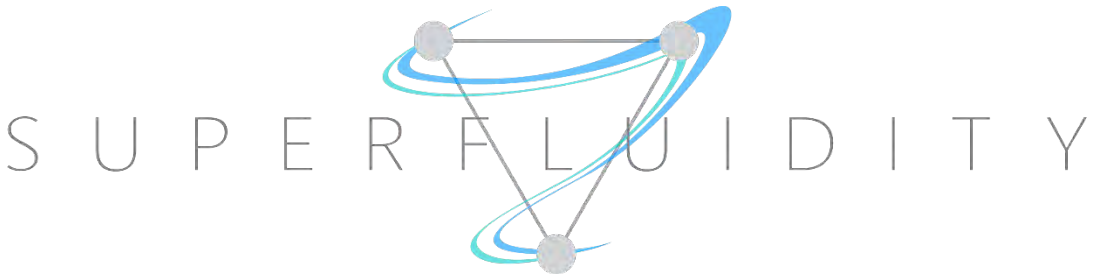

SUPERFLUIDITY

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

Research and Innovation Action GA 671566

DELIVERABLE I6.1B:

DESIGN FOR CONTROL FRAMEWORK

Deliverable Type:	Report
Dissemination Level:	CO
Contractual Date of Delivery to the EU:	31 May 2017
Actual Date of Delivery to the EU:	1 Jun 2017
Workpackage Contributing to the Deliverable:	WP6
Editor(s):	Luis Tomas (Red Hat) Livnat Peer (Red Hat) Daniel Mellado (Red Hat)
Author(s):	Carlos Parada, Isabel Borges, Francisco Fontes (Altice Labs), George Tsolis (Citrix), Michael McGrath, Vincenzo Riccobene (Intel), John Thomson, Julian Chesterfield, Joel Atherley, Manos Ragiadakos (OnApp), Luis Tomas, Livnat Peer, Daniel Mellado (Red Hat), Erez Biton (ALU-IL), Stefano Salsano, Claudio Pisa (CNIT)
Internal Reviewer(s)	Elisa Rojas (Telcaria)

Abstract: This deliverable reports the efforts of task 6.1. It focuses on the control framework: the requirements analysis, the framework design and its components, and the implementation and testing of the provided functionality to the different superfluidity framework components.

Keyword List: Orchestration, Management, VMs, Containers

Table of Contents

Glossary

SUPERFLUIDITY DICTIONARY	
TERM	DEFINITION
UE	User Equipment
OSS	Operation Support System
VIM	Virtual Infrastructure Management
VM	Virtual Machine
MANO	Management And Orchestration
NFV	Network Function Virtualization
KPI	Key Platform Indicator

Table 1: SUPERFLUIDITY Dictionary.

1. Executive Summary

The present document is the second document of its series (first version was I6.1) and covers the next points:

- Updates the requirements gathered for the management and control framework.
- Extends the state of the art revision with new tools/components that have been studied/evaluated/used during this period
- Introduces the Superfluidity management and control framework for multi-site deployments (core, central and edge clouds)
- Presents the contributions made so far to the different components of the ones presented to tackle the main existing gaps required to achieve the Superfluidity objectives.

All requirements are assigned a unique name, for future reference and own the following format: [ReqName-XX] where XX enumerates the same property requirements.

2. Introduction

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

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

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

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

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

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

3. Requirements analysis (**taken from I6.1, just small revision and table formatting**)

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

3.1. NFV Technical Requirements

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

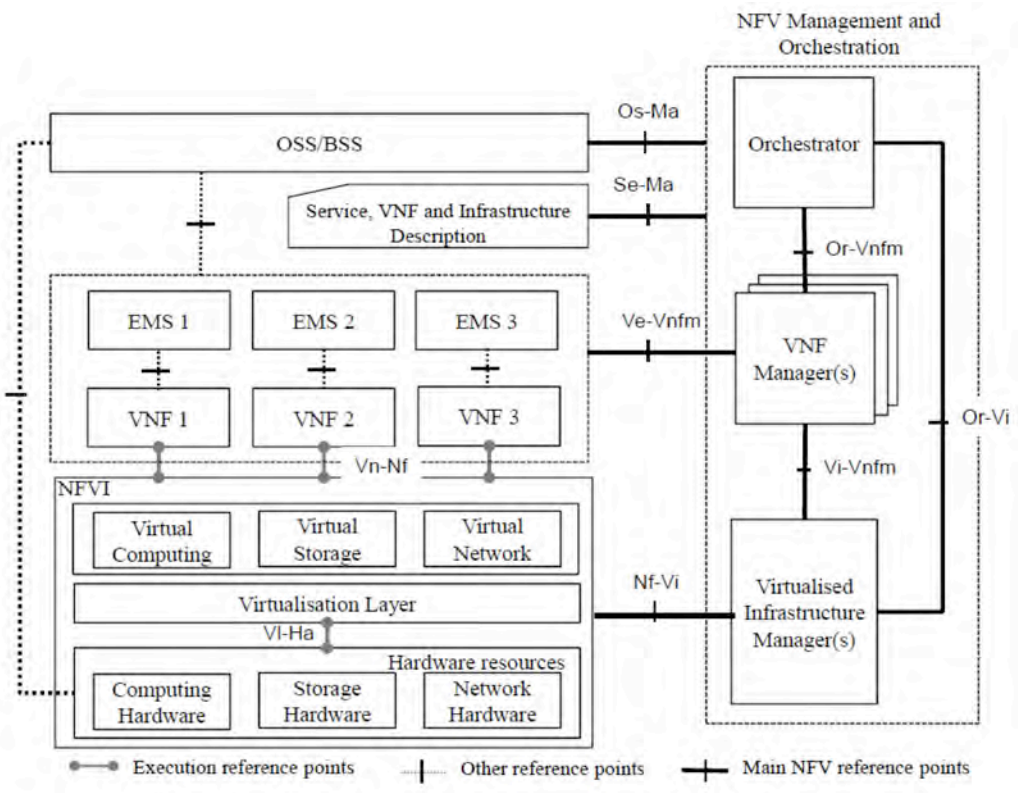

Figure 1: ETSI NFV reference architecture [ETSI-NFV]

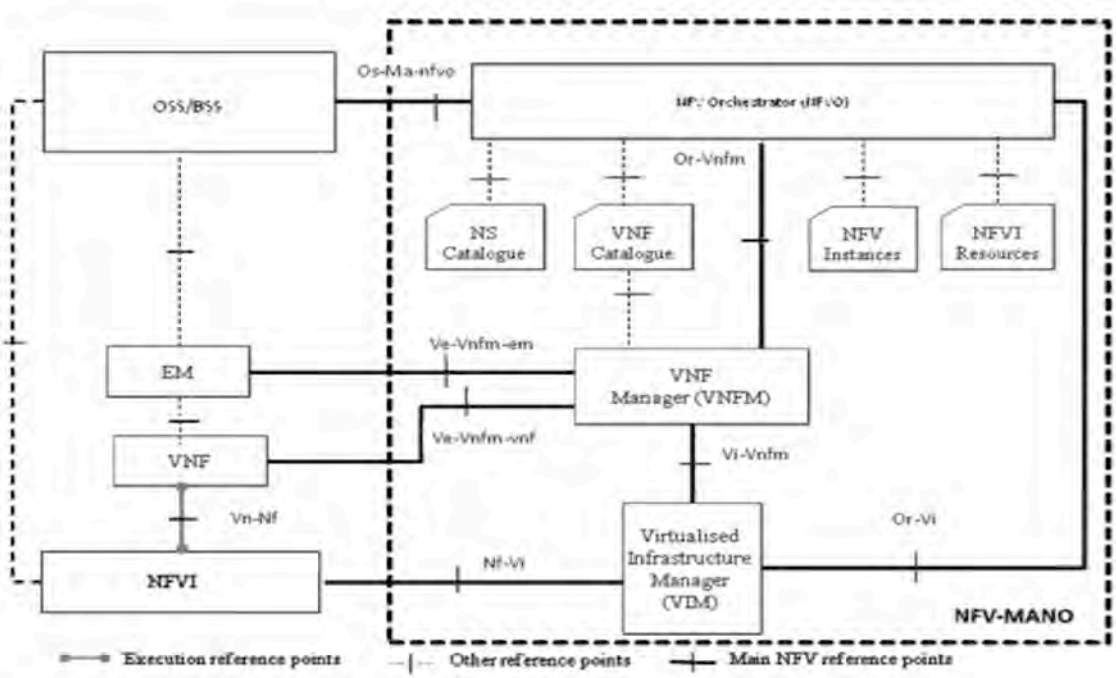

Figure 2: ETSI NFV MANO reference architecture [ETSI-NFV-MANO]

Next table describes high-level technical requirements for a NFV management and orchestration system. More detailed requirements and flows can be found in Annexes 8.1.1 and 8.2.1, respectively, of document I6.1.

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

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

	[LB-03] The MANO framework SHOULD ideally support firewall load balancing mode. However, this MAY require addressing gaps in NFVI/VIM (OpenStack LBaaS doesn't appear to support this case).
Service Function Chaining	[SFC-01] The MANO framework MUST support the creation of Service Function Chains (SFCs), consisting of an ordered sequence of Service Functions (SFs). SFs are virtual machines, or even physical devices, that perform a network function such as firewall, content filter, content cache, or any other function that requires processing of packets in a flow.
	[SFC-02] The MANO framework MUST support SFCs with both simple (i.e. single SF) and complex (i.e. sequence of multiple SFs) Service Functions Paths (SFPs). Materialisation of SFCs requires the cooperation of the NFV Orchestrator, VIM and SDN controller. The NFV-O provides the VNFFG definition (please refer to relevant requirements in this document), the VIM creates the SFC by attaching the SF VM instances to network ports and the SDN controller configures the network overlay fabric that interconnects these network attachment points. According to the OPNFV SFC project (https://wiki.opnfv.org/display/sfc), SFC also depends on the VNF Manager: http://artifacts.opnfv.org/sfc/brahmaputra/docs/design/architecture.html#vnf-manager
	[SFC-03] The MANO VIM MUST support the attachment of SF VM instances to network ports to construct SFPs (for more details on how OpenStack aims to implement this capability, please refer to http://docs.openstack.org/developer/networking-sfc/system_design%20and_workflow.html and http://docs.openstack.org/developer/networking-sfc/api.html).
	[SFC-04] A Service Function (SF) MAY consist of a cluster of VM instances, which can be used for load balancing (please also see 2.2.2.5). The load balancing function MUST have the option to be sticky (i.e. sessions in progress must be sent through the same SF VM instance). The load balancing function MUST also have the option to ensure symmetric return traffic.
	[SFC-05] The MANO VIM MUST be extensible to support the creation ("rendering") of SFPs in conjunction with different SDN controllers and renderers (e.g. OpenFlow, NETCONF, etc.). The support of SFC-related requirements by the OpenDaylight SDN controller is described below: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main

	[SFC-06] The MANO VIM MAY support a network overlay function that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch). For a complete implementation of SFC, the MANO framework would need to also support orchestration of the SFC Classifier, Service Function Forwarder (SFF) and SFC Proxy building blocks. For more information on how OpenStack aims to support these SFC functions, please refer to http://docs.openstack.org/developer/networking-sfc/ovs_driver_and_agent_workflow.html).
	[SFC-07] The MANO VIM SHOULD support orchestration of SFC Classifiers. The MANO VIM MAY offer an implementation of an SFC Classifier that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-08] The MANO VIM SHOULD support the orchestration of Service Function Forwarder (SFF). The MANO VIM MAY also offer an implementation that is part of NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-09] The MANO VIM SHOULD support orchestration of SFC Proxies. The MANO VIM MAY offer an implementation of an SFC Proxy that is part of the NFV infrastructure (OpenStack will provide a reference implementation using Open vSwitch).
	[SFC-10] The reference implementation of the SFF, SFC Classifier and SFC Proxy (if available) SHOULD support the preferred SFC encapsulation scheme, NSH (please see IETF draft-ietf-sfc-nsh). Please note that an initial implementation of a subset of the SFC requirements above was made available in OPNFV Brahmaputra, as a combination of OpenDaylight, OpenStack and Open vSwitch: https://wiki.opnfv.org/display/PROJ/Project+Proposals+Service+Function+Chaining An overview of how OPNFV Brahmaputra puts all the pieces together: http://artifacts.opnfv.org/sfc/brahmaputra/docs/design/index.html Further progress is apparently being made, targeting OPNFV Colorado: https://wiki.opnfv.org/display/sfc/OPNFV+SFC+Colorado+Release+Plan Finally, the requirements for supporting VNF Forwarding Graphs are outlined below: https://wiki.opnfv.org/display/PROJ/Openstack+Based+VNF+Forwarding+Graph

3.2. MEC Technical requirements

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

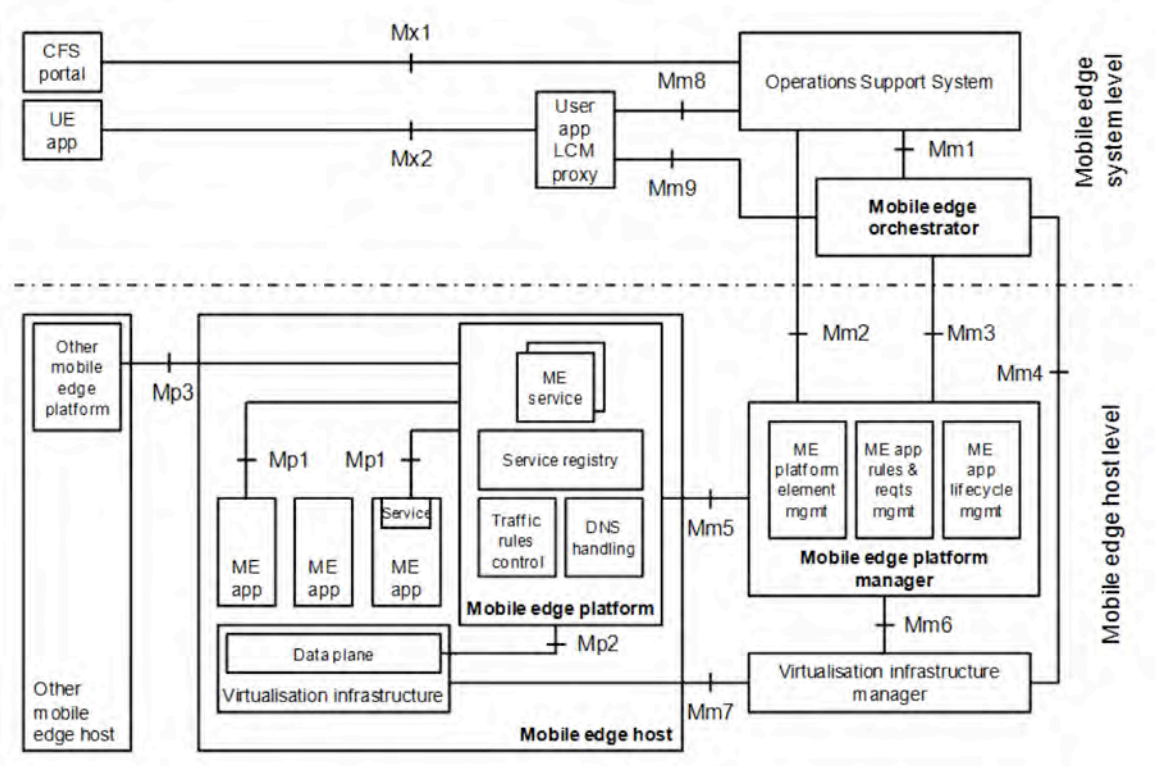

Figure 3: ETSI MEC reference architecture [ETSI-MEC]

The next table describes high-level technical requirements for a MEC management and orchestration system. More detailed requirements and flows can be found in Annexes 8.1.1 and 8.2.1, respectively.

Type	Requirements description
Application lifecycle	[Lifecycle-01] The management system MUST support the following application lifecycle management (LCM) operations: • Instantiation • Scaling • Relocation • Modification • Termination

	[Lifecycle-02] The management system MUST be able to receive and process application LCM requests: ● From the OSS, a third-party, or a UE application ● Based on LCM rules.
	[Lifecycle-03] The management system MUST be able to identify the mobile edge features and services an application requires to run. This will be the input for the decision on which mobile edge host to provision the application.
	[Lifecycle-04] The management system shall support the instantiation and termination of a running application when required by the operator.
Application scheduling and instantiation	[Instantiation-01] The management system MUST be able to deploy the application on mobile edge hosts in various locations, both in a central data center and at the edge of the Core Network.
	[Instantiation-02] The management system MUST support the following deployment application models: ● One App instance per MEC Host, serving multiple users ● Multiple App instances per MEC Host, each serving a single user
	[Instantiation-03] The management system MUST support the indication of the following virtualized resources: ● Compute ● Storage ● Network resources ● Specific hardware support
	[Instantiation-04] The management system MUST support the indication of the following network connectivity resources: ● Connectivity to local networks ● External connectivity to the Internet ● Access to user traffic
	[Instantiation-05] The management system MUST support the indication of the following latency requirements: ● Maximum ● Expected
	[Instantiation-06] The management system MUST support the indication of physical location (edge).

	[Instantiation-07] The management system MUST support the indication of service requirements: • Mandatory - for MEC Apps to be able to operate. • Optional - for MEC Apps can benefit from, if available.
	[Instantiation-8] The management system MUST consider cost requirements, which can be a translation of the operator's estimation for the deployment costs.
Mobility support	[Mobility-01] The management system MUST support multiple MEC Hosts in different locations, including radio sites, aggregation points, or at the edge of the Core Network.
	[Mobility-02] The MEC system MUST guarantee service continuity while the UE moves across the network (between different edges and cells).
	[Mobility-03] The MEC system MUST be able to perform application instance relocation for MEC Apps dedicated to a single user.
	[Mobility-04] The MEC system MUST be able to perform application state relocation for MEC Apps serving multiple users.
KPI's support	[KpiTemplate-01] – The system MUST be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA's and SLO's.
	[KpiScaling- 01] – The system MUST be able to determine the number and types of resources to support workload scaling in order to maintain KPI's and SLO's.
	[Monitoring-01] – The MEC system MUST be able to collect infrastructure and service monitoring information, in order to feed KPI-based automated management and orchestration features.
Network traffic control	[TControl-01] The management system must be able to provide provisioned MEC platforms with guaranteed network bandwidth.
	[TControl-02] The management system must be able to rate limit the provisioned MEC platforms traffic flows.
	[TControl-03] The management system must have the ability to selectively apply the traffic control on different types of traffic, and have the ability of traffic classification.

	[TControl-04] Within the constraints set by the orchestration and management, an authorized mobile edge application shall be able to request the activation, update and deactivation of the mobile edge application traffic rules dynamically.
Scaling	[Scaling-01] – The MEC system MUST be able to scale a MEC App, on OSS request or automatically based on KPIs, in order to increase/decrease the capacity.
	[Scaling-02] The MEC system MUST be able to terminate a MEC App whenever it is no longer required to serve users.

3.3. C-Ran Technical Requirements

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

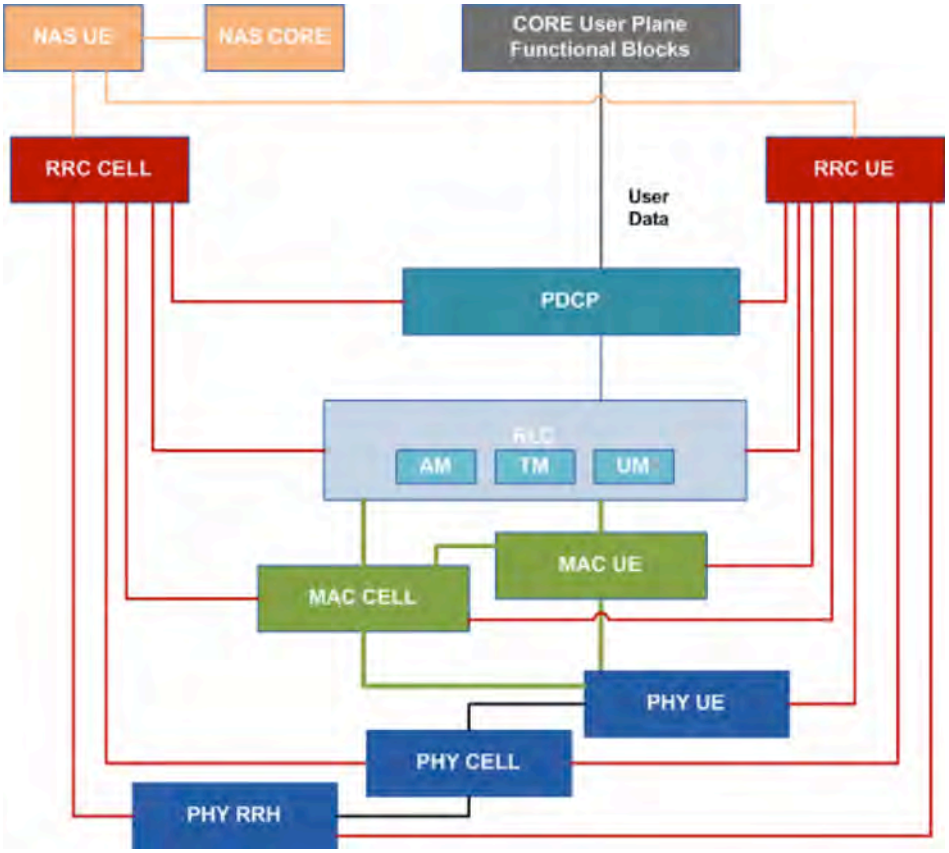

Figure 4: Affinity graph between different C-RAN functional blocks

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

FUNCTIONAL BLOCK (FB)	EXAMPLES OF FB DECOMPOSITION	FB DEPLOYMENT LOCATION	EVENT HANDLING CAPACITY ()	APPLICATION SCALING REQUIREMENT
PHY RRH	Physical NF – not virtualized	Antenna site		Not scalable as application
PHY Cell	all the processes executed for one cell, e.g. FFT/iFFT, Modulation, Cyclic prefix	Antenna site or Front-End Cloud	Every 10 ms (LTE radio frame length)	Scaling decision may be reactive (based on computational latency of previous frame). Less than 10 ms requirement.
	Joint Multiuser Detection – jointly process the received signals from multiple UE from more than one RAP (MTPD, INS)		New UE could arrive or leave asynchronously. Scaling decision should be based on current computational latency and next state prediction	Scale in/out may be dependent on UE mobility. About 5-10 seconds worst case (bus, or train travelling between RAPs)
PHY User (UE)	HARQ must be sent 3 ms after receiving the frame	Front-End Cloud or EDGE cloud	New frame every 10 ms (LTE radio frame length) , but events that results capacity dependent on UE mobility.	Scale in/out may be dependent on UE mobility. About 5-10 seconds worst case (bus, or train travelling between RAPs)
	Convolution coding			
MAC Cell/Scheduling Real Time	ICIC (Intercell Interference Coordination)	Front-End Cloud or EDGE cloud	Every 10 ms (LTE radio frame length), Works with a cluster of RAP's, scaling events not coming in peaks.	Number of minutes in most cases, dependent on UE mobility. About 5-10 sec

	link adaptive part	antenna site	Dependent on current antenna measurements, need to be executed locally on antenna site, latency sensitive	10 ms If implemented in proactive fashion could be less time sensitive
MAC User (UE)	UE Power control	EDGE cloud	LTE case it can happen maximum 1000 times within a second per ue. capacity is Number of users in 1ms	Not coming in peaks, 5-10 seconds worst case
RLC	It includes processes related to segmentation/concatenation of PDCP PDUs based on information exchange with MAC and PDCP. Several modes are supported: Transparent, Acknowledged and Unacknowledged. Each case could be a separate FB	EDGE cloud	*Dependent on the mobility and traffic intensity of UE. For the EDGE cloud slow change in number of ue associated with it.*	Number of minutes to scale
PDCP Packet Data Convergence Protocol	transfer of user plane data, transfer of control plane data, header compression, ciphering, integrity protection.	EDGE cloud or Central cloud	Dependent on ue activity levels: would change through the day in predictable manner (peak in the morning, less activity in the night, etc.)	Scaling not strict time constrained, and predictable. number of times in a day
RRC Cell		EDGE cloud or Central cloud		
RRC User (UE)	Handover UE measurements reporting, QoS management,	EDGE cloud or Central	About 30% of UE are in the	Scaling not strict time constrained, number

	paging	cloud	handover state, so with central deployment number of scaling events in a day	of times in a day
NAS User (UE)	It refers to the user procedures related to signaling between the UE and MME	EDGE cloud or Central cloud	Asynchronous, depends on user mobility. Because of deployment on central cloud slow change in number of the users in the whole network	Scaling not strict time constrained, number of times in a day
NAS Core	MMEs load balancing, MME overload control, GTP-C signaling load control…	EDGE cloud or Central cloud		

3.4. NFV vs. MEC Comparison

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

4. State of the art

4.1. VIM OpenStack Virtual Infrastructure Management (taken from I6.1)

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

Network Traffic Control

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

Scheduling parameters

In order for the orchestration layer to be able of making a 'smart' scheduling decision, the VIM has to expose the required set of parameters for the orchestrator to take into account. However, at this point in time, most of they are not supported. On the upper side - requirements [AppSched-05] (description of the virtualized resources) can be satisfied by the usage of templates provided by such projects as Heat and Tacker as well as [AppSched-06] (Required network connectivity description). However, on the downside requirement [AppSched-08] (Physical location requirements) is hardly fulfilled. The possible solutions to accomplish that can be by made by the usage of OpenStack's Nova (compute project) regions and cells accompanied by custom Nova scheduler filters, a solution we're planning to research and experiment with in the following time frames.

Mobility support

While the OpenStack Nova (compute) project provides support for a subset of functionality for migrating VM instances from one physical host to another, it lacks some of the properties required for full mobility support: [Mobility-01], [Mobility-02]. The user of the migration feature in its current form cannot specify the physical host on which the VM will be migrated, as this decision is left out to the scheduler. In addition to it, this process does not assume that the VM instance has sufficient storage available on the target host, and potentially can fail.

KPI Support

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

Current service orchestration approaches are based on the use of pre-defined configurations for the node(s) hosting the workloads. The Orchestrator then requests instantiation of the pre-

defined configuration to bring the workload into service on specific hardware platform, for instance through usage of pre-compiled deployment templates (i.e. OpenStack Heat Orchestration Templates (HOT), TOSCA descriptors, etc.). These templates are managed by orchestration platforms through the use of catalogues, (for instance, OpenStack Murano project can be used to store and manage HOT templates for OpenStack Heat). However, this approach does not scale efficiently. As the number of different services to be supported by the platform increases as well as the granularity of service specific KPIs (Key Platform Indicators) and SLOs (Service Level Objectives) it results in a huge number of deployment templates to supported deployment of services. A more effect approach may be based around the use of dynamic template definitions at deployment time to meet specified KPI's as described in section 4.1.1.

4.2. VNFM/NFVO

4.2.1. Cloudband (taken from I6.1)

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

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

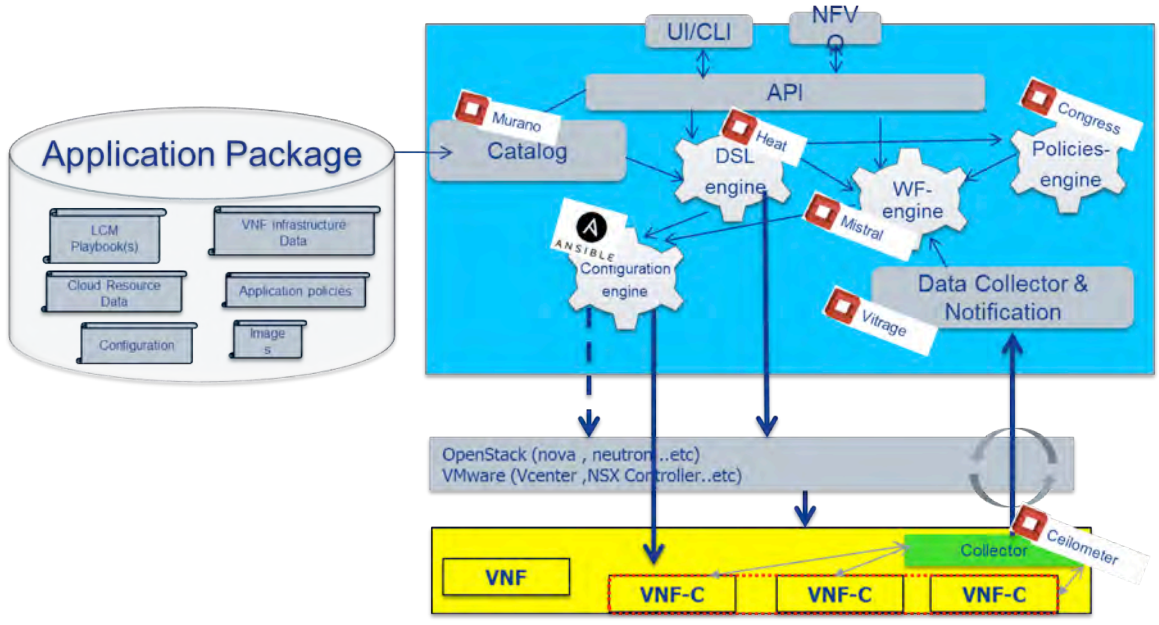

Figure 5: Openstack based generic VNF management system

Figure 5 depicts the architecture for the VNF management system. As indicated, the architecture is based on OpenStack services, such as: Heat, Mistral, Murano, Ceilometer, Vitrage and possibly Congress. In addition, it utilizes Ansible as an open source configuration management. This architecture can support all of the operations that are required for a VNF lifecycle management, including deployment, monitoring, scaling healing and termination (as depicted in Figure 6).

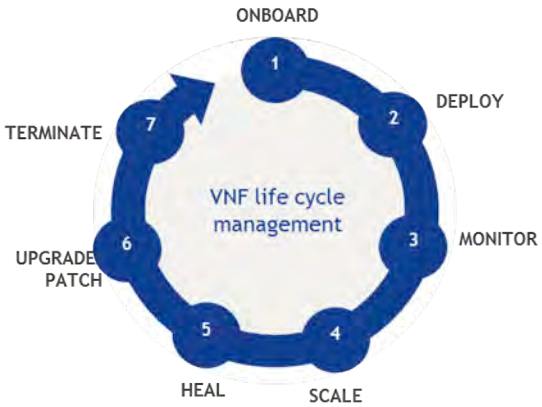

Figure 6: VNF lifecycle operation

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

After the onboarding process is complete, the second LCM stage—Deployment takes place (Figure 7).

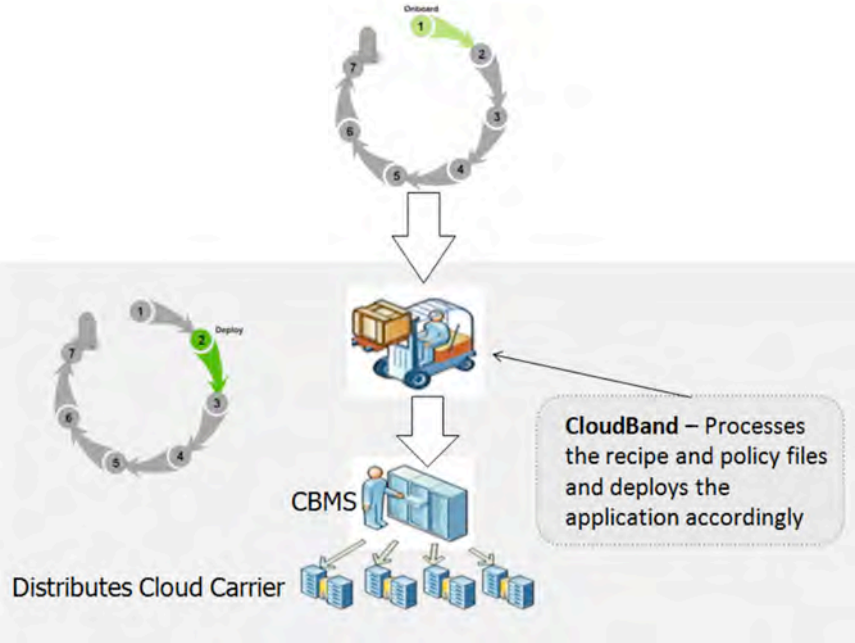

Figure 7: The deployment workflow

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

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

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

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

For each deployment, a job will be created.

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

4.2.2. OpenMANO

OpenMANO implements components from the ETSI NFV MANO stack. Currently, the situation with regards to the requirements outlined in Section 2 is the following:

Network Traffic Control

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

Scheduling parameters

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

Mobility Support

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

KPI Support

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

4.2.3. Open Baton

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

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

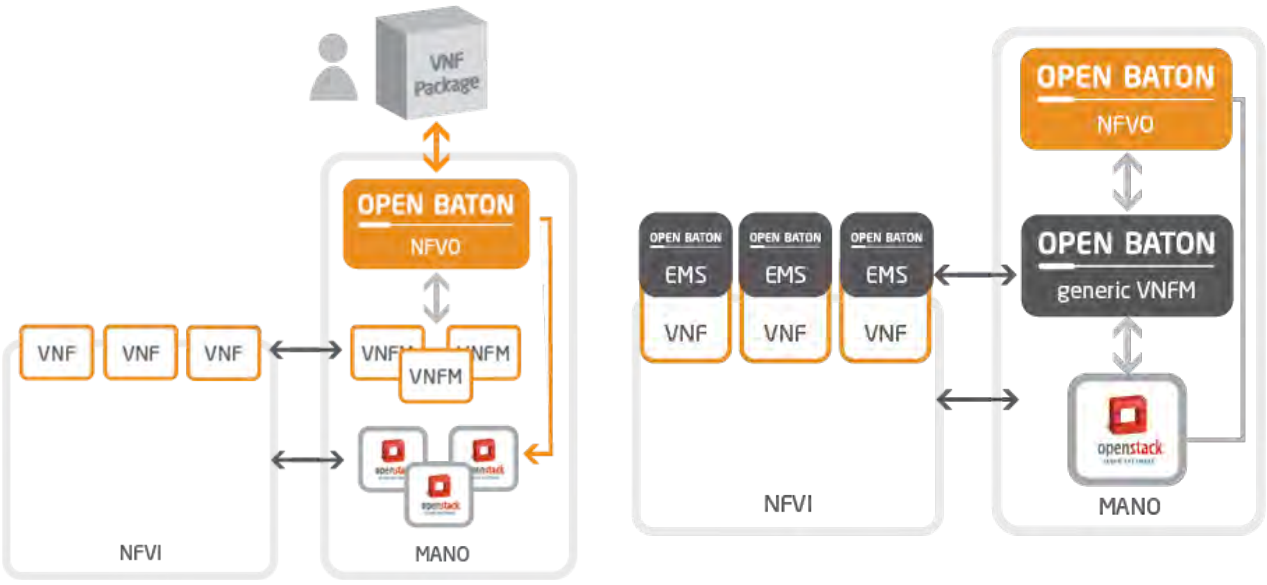

Figure 8: Open Baton high-level view.

The Figure below depicts the Open Baton architecture.

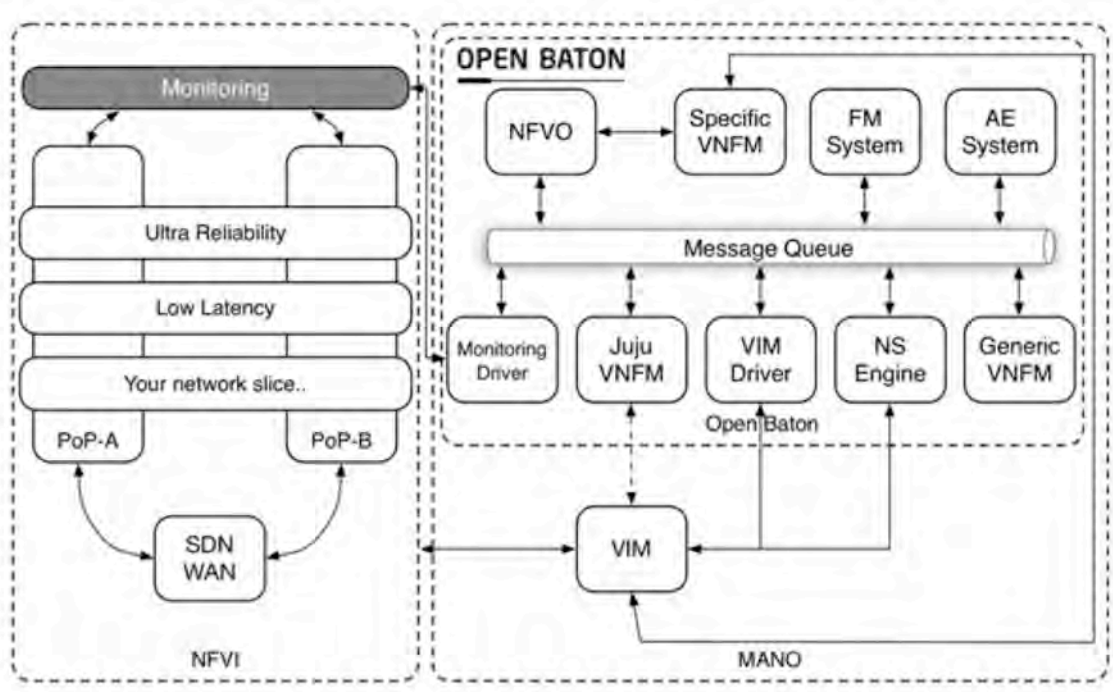

Figure 9: Open Baton architecture.

4.2.3.1. Features

The Open Baton implementation includes the following list of features:

- A Network Function Virtualization Orchestrator (NFVO) completely designed and implemented following the ETSI MANO specification.
- A generic Virtual Network Function Manager extendable (VNFM) able to manage the lifecycle of VNFs based on their descriptors. The Generic VNFM can execute the following operations:
  - Request to the NFVO the allocation of specific resources for the virtual network instance (using a granting mechanism)
  - Can request from the NFVO the instantiation, modification, starting and stopping of the virtual services (or directly to the VIM)
  - Instructs the generic Open Baton EMS to save and to execute specific configuration scripts within the virtual machine instances
- A Juju VNFM Adapter in order to deploy Juju Charms or Open Baton VNF Packages using the Juju VNFM.
- A driver mechanism for adding and removing different type of VIMs without having to re-write anything in your orchestration logic.
- A powerful event engine useful based on a pub/sub mechanism for the dispatching of lifecycle events execution.

- An auto-scaling engine which can be used for automatic runtime management of the scaling operation operations of your VNFs.
- A fault management system which can be used for automatic runtime management of faults which may occur at any level.
- It integrates with the Zabbix monitoring system.
- A set of libraries (the openbaton-libs) which could be used for building your own VNFM. A Marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFMs.
- A user-user friendly dashboard which enables the management of the complete environment.
- It provides also a set of mechanisms which enable the support of external VNFMs. This can be done in the following ways:
  - Publish/Subscribe mechanism using a message queue based on AMQP.
  - REST APIs.
- Open Baton integrates via Plugins to different VIM. By default an OpenStack plugin is provided.
- Docker-based Element Management System
  - It is possible to instantiate a VNF on top of a docker container using the GenericVNFM and VNF Package approach;
  - The compute node needs to use the nova-docker driver.
- Identity Management:
  - Possibility of defining different projects;
  - Possibility of registering users and assign them different roles;
  - Possibility of registering users and assign them to different projects.
- Network Slicing Engine (NSE)
  - The NSE instantiates rules on physical networks for allocating dedicated bandwidth as per Network Service specific requirements;
  - The NSE provides an abstracted view of the inter-datacenter networking topology allowing the instantiation of guaranteed bandwidth levels on top of the physical network elements.

4.2.3.2. Evaluation

Strongest Points:

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

Weakest Points:

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

4.2.4. OSM

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

The OSM implementation is build on top of 3 main components: RIFT.ware, a service orchestrator (NSO); OpenMANO, a resource orchestrator (RO); and Juju, a configuration manager (CM). The integration of those components is the main job of the OSM developers. The Figure below depicts the basics of this integration.

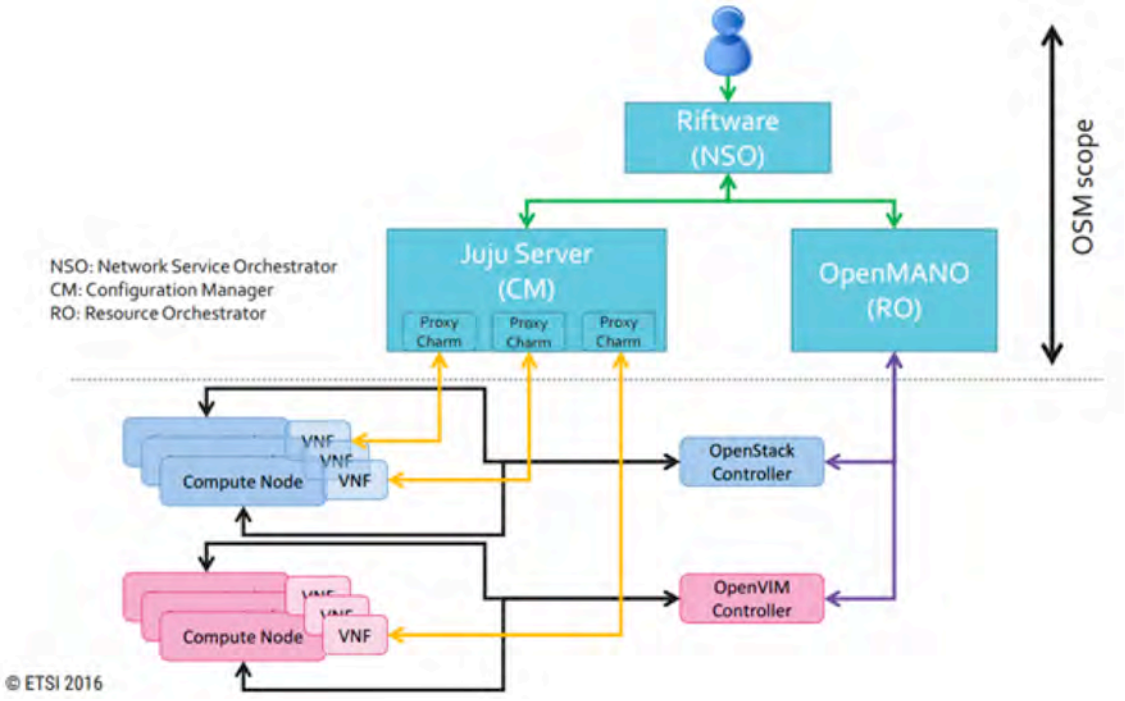

Figure 10: OSM architecture.

4.2.4.1. Features

The OSM implementation includes the following list of features:

- On-boarding experience & VNF Packaging;
  - Allow cloud-init configuration;
  - Create networks at VIM;
  - Remove NSD datacenter network reference;
  - detailed feedback when deploying and configuring & Error Message from RO & VCA;
  - Distinction between template, particularization and instance for NS;
  - Composer should display descriptors in YAML format;
  - Enhance Visual Differentiation Between NS Catalog and VNF Catalog;
  - Restructure layout of service primitive page;
  - Package creation command line utility;
- EPA based resource allocation;
  - Not explicitly captured;
  - May be included as part of an upgrade to OpenStack Mitaka, likely push to R2 timeframe;
- (Networking) Service Modelling;
  - Juju-2.x;
  - Network types in RO;
  - Allow IP parameters for networks;
  - Configuration/Service Primitive model enhancements;
- Multi-VIM;
  - New VIM connector for VMware vCloud Director;
  - Openvim as reference VIM with EPA capabilities;
  - datacenter capabilities;
  - Support for VIM Accounts;
- Multi-Site;
  - Multi-site NS

4.2.4.2. Evaluation

Strongest Points:

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

Weakest Points:

- The documentation is not enough, missing complex examples;
- It not work with the more recent release of OpenStack, it follow the OpenStack releases with delay of one or two (now Mitaka);
- Limited set of functionalities (even less than RiftWare alone, one of the pieces);
- Some limitations identified;
- Evolutions and corrections come slowly sometimes;
- Not very stable and reliable yet.

4.2.5. Cloudify

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

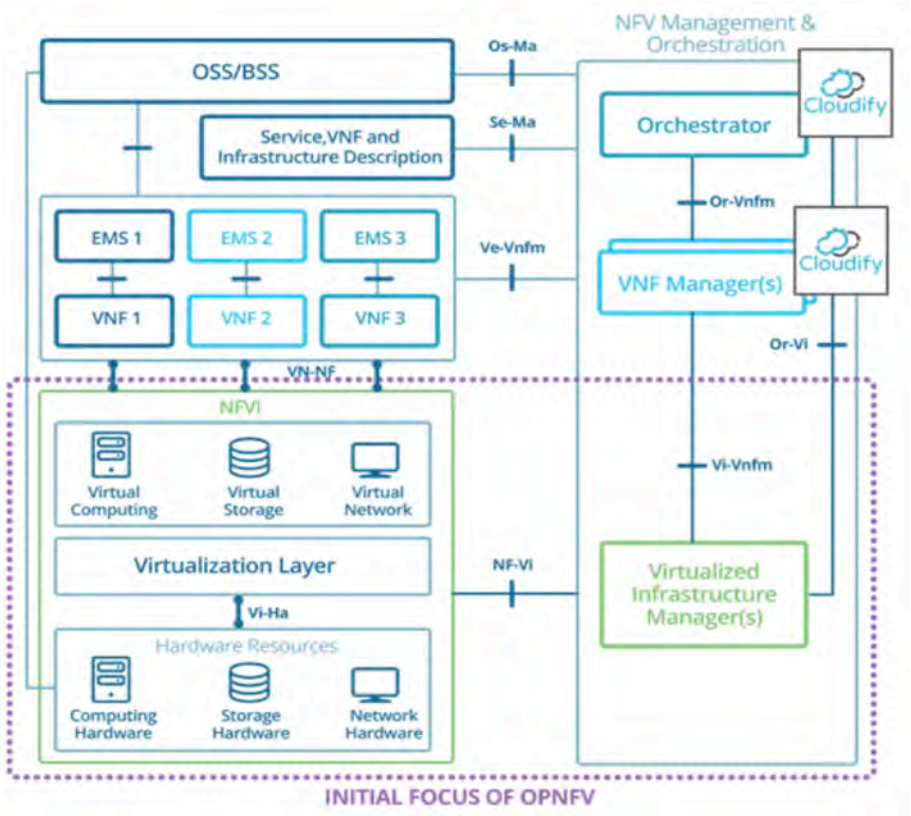

Figure 11: Cloudify architecture

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

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

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

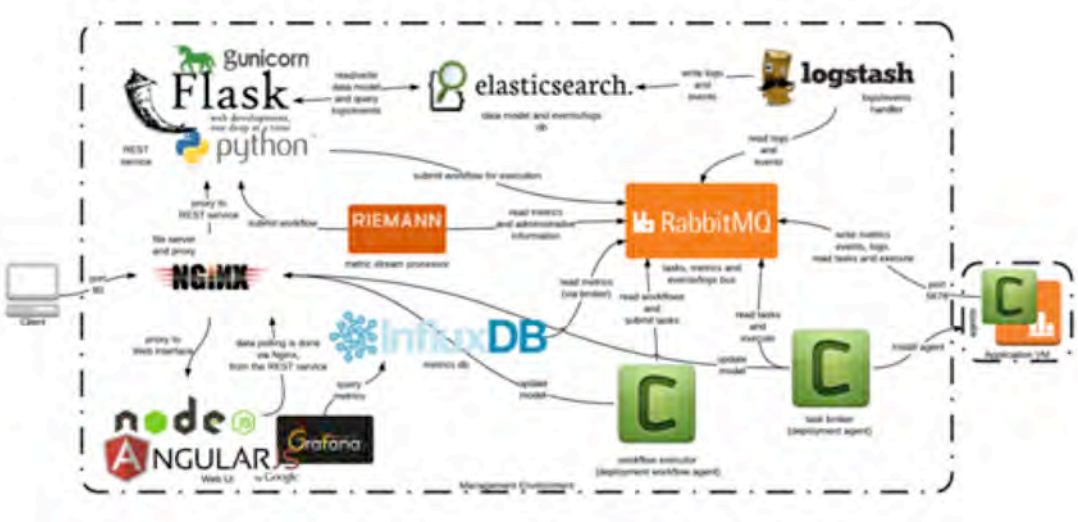

Figure 12: Cloudify components integration.

4.2.5.1. Features

The Cloudify implementation includes the following list of features:

- Full Application Lifecycle Orchestration with Cloudify:
- Pure-play orchestration of applications on the cloud:
- Infrastructure & Tooling Agnostic:
- NFV Orchestration: The only pure-play technology agnostic MANO implementation:
- Hybrid Cloud & Workloads with Cloudify:
- Cloudify Simplifies Cloud Migration & Portability for Enterprises:
- Blueprint Composition:
- Cloudify Telecom Edition: The Telecom Edition is an NFV-specific offering of Cloudify's TOSCA-based orchestration framework. This is the most advanced open source NFV MANO offering and includes built-in multi-VIM support with added extensibility to hybrid workloads and micro services. Included in this release:

- Multi-VIM - Now fully open source vCloud and vSphere plugins, alongside OpenStack native support;
- Overlay Service Chaining;
- Netconf plugin;
- TOSCA/YANG data modelling interoperability;
- Network service management;
- Clearwater vIMS blueprints and plugins;
- F5 BigIP plugin;
- VNF updates for running VNFs and services;
- NIC ordering;
- Enhanced Platform Awareness coupled with Data Plane Acceleration through integration with Intel;
- Drag and drop Cloudify Composer with VNF-specific components;
- Using ARIA as the kernel for TOSCA Orchestration;
- Support installation within environments with no internet access;
- Support for Cloud Native services through Kubernetes plugin.

4.2.5.2. Evaluation

Strongest Points:

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

Weakest Points:

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

4.2.6. Tacker

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

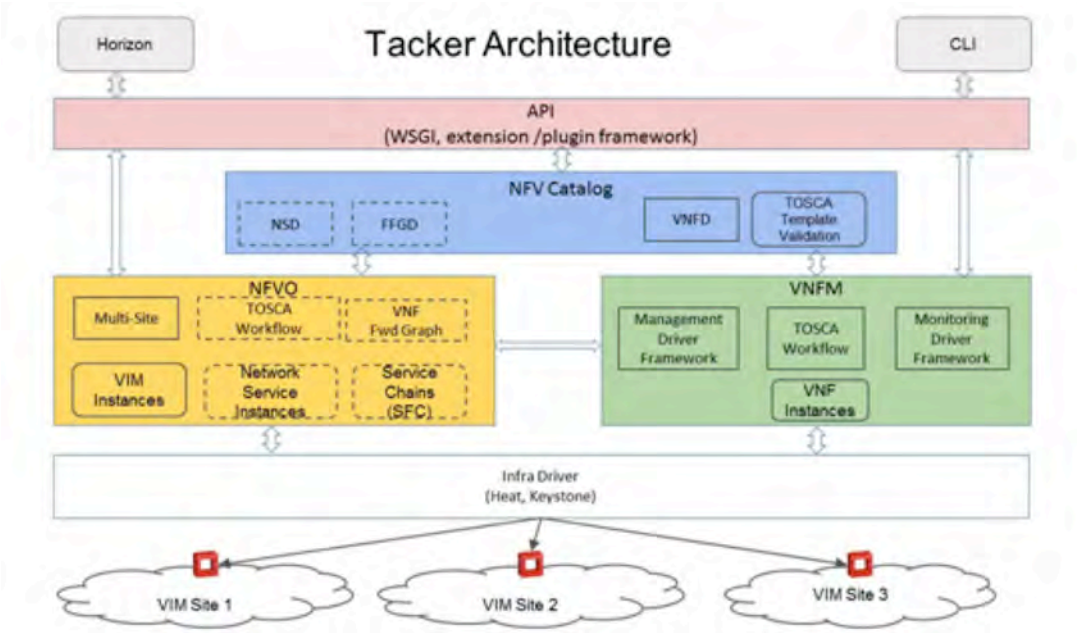

Figure 13: Tacker architecture.

4.2.6.1. Features

The Tacker implementation includes the following list of features:

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

- VNF Auto Configuration
  - Tacker provides a Management Driver Framework
  - Facilitates VNF configuration based on Service selection
  - Inject initial configuration using:
    - Config-drive
    - custom mgmt-driver (connect using ssh / RESTapi and apply configuration)
  - Update configuration in active state
  - Extendable
- VNF Self-Healing
  - Tacker health check starts as VNF becomes ready
  - Ongoing network connectivity check
  - Auto-restart on failure (based on VNFD policy)
  - Extendable Vendor and Service specific Health Monitoring Driver framework
- VNF Auto-Scaling
  - Auto-Scale VNF based on policy
  - Continuous performance monitoring according to KPI described in VNFD
  - Basic Auto-Scaling using common VM metric
    - CPU threshold
    - Custom Monitoring Metric
  - Alarm-based monitoring driver using Ceilometer
  - Manual-scaling option to scale in/out the VDUs
- Multisite VIM Usage
  - Manage multiple OpenStack sites
  - Deploy VNFs in multiple OpenStack sites
- VNF Forwarding Graph
  - TOSCA NFV Profile based FG Descriptor can be uploaded to VNF-FGD Catalog
  - VNF-FFGD template describes both Classifier and Forwarding Path across a collection of Connection Points described in VNFDs
- Recently NS support was introduced
  - NSD Catalogue
  - NS instantiation

4.2.6.2. Evaluation

Strongest Points:

- Active development in Openstack environment and will be upgrade in every OpenStack version;
- Roadmap well defined and large community involved;
- Aligned with OpenStack releases;
- Good documentation;
- Aligned with NFV;

- Uses TOSCA;
- Integration with other tools (ManageIQ);
- Auto-healing and auto-scaling capabilities;

Weakest Points:

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

4.2.7. ManageIQ

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

4.2.7.1. Features

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

4.2.7.2. Evaluation

Strongest Points:

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

Weakest Points:

- Not aligned with NFV;

- It doesn't make direct use of TOSCA descriptors;

4.3. Comparison between Orchestrators

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

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

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

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

5. Superfluidity Provision and Control Framework

Some major international operators and vendors started the ETSI ISG (Industry Specification Group) on Mobile Edge Computing, which advocates for the deployment of virtualized network services at remote access networks, which are placed next to base stations and aggregation points, and run on x86 commodity servers. In other words, its task is to enable services running at the edge of the network, so that services can benefit from higher bandwidth and low latency.

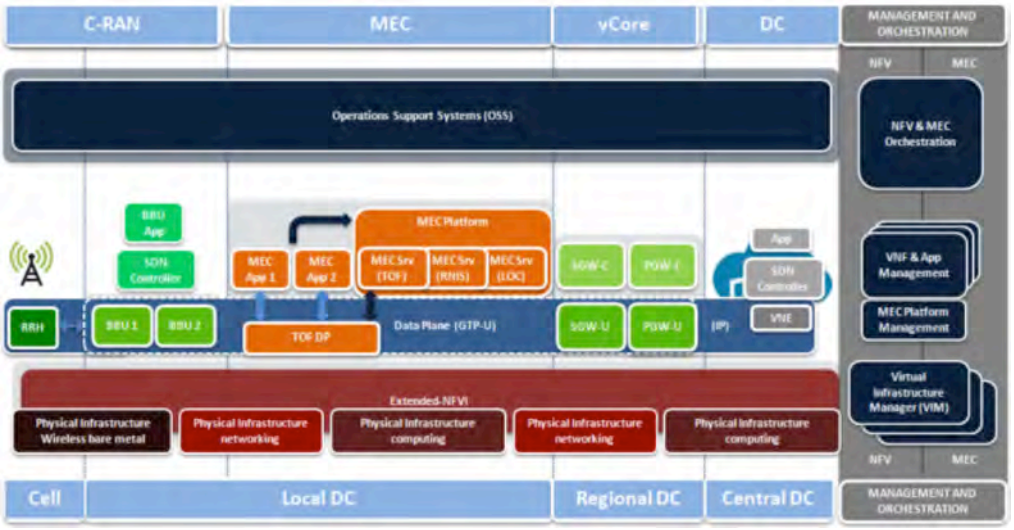

Figure 14: 5G Network Components Overview

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

Based on the current status of available upstream components (OpenStack, Kubernetes, OSM, ManageIQ, ODL, …), the requirement analysis, and the management and orchestration study presented above, we think VMs may not always be a proper approach for all the needs. Instead, other solutions such as the unikernel VMs and containers should be used. So, we forecast a mixed container and VM scenario, at least for the following years. Note, even though there is a high interest in moving more and more functionality to containers over the next years, the priority so far is still set on new applications rather than the legacy ones. And not all the applications will/can

be migrated at the same time. On top of that, there is a belief that containers and virtualization are essentially the same thing, while they are not. Although they have a lot in common, they have some differences, too. They should be seen as complementary, rather than competitive technologies. For example, VMs can be a perfect environment for running containerized workload (it is already fairly common to run Kubernetes or OpenShift on top of OpenStack VMs), providing a more secure environment for running containers, as well as higher flexibility and even improved fault tolerance, and also taking advantage of accelerated application deployment and management through containers. This is commonly referred to as "nested" containers.

Consequently, with this blend of VMs and Containers, the different components (and their actuation level) of the proposed orchestration framework are depicted in the next figure.

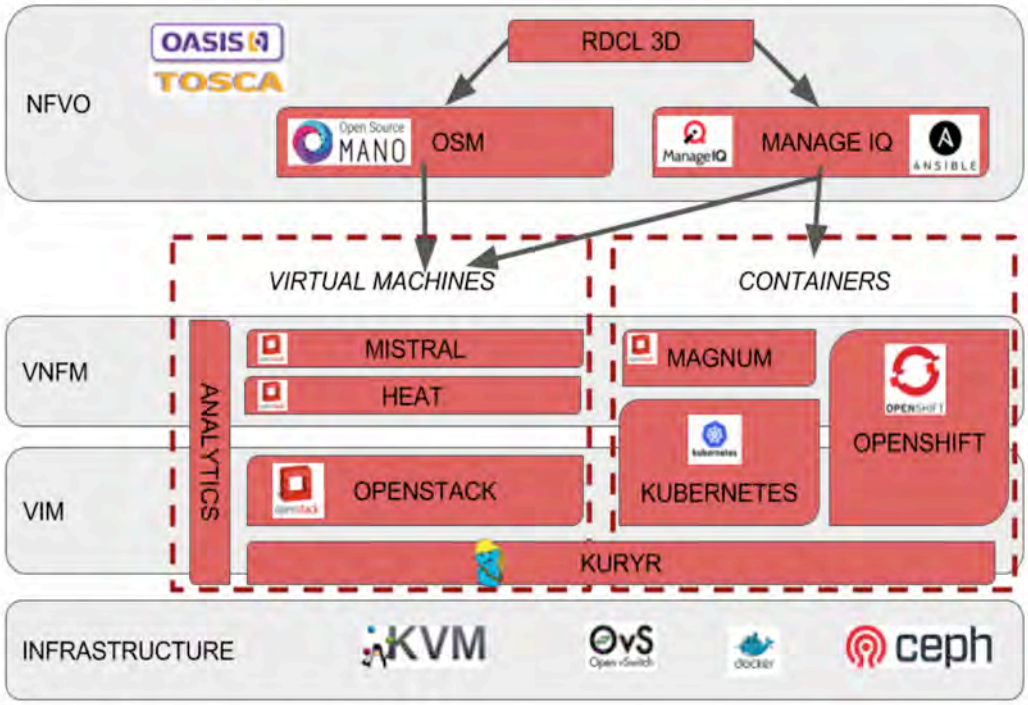

Figure 15: Superfluidity NFVO Components

5.1. VIM Option: OpenStack, Kubernetes and Kuryr

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

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

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

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

5.2. VNFMs

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

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

On the other hand, OpenShift already has other extra container management functionalities on top of Kubernetes that can be used through its API. Moreover, as OpenShift leverages Kubernetes functionality, the management capabilities available in Kubernetes/Magnum can be used on OpenShift deployments too.

5.3. NFVO

Finally, in the upper level, we may made use of different NFV orchestrators. There is no complete solution yet and, as detailed before in this document, different options have been studied. Among them, we decided to make Superfluidity project to target the 2 most promising ones. On the one hand, OSM seems to have some momentum and has a large support by the NFV community. On the other hand, we decided to explore/extend ManageIQ as it is the only one capable of dealing with different VM and container providers, which, as mentioned before, is a need for the 5G deployments. In addition, ManageIQ allows to easily handle in a single point multiple deployments, as it is the target of Superfluidity, where we can have different cloud deployments at the edges, and the core. Moreover, it provides enough flexibility to include new orchestration actions by relying on Ansible playbooks to trigger/execute new required actions. Finally, there is also another component, named RDCL 3D. This component provides an UI to create the RFB and translate them from TOSCA templates to Heat templates for the VMs and Ansible playbooks for the containers that can be later process by the NFVOs and pushed to the lower layers in their respective template formats.

5.4. Deployment

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

provided by Magnum OpenStack project which allows to install Kubernetes on top of the OpenStack VMs, as well as some extra orchestration actions over the containers deployed through the virtualized infrastructure. A multi-side deployment example fitting the Superfluidity use cases is presented in the next figure.

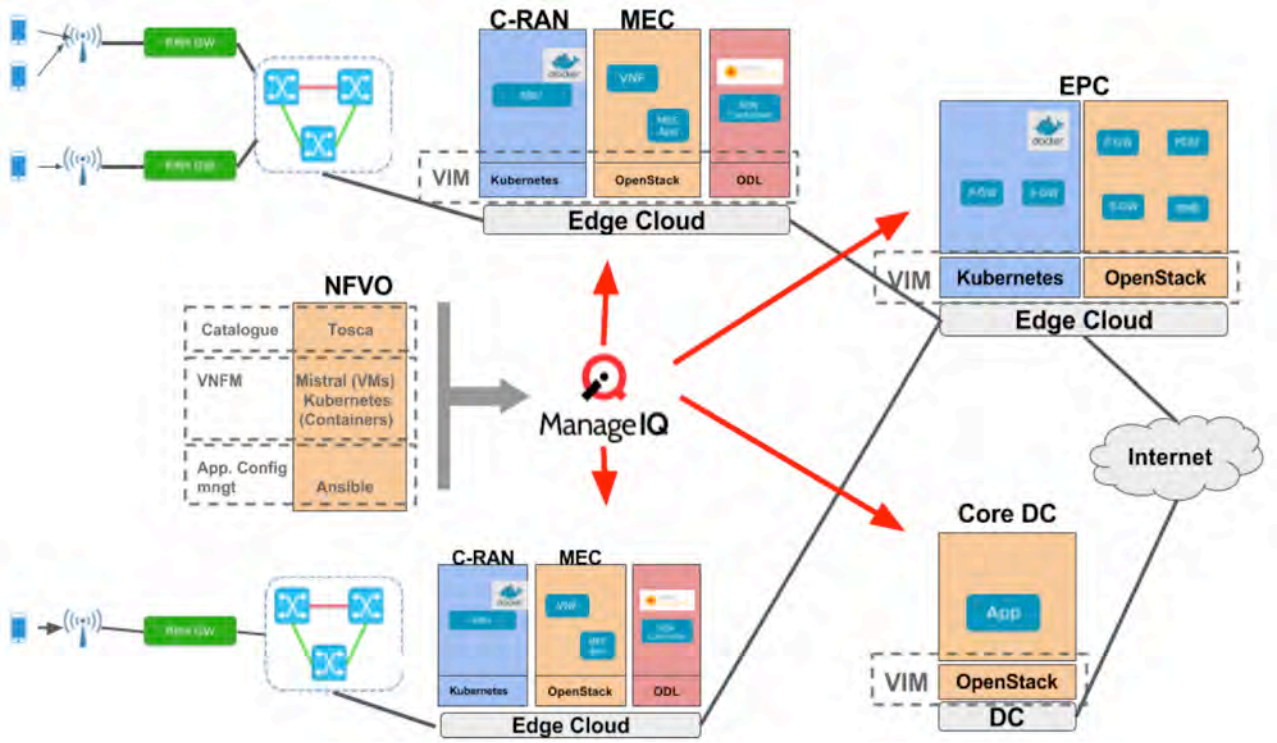

Figure 16: Multi Side Superfluidity Deployment Overview

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

There is a VIM in all of them, but it could be different in different clouds. For instance, it is more common to have OpenStack as a VIM in big data centers, as the virtualization overhead is not a big concern at that scale. By contrast, at the edge side, the amount of resources is more limited, but the responsiveness needs to be higher. Therefore, containers are needed, but so VMs are. In such

case the VIM is composed of both OpenStack and Kubernetes/OpenShift, and even SDN controllers (which may also run inside VMs, containers, or baremetal).

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

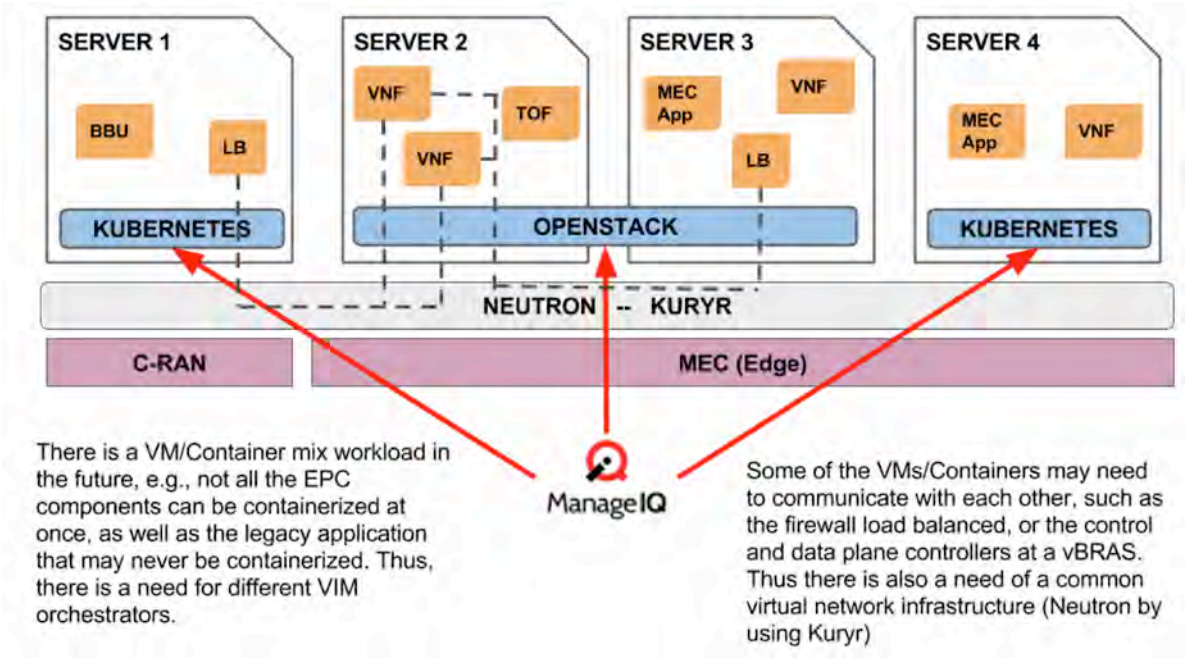

Figure 17: MEC deployment options overview and motivation

6. Superfluidity Contributions

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

6.1. Kuryr

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

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

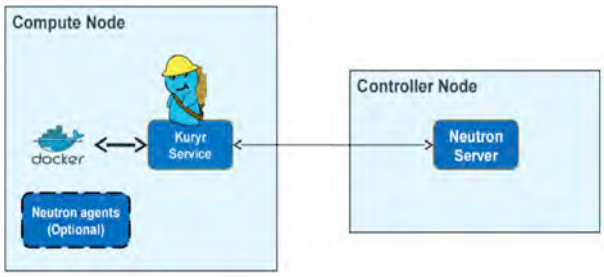

Figure 19: Kuryr Service Overview

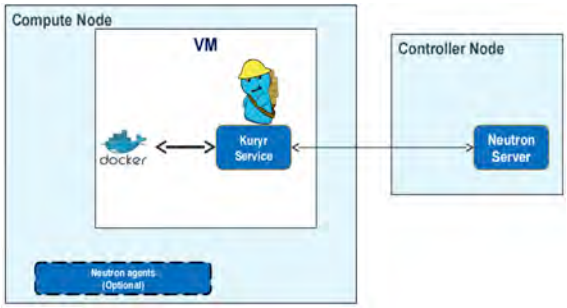

Figure 18: Nested Kuryr Service Overview

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

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

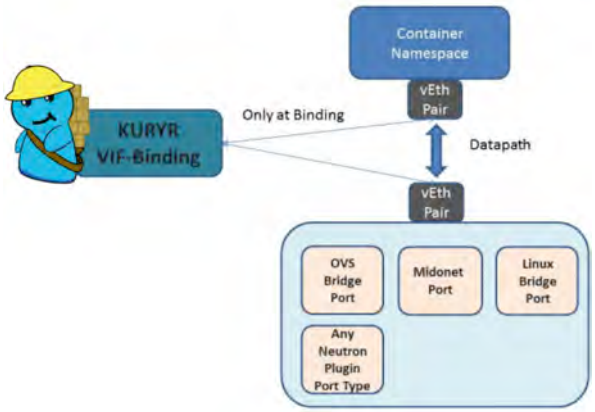

Figure 20: Kuryr Binding Options

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

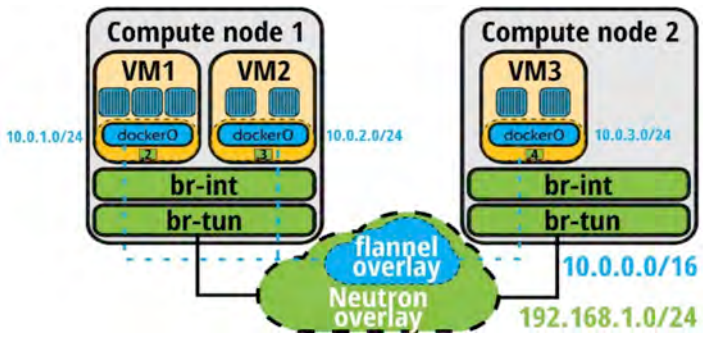

Figure 21: Double encapsulation problem

6.1.1. Superfluidity Contributed Features

Side by side OpenStack and OpenShift/Kubernetes deployment

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

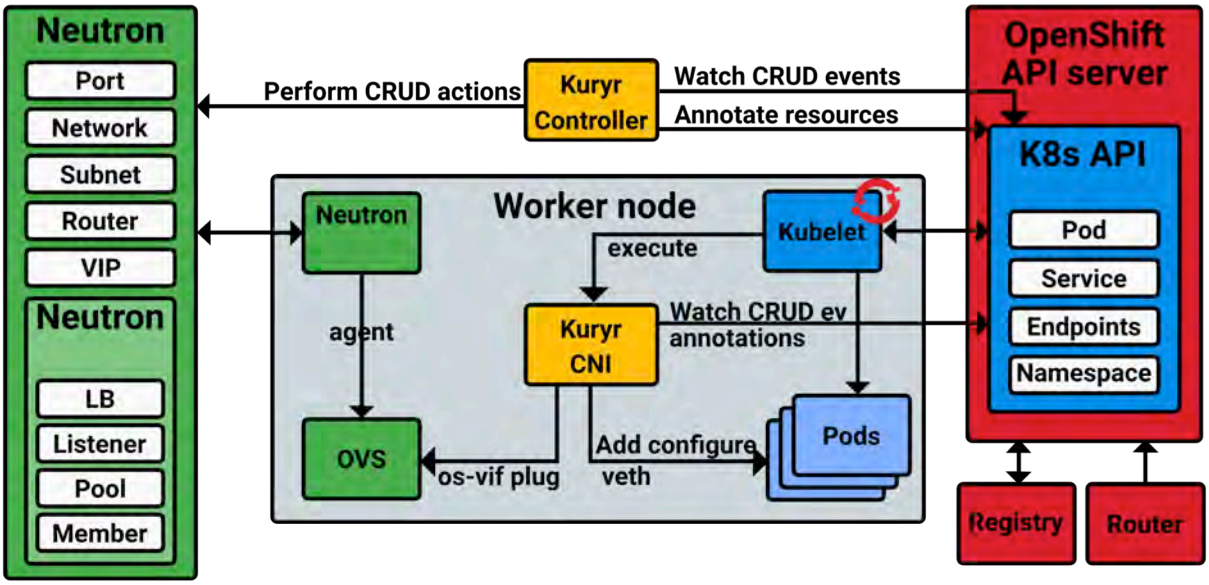

Figure 22: Kuryr for OpenShift/Kubernetes Clusters

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

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

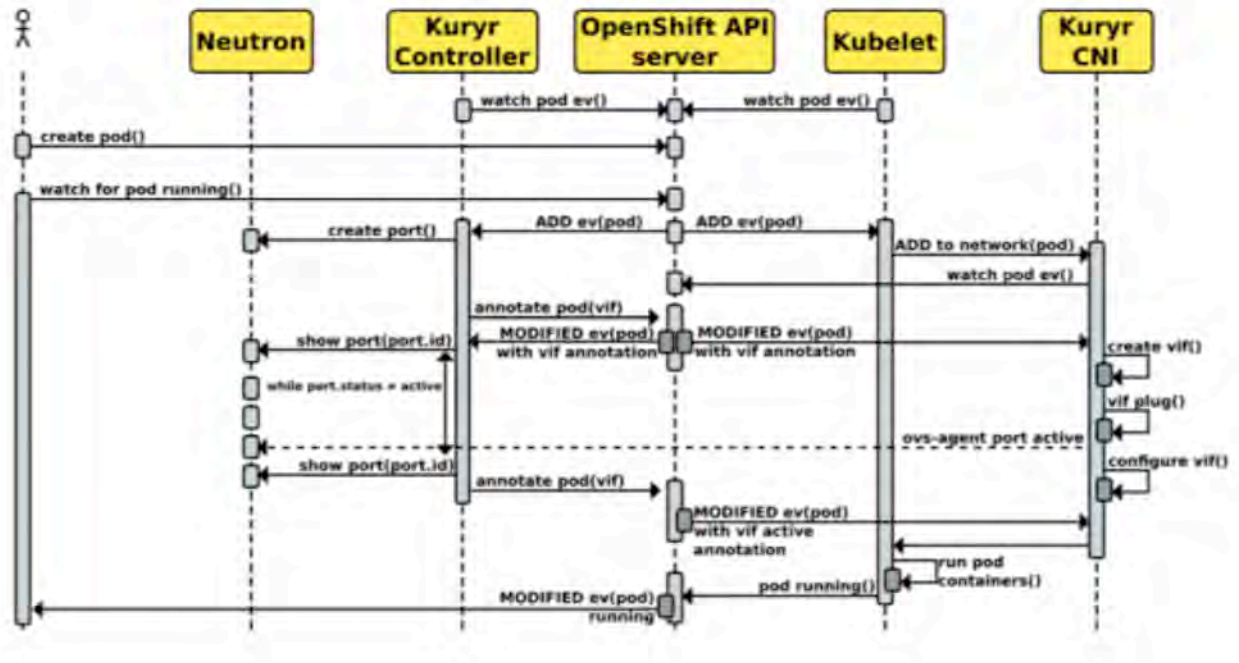

Figure 23: Sequence Diagram: Booting container with Kuryr

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

Nested deployment: OpenShift/Kubernetes on top of OpenStack

There were still some gaps regarding nested containers that we are addressing at the Superfluidity project. Note these VMs are not managed directly by Nova and hence do not have an OpenStack

agent inside them. This means we need a mechanism to perform the VIF binding inside the VM and it needs to be initiated by the local Docker remote driver or Kuryr-Kubernetes CNI for the kubernetes case.

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

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

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

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

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

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

Furthermore, the modifications at the Kuryr-CNI (and kuryr-libnetwork for the docker case) are targeting the new way to bind the containers to the network, as in this case, instead of being

added to the OVS (br-int) bridge, they are connected to the VM 's vNIC in the specific vlan provided by the Kuryr-Controller (subport).

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

Ports Pool Optimization

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

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

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

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

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

- Available pools: There will be a pool of ports for each tenant, host (or trunk port for the nested case), and security group, ready to be used by the new pods being created. Note at the beginning there is not pools, and once a pod is created at a given host/VM by a tenant, with a specific security group, a corresponding pool gets created, and populated with the desired minimum amount of ports.

- Recyclable pool: Instead of deleting the port during pods removal, the port will be included into this pool. The ports in this pool will be later recycled by this driver and put them back into the corresponding available pool, after reapplying security groups to avoid any security breach.

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

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

Similarly, the pool driver ensures that ports are regularly recycled after pods are deleted and put back in the corresponding *available_pool* pool to be reused. Therefore, Neutron calls are skipped as there is no need to delete and create another port for a future pod. The port cleanup actions return ports to the corresponding available_pool after re-applying security groups and changing the device name to 'available-port'. A maximum limit for the pool can be specified to ensure that once the corresponding available_pool reach a certain size, the ports above this number get deleted instead of recycled. This upper limit can be disabled by setting it to 0. In addition, a Time-To-Live (TTL) could be set to the ports at the pool, so that if they are not used during a certain period of time, they are removed -- if and only if the disposable pool size is still larger than the target minimum.

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

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

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

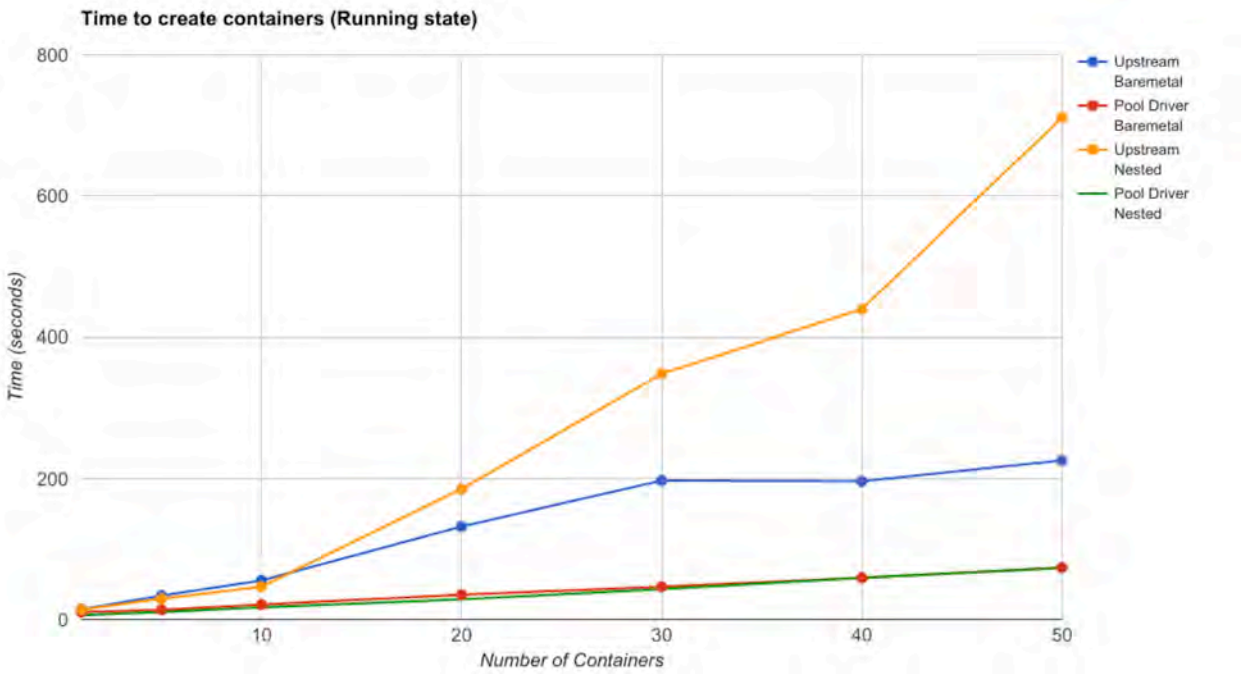

Figure 24: Booting time comparisson

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

More instructions about how to enable it and use it are available at:
https://ltomasbo.wordpress.com/2017/05/09/kuryr-ports-pool-speeding-up-containers-booting-time-on-neutron-networks/

Load Balancer as a Service (LBaaS) Integration

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

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

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

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

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

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

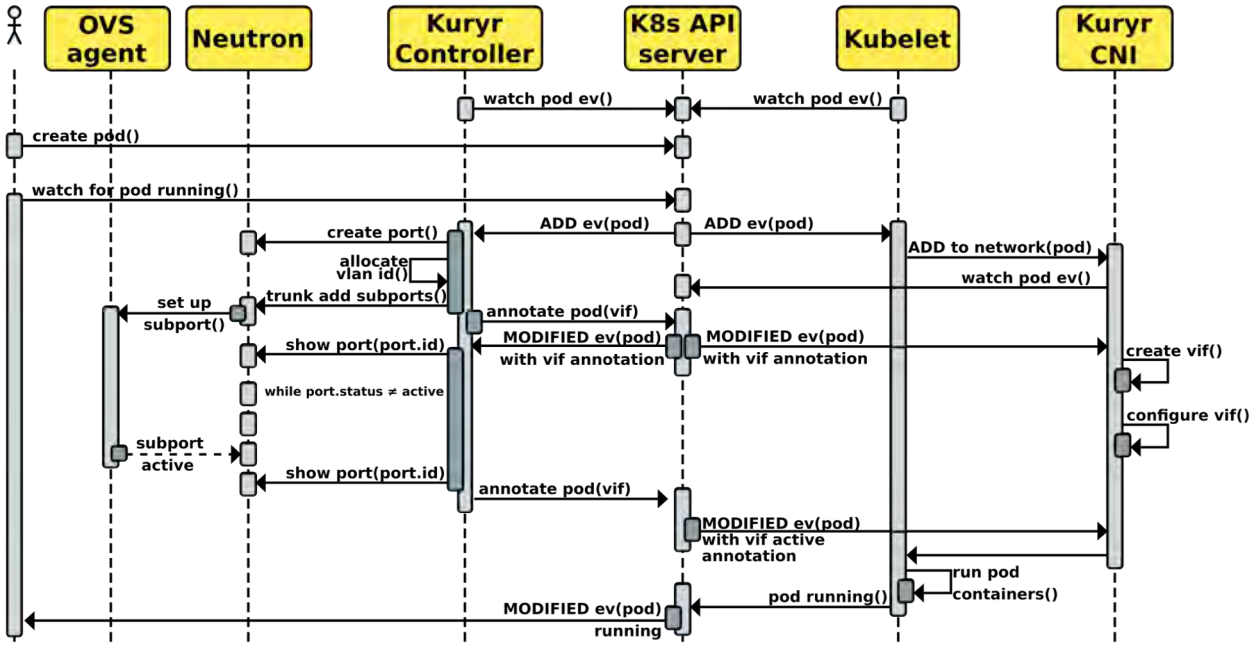

Figure 25: Sequence Diagram: Booting a nested contain in a VM with Kuryr

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

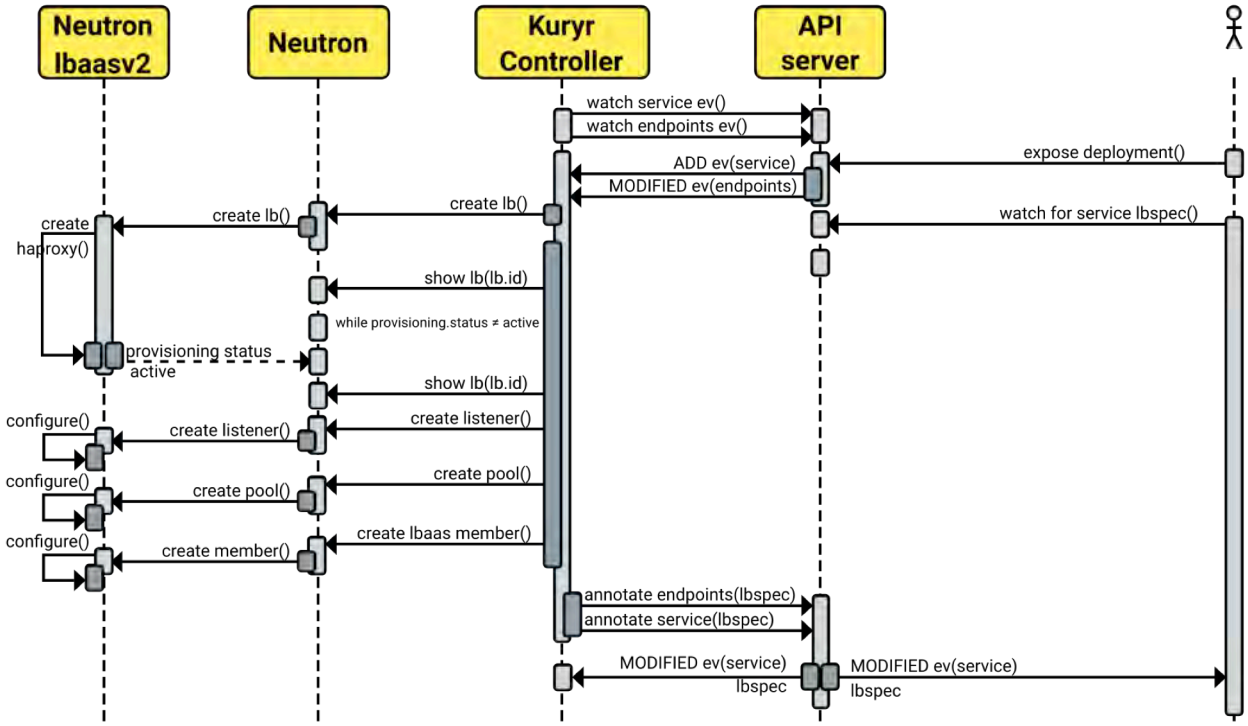

Figure 26: Sequence Diagram: Neutron Load Balancer Creation from Kuryr

6.1.2. Usage

The steps required to create and configure an all-in-one devstack deployment that includes Kubernetes, OpenStack and Kuryr installation, including the nested deployment too (Kuryr and Kubernetes running in a VM inside it) are presented in the next blog posts:

https://ltomasbo.wordpress.com/2017/01/29/side-by-side-and-nested-kubernetes-and-openstack-deployment-with-kuryr/

Note, the all-in-one DevStack deployment is called ***undercloud*** deployment. And the installation of Kubernetes and Kuryr inside a VM in the undercloud deployment, to enable the nested container use cases, is named ***overcloud***.

6.2. Mistral Orchestration

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

Using TOSCA as the main DSL for the Network Service Descriptor provides a hierarchical view of services, components and their relationships, which is decoupled from the underlying VIM/NFVI.

When combining TOSCA interfaces and a Mistral workflow, a full network service lifecycle can be achieved.

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

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

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

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

Mistral, originally, was configured to execute workflows on a specific cloud VIM. In order to execute workflows on a different VIM, one needed to start a new Mistral instance. To allow a more fluent support for the distributed architecture of Superfluidity, we extended Mistral's workflow execution parameters to make it possible to target a specific cloud without modifying the configuration of the Mistral service.

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

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

6.3. OSM

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

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

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

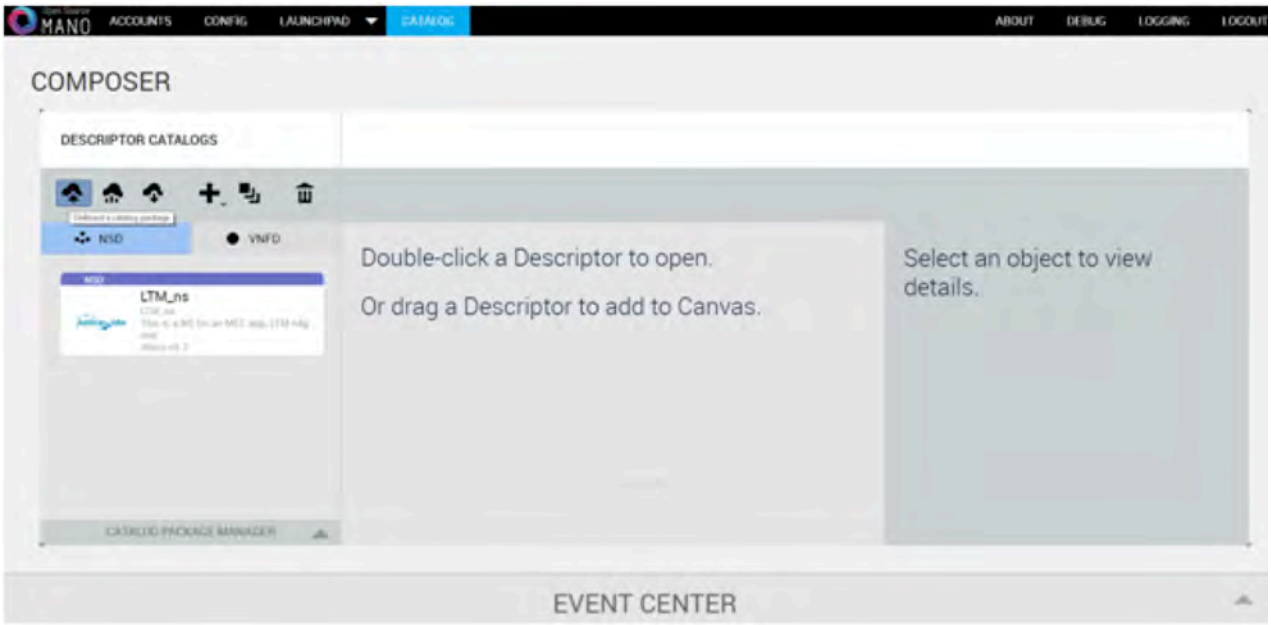

Figure 27: Onboarding of ME Apps on OSM.

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

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

The OSM OpenMANO (RO) component is responsible to interact with the VIM and can be mapped directly to MEC environments. As OSM is multi-VIM, it allows us to select the VIM where the ME

App is deployed. In the instantiate process the user can choose one of the available VIMs, previously configured. In MEC environments, assuming one VIM per edge, this permits a single orchestration tool to manage the multiple edges.

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

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

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

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

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

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

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

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

6.4. ManageIQ

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

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

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

In our ManageIQ catalog item, we have defined a dialog that takes, on the one hand, the ansible playbook to execute (usually a site.yml file), and on the other hand the provider where you want

to execute it -- being the only requirement to have ansible installed at the ManageIQ appliance and ssh-passwordless towards the providers master nodes.

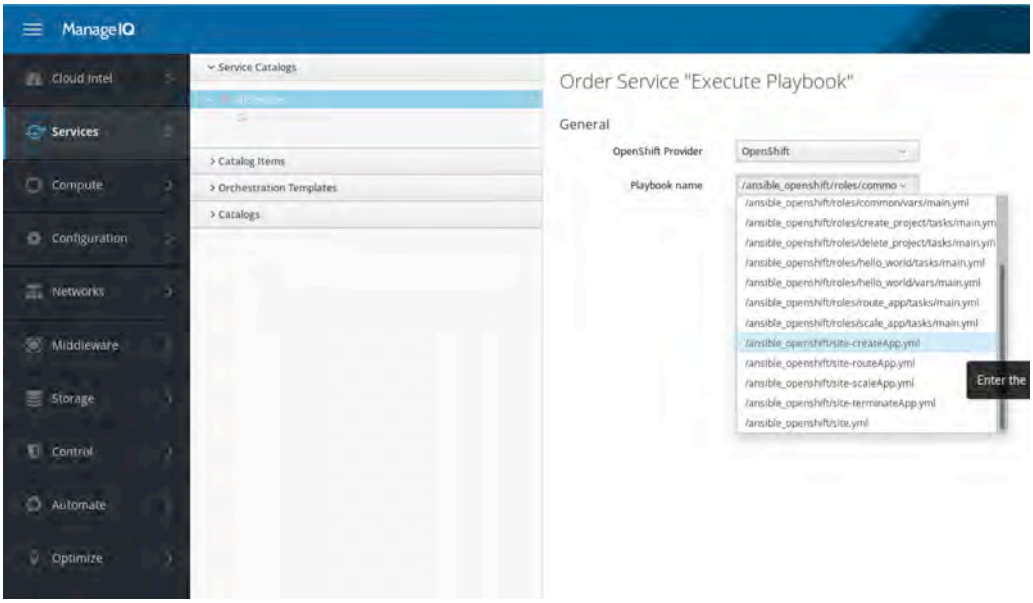

Figure 28: Playbook execution support at ManageIQ

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

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

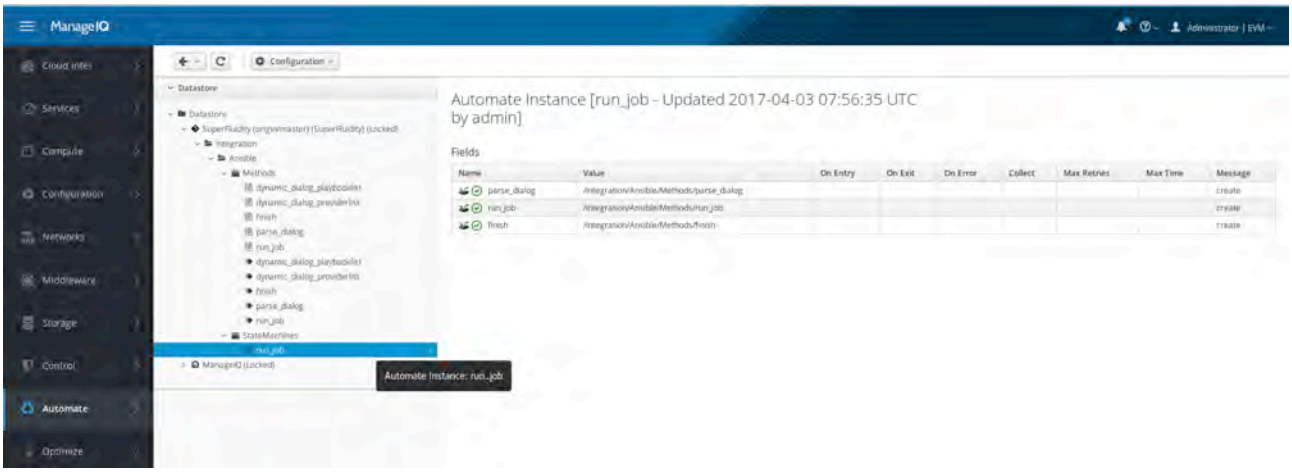

Figure 29: Playbook Execution State Machine

6.5. Load Balancing as a Service

As identified in previous deliverables (Deliverable D2.1), many of the use cases of interest depend on common load balancing capabilities, to fulfil the scalability and high-availability requirements. The requirements of Load Balancer, as a functional block, are analyzed further in Deliverable D2.2. Additionally, the requirements, as well as potential gaps (e.g. “Firewall” Load Balancing), from a MANO framework standpoint, are summarized in the table of section 3.1.1.

OpenStack offers a few open source options for implementing the infrastructure load balancing capability, most notably HAProxy (upstream open source project) and Octavia (OpenStack project). Deliverable I5.3 provides more information on these options.

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

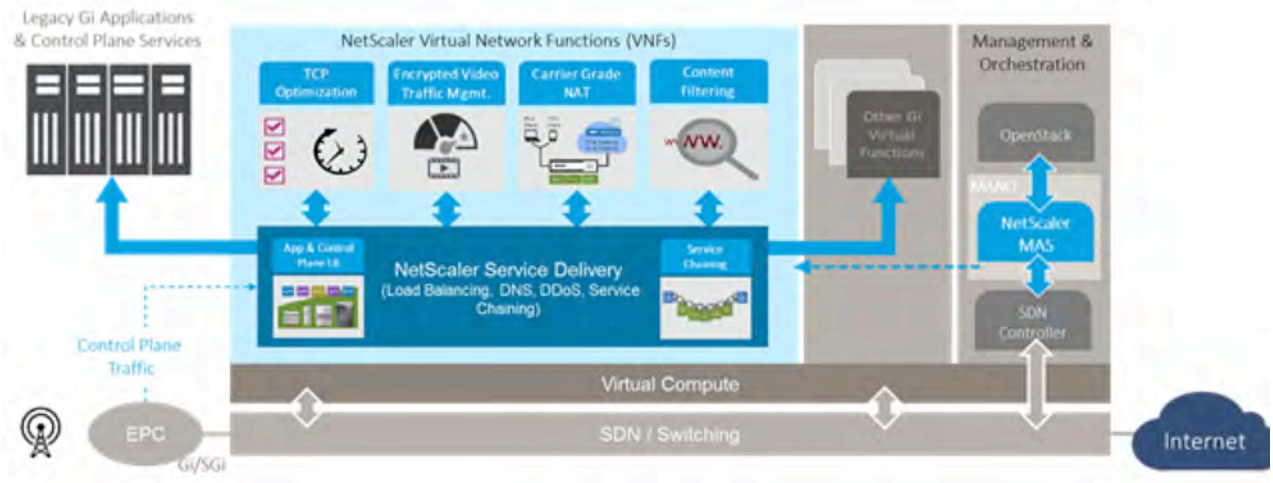

Figure 30: NetScaler at NFV Architecture

NetScaler MAS (https://www.citrix.com/products/netscaler-management-and-analytics-system/) acts as an Element Manager for NetScaler ADC. It integrates using standard APIs with OpenStack and supported SDN Controllers, translating them to the RESTful APIs supported by NetScaler ADC (https://docs.citrix.com/en-us/netscaler/12/nitro-api.html). More information on the integration between NetScaler MAS and OpenStack can be found at: http://docs.citrix.com/en-us/netscaler-mas/12/integrating-netscaler-mas-with-openstack-platform.html

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

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

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

- Load Balancing Algorithms: https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-customizing-algorithms.html
- Persistence Types: https://docs.citrix.com/en-us/netscaler/12/load-balancing/load-balancing-persistence/persistence.html

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

As part of this activity, we have leveraged enhancements in both the NetScaler LBaaS driver (OpenStack Neutron/LBaaS project: https://wiki.openstack.org/wiki/Neutron/LBaaS/NetScaler,

https://github.com/openstack/neutron-lbaas/tree/master/neutron_lbaas/drivers/netscaler), as well as NetScaler MAS and ADC (commercial software). The VPX (virtual) edition of NetScaler ADC was used for that purpose, specifically the 11.1 release, and the respective NetScaler MAS release. The above outcomes will be integrated with the Superfluidity platform as part of WP7 activities.

6.6. Service Function Chaining

As identified in the requirements analysis (Deliverable D2.1), many complex use cases consist of compositions of in-network services, which require Service Function Chaining (SFC) to materialize. The components of the IETF SFC architecture, namely the Service Functions (SF), SF Forwarder, Network Overlay, SFC Proxy and SFC Classifier are analyzed further in Deliverable D2.2. In addition, the MANO requirements, as well as the status (as of June 2016) of SFC support in OpenStack, OVS, OpenDaylight and OPNFV are summarized in the table of section 3.1.1.

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

- OpenStack SFC support (https://docs.openstack.org/ocata/networking-guide/config-sfc.html) is maturing, adding new capabilities (https://docs.openstack.org/developer/networking-sfc/): New SFC Driver for OVN (http://openvswitch.org/support/dist-docs/ovn-architecture.7.html), support for Symmetric Port Chains, Service Function Tap for Port Chains, etc.
- The OPNFV OVS project introduced "NSH for VXLAN" support in the OPNFV Colorado release. However, this required a special release of OVS (which included Yi Yang's OVS NSH patches: https://github.com/yyang13/ovs_nsh_patches). Actually, this is still the case today, it seems.
- The OPNFV Colorado and Danube versions support the SFC os-odl_l2-sfc-noha, os-odl_l2-sfc-ha scenarios, i.e. OpenStack + OVS + OpenDaylight + SFC, at least for the Apex and Fuel installers. The version of OpenDaylight OPNFV Danube is dependent on is Boron SR2 (SR3 is the latest). The latest documentation of the OPNFV SFC submodule: http://docs.opnfv.org/en/stable-danube/submodules/sfc/docs/development/design/index.html
- As foreseen in section 3.1.1, the VNFM has to be involved as well. The scenarios implemented by OPNFV SFC utilize OpenStack Tacker for that purpose (see http://docs.opnfv.org/en/stable-danube/submodules/sfc/docs/development/design/architecture.html#vnf-manager). More information on how this is implemented: https://specs.openstack.org/openstack/tacker-specs/specs/newton/tacker-networking-sfc.html
- Lastly, when using NSH with VXLAN tunnels, it is important that the VXLAN tunnel is terminated in the SF VM. This allows the SF to see the NSH header, allowing it to decrement the NSI and also to use NSH metadata. When using VXLAN with OpenStack, the

tunnels are not terminated in the SF VM, but in the OVS bridge. Workaround: http://docs.opnfv.org/en/stable-danube/submodules/sfc/docs/development/design/architecture.html#ovs-nsh-patch-workaround.

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

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

6.7. RDCL 3D

The vision of the Superfluidity project is to orchestrate functions dynamically over and across heterogeneous environments. It is not possible to consider a single language and tool to cover the diversity of the heterogeneous environments. In the Superfluidity architecture, the different languages for the description, composition and orchestration of services and service components are referred to as RDCL (RFB Description and Composition Languages). The project has designed and developed a tool called RDCL 3D (Design, Deploy and Direct) to assist in the editing of the descriptors of services / service components and in the interaction with different types of orchestration tools. The RDCL 3D tool is not focused on a specific data model / description language but it is meant as a framework that can be specialized for any data model / description language. The RDCL3D tool has already been described in Deliverable I5.3; we here provide an updated description.

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 31 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 [1] is

released under the Apache 2.0 Open Source license, to facilitate its uptake by research and industrial communities [2]. 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.

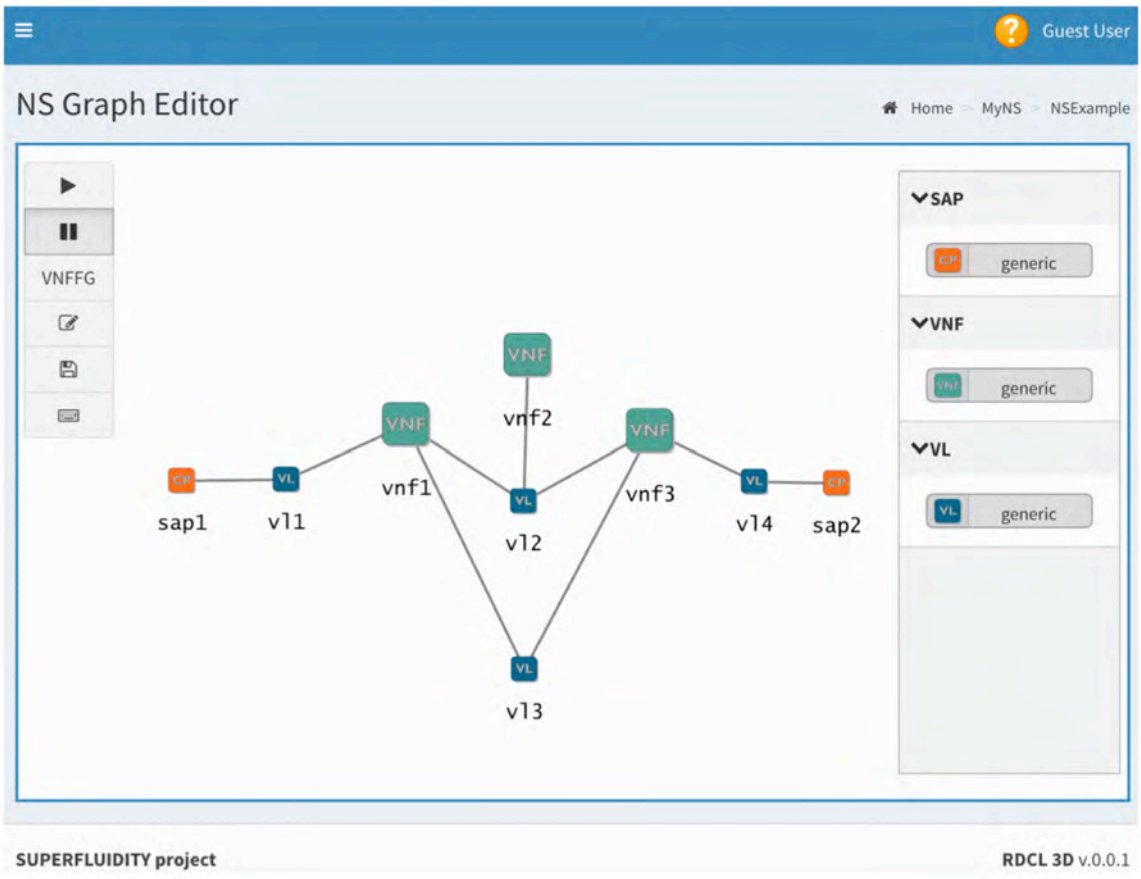

Figure 31: Network Service design using the RDCL 3D GUI

Currently the RDCL framework supports the models defined by the latest ETSI NFV ISG specifications [3,4], the TOSCA Simple Profile for NFV [5], the TOSCA Simple Profile in YAML [6] and the network elements described in the Click configuration language [7]. However, RDCL 3D is designed for extensibility in order to support new models or combine existing ones.

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

1. RDCL 3D can be used as a standalone tool to edit the NS and VNF descriptors, as shown in <1> in Figure 32. This approach is currently implemented in the demo for the ETSI and

TOSCA models. The produced descriptor files can be manually retrieved and provided to an Orchestrator (NFVO).

2. RDCL 3D can support the direct interaction with programmatic APIs of external Orchestrators (<2> in Figure 32) by including the logic for such interaction in an Agent module. In this case, the Agent module receives instructions from the RDCL 3D web GUI, hands the descriptor files to the external Orchestrator, and provides feedback to the user on the GUI. Note that this scenario can be extended when there is the need to combine different orchestration platforms with different description languages, so that RDCL 3D can become a *meta-orchestrator*.
3. Another usage scenario that we have considered is to use the platform to build Orchestrator prototypes, so that RDCL 3D plays the role of the NFVO (<3>in Figure 32). This is especially useful when one needs to explore new functionalities and it is easier to have a small stand-alone proof-of-concept implementation rather than integrating the new functionality into a fully-fledged NFVO.
4. A different usage of the proposed network is to be integrated as a library within Orchestrators that do not yet support a GUI as shown in <4> in Figure 32.

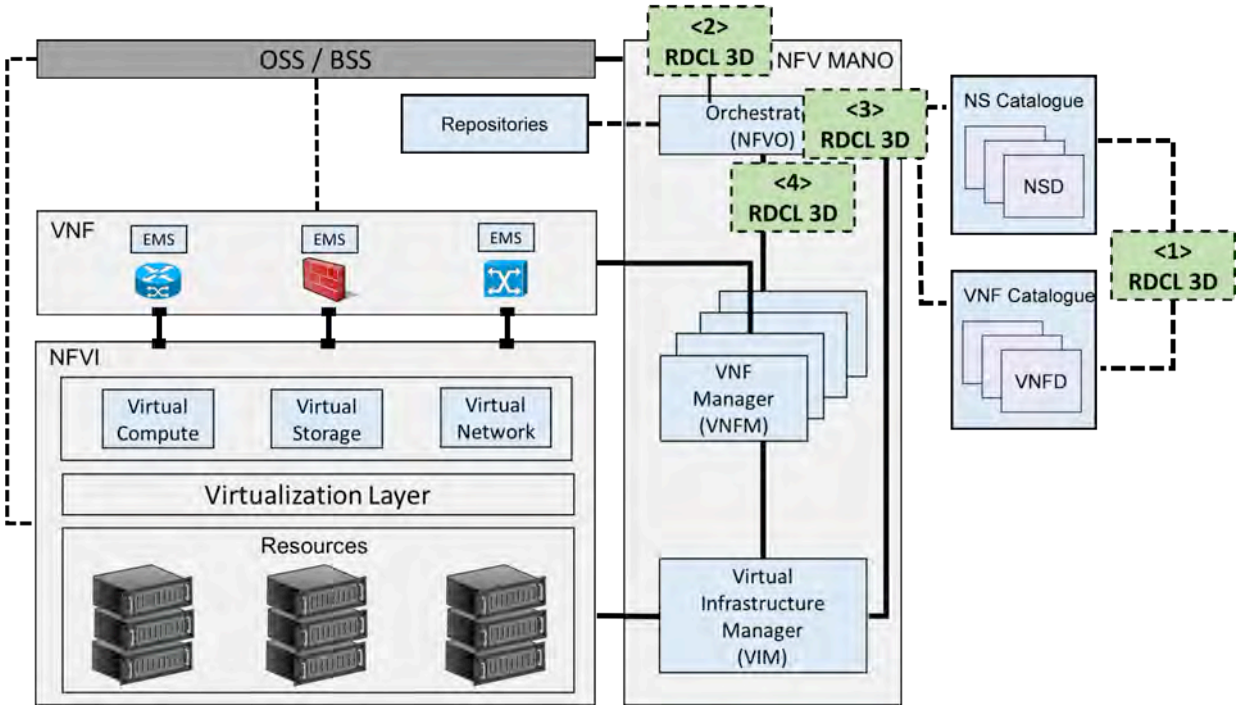

Figure 32: Positioning RDCL 3D in the ETSI MANO architecture

Software Architecture

RDCL 3D is a web application with backend and frontend components, as depicted in Figure 33. 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 33). 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 front-end (<2> in Figure 33), where is it processed and filtered to produce the different graphical views on the browser GUI (<3> in Figure 33). 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). The backend can also optionally deploy the designed network services through an RDCL 3D Agent. An RDCL 3D Agent can deploy the designed network services by either interacting directly with the APIs of an Orchestrator or a VIM or by pushing the network service descriptors to a git repository to which the Orchestrator/VIM has access. Moreover, to allow the interaction with specific VIMs, the RDCL 3D Agent can also perform online translation between descriptor formats. The RDCL 3D Agent exposes REST APIs, which are invoked by the Deployment Handler plugin. Our RDCL 3D Agent implementations are based on node.js, but any technology which allows to expose HTTP REST APIs can be employed.

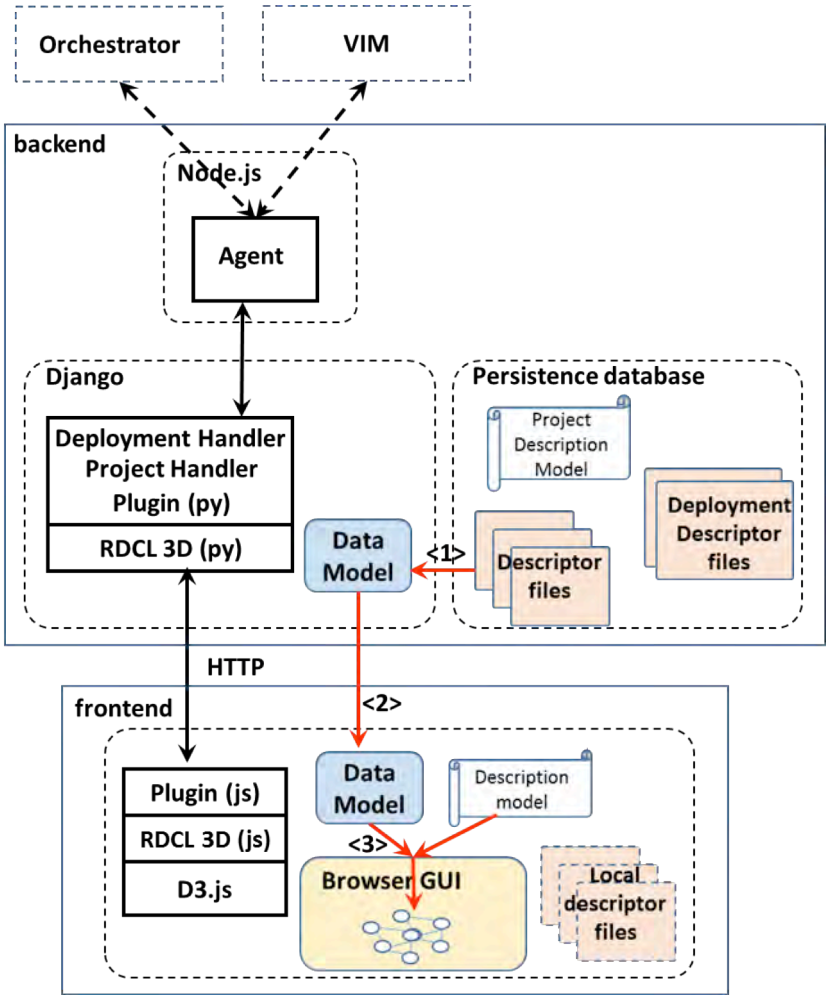

Figure 33: RDCL 3D Software Architecture

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 [1].

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

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

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

6.8. Intel Characterization Framework

A key focus of interest in Task 6.1 is the control framework which focuses on resource allocation for virtual network services. Resource allocation is important in the context of “virtualization” of network services as it plays a very important role in both service assurance and Total Cost of Ownership (TCO). On the one hand it is of crucial importance to allocate the right quantity and type of resources for service execution in order to support the required service level, which is usually measured against Service Level Objectives (SLOs); On the other hand, the overprovisioning of resources leads to higher TCO, due to unused resources and, in some cases, could also lead to service level degradation.

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

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

The main focus of the Characterisation Framework is to support automated characterisation of the relationship between service performance and resource allocation cost on an NFVI. Characterisation is carried out prior to initial placement of a new service in production, resulting in performance optimisation of the services based on the available hardware ingredients. For this reason, this approach is part of the pre-deployment characterisation of a service.

In the context of Task 6.1, the design and implementation of an automation framework is envisioned in order to automate some aspects related to the generation of placement insights for an orchestrator. The primary goal therefore is to automatically define a set of rules that can be interpreted by an orchestrator in order to make intelligent decisions on the quantity and type of resources to be allocated to a service by a VIM (Virtual Infrastructure Manager). Automation is key to determine the best composition of quantity and types of resources to be allocated to a service according to its KPIs and SLOs and considering changing workload conditions such as variations in user load. Making automated and performant deployments decisions enables support for performance requirements in a scalable manner, which results in increased efficiency in the management of features exposed by the platform and the infrastructure resources.

The results of the characterisation procedure are an optimized service descriptor which contains the values for each configuration parameter of interest that corresponds to the most efficient resource allocation option in order to satisfy the customer specified SLOs. This supports the **[KpiTemplate-01]** requirement: “The system must be able to dynamically define a workload deployment template to ensure that resource allocations can support required SLA’s and SLO’s”, as outlined in deliverable i6.1a.

6.8.1. Characterization Lifecycle

There are three phases necessary to characterizing a service using the Characterization Framework:

- **Experiment Execution:** automatic deployment of the service and execution of stress tests.

- **Data Collection:** Exploit metric collected by workload generators and telemetry agents to develop an understanding of the service's behavior under representative operational scenarios.
- **Data Analysis**: extract insights from the data and generate orchestration insights.

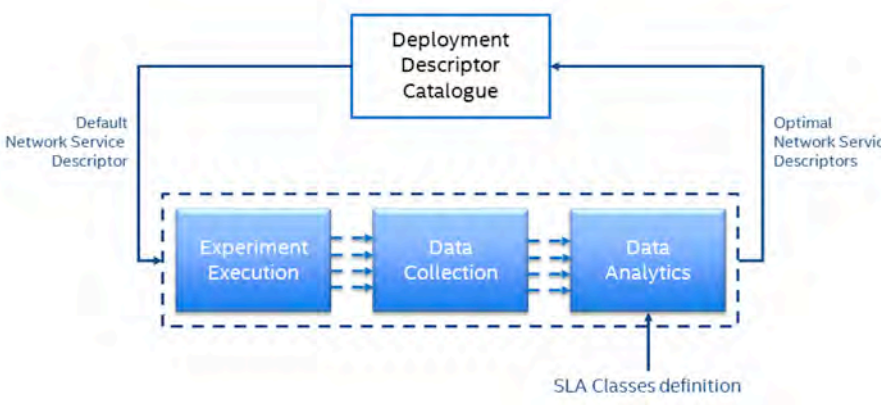

Figure 34: Characterization Framework phases.

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

The user selects a service descriptor which is taken as an input by the Characterization Framework and is used for the characterisation of the service in the three phases listed above.

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

Experiment Execution

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

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

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

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

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

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

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

Data Collection

The collection of data is a very important aspect of the characterization process. There are two types of data that are of relevance to characterization analysis, namely:

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

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

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

Data Analysis

The main scope of the data analysis pipeline for the optimization of the deployment template is to determine the rules that will be used to eventually generate the configuration.

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

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

For each configuration option it is necessary to calculate the contribution to the total cost. For instance, selecting SR-IOV instead of OvS will increase the cost due to the fact that the availability of SR-IOV channels is finite within a given NFVI. The cost does not need to be expressed in an actual currency, but it needs to be proportional, such that the comparison between the different configuration options is meaningful (for instance, it could be based on availability of resources).

The data analytics pipeline then needs to identify the configuration that satisfy the constraints on the SLOs while providing the lowest cost.

6.8.2. Characterization Framework Implementation

In the previous sections the characterization process was described together with a number of requirements. Among them, a number of similarities have been identified with the framework developed during the T4.1 activities related to automated KPI mapping, which has been used for KPI Mapping of the Unified Origin Transmuxing service. As a consequence, the framework previously developed has been reengineered and extended to accommodate the new requirements identified in the context of Task 6.1 i.e. automated optimization service deployment templates. The current high-level architecture of the framework is depicted in Figure 35.

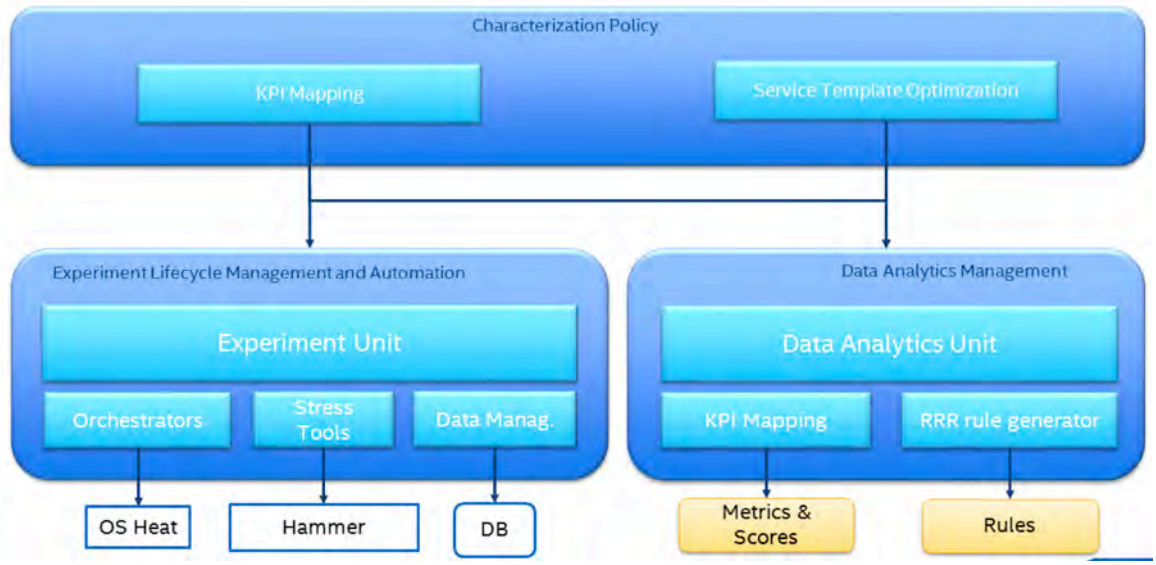

Figure 35: Characterization Framework high-level architecture

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

The **Experiment Unit** provides automatic management of the experiment lifecycle as described before: it supports the deployment of a service, the deployment of any desired noisy neighbour e.g. CPU, network I/O etc., the execution of a stress test and the termination of the services. In order to implement this, a heat template to deploy Unified Origin has been defined that enables the automated deployment of the service. The Experiment Unit manages the generation of all the user required permutations and their execution lifecycle.

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

To run stress tests appropriate for Unified Origin the framework has been integrated with Citrix's workload generator Hammer. The Framework creates a Hammer profile for a simulated users (simuser) profile in accordance to a specific user definition e.g. request type, number of users, linear to stepwise user ramp etc., which is used to stress test different deployment instances of Unified Origin. The user can in fact specify to the framework which kind of simuser profile to use, defining the name and all the necessary details of the user ramp dynamically. Additionally, integration allows at the end of an experiment the collection the metrics stored by Hammer. Those metrics are collected by the framework by automatically establishing an SSH session to the

Hammer node and copying the data file (CSV format) for extraction and storage in an InfluxDB database in time series.

The **Data Analytics Unit** provides automatic analysis of the experiment data. It loads the experiments data according to the specific type of processing required. The data analytics module to support the KPI Mapping has been previously completed (see I4.1 and I4.1b), the Service Template Optimization remains a work in progress and will be finalized in the coming months, according to the requirement analysis described above.

Unified Origin Deployment Optimisation

The definition of a Heat template has been finalized, as well as the framework's integration with Citrix Hammer, enabling the automated deployment in conjunction with stress test execution in an OpenStack cloud environment. An initial experimental campaign has been completed to identify performance ranges and the impact that the configuration parameters have on it.

The results that are shown were collected from experiments run on the same testbed that was used for the KPI Mapping activities described in I4.1a.

A set of 16 different configurations have been executed as shown in next table.

	vCPUs	RAM	vNIC type	Memory size	Page
Configuration 1	4	4	SR-IOV	Large	
Configuration	4	4	OvS	Large	
Configuration 3	4	4	SR-IOV	Small	
Configuration 4	4	4	OvS	Small	
Configuration 5	2	4	SR-IOV	Large	

Configuration 6	2	4	OvS	Large
Configuration 7	2	4	SR-IOV	Small
Configuration 8	2	4	OvS	Small
Configuration 9	4	2	SR-IOV	Large
Configuration 10	4	2	OvS	Large
Configuration 11	4	2	SR-IOV	Small
Configuration 12	4	2	OvS	Small
Configuration 13	2	2	SR-IOV	Large
Configuration 14	2	2	OvS	Large
Configuration 15	2	2	SR-IOV	Small

Configuration 16	2	2	OvS	Small

The obtained results are shown in Figure 36 and 37, which are snapshots of the data in InfluxDB.

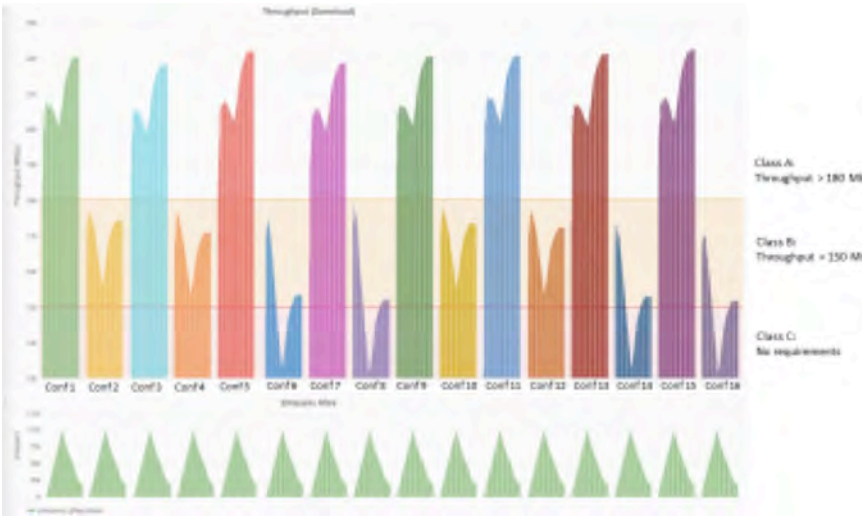

Figure 36: Throughput results for all the different configurations.

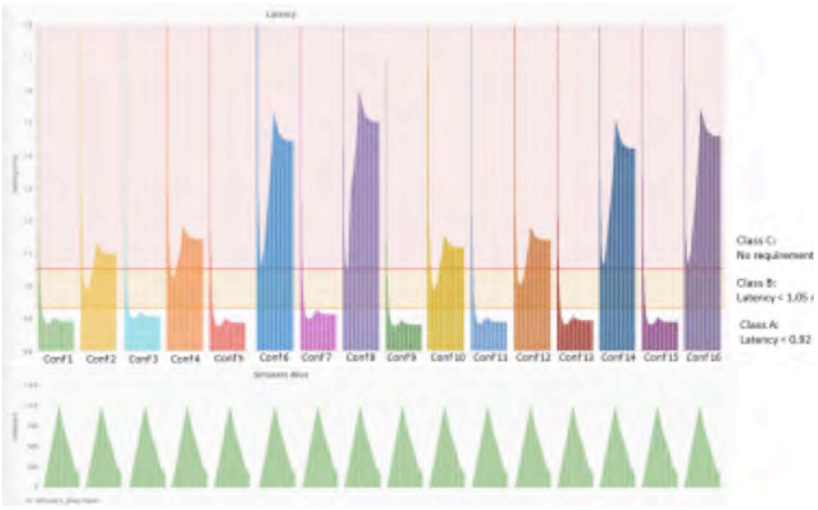

Figure 37: Latency results for all the different configurations.

The results obtained to date show that varying the combination of the four parameters it is possible to classify the configurations into 3 unique performance classes:

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

In Figures 4 and 5, the simuser profile used for the experiment is shown. The simuser ramp scales from 0 to 1000 users in the first phase of the experiments and then ramps back down to 0 in the second phase of the experiment. The results also show the impact of increasing the number of users on the KPIs, since the throughput increases and the latency decreases when the number of user increases and vice versa.

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

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

In future work, the automatic methodology will be extended to provide a reasoning capability over the results and to determine the optimal configuration to use for each service in a fully automated fashion.

6.9. MicroVisor Orchestration (taken from I6.1)

One of the most challenging requirements captured in Section 2 is [AppSched-03], that states that an application must be provisioned in <10ms. For a VIM such as OpenStack, this poses a number of challenges as this Python-based framework was designed for 'traditional' VMs that usually comprise a Linux or Windows-based guest OS. These VMs are heavy-weight and need miniaturization before they could start in the order of seconds. Containers and other light-weight virtualization techniques as those currently investigated in Superfluidity can start up much faster. When looking to approach fast provisioning and orchestration tools it is important to profile all aspects of the virtualization workflow. This will be reported by T5.2, where an analysis of different virtualization techniques is being carried out.

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

6.9.1. UI design for managing a large collection of resources

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

Computer assisted workload placement will therefore move from being just an optimization effort, to being a tool to help manage the workloads at the scales that are expected. A mockup diagram showing a possible visualization of the physical to virtual workloads is shown in Figure 38. This Figure captures the physical, network overlay and virtual resources and how they relate to each other. The work is ongoing to determine which visualization mechanisms users and administrators will find useful.

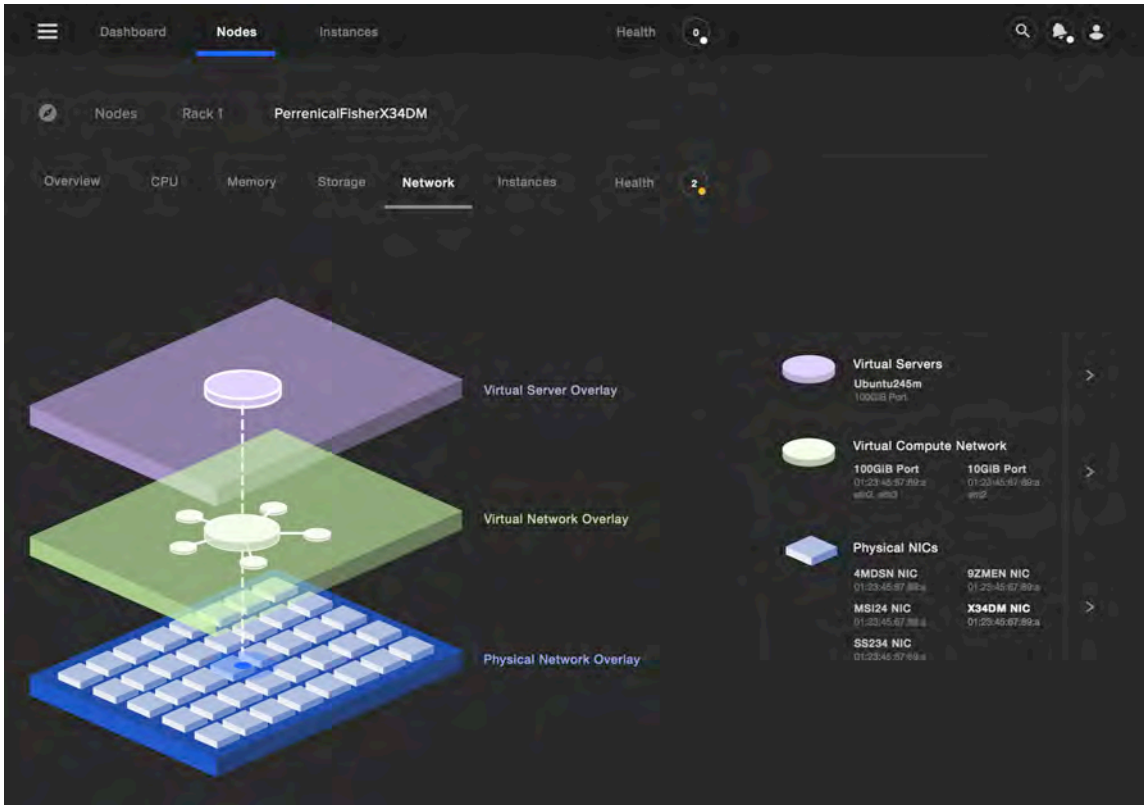

Figure 38: Mock-up diagram showing a UI that relates virtual to physical resources

In Figure 39 a visualization mock-up of the administration panel is shown. In this visualization, the physical racks have a number of rack servers that are numbered and can be probed for more

information. Each rack then has a number of compute nodes that can be contained within a single physical server. The CPU load of each compute unit is then visualised, with standard traffic light coloring used to indicate low-utilization (green), through to heavily loaded compute units (red). This gives an administrator a powerful tool to quickly identify if there are any servers that are struggling and to indicate issues that potentially need to be resolved either through computer assisted orchestration, or manual intervention.

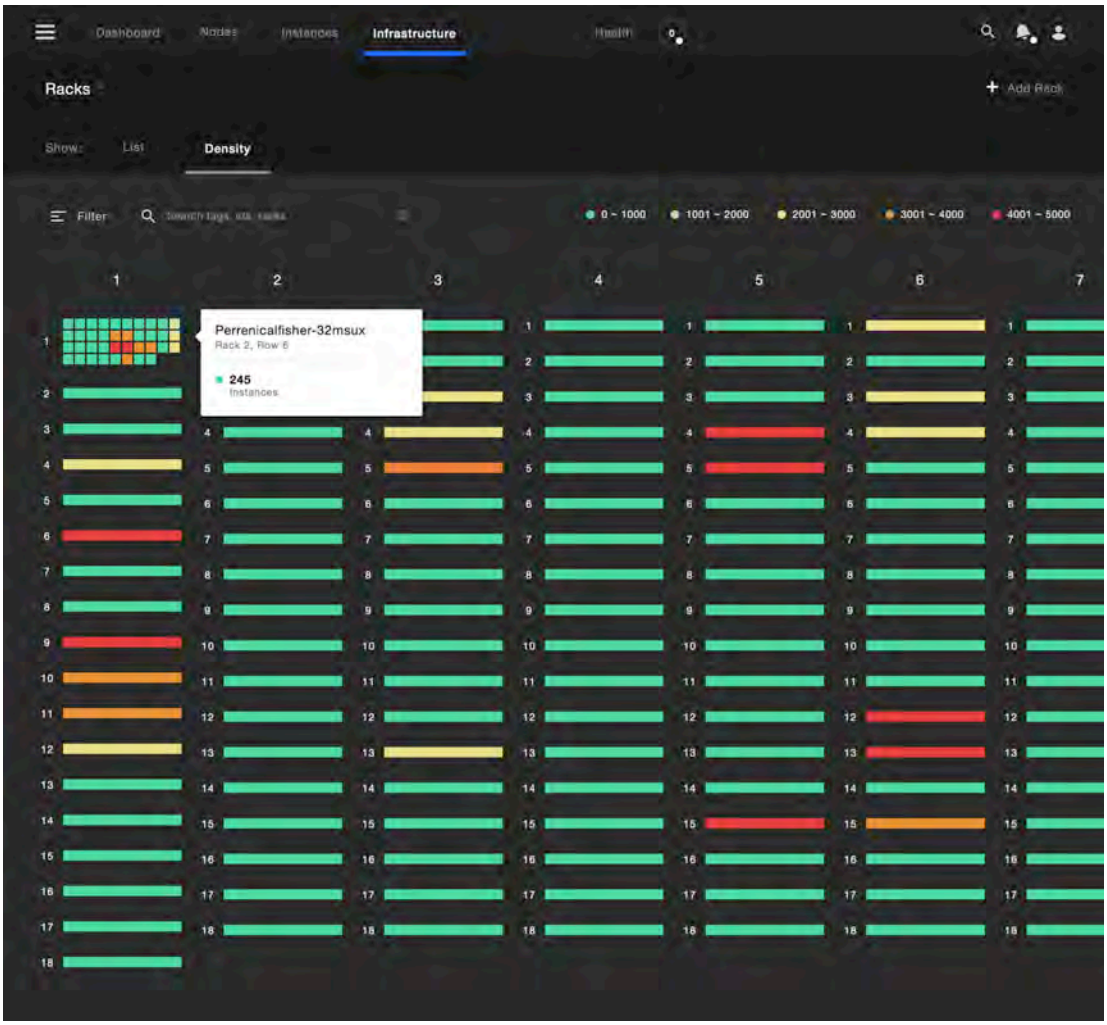

Figure 39: Mock-up diagram showing the rack utilization

Aside from the physical to virtual mapping and CPU load that have been shown in the previous Figures, it is also important to show the utilization of the storage resources. A mock-up Figure showing the utilization of the storage can be seen in Figure 40. Disks that are close to being full are shown in red with the less utilized disks being coloured in blue. All of the storage resources are associated with particular racks and are separated accordingly. Also shown in the diagram is the

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

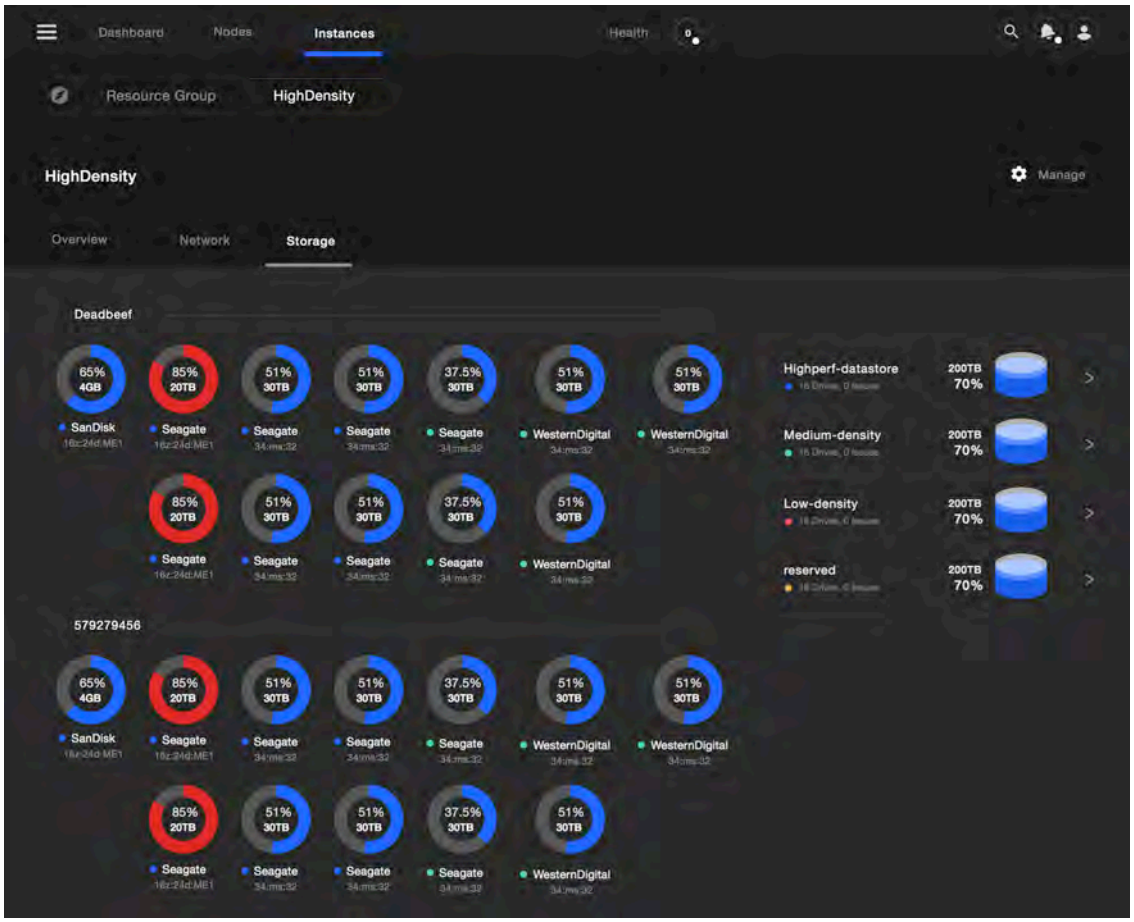

Figure 40: Mock-up diagram showing the storage utilization in the management UI

Given that SDN networking will also allow reconfiguration of a network, it is important for both the management platform and the orchestration system to be able to capture and possibly modify the network topology. This will allow maximization of the performance for a given set of workloads and configurations decided by the administrator. In Figure 41 a mock-up of the network mapping UI is shown. This can be used to visualize the current network topology and also could be used to capture modifications required of the network that could be then mapped to the network routers and hypervisors through tools such as OpenDaylight or others.

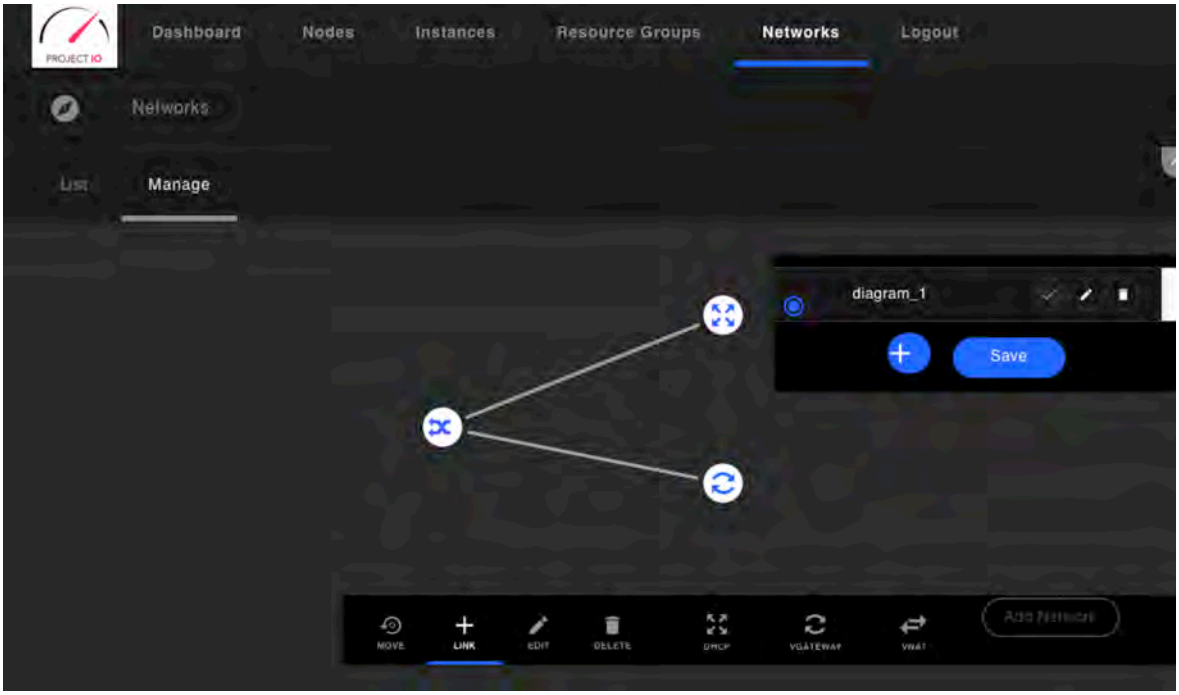

Figure 41: Mock-up showing the network planner UI

7. Conclusions

The progress has been made on several challenges: recognition of the requirements from the control framework, analysis of the existing work of MEC workgroup, looking into subset of the currently existing solutions on the market and highlighting the gaps between the requirements and the solutions. A very important progress has been made on the management and orchestration design side with most of the components already selected. Additionally, considerable work/progress has been done on these components to provide support for the missing features to achieve Superfluidity objectives.

References

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

[2] 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

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

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

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

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

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