily added to the resource pool, and the load balancer will immediately begin sending traffic to the new server. This process is crucial to all deployments with


more than one physical or virtual servers which intend to reach certain scalability standards. It is therefore expected from cloud-oriented software to natively support load balancing techniques. OpenStack implements load balancing solutions as part of the Neutron Project, that allow simplistic or more advanced network traffic scenarios to be realized.

2.2.4.1 HAProxy

HAProxy is a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler. Designed with a data forwarding orientation, its architecture is optimized to move data as fast as possible with the least possible operations by implementing a layered model of distinct bypass mechanisms. This approach ensures that data will be forwarded by the most appropriate level, rendering it suitable for TCP and HTTP-based applications where introducing additional encapsulation comes with a great computational and complexity cost. HAProxy only requires the haproxy [15] executable and the necessary configuration file to run. The configuration file is parsed before starting, then HAProxy tries to bind all listening sockets and once initiated it processes incoming connections, periodically checks the server's status and exchanges information with other HAProxy nodes.

HAProxy offers a fairly complete set of load balancing features with several supported load balancing algorithms, dynamic weight and slow-start support, and a large number of internal metrics for creating advanced load balancing strategies. Some of the algorithms include **round-robin**, (for short connections, pick each server in turn), **leastconn** (for long connections, pick the least recently used of the servers with the lowest connection count), **source** (for SSL farms or terminal server farms, the server directly depends on the client's source address), **uri** (for HTTP caches, the server directly depends on the HTTP URI), **hdr** (the server directly depends on the contents of a specific HTTP header field), **first** (for short-lived virtual machines, all connections are packed on the smallest possible subset of servers so that unused ones can be powered down) [16].

OpenStack fully supports HAProxy deployment in its controller nodes where each instance of the software configures its frontend to accept connections only to the virtual IP (VIP) address. The HAProxy backend (termination point) is a list of all the IP addresses of instances for load balances. When integrated with OpenStack, HAProxy provides a fast and reliable HTTP reverse proxy and load balancer for TCP or HTTP applications. It is particularly suited for web crawling under very high loads while needing persistence or Layer 7 processing and can realistically support tens of thousands of connections with recent hardware [17]. An alternative LB implementation is OpenStack Octavia [55].

2.2.4.2 Load Balancing as a Service

Load Balancing as a Service (LBaaS) is currently an advanced service of Neutron, which allows both proprietary and open-source load balancing technologies to drive the actual overall request load. This

renders OpenStack operators capable of using the back-end technology of choice. However, the long term vision for this service is to provide a single API that allows users to seamlessly move between load balancing technologies [18]. All OpenStack distributions after Liberty integrate LBaaS v2, which addresses certain limitations of the previous version, including lack of scalability and configuration flexibility.

LBaaS builds on top of HAProxy by leveraging agents that control HAProxy configuration and manage the HAProxy daemon. However, the significant changes of LBaaS v2 is the addition of new entities and concepts such as the Listener, the Member, the Pool and the Health monitor. A Listener is an entity mapped to and listens for requests made to a specific network port. Users can create multiple Listeners per load balancer, allowing monitoring of requests originating from multiple network ports. Members are servers that handle traffic behind the load balancer. Each member is specified by the IP address and port that uses for traffic handling. Pools hold a list of Members that serve content through the load balancer, while Health monitors divert traffic away from non-responsive Members and are directly associated with Pools [19]. The overall LBaaS architecture is shown in **Error! Reference source not found.**.

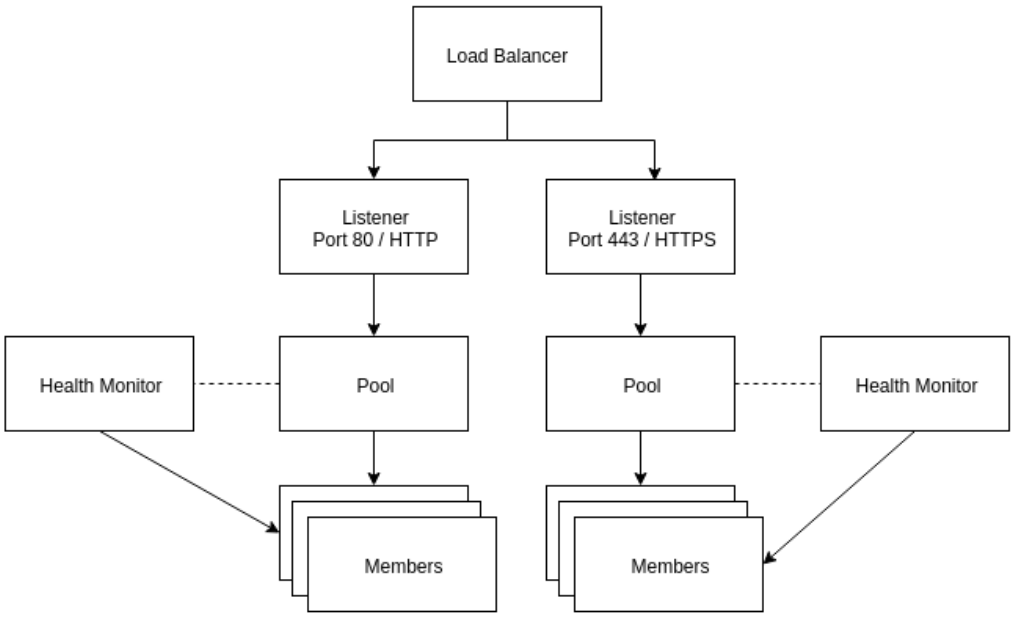

Figure 5: Load-Balancing-as-a-Service Architecture

The new set of features introduced by LBaaS enables OpenStack Networking to distribute incoming requests evenly between designated instances. This ensures that the workload is shared predictably among instances, and allows more effective use of underlying system resources.

2.2.5 Heat

Heat is the cornerstone of the overall OpenStack orchestration project [20]. It implements an orchestration engine allowing users to launch multiple composite cloud applications through detailed

deployment descriptions in specific text files called *templates* that can be treated as code. A Heat template describes the infrastructure a cloud application requires to become operational, such as servers, floating IPs, volumes, security groups or users, as well as the relationships between them. This pre-defined correlation of all application elements enables the Heat engine to interact with the virtual hardware through the OpenStack API towards creating the proper deployment environment. Heat manages the whole lifecycle of the application, therefore when infrastructure changes are necessary, users must only modify the template and use its latest version to update the existing stack. All corresponding changes are handled by Heat, which will also delete the reserved resources once the application lifespan ends. Heat primarily manages infrastructure; however, templates integrate well with software management tools such as Puppet [23], Chef [24] and Ansible [56] while also providing an autoscalling service.

Heat was born as the counterpart to the CloudFormation service in AWS. It accepts AWS templates and provides a compatible API, yet in recent OpenStack releases evolved to a new format called Heat Orchestration Template (HOT) with nicer template syntax and new features not supported by its competitor. HOT is considered reliable, supported and standardized as of OpenStack Icehouse (05/2013) release, offering extensive backwards compatibility. Since OpenStack Juno (10/2014) release, Heat supports multiple different versions of the HOT specification [21].

HOT templates are defined in YAML [22] and follow the structure outlined in Figure 6, while the template structure is further explained in Table 2.

```
heat_template_version: 2016-10-14
description:
  # a description of the template
parameter_groups:
  # a declaration of input parameter groups and order
parameters:
  # declaration of input parameters
resources:
  # declaration of template resources
outputs:
  # declaration of output parameters
conditions:
  # declaration of conditions
```

Figure 6: Heat Orchestration Template

Table 2: Heat Orchestration Template structure

KEY NAME	DESCRIPTION
heat_template_version	This key value indicates that the YAML document is a HOT template of the specified version
description	This optional key allows for giving a description of the template, or the workload that can be deployed using the template
parameter_groups	This section allows specifying how the input parameters should be grouped and the order to provide the parameters in. This is an optional section.
parameters	This section allows for specifying input parameters that must be provided when instantiating the template. This is an optional section.
resources	This section contains the declaration of the single resources of the template. This section with at least one resource should be defined in any HOT template, or the template would not do anything when being instantiated.
outputs	This section allows for specifying output parameters available to users once the template has been instantiated. This is an optional section.
conditions	This optional section includes statements which can be used to restrict when a resource is created or when a property is defined.

2.2.6 Mistral

Mistral [25] can be defined as a simple yet scalable workflow service for cloud automation, featuring an intuitive YAML-based workflow definition language, a REST API for operating the workflows and an extensive set of actions including OpenStack operation, REST HTTP call and SSH protocol support. Mistral mostly targets administrators who are automating their operation and maintenance procedures, and integrating private clouds with their broader infrastructure. A user typically assembles a workflow using the YAML-based language and uploads the workload definition to Mistral via its REST API. Then the user can manually initiate the uploaded workflow using the same API or configure a trigger to start the workflow based on a specific event. Mistral typically provides all the necessary control points to manage task execution (i.e. suspend/resume) and observe their state, for instance, to find out whether a particular task or a sequence of tasks has successfully finished or failed.

As described in the previous paragraph, Mistral intends to add capabilities resembling to “Cloud Cron” to OpenStack, thus facilitating periodic cloud task scheduling and alleviate automated business functions consisting of multiple distributed processing steps. For efficiently tackling all issues that rise in the process, Mistral workflow service introduces the Domain Specific Language (DSL) v2 (as of 04/2015) which besides being based on YAML, takes advantage of additional query languages, for instance YAQL [26] and Jinja2 [27], to define expressions in workflow and action definitions. Mistral DSL v2 introduces different workflow types and the structure of each workflow type varies according

to its semantics. Basically, workflow type encapsulates workflow processing logic, a set of meta rules defining how all workflows of this type should work. Currently, Mistral provides two workflow types: Direct workflow, where each task starts based on the previous task's result, and Reverse workflow, where all relationships in the workflow task graph are dependencies. When executing reverse workflows, Mistral Engine must recursively identify all dependencies that need to be completed first. An example workflow which simply sends a command to OpenStack Compute service to start creating a VM and wait until the VM is created using a special "retry" policy is presented in Figure 7.

```
---
version: '2.0'

create_vm:
  description: Simple workflow example
  type: direct

  input:
    - vm_name
    - image_ref
    - flavor_ref
  output:
    vm_id: <% $.vm_id %>

  tasks:
    create_server:
      action: nova.servers_create name=<% $.vm_name %> image=<% $.image_ref %>
flavor=<% $.flavor_ref %>
      publish:
        vm_id: <% task(create_server).result.id %>
      on-success:
        - wait_for_instance

    wait_for_instance:
      action: nova.servers_find id=<% $.vm_id %> status='ACTIVE'
      retry:
        delay: 5
        count: 15
```

Figure 7: Mistral Workflow using YAML

2.2.7 Telemetry

As described in [28] the mission of OpenStack Telemetry is to reliably collect data on the utilization of both physical and virtual resources comprising deployed clouds, to persist the collected data for subsequent retrieval and analysis, as well as to trigger actions when pre-defined criteria are met. A convoluted environment such as OpenStack has vast telemetry requirements, including perplexed


use cases involving metering, monitoring and alarming functions. In order to properly support dedicated components for all prerequisites, OpenStack Telemetry consists of various projects each designed to provide a discrete service in the telemetry space, and is built on an agent-based architecture [29]. Several modules combine their responsibilities to collect data, store samples in a database and provide an API service for handling incoming requests. In particular, (i) Aodh is an alarming service, (ii) Gnocchi is a time-series database and resource indexing service, (iii) Panko is an event and metadata indexing service and (iv) Ceilometer, which acts as OpenStack Telemetry's data collection service.

2.2.7.1 Ceilometer

Ceilometer is a data collection service that provides the ability to normalize and transform data across all current OpenStack core components [30]. Operating as a critical element of OpenStack's Telemetry, its data can be used to provide information that can be transformed into billable items. Ceilometer collects data using two methods:

(i) **Notification agent**, which takes messages generated on the notification bus and transforms them into Ceilometer samples or events. This is the preferred method of data collection.

(ii) **Polling agents**, will poll some API or other tool to collect information at a regular interval. The polling approach is less preferred due to the load it can impose on the API services.

The first method is supported by the ceilometer-notification agent, which monitors the message queues for notifications. Polling agents can be configured either to poll the local hypervisor or remote APIs (public REST APIs exposed by services and host-level SNMP/IPMI daemons). Ceilometer offers the ability to take data gathered by the agents, manipulate it, and publish it in various combinations via multiple pipelines. This functionality is handled by the notification agents. The overall summary of Ceilometer's logical architecture is shown in Figure 8 along with a representation of how polling and notification agents gather data from multiple sources.

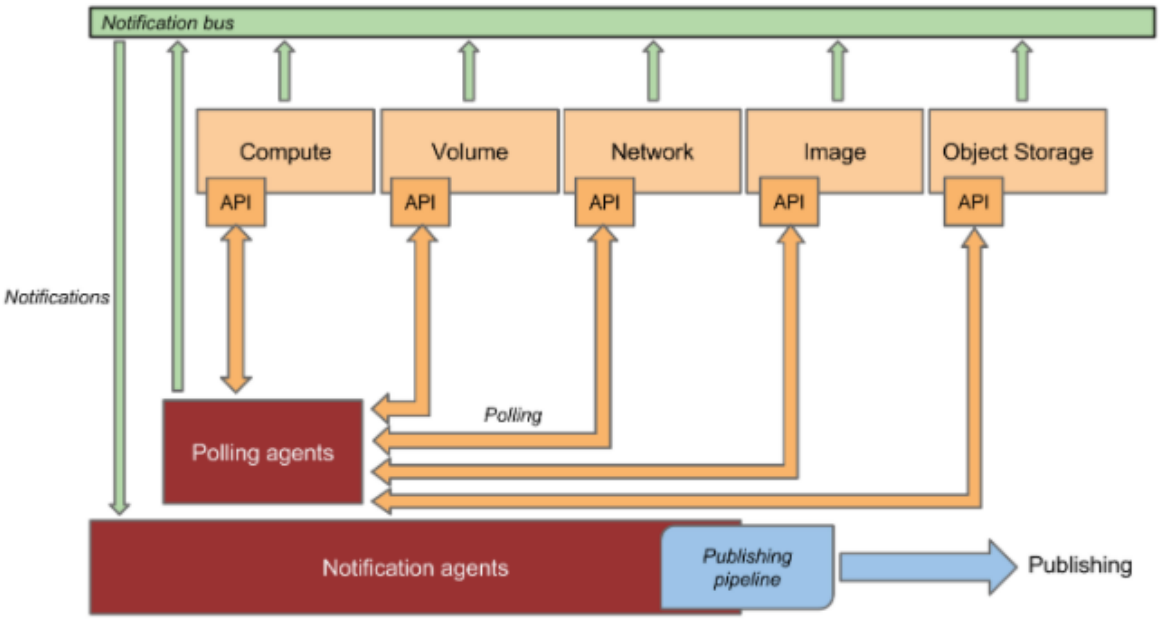

Figure 8: OpenStack Ceilometer data collection mechanisms

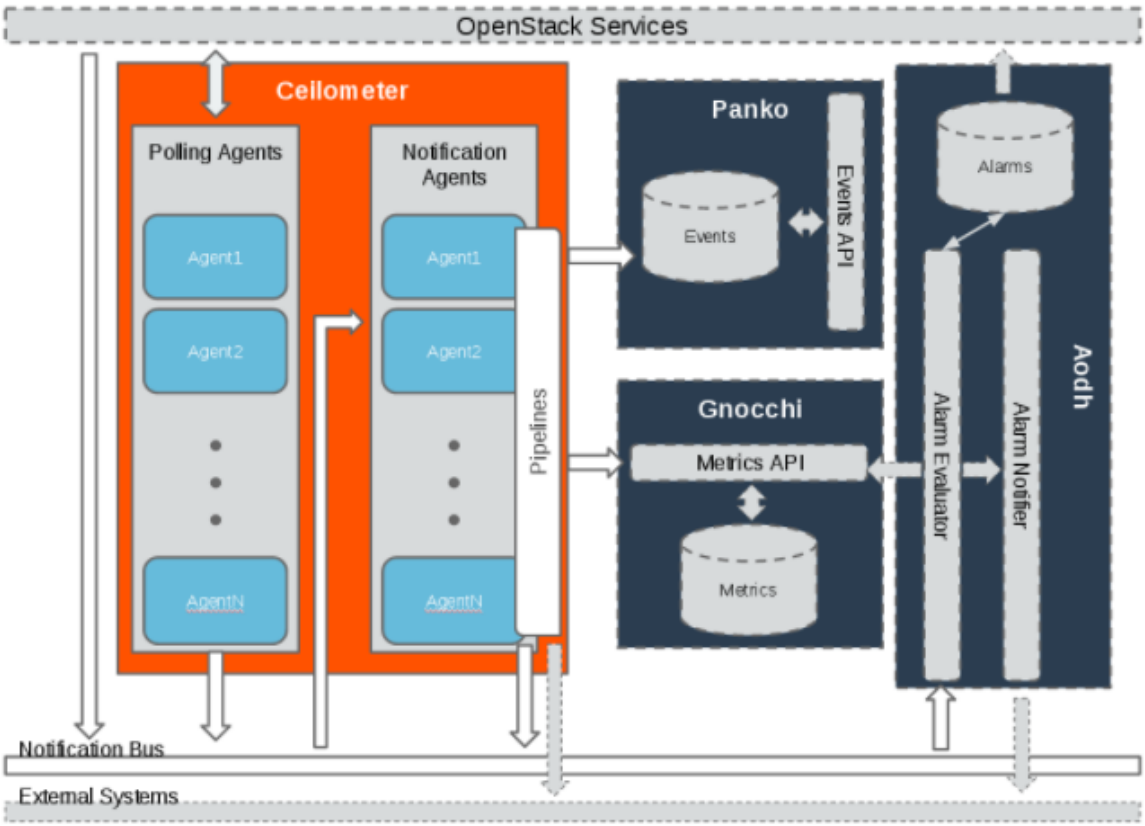

Figure 9: OpenStack Ceilometer architecture and communication interfaces

3 Communication Interfaces and API Design

Application deployment in Superfluidity architecture must rely on a well-defined API which operates over specific pre-defined interfaces, as those defined by the ETSI NFV Industry Specification Group (ISG) [31]. As already presented in D3.1 [54] any proposition related to User-Management (UM) API development must build on the functions and reference points of the NFV Orchestrator (NFVO), augmenting them to expose the unique capabilities of the Superfluidity architecture and the newly introduced components and functional blocks. In this deliverable, our intention is to map and elaborate on specific API calls that need to be implemented on each and every main NFV reference point [31]. In the context of specifying the API to third-parties, one rather important factor to consider is the concrete textual representation of the information model elements. Practically speaking, this will specify the syntax of the files that will represent the network service templates, VNF descriptors, VNF forwarding graph descriptors, etc. For properly addressing all issues, an analysis of the actual ETSI NFV Architecture is needed.

3.1 ETSI NFV Architecture

Over the last few years, the ETSI NFV ISG [31] identified the functional blocks of a holistic NFV architecture along with the main reference points between them. Some of these blocks are already present in current deployments, while others are identified as necessary additions for enhanced virtualization-based functionality. Figure 10 depicts an overview of the particular architecture with all major functional blocks as well as the reference points in between. VNFs rely on the resources provided by the Network Functions Virtualization Infrastructure (NFVI), which is composed by the compute, networking and storage hardware resources. These resources are managed by one or more Virtualized Infrastructure Managers (VIMs), which expose northbound interfaces to the VNF Managers (VNFM) as well as the Network Functions Virtualization Orchestrator (NFVO).

The VNFM performs the lifecycle management of the VNFs through an associated VNF catalogue that stores the VNF Descriptors (VNFDs). These descriptors contain the structural properties of the VNFs, for instance the number of available ports or the internal component decomposition, along with their deployment and operational behaviour. A VNF is decomposed in one or more Virtual Deployment Units (VDUs) which can be mapped to Virtual Machines (VMs) or containers, deployed and executed over the NFV Infrastructure. The NFVO is responsible for the overall orchestration and lifecycle management of the Network Services (NS) topology, which is represented by an NS Descriptor (NSD) capturing the relations between VNFs and are stored in an NS Catalogue, parsed by the NFVO during the deployment and operational management of all services. The NFVO is also responsible for the on-boarding and validation of the descriptions of VNFs and NSs.

The main (named) reference and execution points shown by solid lines, are potential standardization targets and the communication interfaces that any API must utilize. The dotted reference points

despite being present in deployments already, will need minor extensions to handle network function virtualization, but are not the focus of NFV at the moment. As stated in [31] the architectural framework focuses on the necessary functionality for the virtualization and the consequent operation of an operator's network but does not strictly specifies which network functions should be virtualized, leaving the network owner to decide at will. In the same manner, no direct guideline is issued for communication interface functionality besides certain generic principles, thus providing an additional degree of freedom somehow suitable for the novel concepts and functional elements of Superfluidity.

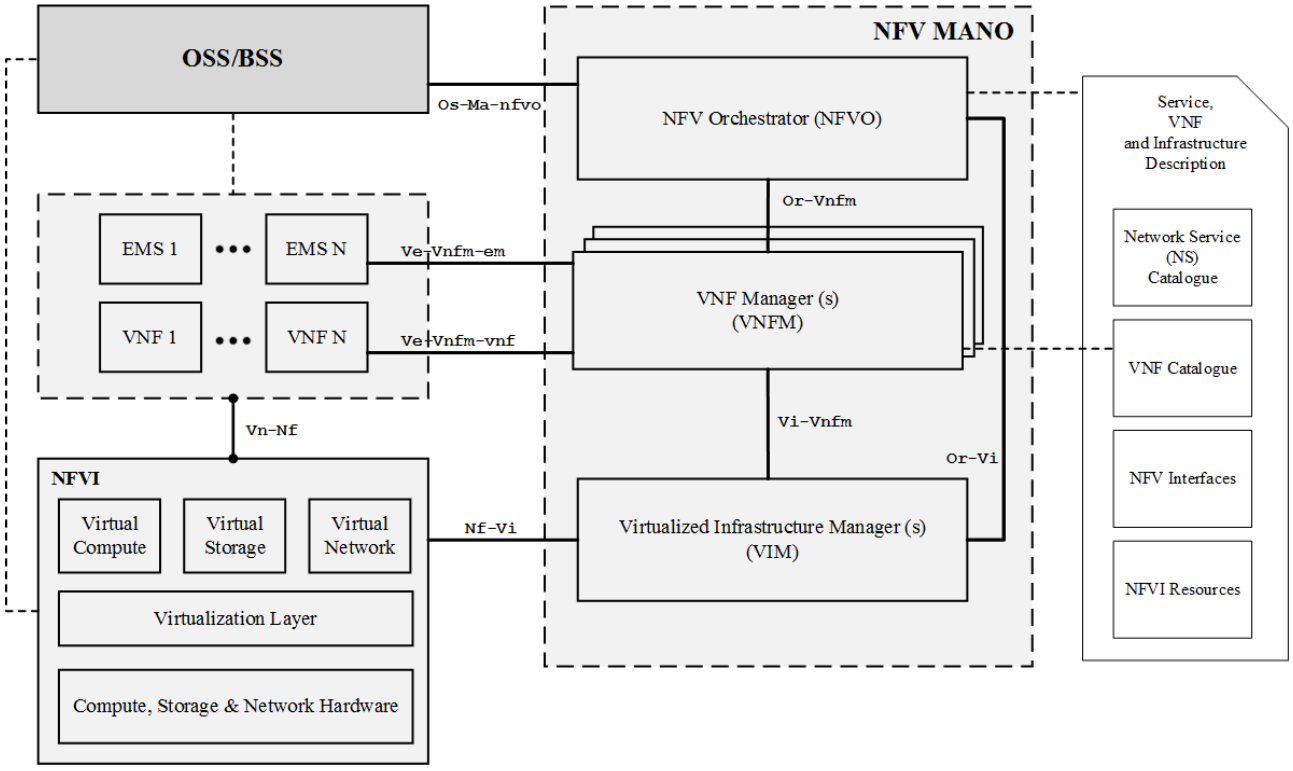

Figure 10: ETSI NFV Reference Architecture diagram

3.2 Network Service Interfaces

Network Service communications in the ETSI NFV Architecture are mostly traverse through the Os-Ma-nfvo [32], which is the sole external reference point of NFVO. This renders NFVO functionality available to OSS/BSS systems, through implementing necessary functions, allowing NFV MANO to handle:

- Network Service Lifecycle Management messages
- Network Service Lifecycle Change Notifications
- Network Service Descriptors
- Network Service Performance Management messages
- Network Service faults

In addition, the latest ETSI NFV MANO Os-Ma-Nfvo reference point specification [32] incorporates virtual network function package functionality, such as enable, disable, delete, query, subscribe/notify, fetch and abort package deletion. As also described in D3.1 [54] the top-level User-Manager API of Superfluidity must implement most of the above functions, with some of them further enhanced to leverage the project's innovations. The functions exposed by the Os-Ma-Nfvo reference point make use of a multitude of elements from the NFV Information Model [33], while NFVO must also be able to manage Network Service Templates [34], located in the NFV Service Catalogue.

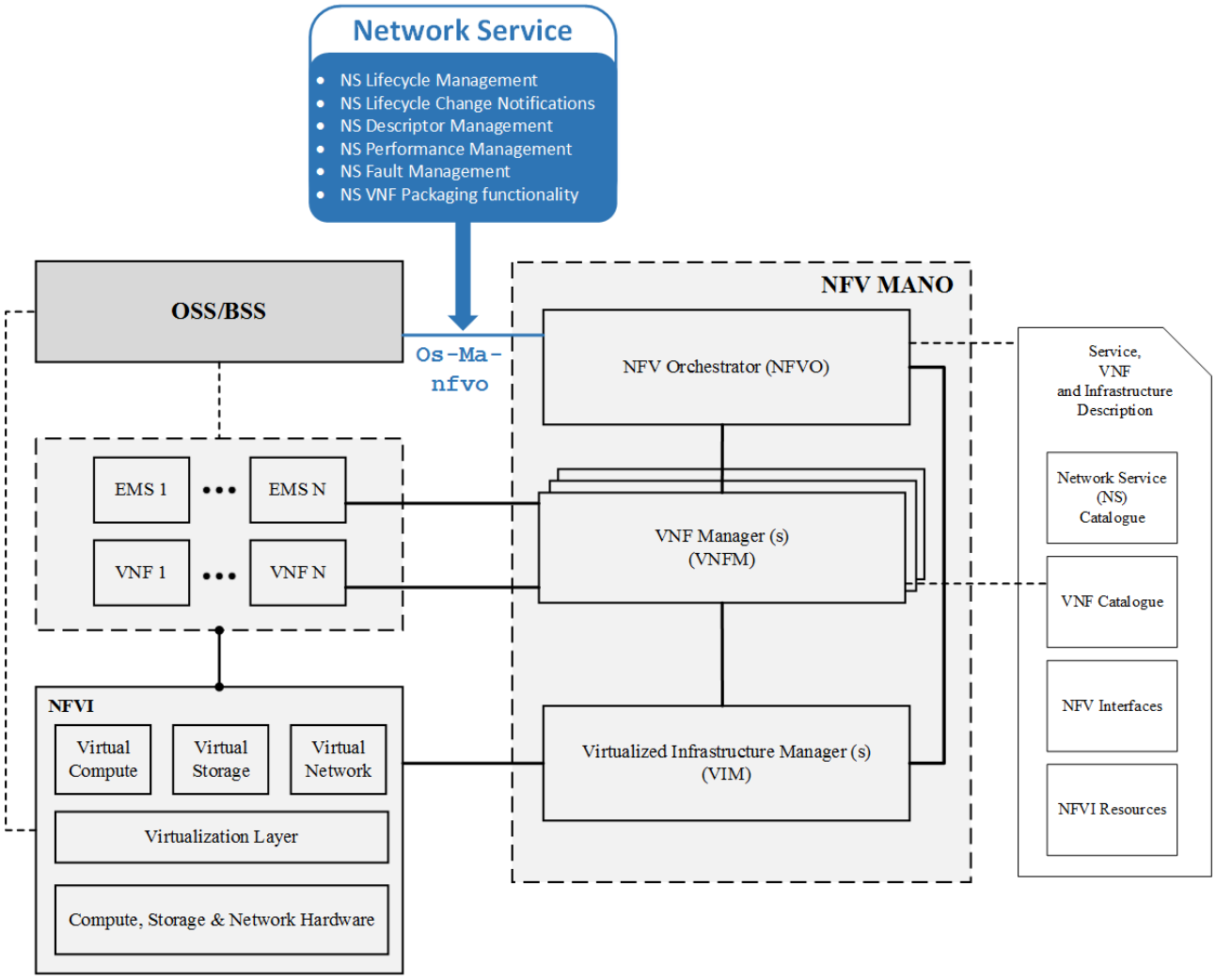

Figure 11: ETSI NFV Network Service Reference Point

3.3 Management and Configuration VNFs

ETSI NFV Architecture defines a large variety of VNFs related to management and configuration tasks. All ETSI NFV Architecture blocks connected to the VNF Manager through main NFV reference points must facilitate VNF handling, therefore support related API calls. The interconnected blocks to the VNFM are the NFV Orchestrator through the Or-Vnfm reference point, the Virtual Infrastructure

Management (VIM) through the Vi-Vnfm reference point and well as EMS/VNF blocks attached through Ve-Vnfm reference point. In addition, since NFVO also handles VNF internally, must support all corresponding functionality via the Os-Ma and Or-Vi reference points towards OSS/BSS and VIM blocks respectively. VNF actions include but are not limited to:

- VNF Package Management
- Software Image Management
- VNF Lifecycle Management and Operation Granting, along with the necessary Notifications
- VNF Configuration and Indicators
- VNF Configuration, Performance and Fault Management

Despite the inherent necessity for end-to-end functionality support, not all reference points need to integrate mechanisms for the whole set of VNF actions listed before. Figure 12 presents the specific VNFs per reference point that need to be implemented, as described in each reference point interface and information model specification document [35,36,37,38].

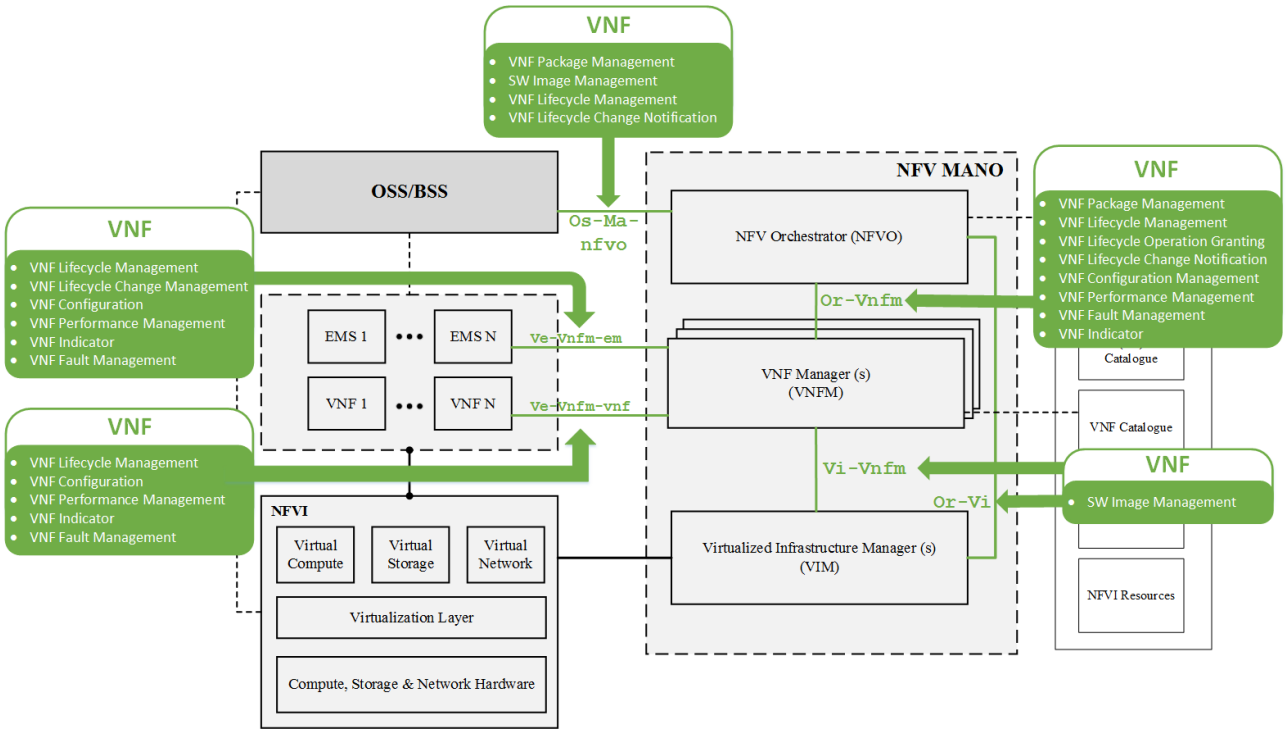

Figure 12: ETSI NFV VNF Reference Points

3.4 Virtual Resource Interfaces

The ETSI NFV Reference points related to Virtual Resource functionality are located in the NFV MANO block, consists of the NFV Orchestrator, the VNF Manager and the Virtualized Infrastructure Manager. According to [31] the NFVO communicates (i) with the VNFM via the Or-Vnfm reference point and (ii) with the VIM via the Or-Vi reference point. In addition, the VNFM exchanges information with the

VIM through the Vi-Vnfm reference point. Virtual Resource-related information, notifications, requests and responses traverse through the three reference points, it is therefore necessary to implement corresponding functions in the API, aligned with the guidelines of [35, 36, 37]. Virtual resource-related tasks are listed below:

- Virtual Resource Management
- Virtual Resource Notifications regarding available quota and possible status changes
- Virtual Resource Information, Capacity, Performance, Reservation and Fault Management
- Network Forwarding Path Management

Figure 13 presents the virtual resource information chunks per specific NFV main reference point, since similar to the Management and Configuration VNF case, not all functional blocks demand access to the same information subset available in the NFV MANO.

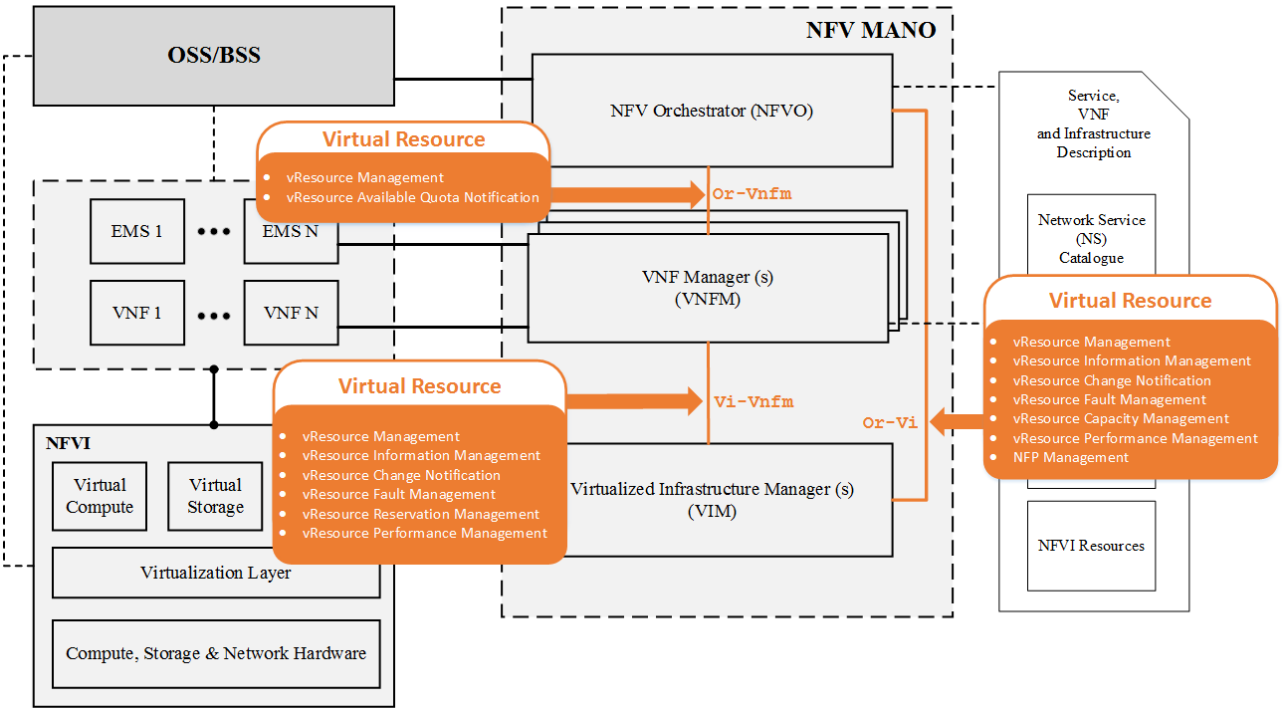

Figure 13: ETSI NFV Virtual Resource Reference Points

3.5 NFVI Functions

The ETSI NFV Architecture only defines a single reference point which facilitates communication between the NFVI and the Virtualized Infrastructure Manager, namely the Nf-Vi reference point. This leads to a unique interface on each side for NFVI-related information exchange, in particular messaging associated to NFVI Networking, Storage and Hypervisor resources management. Figure 14

depicts the exact information elements that Nf-Vi reference point handles, are therefore need to be integrated to the Superfluidity API.

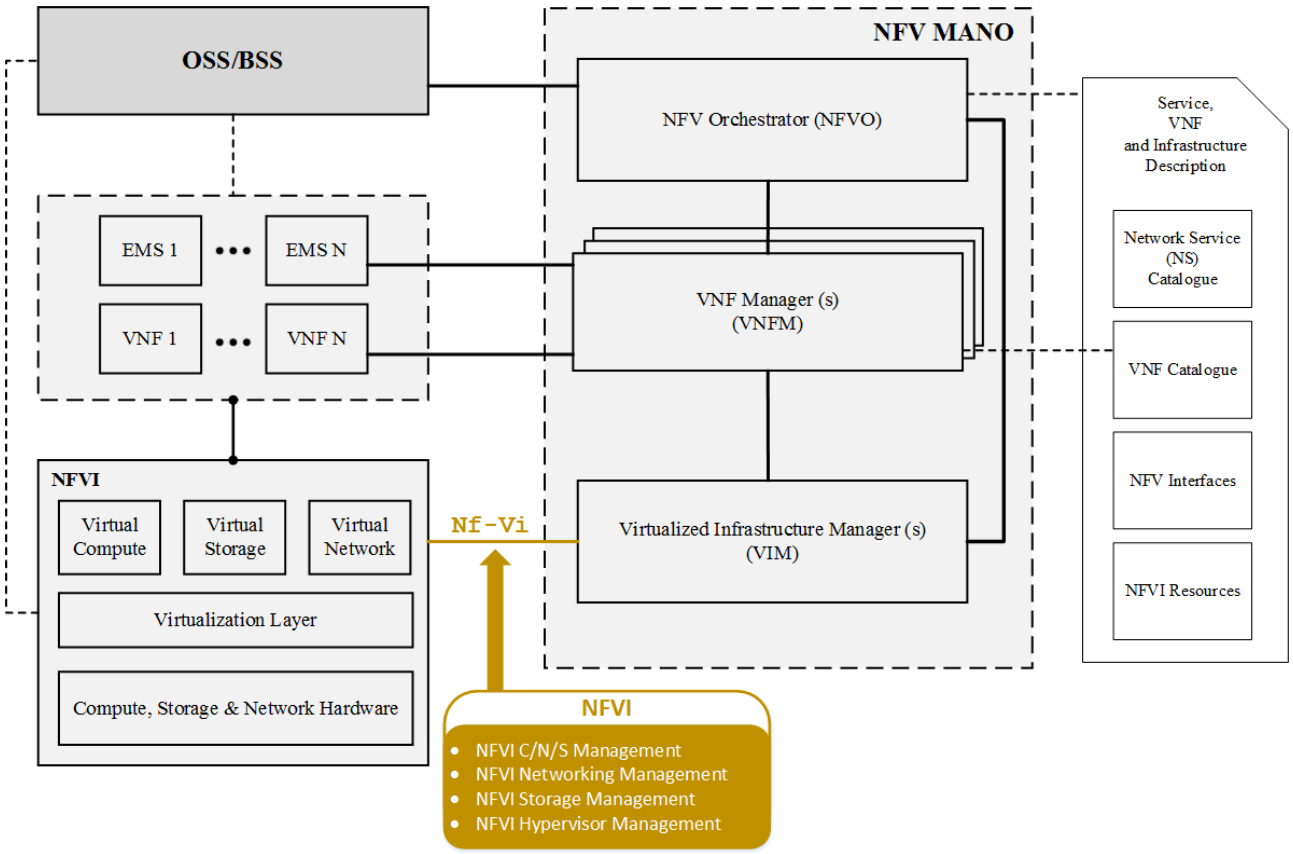

Figure 14: ETSI NFV NFVI Reference Point

3.6 External and Internal Interfaces of ETSI NFV Architecture

ETSI NFV ISG published a series of documents clarifying all specification issues derived from missing attributes of the now deprecated Release 1 of the ETSI NFV Architectural Framework. All Main NFV reference points were analyzed, in documentation containing all specification, interfaces and functions per architectural block. In April 2016, the first set of Release 2 specification was published, namely ETSI GS NFV-IFA 002 [39], ETSI GS NFV-IFA 003 [40] and ETSI GS NFV-IFA 004 [41] containing specification regarding Acceleration Technologies for VNF Interfaces, vSwitch Benchmarking and Management aspects respectively, ETSI GS NFV-IFA 005 [35] with specification regarding the Or-Vi reference point, ETSI GS NFV-IFA 006 [36] with specification regarding Vi-Vnfm reference point and ETSI GS NFV-IFA 010 v2.1.1 [42] with the necessary Functional Requirements specification. In September 2016, the remaining Release 2 requirements, interfaces, and information model specifications were completed, in particular ETSI GS NFV-IFA 007 [37] related to Or-Vnfm reference point, ETSI GS NFV-IFA 008 [38] describing the Ve-Vnfm reference point, ETSI GS NFV-IFA 010 v2.2.1 [43] including certain updates on the previous version, ETSI GS NFV-IFA 011 [44] with the VNF

Packaging Specification, ETSI GS NFV-IFA 013 [32] related to Os-Ma-Nfvo reference point, ETSI GS NFV-IFA 014 [34] containing the Network Service Templates specification and in November 2016, the ETSI GS NFV-IFA 015 [33] with the Report on the NFV Information Model. Additional specifications addressing Security, Reliability, Use Cases and other features were also available, some of which contain information related to Vn-Nf execution point and other auxiliary reference points that were not addressed recently by the ETSI NFV ISG, such as Nf-Vi. Figure 15 summarizes the corresponding documents per main reference point and functional block.

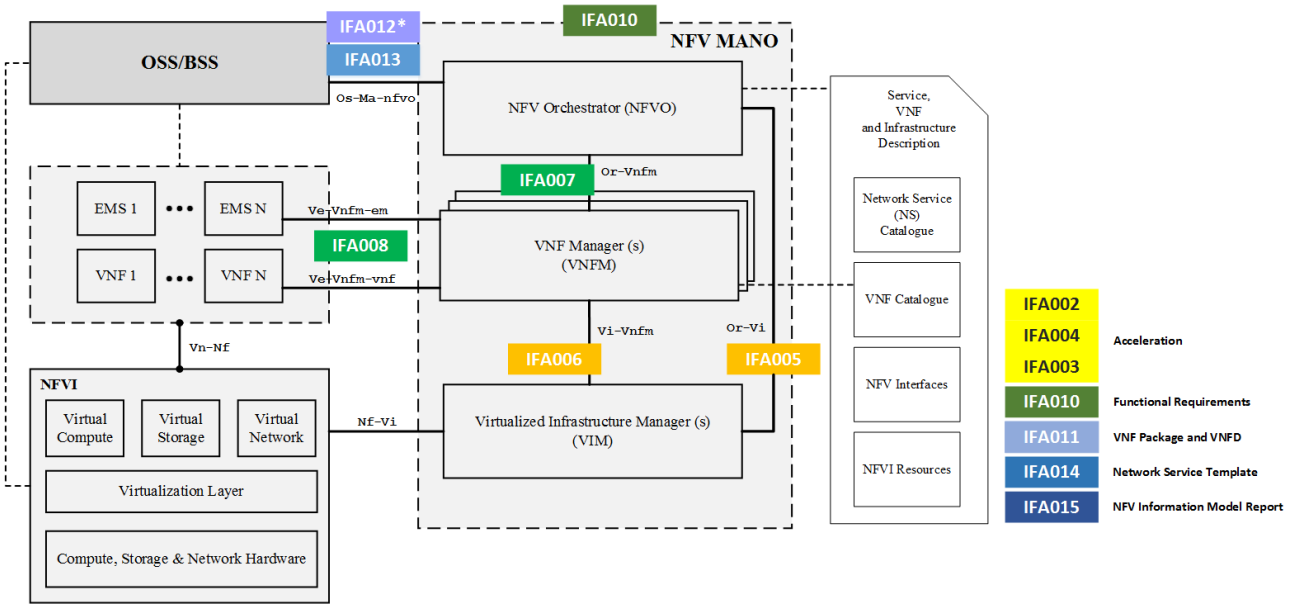

Figure 15: ETSI MANO Architecture - Specifications and Reference Point Analysis

3.7 NFV Orchestrator API/Functions

The approval of the ETSI GS NFV-IFA specifications was a major step towards enabling interoperability between Management and Orchestration functions, as well as between VNFs and Management & Orchestration functions. However, further steps are needed, related to migrating from the functional descriptions of Management and Orchestration Interfaces to Protocols/API specifications. Additionally, Network Service and VNF descriptors need to evolve from information to data models before any implementation attempt begins. An actual step towards a pragmatic solution on developing a protocol, API and data model specifications is most likely to endorse/profile externally defined solutions when most of the ETSI requirements are met. Such an attempt is made in the Open Stack APIs, which are considered as a Stage 3 solution of the VIM Northbound interfaces, currently defined by main reference points Vi-Vnfm and Or-Vi, analyzed in ETSI GS NFV-IFA 006 [36] and ETSI GS NFV-IFA 005 [35] respectively. Furthermore, certain specification development activity inside ETSI NFV is in progress, with the pending GS NFV-SOL 002 document which will specify RESTful protocols

for the Ve-Vnfm Reference Point, as well as the GS NFV-SOL 003 document which will specify RESTful protocols for the Or-Vnfm Reference point.

3.7.1 Top Level User-Manager API Implementation Challenges

As described in Supefluidity Project deliverable D3.1 [54] and in Section 3.2 of the present document the top-level User-Manager API of Superfluidity should implement a large variety of (i) network service descriptors, (ii) network service life-cycle calls, (iii) network service performance functions (iv) network service fault identifiers and (v) virtual network function packages, the majority of which is summarized in Table 3.

Table 3: Common API Calls

PURPOSE	FUNCTION/API CALLS
NS Descriptors	on-board, enable, disable, update, delete, query, subscribe/notify
NS Life-cycle calls	create, instantiate, update, query, terminate, delete, heal, get status, subscribe/notify
NS Performance Functions	create job, delete job, subscribe/notify, query job, create threshold, delete threshold, query threshold
NS fault identifiers	subscribe/notify, get alarm list
VNF Packagers	on-board, enable, disable, delete, query, subscribe/notify, fetch package, fetch package artifacts, abort package deletion

However, since the corresponding reference point specification was only recently published, and the protocol is still under development, there is no NFVO that implements this specification in a standards-compliant way. Upcoming RESTful protocol specification documents, will address similar cases for other reference points i.e. Or-Vnfm and Ve-Vnfm, but at the moment, all functions will be implemented by the toolchain of the NFVOs of choice. This may lead to non-standards compliant implementation that will limit the overall functionality of the platform, compared to what ETSI NFV ISG Architecture described in the first place.

3.7.2 Relation to NFV Information Model

According to [33], the NFV Information model is organized in an NFV Core Model and certain extensions which enrich the NFV Core Model functionality for specific needs. Each model, including the Core, follows a Domain-based structure with four distinct domains defined up until today namely:

- NFV Common Domain
- Virtualized Resource Domain
- VNF Domain
- NS Domain

In addition, the NFV Information Model includes three types of view: (i) Logical, which is concerned with the functionality that the system provides to end-users, (ii) Deployment, which is concerned with the functionality that is needed to deploy the provided system to the end users and (iii) Application, concerned with the functionality that the application provides to end-users.

Despite the significant level of flexibility the proposed NFV Information model supports, the Reusable Functional Block (RFB) model proposed by the Superfluidity is even more generic. The notion of recursive support of arbitrary depth of sub-classing and decomposition needs additional extensions to the NFV information model to be properly supported. Moreover, similar changes will also need to be introduced to the MANO reference points (APIs) or the corresponding NFVO toolchain. Last but not least, Superfluidity API must tackle the so called "optimal function allocation problem" consists in mapping a set of network services (decomposed into RFBs) to the underlying, available hardware block implementations in the RFB Execution Environment (REE). It aims at minimizing resource usage while meeting individual SLAs. Solving the optimal function allocation problem is a work-in-progress, and its API is by no mean definitive.

3.7.3 VNF Packaging Specification

NFVO is also responsible for managing VNF Packages, stored in the VNF Catalogue. Details are published in the VNF Packaging Specification [44], but at a high level they contain:

- VNF descriptor that defines metadata for package on-boarding and VNF management
- Software images needed to run the VNF
- Optional additional files to manage the VNF (e.g. scripts, vendor-specific files etc.)

Based on our detailed review of the NS template, VNF descriptor and VNF packaging specification docs, the information model entities support decomposition up to the VNF component (VNFC).

4 Platform Orchestration

Orchestration is the automated arrangement, coordination and management of computer systems, middleware and services. Especially in cloud computing and contemporary telecommunication infrastructure environments, virtualized networks can span a large number of networks, software elements, and hardware platforms, NFV orchestration tools must be powerful and able to work with many different standards. This leads to an overall orchestration definition, expanded to incorporate elements such as architectural composition of tools and processes, software and hardware component stitching and workflow connection and automation towards delivering a pre-defined service. Any NFV orchestrator must support functions for delivering: (i) service coordination and instantiation through software able of communicating with the underlying NFV platform, (ii) service chaining, allowing a service to be cloned and multiplied when needed, (iii) service scaling by allocating sufficient resources for service delivering and (iv) service monitoring, where the performance of both platform and resources are tracked to ensure meeting certain quality standards.

These requirements fuelled the management and organization (MANO) working group of ETSI decision to create a framework for NFV, which breaks down the management and orchestration needs of the NFV architecture, introducing an NFV Orchestrator (NFVO). The NFV Orchestrator provides management of the NFV services, which is responsible for new Network Service (NS) and VNF package onboarding, together with NS lifecycle and global resource management, validation and authorization of network functions virtualization infrastructure (NFVI) resource requests.

Superfluidity Project focused on the evaluation of such a framework, the Open Source MANO (OSM), which delivers an open source management and orchestration stack aligned with ETSI NFV Information Models, providing an a regularly updated implementation of a production quality MANO stack that meets current and future requirements of commercial NFV networks.

4.1 OSM Release 0

The architectural components of Open Source MANO Release Zero are presented in Figure 16. OSM integrated OpenMANO [45], Canonical's Juju Server [46] and RiftWare [47] to operate as Resource Orchestrator, Configuration Manager and Service Orchestrator respectively, all connected through a dedicated VLAN. The minimal infrastructure requirements for installing and running OSM Release Zero was a single server split into three different VMs. In order to efficiently divide the server into VMs a hypervisor was required, allowing optimal VM configuration and virtual resource allocation.

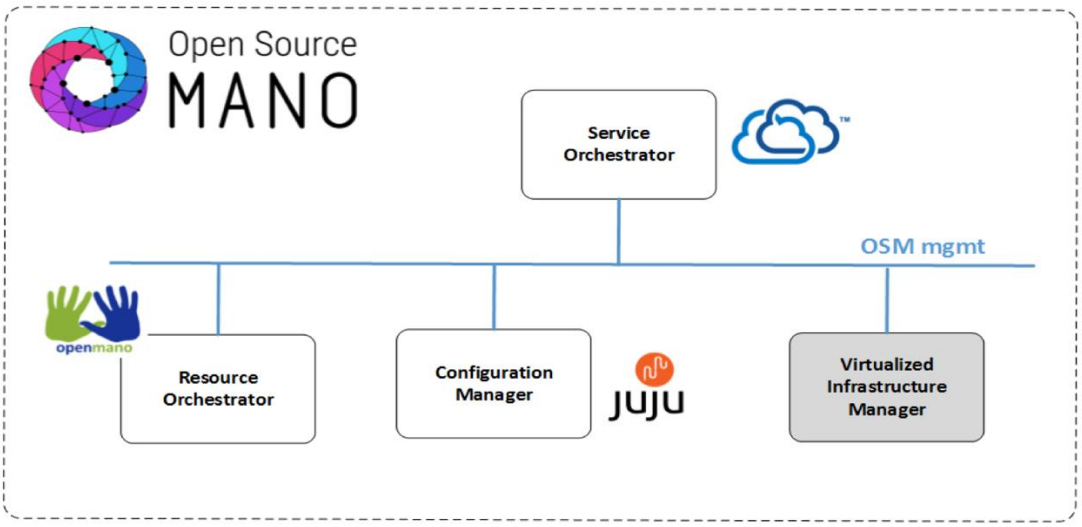

Figure 16: Open Source MANO - Release Zero

OpenMANO Installation

In order to install OpenMANO, a virtual node with minimum 1 vCPU, 2 GB of RAM, 40 GB of available disk space and 1 network interface connected to the OSM management network, is needed. The necessary base operating system namely Canonical's Ubuntu 14.04.4 LTS should be installed in advance, for the OpenMANO installation process to begin. One could install OpenMANO automatically using certain installation scripts or manually, by installing all required packages including mysql-server, configure python-argcomplete, clone the git repository and situate in the v0.0 tag, configure the previously installed MySQL by granting access privileges from localhost and add openmano client and scripts to the PATH. Once the OpenMANO installation was finished, users need to configure the openmano server, the openmano local CLI client, create an openmano tenant and attach the newly created openmano tenant to the virtual infrastructure manager datacenter, in our case an OpenStack Mitaka-based deployment.

Juju Installation

For installing a single VM with both Juju server and client, a total of 4 vCPUs, 4 GB of RAM, 40 GB of available disk space, along with the necessary OSM mgmt. network interface, are needed. Similar to OpenMANO, the base operating system is also Canonical's Ubuntu 14.04.4 LTS. Unlike OpenMANO, the only necessary package and libraries pre-requisites are Juju Server and Client which also need to be configured accordingly, as described in [48].

RiftWare Installation

The specific OSM component requires 1 vCPU, 8 GB of RAM, 40 GB of available disk space on the hypervisor to hold the disk image as well as a network interface while the base operating system is Fedora 20. In order to setup a build environment suitable for building RiftWare's major component,

RIFT.ware, there are two different approaches: (i) download a pre-built image or (ii) connect to the dedicated ETSI repository and download the RIFT.ware source code. Then, initiate the build and once this process is concluded, RIFT.ware Launchpad service can be deployed. It is possible to connect to the service via a web browser, seamlessly through a self-signed certificate.

When all three virtual machines are properly instantiated, the VIM of choice can be installed and configured. Even in this preliminary OSM release, the community offers specific VNFs for testing which can be downloaded for the application's repository. Despite the fact that each OSM release is deprecated once the next release is available, all necessary documentation wikipages are still available.

4.2 OSM Release One

Open Source MANO Release One, greatly simplifies the overall installation process by using an all-in-one installation script. One of the major configuration changes of the specific release is that all services that were deployed in VMs in Release Zero, are now launched in containerized format. The installation script, also handles the Linux Bridge configuration, allowing the OSM node to establish connectivity with the outside world and exchange configuration information. OSM Release One now only requires a single server or VM with 8vCPUs, 16 GB of RAM, 100 GB of hard disk space and a single interface with internet access. The base OS is now Canonical's Ubuntu 16.04, configured to run LXD containers. Figure 17 summarizes the new architectural approach of OSM Release One, while Table 4 compares the amassed hardware and software requirements of Releases One and Zero.

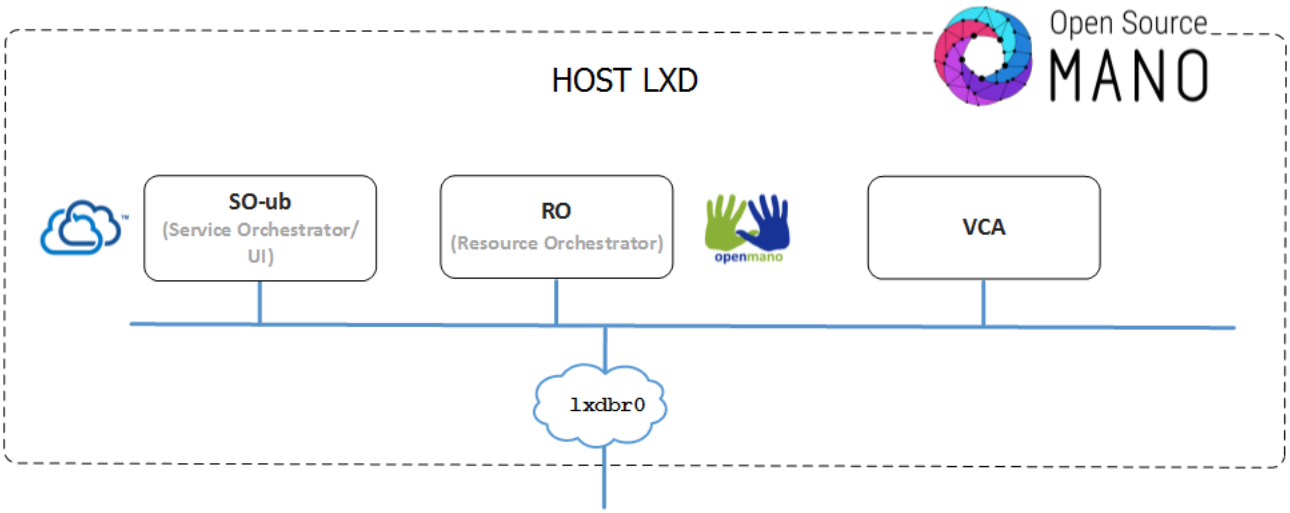

Figure 17: Open Source MANO - Release One

Table 4: OSM Release Comparison

OSM Release	vCPU (min/rec)	RAM (min/Rec)	HDD
Zero	6/24	14/40 GB	120/140 GB
One	**8**	**16 GB**	**100 GB**

Component	Linux Distro	Rec. Version	Avail. Updates
RO/CM	Ubuntu 14.04.4	0.4.38/1.25	-/2.0 (16.04 LTS)
SO	Fedora 22	4.2.0	4.2.1
HOST LXD	Ubuntu 16.04 LTS	-	-

Open Source MANO Release One Deployment steps can be summarized as follows:

- Pre-requisites
  - Configure VM
  - Install Ubuntu 16.04 LTS
  - Configure Networking
  - Configure LXD container operation
- Main OSM Installation Process
  - Obtain/execute installation script
  - Access the UI using Chrome to avoid additional SSL certificate configuration issues

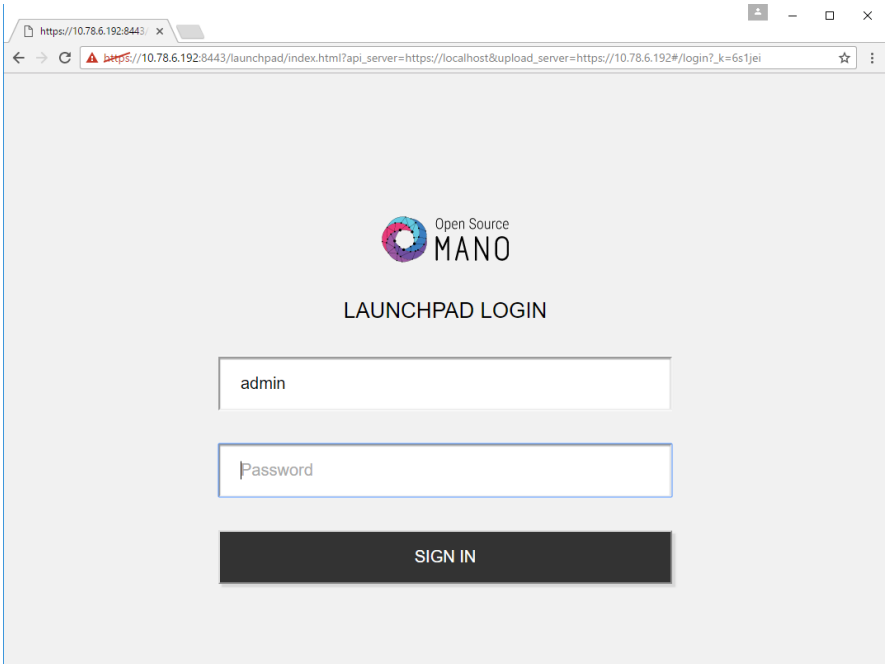

Figure 18: OSM Release One Login Screen

During the evaluation phase of OSM for the Superfluidity project, we decided to install a later OpenStack distribution than Mitaka, in order to efficiently allocate possible compatibility issues.

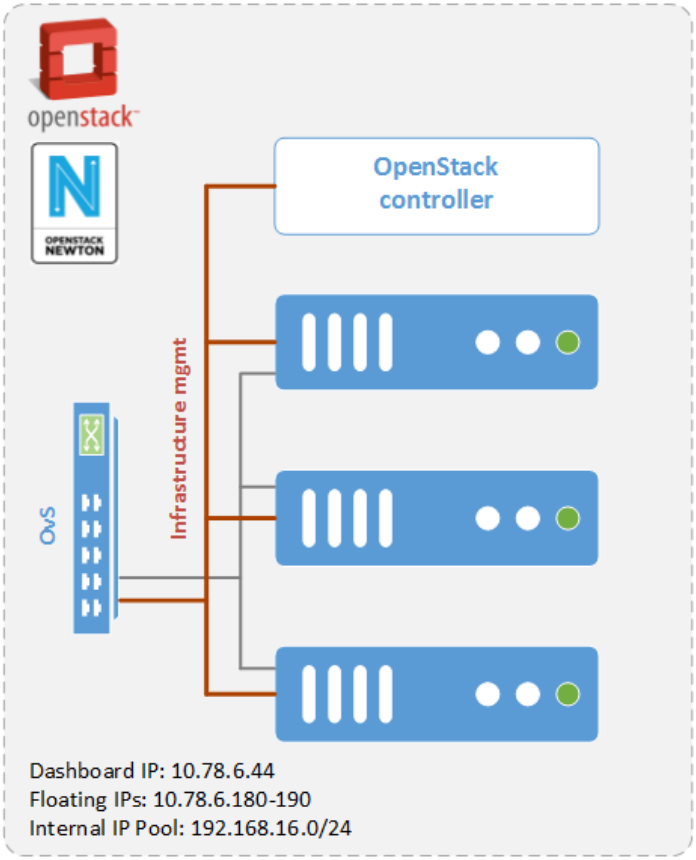

Figure 19: OpenStack Newton Deployment for OSM-VIM testing

Once the OpenStack node is deployed, additional steps are needed for configuring the node to be used by OSM, as well as attach it to the OSM itself. In particular, it is required to access the OpenStack node, create the management network along with a valid tenant/user, upload the images that will later be used by the OSM through VNFs for spawning VMs, and modify the default security group. It is advised to save all preliminary testing images under the /mnt/powervault/virtualization directory, since it is tested to be identified seamlessly by the VNFD provided by the OSM community for validation purposes.

Certain configuration steps are needed on the OSM node as well. In particular one must access the RO container, export the tenant and use CLI commands to add the OpenStack node. However, two issues are present in OSM Release One, that somehow might lead to confusion. Despite the available menu for adding OpenStack (VIM in general) nodes via the GUI, the whole process fails. At the moment, CLI commands are the only functional option for adding VIMs in the OSM node and even so, added datacenters do not appear in the GUI at all. Executing CLI commands for discovering attached datacenters reveals the available VIMs, however GUI and CLI inconsistency is present throughout the specific release. Figure 20 presents the integrated OpenStack Newton / Open Source MANO Release One testbed in which the Network Service of Figure 21 consisting of two CirrOS-based

VNFs connected by a simple VLD will be deployed. In order to proceed (i) download the required VNF and NS packages from this URL: https://osm-download.etsi.org/ftp/examples/cirros_2vnf_ns/, (ii) obtain a CirrOS 0.3.4 image (iii) upload the image into the /mnt/powervault/virtualization directory of the OpenStack node and (iv) onboard the image into OpenStack using the following command:

```
openstack image create --file="./cirros-0.3.4-x86_64-disk.img" --container-format=bare --disk-format=qcow2 cirros034
```

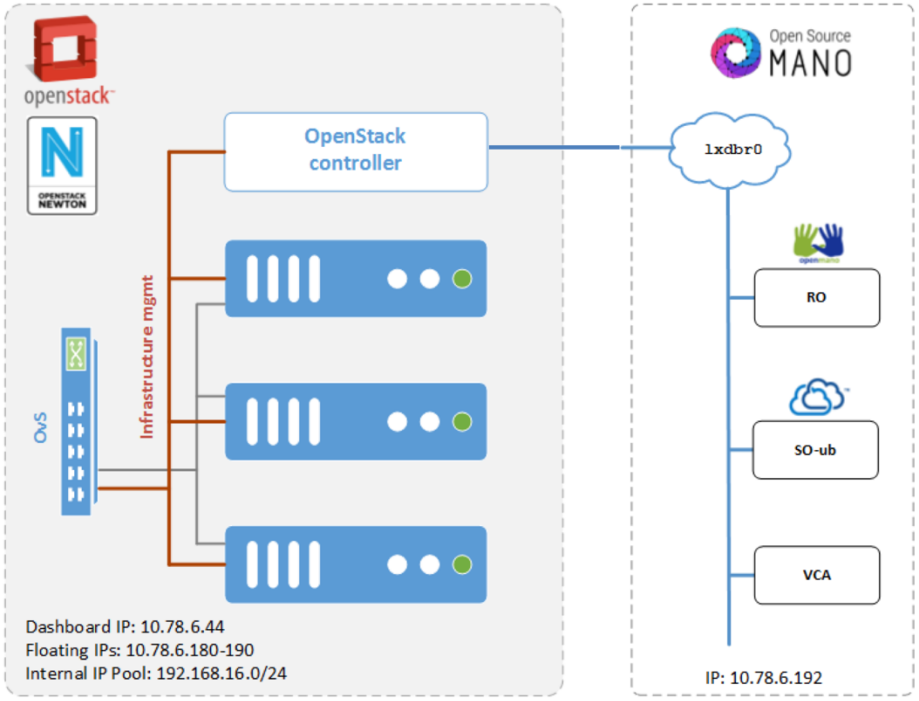

Figure 20: OpenStack Newton - OSM Release One testbed

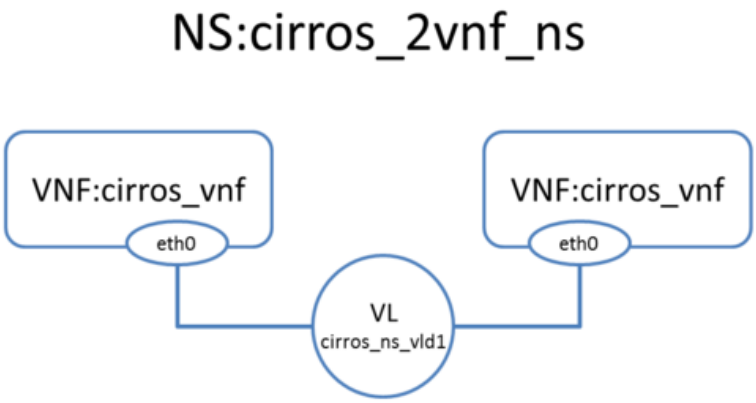

Figure 21: Validation Network Service deployment

Onboarding VNF and NS

In order to successfully onboard VNF and NS it is required to (i) access the OSM node GUI using Chrome, (ii) go to CATALOG > Import, (iii) select the element using the corresponding tab as shown

in Figure 22 VNFD for VNF or NSD for NS, (iv) Drag and drop the corresponding package in the importing area, cirros_nfv.tar.gz for VNF or cirros_2vnf_ns.tar.gz for NS.

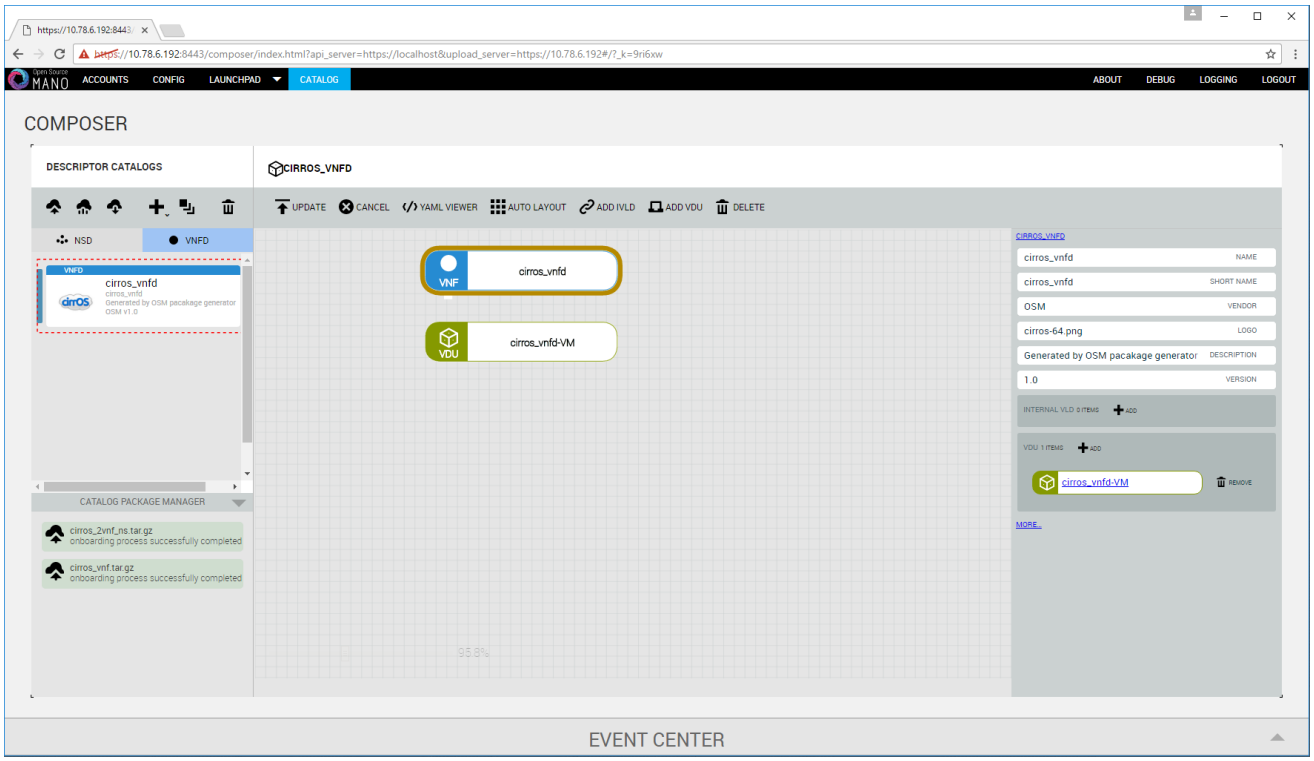

Figure 22: VNFD Onboarding

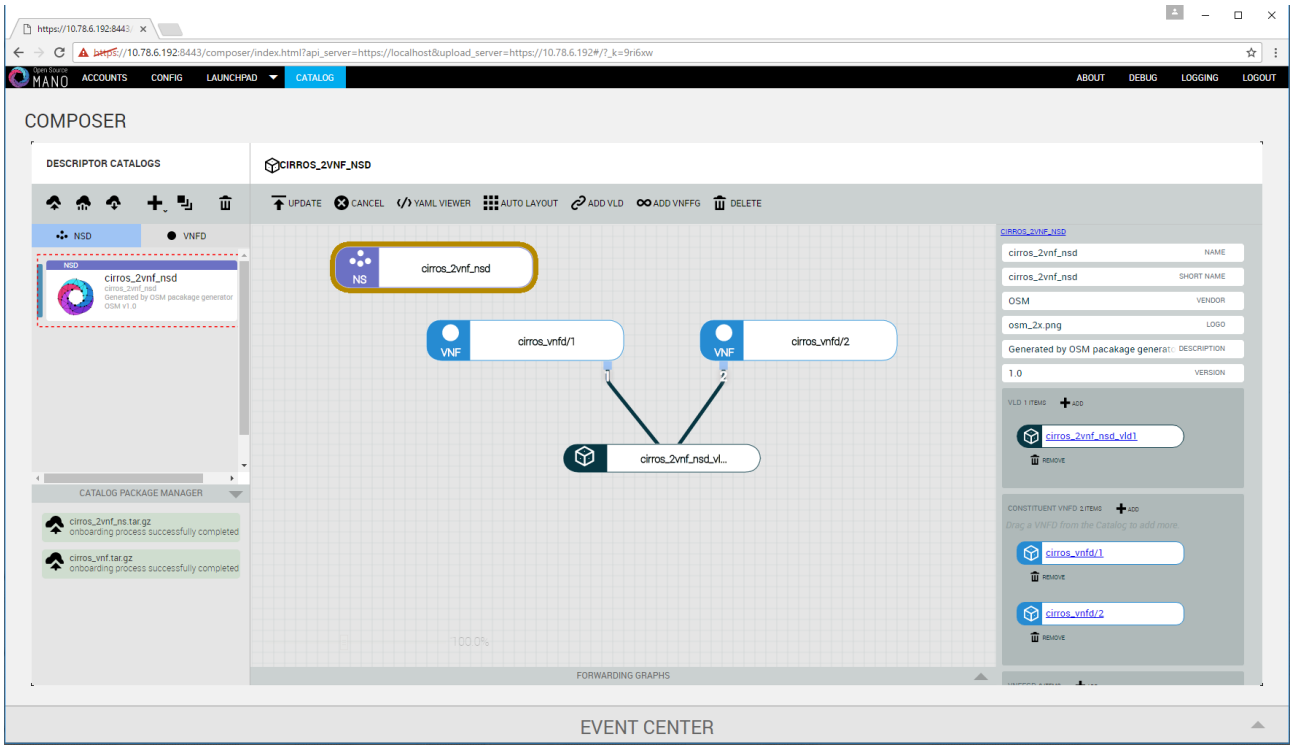

Figure 23: NSD Inspection

Instantiating the NS

For instantiating the NS (i) go to Launchpad > Instantiate, (ii) Select the NS descriptor to be instantiated and click Next (Figure 24), (iii) add a name to the NS instance and click Launch (Figure 25).

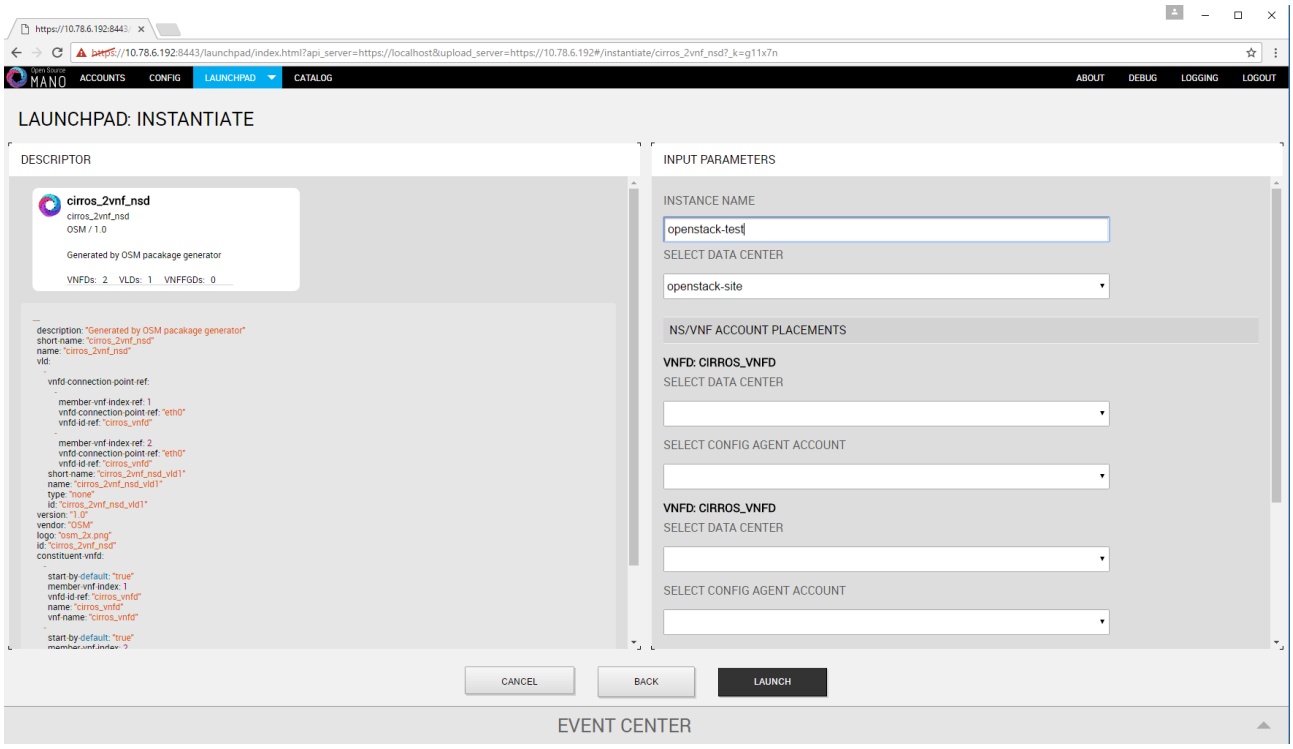

Figure 24: NS Instantiation

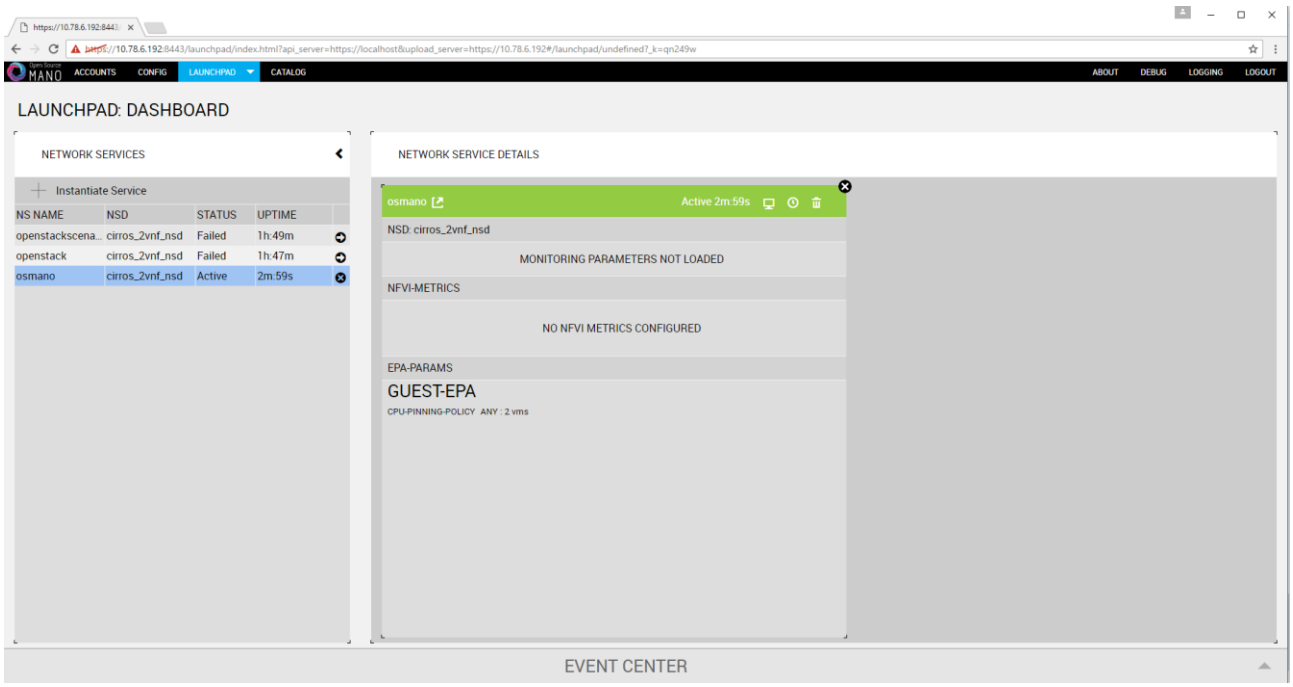

Figure 25: Successfully Instantiating the NS from the OSM Node perspective

Any successful Network Service instantiation made via the Open Source MANO Orchestrator is also visible through the Network Topology graph of the attached OpenStack VIM, as shown in Figure 26, 27 and 28 for the validation NS scenario previously deployed.

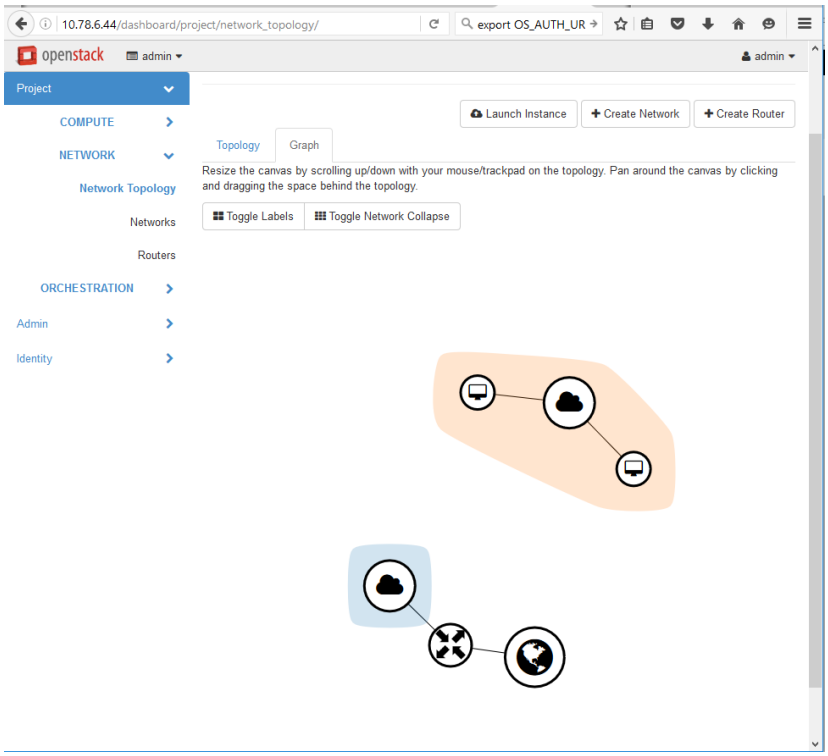

Figure 26: OpenStack Graph representing the validation NS deployment

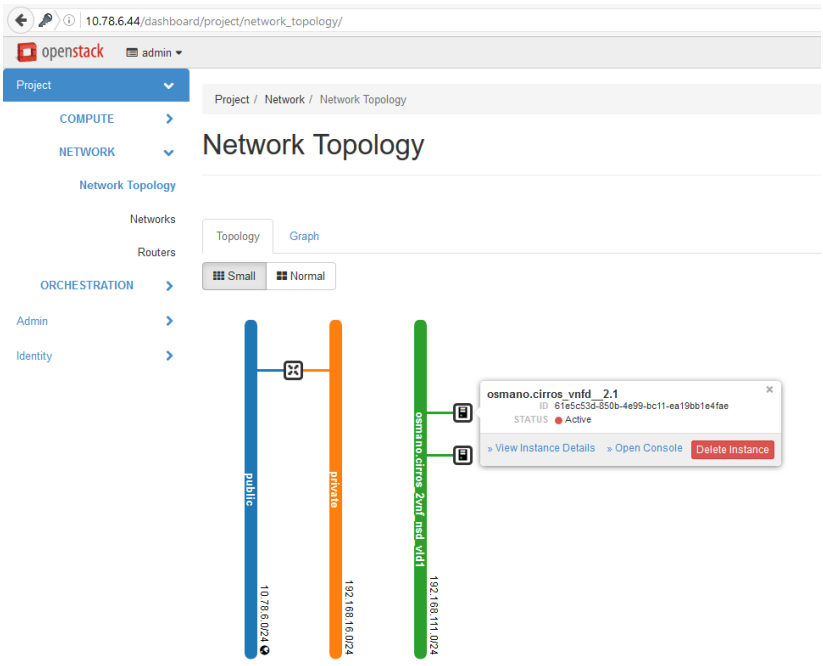

Figure 27: OpenStack Network Topology representing the validation NS deployment

4.3 OSM Release Two

Continuing to meet operators' needs for predictable, high-quality Open Source MANO releases, the ETSI Open Source MANO group (ETSI OSM) announced in April 27, 2017 the OSM Release TWO. This release brings significant improvements in terms of interoperability, performance, stability, security, and resources footprint to meet operators' requirements for trials and upcoming RFx processes.

According to the press release new features of OSM Release TWO include:

- SDN assistance to interconnect traffic-intensive virtual network functions with on-demand underlay networks
- Support for deployments in hybrid clouds through a newly developed Amazon Web Services plugin
- OSM's plugin model for major SDN controllers has been extended with ONOS, which joins ODL and FloodLight in the list of supported controllers
- Dynamic network services to scale resources on demand
- Multiple installer options to ease OSM installation in different environments

The new SDN capabilities enable advanced types of underlay connectivity that are often unavailable in a non-customized, virtualized infrastructure manager (VIM), thus avoiding performance degradation. This is transparent for operators, who only need to request the right type of connectivity in their virtual network functions or network service descriptors without concerns about the need for special hardware, server wiring or manual post-deployment intervention. OSM Release Two will be evaluated by the Superfluidity project partners and the results will be presented in an upcoming version of this deliverable.

5 Superfluidity Mechanisms, Algorithms and Innovations

Any network operator is able to build services by combining Virtual Network Functions using an orchestrator (NFVO). The orchestrator should be able of processing the descriptors of both services (NSDs) and VNFs (VNFDs), as well as interact with the manager (VIM) of the Virtualized Infrastructure (NFVI) towards deploying the components which implement the service and interconnect them properly. In particular, the virtual network function descriptors incorporate the Virtual Deployment Units (VDUs), mapped into virtual machines capable of being deployed over the NFVI. Superfluidity Project proposes a generalization and an extension of those architectural models through the introduction of the Reusable Functional Blocks (RFBs), the RFB Execution Environment (REE) and the RFB Description and Composition Languages (RDCLs). An RFB is a generalization and an abstraction of VDUs, VNFs and NSs. In the ETSI NFV MANO reference architecture, there is a fixed hierarchy amongst these three elements, however Superfluidity proposes a more general concept for RFBs where nested functionality and iteration is supported, the REE not only represents the abstraction of the actual execution environment but also includes the tools used for deploying and running components and services, and RDCLs are used to vaguely describe all participating components inside an execution platform, their mutual interactions as well as their composition towards building other components or final services [52]. For a more detailed analysis of these notions and technologies, please refer to Superfluidity Deliverable 3.1.

5.1 RDCL 3D

The vision of Superfluidity project is to orchestrate functions dynamically over and across heterogeneous environments, hence the need to interpret and operate with different RDCLs. At least in the short-medium term, it is most likely that a variety of RDCLs will coexist and together with “meta-orchestrators” will coordinate the configuration mapping of resources over different REEs. To facilitate the transition even further, our project proposes RDCL 3D – Reusable Functional Blocks Description and Composition Language Design, Deploy and Direct, a novel “model-agnostic” web framework for descriptor design.

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 28 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 [51] is released under the Apache 2.0 Open Source license, to facilitate its uptake by research and industrial communities [52]. In addition, an alpha release 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.

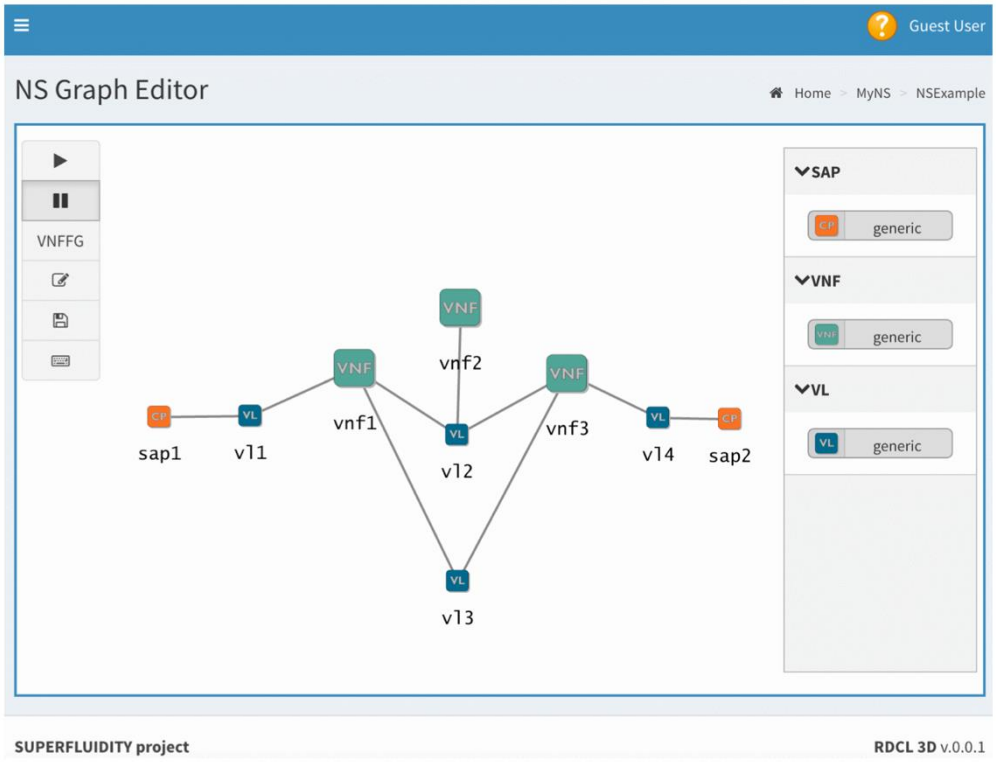

Figure 28: Network Service design using RDCL 3D GUI

Currently the RDCL framework supports the models defined by the latest ETSI NFV ISG specifications [34, 44], the TOSCA Simple Profile for NFV [49], the TOSCA Simple Profile in YAML [57] and the network elements described in the Click configuration language [50]. However, RDCL 3D is designed for extensibility in order to support new models or combine existing ones. With reference to ETSI MANO architecture, RDCL 3D can be used as a standalone tool to edit the NS and VNF descriptors, as shown in Figure 29 (1). This approach is currently implemented in the demo for the ETSI and TOSCA models. The produced descriptor files can be retrieved and provided to an Orchestrator (NFVO) but this is not automated in the current implementation. We are currently working on extending RDCL 3D to support the direct interaction with the programmatic APIs of relevant Orchestrators. This is an interesting direction which supports one additional use case, where RDCL 3D becomes the meta-orchestrator. 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 Figure 29 (2).

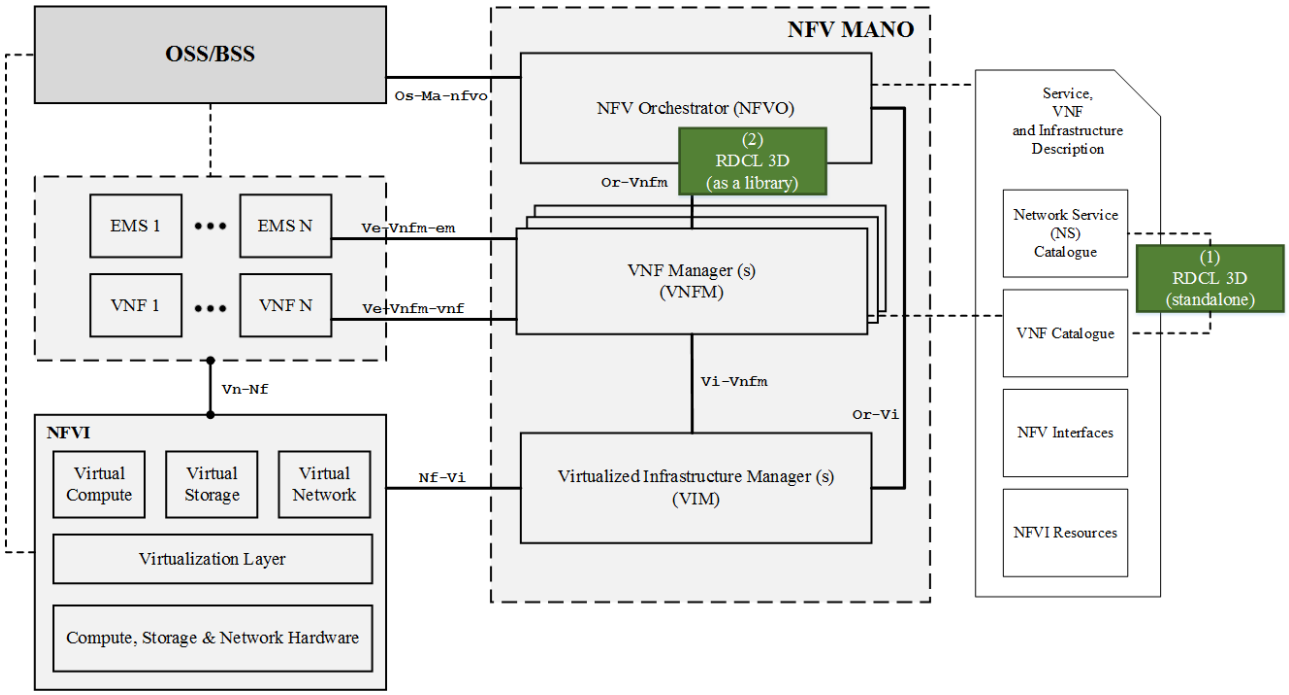

Figure 29: Positioning the RDCL 3D in the ETSI MANO architecture

Software Architecture

RDCL 3D is a web application with backend and frontend components, clearly depicted in Figure 30. 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 30). 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 30), where is it processed and filtered to produce the different graphical views on the browser GUI (<3> in Figure 30). 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).

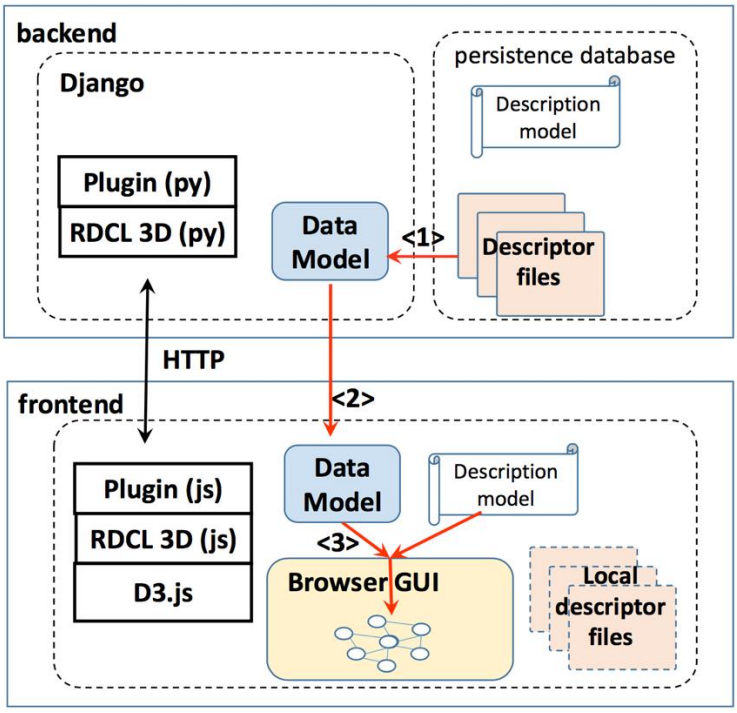

Figure 30: RDCL 3D Software Architecture

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 [51]

For simplifying 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:

I. include the parser to translate the descriptor files into the Data model representation in the Python plugin (backend);

II. 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;
III. develop the project type specific processing operation in the JavaScript plugin (frontend).

Overall, RDCL 3D is a web framework for visualizing and edit 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.

5.2 SEFL and Symnet

Software verification is impossible unless one accurately identifies the properties the final product should meet. Analysing a program's specifications through program annotations, is a popular method amongst system designers and developers since it allows condition placement prior, after or in parallel with function calls. The verification process occurs during runtime – whenever a call to a function that carries annotations is encountered, the preconditions are checked, and then, upon return from the function, the post-conditions on the result are checked. A program is considered correct with respect to a set of annotations if at any given time during its execution, all conditions are true. For further validating these conditions, approaches based on static verification (prior to deployment) and/or runtime checking are often used.

Symnet [53] is a symbolic execution tool tailored for static network analysis. The tool rapidly analyses networks by injecting symbolic packets and tracing their paths through the network, even in cases of increased network complexity where networks contain routers with hundreds of thousands of prefixes and NATs. The overall process is completed in a few seconds, while ensuring packet header memory-safety and capturing network functionality such as dynamic tunnelling and encryption. Symbolic execution is a rather powerful method, since it explores all possible paths through the program, providing possible values for each (symbolic) variable at every point. For static network analysis, the power of symbolic execution lies in its ability to relate the outgoing packets to the incoming ones: even if all incoming packet headers are unknown, a symbolic execution engine can detect which parts of the packet are invalid through the network, and can tell how the modified headers depend on the input when their change occurred. The only limitation of symbolic execution is its inherently poor scaling ability due to the exponential complexity introduced by highly complex networks.

Superfluidity managed to address this limitation of symbolic execution by introducing SEFL – Symbolic Execution Friendly Language, a language specifically developed to model network processing in way

that is amenable to symbolic execution. As described in D3.1 [54], in SEFL any header field can be accessed using either the numeric offset of the header inside the packet or using a string alias identifying the header (IPSrc, MACDst etc). To specify conditions on fields SEFL provides support for the usual boolean operators (and, or, not etc), arithmetic comparison operators (>, <, !=, ==) etc. A brief overview of the SEFL syntax is listed in Table 5.

Table 5: SEFL Syntax

INSTRUCTION	DESCRIPTION
`Allocate(v[,s,m])`	Allocates new stack for variable v, of size s. If v is a string, the allocation is handled as metadata and the optional m parameter controls its visibility: it can be global (default) or local to the current module. If v is an integer it is allocated in the packet header at the given address; size is mandatory.
`Deallocate(v[,s])`	Destroys the topmost stack of variable v; if provided, the size s is checked against the allocated size of v. The execution path fails when the sizes differ or there is no stack allocated for variable v.
`Assign(v,e)`	Symbolically evaluates expression e and assigns the result to variable v. All constraints applying to variable v in the current execution path are cleared.
`CreateTag(t,e)`	Creates tag t and sets its value e, where e must evaluate to a concrete integer value.
`DestroyTag(t)`	Destroys tag t
`Constrain(v,cond)`	Ensures that variable v always satisfies expression cond. The execution path fails if it doesn't.
`Fail(msg)`	Stops the current path and prints message msg to the console.
`If (cond, instr1, instr2)`	Two execution paths are created; the first one executes instr1 as long as cond holds. the second path executes instr2 as long as the negation of cond holds.
`For (v in regex, instr)`	Iterates through all variables that match the name given by regex, executing instruction instr.
`Forward(i)`	Forwards this packet on exit port i. Used in the context of Click elements.
`InstructionBlock(i1,…)`	Groups a number of instructions that are executed in order
`NoOp`	Nothing occurs

Every instruction implicitly takes as parameter the current execution state and outputs a new one. The state includes header variables and metadata together with their values and constraints. The `Allocate` and `Deallocate` instructions create both header fields and metadata, depending on the parameter provided. If `v` is a string, the variable is metadata and is not aligned in any way, and


memory safety checks do not apply. `v` acts as a key in a Map managed by the symbolic execution engine. If `v` is an integer (or an expression that evaluates to an integer), it is treated like a header field, and the associated memory checks are performed. SEFL also offers instructions to create and destroy tags. New tags can be created at absolute values (used when the packet is created), or relative to other tags (used for encapsulation of an existing packet). SEFL includes two instructions that constrain the execution of the current path, that have no direct correspondent in C. `Fail` stops the current execution path and prints an error message. `Constrain` applies a constraint to a variable, stopping the current path if the constraint does not hold. `Constrain` allows programmers to model filtering behaviour without branching. If forks the current execution state. On one path, it applies the constraint and executes instr1. On the "else" branch is applies the negated constraint and executes in- str2. If more than one instruction must be executed on any branch, an `InstructionBlock` should be used that groups more instructions into a single compound instruction. If any branch is empty, `NoOp` can be used instead. `For` iterates over all variables that match the given regular expression. The match is computed when the for starts, and the loop in unfolded before it is executed, and creates no branches by itself.

5.2.1 OpenStack Neutron Configurations

We have written an OpenStack plugin that takes the router and firewall configurations and translates them into SEFL models. These could be used to check reachability before deployment, or to check that the actual deployment matches the user's intent; we are still working on integrating the results from symbolic execution back into Neutron to make it easily available to the users.

5.3 NEtwork MOdelling language

NEMO, the NEtwork MOdelling language has already been described in the scope of Superfluidity in D3.1 (section 4.6). In the architecture description it was shown how NEMO could be used for describing the composition of network resources. In order to make them more powerful, it was described how NEMO would be extended in order to cope with Virtual Network Function Descriptors (VNFDs).

Making use of this extension, we can use the NEMO language as a description model for the previously described RDCL 3D tool, using Nodemodels in a similar way to VNFs, and Network Intents as NSDs, as can be seen in Figure 31.

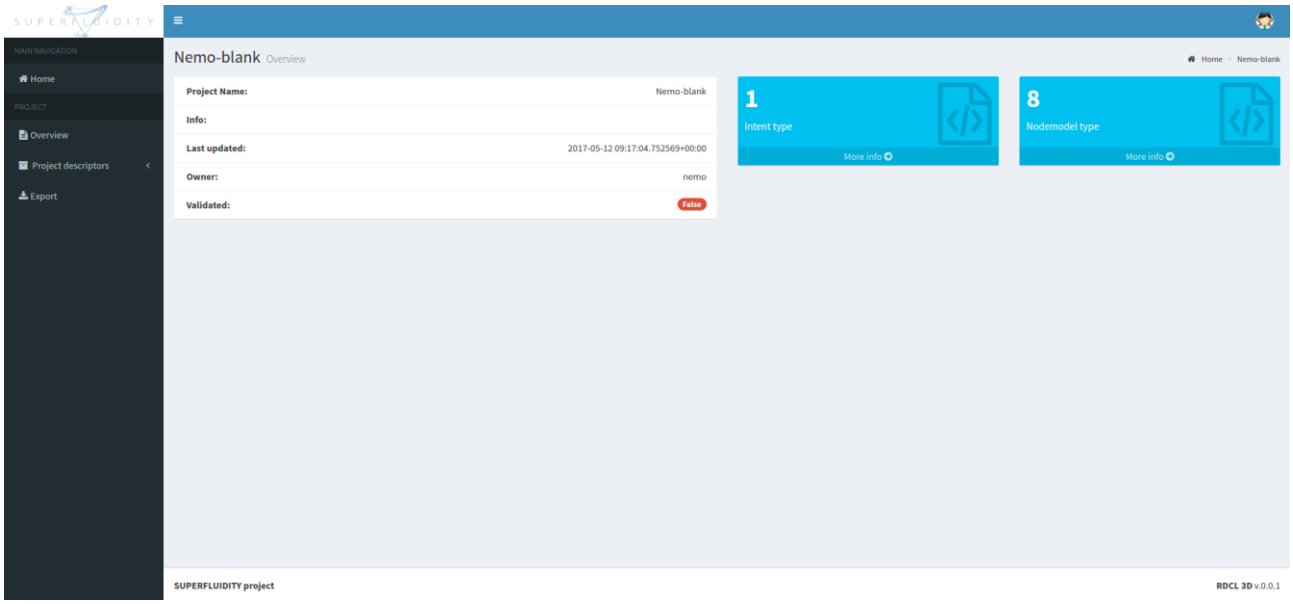

Figure 31: Overview of a NEMO project in the RDCL 3D tool

With the RDCL 3D tool, it is possible to create Nodemodels and Intents from files or direct text input, editing them later like in Figure 32 and, in a near future, visualize a graph of the described network.

Figure 32: Editing a NEMO intent using the RDCL 3D tool

6 Conclusion

Based on our detailed review of the Network Service template, VNF descriptor and VNF packaging specification docs, the information model entities support decomposition up to the VNF component (VNFC). However, as we have described previously, the RFB model proposed by Superfluidity is more general than the NFV model, since it can recursively support an arbitrary depth of sub-classing and decomposition. As a consequence, we will have to amend the NFV information model to support the RFB model of Superfluidity. Moreover, similar changes will have to be introduced to the MANO reference points (APIs), or the corresponding toolchain of the NFVO.

Something else that is important to consider is the concrete textual representation of the information model elements. Practically speaking, this specifies the syntax of the files that will represent the NS templates, VNF descriptors, VNF forwarding graph descriptors, etc. Based on the current practice of NFVOs and VNFMs, which seem to also be adopted by the ETSI standardisation efforts, the elements of the NFV Information Model are represented using a YAML syntax, and increasingly compliant with the OASIS TOSCA Simple Profile for NFV. Based on the current discussions, and under the assumption that the NFVOs of choice are aligned with this decision, Superfluidity will adopt YAML-based file formats, and, to the extent possible, TOSCA-based. But given the abovementioned requirement for a recursive composition of RFBs, the top-level RDCL is envisioned to utilize the NEMO language instead.

Finally, the descriptors of RFBs, for which formal analysis of correct behaviour is available, can be extended to include the relevant representation. In that context, we aim to extend them to optionally embed the domain-specific language adopted by Superfluidity, SEFL. This will facilitate automatic execution of the symbolic execution engine, SymNet, for the subset of life-cycle transitions that would require (re-)verifying the composition of RFBs.

7 References

[1] IBM, "Hypervisors, virtualization, and the cloud: Learn about hypervisors, system virtualization and how it works in a cloud environment" [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/cl-hypervisorcompare-pdf.pdf [Accessed: 26/04/2017]

[2] Kernel Virtual Machine [Online]. Available: http://www.linux-kvm.org/page/Main_Page [Accessed: 26/04/2017]

[3] Tselios C., Tsolis G. (2016) A Survey on Software Tools and Architectures for Deploying Multimedia-Aware Cloud Applications. In: Karydis I., Sioutas S., Triantafillou P., Tsoumakos D. (eds) Algorithmic Aspects of Cloud Computing. Lecture Notes in Computer Science, vol 9511. Springer

[4] Open Networking Foundation “OpenFlow” [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow [Accessed: 28/04/2017]

[5] Red Hat, “Understanding OpenStack” [Online]. Available: https://www.redhat.com/en/topics/openstack [Accessed:28/04/2017]

[6] Linux Foundation, “Open vSwitch” [Online]. Available: http://openvswitch.org/ [Accessed: 27/04/2017]

[7] OpenStack Project “Scope of the Nova Project” [Online]. Available: https://docs.openstack.org/developer/nova/project_scope.html [Accessed 28/04/2017]

[8] OpenStack Project, “OpenStack Bare Metal Provisioning” [Online]. Available: https://wiki.openstack.org/wiki/Ironic [Accessed 28/04/2017]

[9] Cisco, “Introduction to Cisco IOS NetFlow”, White Paper [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.pdf [Accessed: 28/04/2017]

[10] OpenStack Project “Magnum” [Online]. Available: https://wiki.openstack.org/wiki/Magnum [Accessed 28/04/2017]

[11] OpenStack Project Wiki "Neutron" [Online]. Available: https://wiki.openstack.org/wiki/Neutron [Accessed 28/04/2017]

[12] Open vSwitch Documentation [Online]. Avaialble: http://docs.openvswitch.org/en/latest/intro/why-ovs/ [Accessed 29/04/17]

[13] Open vSwitch "Classic OvS scenario" [Online]. Avaialble: https://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html [Accessed 29/04/17]

[14] sFlow Protocol [Online]. Available: http://www.sflow.org/ [Accessed 29/04/17]

[15] HAProxy [Online]. Available: http://www.haproxy.org/ [Accessed 28/04/17]

[16] HAProxy Documentation [Online]. Available: http://cbonte.github.io/haproxy-dconv/1.7/intro.html [Accessed 28/04/17]

[17] OpenStack Documentation “HAProxy” [Online]. Available: https://docs.openstack.org/ha-guide/controller-ha-haproxy.html [Accessed 28/04/17]

[18] OpenStack Project “Neutron/LBaaS” [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS [Accessed 28/04/17]

[19] OpenStack Project “Load Balancer as a Service (LBaaS)” [Online]. Available: https://docs.openstack.org/mitaka/networking-guide/config-lbaas.html [Accessed 28/04/17]

[20] OpenStack Orchestration [Online]. Available: https://wiki.openstack.org/wiki/Heat [Accessed 28/04/17]

[21] Heat Orchestration Template (HOT) specification [Online]. Available: https://docs.openstack.org/developer/heat/template_guide/hot_spec.html? [Accessed 28/04/17]

[22] YAML Ain't Markup Language (YAML) [Online]. Available: http://yaml.org/ [Accessed 29/04/17]

[23] Puppet [Online]. Available: https://puppet.com/ [Accessed 29/04/17]

[24] Chef [Online]. Available: https://www.chef.io/chef/ [Accessed 29/04/17]

[25] OpenStack Mistral Wiki [Online]. Available: https://wiki.openstack.org/wiki/Mistral [Accessed 28/04/17]

[26] Yet Another Query Language (YAQL) [Online]. Available: https://pypi.python.org/pypi/yaql/1.0.0 [Accessed 29/04/17]

[27] Jinja 2 : Python Templating Engine [Online]. Available: http://jinja.pocoo.org/docs/dev/ [Accessed 29/04/17]

[28] OpenStack Telemetry Wiki [Online]. Available: https://wiki.openstack.org/wiki/Telemetry [Accessed 28/04/17]

[29] OpenStack Telemetry Documentation [Online]. Available: https://docs.openstack.org/admin-guide/telemetry.html [Accessed 28/04/17]

[30] OpenStack Ceilometer Developer Documentation [Online]. Available: https://docs.openstack.org/developer/ceilometer/ [Accessed 28/04/17]

[31] ETSI Industry Specification Group (ISG) for NFV Home Page, http://www.etsi.org/technologies-clusters/technologies/nfv

[32] ETSI GS NFV-IFA 013, "NFV MANO, Os-Ma-Nfvo reference point - Interface and Information Model Specification", V (2016-10)

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

[34] ETSI GS NFV-IFA 014, "NFV MANO, Network Service Templates Specification", V2.1.1 (2016-10)

[35] ETSI GS NFV-IFA 005: "NFV MANO, Or-Vi reference point - Interface and Information Model Specification". V2.1.1 (2016-04)

[36] ETSI GS NFV-IFA 006: "NFV MANO, Vi-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-04)

[37] ETSI GS NFV-IFA 007: "NFV MANO, Or-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-10)

[38] ETSI GS NFV-IFA 008: "NFV MANO, Ve-Vnfm reference point - Interface and Information Model Specification". V2.1.1 (2016-10)

[39] ETSI GS NFV-IFA 002 "NFV Acceleration Technologies, VNF Interfaces Specification", V2.1.1 (2016-03)

[40] ETSI GS NFV-IFA 003 "NFV Acceleration Technologies, vSwitch Benchmarking and Acceleration Specification " V2.1.1 (2016-04)

[41] ETSI GS NFV-IFA 004 "NFV Acceleration Technologies, Management Aspects Specification" V2.1.1 (2016-04)

[42] ETSI GS NFV-IFA 010: "NFV MANO, Functional requirements specification" V2.1.1 (2016-04)

[43] ETSI GS NFV-IFA 010: "NFV MANO, Functional requirements specification" V2.2.1 (2016-10)

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

[45] OpenMANO [Online]. Available: https://github.com/nfvlabs/openmano [Accessed 28/04/17]

[46] Canonical, Juju [Online]. Available: https://www.ubuntu.com/cloud/juju [Accessed 28/04/17]

[47] Rift.io, Rift.Ware [Online]. Available: https://www.riftio.com/tag/rift-ware/ Accessed 28/04/17]

[48] OSM, "Juju Installation in R0" [Online]. Available: https://osm.etsi.org/wikipub/index.php/Juju_installation_(release_0) [Accessed 28/04/17]

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

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

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

[52] Salsano S, et al “RDCL 3D, a Model Agnostic Web Framework for the Design of Superfluid NFV Services and Components”, arXiv preprint arXiv:1702.08242, 2017

[53] Stoenescu R, et al “Symnet: scalable symbolic execution for modern networks”, Available: https://arxiv.org/pdf/1604.02847.pdf

[54] Superfluidity Project, Deliverable D3.1, “Final system architecture, programming interfaces and security framework specification”

[55] OpenStack Octavia [Online] Available: https://wiki.openstack.org/wiki/Octavia [Accessed 28/04/17]

[56] Red Hat Ansible [Online], Available: https://www.ansible.com/[Accessed 28/04/17]

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

.