Ref. Ares(2019)7659925 - 12/12/2019

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017

# SEMIoTICS

# Deliverable D3.6
# Field-level middleware & networking toolbox (second draft)

| | |
|---|---|
| Deliverable release date | 12/12/2019 |
| Authors | 1. Kostas Ramantas, Prodromos Vasileios Mekikis (IQU)<br>2. Luis Sanabria-Ruso, Jordi Serra, David Pubill, Angelos Antonopoulos, Christos Verikoukis (CTTC)<br>3. Kostas Fysarakis, Jason Somarakis (STS)<br>4. Manos Papoutsakis, Manolis Michalodimitrakis (FORTH)<br>5. Ermin Sakic, Darko Anicic, Arne Boering (SAG) |
| Responsible person | Kostas Ramantas (IQU) |
| Reviewed by | Kostas Fysarakis, Ermin Sakic (SAG), Eftychia Lakka (FORTH), Jordi Serra (CTTC), Urszula Rak (Bluesoft), Nikos Petroulakis (FORTH) |
| Approved by | PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Mirko Falchetto, Domenico Presenza, Verikoukis Christos)<br>PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas) |
| Status of the Document | Final |
| Version | 1.0 |
| Dissemination level | Confidential |

# Table of Contents

Acronyms Table

| Acronym | Definition |
|---|---|
| CPU | Central Processing Unit |
| ICT | Information Communication Technology |
| IoT | Internet of Things |
| IIoT | Industrial Internet of Things |
| JSON | JavaScript Object Notation |
| KPI | Key Performance Indicator |
| KVM | Kernel-based Virtual Machine |
| LCVNF | Latency Critical VNF |
| LTVNF | Latency Tolerant VNF |
| LXD | Linux Containers |
| NETCONF | Network Configuration Protocol |
| NFV | Network Functions Virtualization |
| NFVO | NFV Orchestrator |
| MEC | Mobile Edge Computing |
| OFCONF | OpenFlow Configuration |
| ODL | OpenDaylight |
| OvS | Open vSwitch |
| OVSDB | Open vSwitch Database Management Protocol |
| POP | Point of Presence |
| QoS | Quality of Service |
| SDN | Software-Defined Networking |
| SSC | SEMIoTICS SDN Controller |
| SARA | Socially Assistive Robotic Solution for Mild Cognitive Impairment or mild Alzheimer's disease |
| SEMIoTICS | Smart End-to-end Massive IoT Interoperability, Connectivity and Security |
| SFC | Service Function Chaining |
| SPDI | Security, Privacy, Dependability, and Interoperability |
| TD | Thing Description |
| UC | Use Case |
| VIM | Virtualized Infrastructure Manager |
| VLAN | Virtual Local Area Network |
| VM | Virtual Machine |
| VNF | Virtual Network Function |
| vSwitch | Virtual Switch |
| VTN | Virtual Tenant Network |
| WoT | Web of Things |

# 1 INTRODUCTION

SEMIoTICS aims to enhance the connectivity, latency and bandwidth in industrial environments, while reducing the cost of their Information and Communications Technology (ICT) systems through a set of technologies under the umbrella of virtualization. On the other hand, network function virtualization (NFV) is a technique that can significantly benefit industries by optimizing their network services. It allows for a software-defined implementation of networks as it decouples several network functions from previously required network devices, such as firewalls, and runs them as software, i.e., Virtual Network Functions (VNFs), at a data center. In this way, the NFV infrastructure (NFVI) does not only drop the deployment cost, as less equipment and installation personnel are needed, but it also reduces the service creation time from hours to minutes resulting in an extensively more efficient procedure.

To automate even further the networking procedures in the Industrial Internet of Things (IIoT), software-defined networking (SDN) can be employed, which is a complementary approach to NFV that separates the control and forwarding planes to offer a centralized view of the network. Moreover, for the handling of the physical and virtual resources that support the network virtualization, an NFV management and orchestration (MANO) is responsible for the lifecycle management of the VNFs and it focuses on all virtualization-specific management tasks necessary in the NFV framework. To that end, a service chain of connected VNFs, i.e., a service function chain (SFC), can be created to automatically run a requested application based on the current traffic demand. This capability can be employed by industries to set up sets of connected VNFs that allow the use of a single network connection for many services that have different characteristics.

Although the set of aforementioned technologies can substantially improve the efficiency of the network layer in IIoT, there is still the obstacle of the ability of things to interact in a meaningful way. Knowing that there are so many diverse IIoT devices and even more possible ways of their interaction using diverse communication protocols, there is a need to define novel technologies that introduce interoperability in IIoT environments. One way to achieve this is to describe things, their capabilities, and data they produce or consume in a machine understandable form. Such a description could be then used to discover things relevant for an application. It can also serve to figure out how these things could interact. The description should be formalized, with a clear semantic meaning, so that both humans and machines can interpret it. In this way we would not have just Internet of mere things. Instead, IoT would be the Internet of semantically-described things. Semantics for IoT is the key enabler of applications that operate on physical world objects. It is a prerequisite for achieving the interoperability of things, and thus for realization of a new class of IoT applications.

In this deliverable, we investigate the introduction and adaption of SDN/NFV, semantic bootstrapping and interoperability technologies in industrial environments. Furthermore, we employ the well-defined SEMIoTICS architecture[1] and Middleware to build an experimental platform that consists of open-source software and novel SEMIoTICS modules and frameworks. Hence, the contribution of this deliverable is the following:

i) In Section 2, we contribute the design of the SEMIoTICS field-level middleware and define the SEMIoTICS development and release procedure.
ii) In Section 3, we discuss how concepts like NFV and SDN can be leveraged in the Industrial IoT domain.
iii) In Section 4, we study standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic bootstrapping and interoperability.
iv) Finally, in Section 5, we present the design of the SEMIoTICS integration testbed and contribute performance evaluation results that provide useful insights for the deployment of IIoT applications on top of virtualized, programmable and pattern-driven infrastructures.

## 1.1 What has changed in the second cycle deliverable

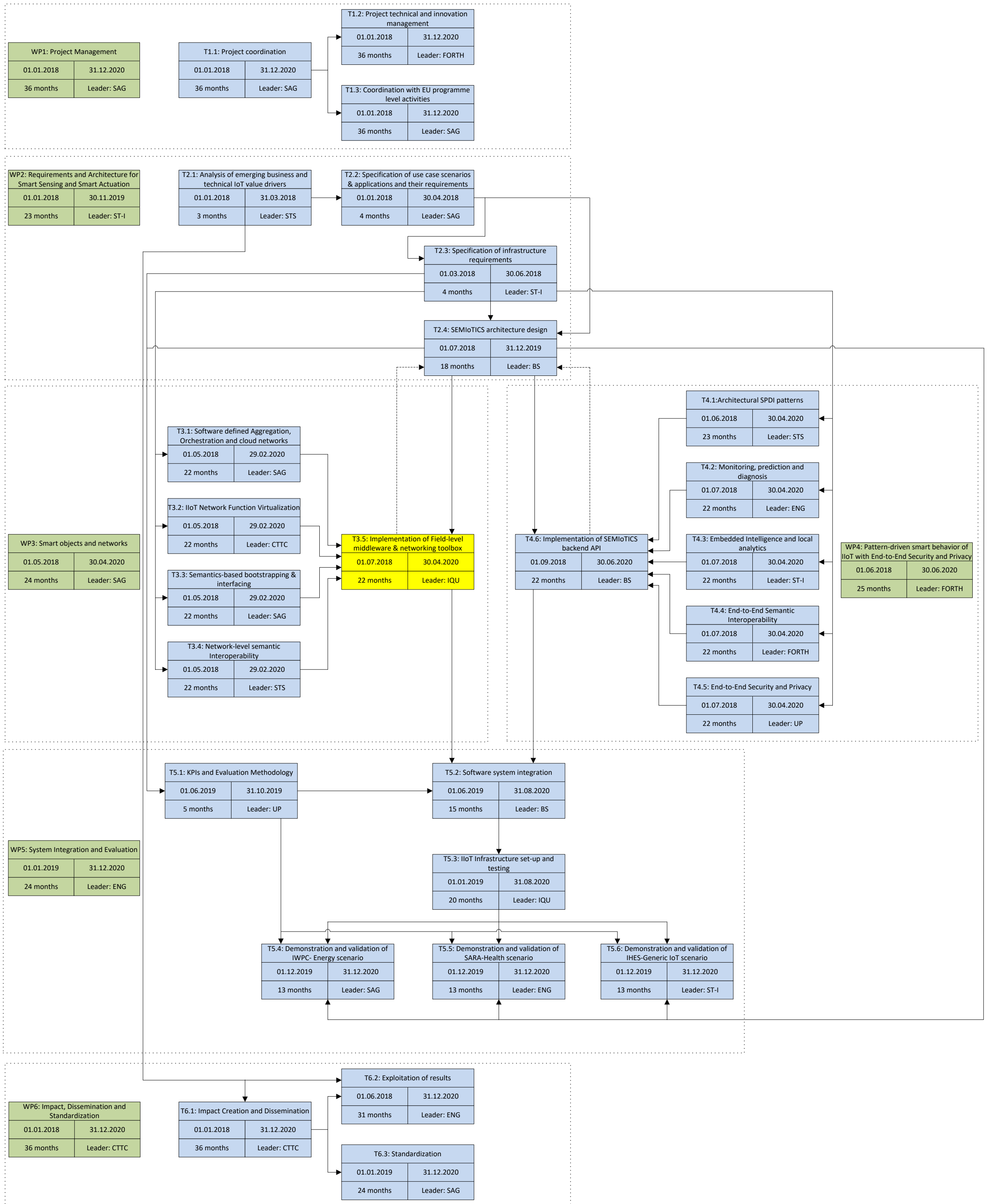
Deliverable D3.6 is the Second Cycle deliverable of D3.5. In this section, we detail how the previous version of the deliverable was updated, and which new sections were added:

- A major update was performed to section 2.1, to be in-line with the SEMIoTICS architecture which was introduced in D2.4 and further updated in D2.5. Furthermore, section 2.1.2 was added to highlight how the Middleware is used in individual use cases.

[1] SEMIoTICS webpage: https://www.semiotics-project.eu/

- Table 1 was added, which lists all SEMIoTICS specifications that are relevant to the Middleware implementation, and details the steps partaken to satisfy the respective specifications.
- Section 3.1.2.4 was added which presents a new telemetry feature that collects measurements from diverse virtualized elements of the architecture.
- Section 3.2 was added, with a focus on NFV Orchestration at the Middleware layer. In the Second cycle, support for ETSI OSM NFVO was added, to manage individual IIoT services, implement automated VNF placement and VNF scheduling with scale-out and load balancing operations.
- Section 3.3 was revised, including new features implemented in the SEMIoTICS SDN Controller during the second implementation cycle. Furthermore, the SDN testbed shown in a video presentation during the Mid-Term review of SEMIoTICS is presented in this section.
- Section 4.2.1 was added, which describes implementation aspects of the Pattern Engine that is integrated with the SEMIoTICS SDN controller.
- Section 4.2.2 was added with a use case example that relies on the SEMIoTICS SDN controller and the concept of pattern-driven network interfaces which was also presented during the Mid-Term review.
- Section 4.3 was added to describe the current state of the implementation of SEMIoTICS IoT Gateway and the process of bootstrapping and interfacing of field devices. Moreover, it demonstrates the semantic edge platform for device scanning, configuration and exposure for IIoT applications, also presented during the Mid-Term review.
- Section 5.1.1 was revised, to reflect how the SEMIoTICS integration testbed is in line with the SEMIoTICS architecture.
- Section 5.1.2 was updated, detailing the deployment of the semantic interoperability functions at the integration testbed. Furthermore, the newly implemented support of edge virtualization at ARM-based IIoT devices is detailed.
- Section 5.1.3 was added, with an NFV demonstration scenario presented during the Mid-term review.
- Section 5.3 was added, with the experimental evaluation of the NFV Orchestration subsystems for VNF placement and scheduling in the integration testbed.

## 1.2 PERT chart of SEMIoTICS




Please note that the PERT chart is kept on task level for better readability.

# 2 MIDDLEWARE DESIGN AND IMPLEMENTATION

The sheer number of smart objects that are expected to connect to the Internet by will increase network traffic dramatically and introduce more diversity of network traffic. A series of innovations across the IoT landscape have converged to make IoT products, platforms and devices technically and economically feasible. Specifically, Integrating IoT and SDN will increase network efficiency as it will make it possible for a network to respond to changes or events detected at the IoT application layer through network reconfiguration. Moreover, NFV architectures allow monitoring, caching, security and data analytics functions to be virtualized and placed in a local and remote clouds, or event directly at IoT smart objects and Field level IoT gateways. Finally, intelligent data analytics running locally at the Field layer are needed to implement autonomic behaviour, but considering IoT smart objects' limited resources, specialized lightweight algorithms are required. The aforementioned complexities must be abstracted from the IIoT applications and field-level devices, simplifying the development and deployment of applications. Hence, SEMIoTICS has proposed the development of a Field-Level Middleware that will integrate the application modules and networking APIs implemented in T3.1-3.4 and provide, ensuring interoperability and simplifying application development.

## 2.1 SEMIoTICS Middleware

### 2.1.1 MIDDLEWARE DESIGN AND SPECIFICATIONS

This section will focus on the design of the SEMIoTICS unified Middleware, shown in Figure 1. Furthermore, it details all SEMIoTICS specifications that are relevant to the Middleware implementation, and how they were addressed. It must be noted that the Middleware is not a separate component of the SEMIoTICS architecture, which is detailed in D2.4 and D2.5, but rather the implementation of frameworks and APIs designed within T3.1-T3.4 and include the NFV, SDN, Semantic Interoperability and Pattern Engine frameworks. These are deployed and evaluated in a testbed environment as part of T3.5. The Middleware ensures that functionalities such as establishing connectivity to a service, negotiating transport protocols and networking paths, as well as service scale-out and load balancing functions will be totally transparent for IoT applications, and are handled by the respective Middleware frameworks. The Middleware also serves the purpose of bridging these frameworks with the SEMIoTICS Backend layer and the Pattern orchestrator. The Pattern Orchestrator is responsible for defining policies leveraging the pattern language, which are enforced by the relevant Pattern Engine. These policies and requirements are then implemented in each domain by the respective framework (e.g., the networking policies are implemented by the SDN framework, Service policies by the NFV framework, etc.).

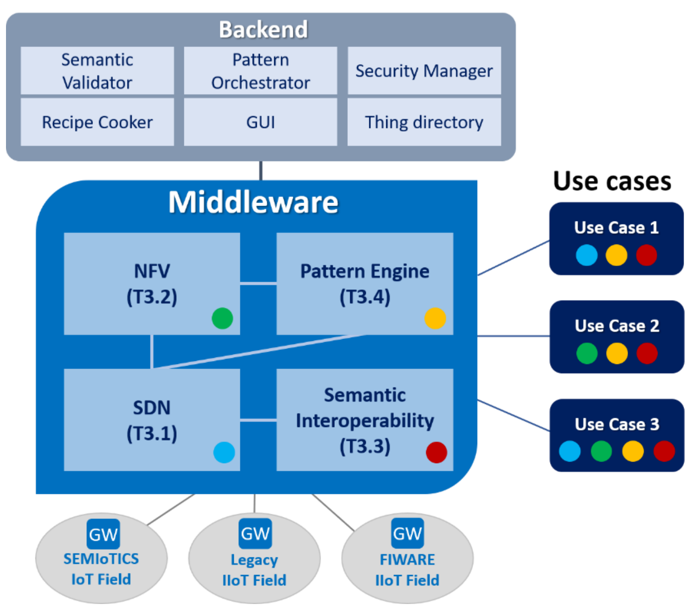



FIGURE 1: FIELD LEVEL MIDDLEWARE

At the Field layer, SEMIoTICS leverages semantically annotated messages transmitted from the Field devices, that include Context Information (e.g., sensor values). To counter the fragmentation which is inherent in the IoT ecosystem, the IIoT gateway performs Semantic Mapping of these messages, employing the W3C Web of Thing (WoT) data model. The semantically annotated messages are then delivered to the Backend where the context information is processed and stored. A Context API, which is part of the SEMIoTICS Backend and implemented in WP4, provides access to the context data via a REST API, which includes:

- Context queries, e.g., for sensor data stored at the local database
- Context updates, e.g., to update the local database with sensor values
- Context subscriptions, to receive updates when a certain device status (or a certain topic) is updated

Table 1 lists all SEMIoTICS specifications that are relevant to the Middleware implementation, and lists the steps partaken to satisfy the respective specifications. For each specification we include its Req-ID, which maps with the respective specification defined in D2.3, and its current status which can be "Done", or Work In Progress (WIP).

TABLE 1. SEMIOTICS MIDDLEWARE REQUIREMENTS SPECIFICATION

| Topic | Req-ID | Status | Description | Steps partaken to satisfy the requirement |
|---|---|---|---|---|
| **General Platform Requirements** | R.GP.1 | Done | End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level) | The SSC implements interfaces for providing best-effort and QoS-constrained service addition and thus addresses the requirement for end-to-end connectivity. Delay, bandwidth, resilience requirements be met by implemented approach. |
| | R.GP.2 | Done | Scalable infrastructure due to the fast-paced growth of IoT devices | Multiple SSC instances can be deployed for purpose of domain partitioning and thus achieving higher scalability. VNF scaling out operations provide scalability to IIoT apps. |
| | R.GP.3 | Done | High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication) | The SSC's interface for connectivity instantiation can be used at runtime and it is enabled to adapt the state of reservations and implement new flows without service guarantee loss for existing flows. |
| | R.GP.4 | Done | Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern | SSC's Pattern Engine aggregates connectivity and QoS related inputs, triggering reasoning and adaptation to react to failures, reporting the invalidated connectivity-related pattern instances and associated changes to the backend (Pattern Orchestrator). |

| | | | | |
|---|---|---|---|---|
| | R.GP.5 | Done | Interaction between SDN controller and IoT backend cloud through a dedicated interface (called northbound software interface) | Northbound interface of the SSC enables the communication to backend cloud via Pattern Orchestrator. To that end, SDN Controller exposes the REST interface used to specify pattern rules in Drools format. |
| | R.GP.6 | Done | Interaction between SDN controller and network nodes (e.g. switches, routers or IoT Gateways) through dedicated interface (called southbound software interface) | The controller is capable of interacting with switches using OpenFlow southbound protocol. |
| | R.GP.7 | Done | SDN controller giving feedback for a future generation of SPDI patterns to avoid using the same pattern in case of failure | SSC's Pattern Engine is enabled to react to failures and report the invalidated connectivity-related pattern instances to Pattern Orchestrator |
| **Backend/Cloud Layer Requirements** | R.BC.1 | Done | Controller Node requirement: At least 6 CPU cores and 32 GB RAM | All hardware and software requirements have been met. Controller and Hypervisor hardware requirements (RAM, HDD size, network interfaces) were derived from the requirements of the respective MANO and OpenStack management services. |
| | R.BC.2 | Done | Controller Node requirement: At least 2 Network interfaces | |
| | R.BC.3 | Done | Controller Node Requirement: Linux OS | |
| | R.BC.4 | Done | Controller Node Requirement: Solid State Disk (SSD) of at least 256 GB | |
| | R.BC.5 | Done | Hypervisor Requirement: At least 4 CPU cores and 8 GB RAM | |
| | R.BC.6 | Done | Hypervisor Requirement: At least 2 Network interfaces | |
| | R.BC.7 | Done | Hypervisor Requirement: Virtualization Extensions (Intel VT-x/AMD-V) must be supported by the Hypervisor CPU for hardware acceleration of VMs. | |
| | R.BC.8 | Done | Hypervisor Requirement: KVM must be supported by the Hypervisor Linux OS | |
| | R.BC.9 | Done | Hypervisor Requirement: Linux Containers (LXD) must be supported by the Linux OS | |
| | R.BC.10 | Done | Virtual Switch requirement: Support for OpenFlow protocol | |
| | R.BC.11 | Done | Virtual Network requirement: Support for GRE, VLAN, and VXLAN tunnels for virtual tenant networking. | |

| | | | | |
|---|---|---|---|---|
| | R.BC.12 | Done | The VIM and Virtual Network frameworks must support Interfaces that enable VM tenant networking | |
| | R.BC.13 | Done | Interface between the VIM and the SDN controller to allow Tenant Network Slicing | Slicing module implementation with OpenStack Neutron APIs and verification completed. |
| | R.BC.14 | WIP | Interfaces among the MANO entities (NFO, RO, NFVO) and the VIM must ensure seamless interoperability among different entities of the Backend Cloud | Functionality provided by the Pattern-driven NBI |
| | R.BC.15 | WIP | Secure communication among the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules) | Current status uses Neutron virtual firewall service, integration with Security manager under way |
| **Network Layer Requirements** | R.NL.1 | Done | Controller Node requirement: At least 6 CPU cores and 32 GB RAM | All hardware requirements of the SDN and NFV have been provisioned. |
| | R.NL.2 | Done | Controller Node requirement: At least 2 Network interfaces | |
| | R.NL.3 | Done | Controller Node Requirement: Linux OS | |
| | R.NL.4 | Done | Controller Node Requirement: Solid State Disk (SSD) of at least 1 TB | |
| | R.NL.5 | Done | Data paths / Hypervisor Nodes Requirement: At least 4 CPU cores and 8 GB RAM, at least 2, 1Gbps Network interfaces, Virtualization Extensions (Intel VT-x/AMD-V) must be supported by the Hypervisor CPU for hardware acceleration of VMs. | |
| | R.NL.6 | Done | Data paths / Hypervisor Nodes: KVM and Linux Containers (LXD) must be supported by the Hypervisor Linux OS | |
| | R.NL.7 | Done | Virtual Switch requirement: Support for OpenFlow v1.3 protocol or greater | Virtual and Physical SDN switches used in SEMIoTICS support OpenFlow1.3 protocol, as well as the developed SSC. |
| | R.NL.8 | Done | The VIM and Virtual Network frameworks must support Interfaces that enable VM tenant networking | VM tenant networking is enabled by the SDN controller through its VTN Manager implementation, allowing for |

| | | | | |
|---|---|---|---|---|
| | | | | VTN realization and Layer 2 isolation of virtual network participants. |
| | R.NL.9 | Done | Interface between the VIM and the SDN controller to allow VTN | VM tenant networking is enabled by the SDN controller through its VTN Manager implementation, allowing for VTN realization and Layer 2 isolation of virtual network participants. |
| | R.NL.10 | WIP | Interfaces among the MANO and the VIM must ensure seamless interoperability among different entities of the Backend Cloud | Functionality provided by the Pattern-driven NBI, along with standard interfacing capabilities of selected SDN Controller (ODL) and backend MANO capabilities |
| | R.NL.11 | WIP | Secure communication with the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules), as well as the communication between VIM, SDN Controller, and MANO, with data paths acting as computing nodes for VNF spinoff. | Distributed compute nodes are used for VNF spinoff that enable data paths throughout the platform. The capability is enabled, its status is subject to the integration and UC implementation plans. |
| **Field Layer Requirements** | R.FD.5 | WIP | Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components | |
| | R.FD.6 | WIP | Field devices MUST interoperate using a standard communication protocol like Rest APIs, COAP, MQTT. | |
| | R.FD.7 | WIP | Field devices MUST support secure bootstrapping / registration protocol. | |
| | R.FD.8 | WIP | Field devices MUST support secure bootstrapping / registration protocol. | |
| | R.FD.9 | WIP | Field devices MUST be able to communicate with the IIoT Gateway / other architectural components. | |
| | R.FD.12 | WIP | Greenfield device is expected to expose its capability over a W3C Thing Description, which semantically describes field | |

| | | | | |
|---|---|---|---|---|
| | | | resources, and to be computationally powerful enough to run a node-wot servient (that exposes the TD). | |
| | R.FD.13 | WIP | Brownfield device is assumed to consist of a sensor/actuator and a controller (PLC). The controller is expected to expose capability of its sensor/actuator over a native brownfield protocol (without the need for IIoT Gateway to interact directly with them). | |
| **IoT Security and Privacy Requirements** | R.S.1 | WIP | The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms. | SEMIoTICS Security manager leverages SoTA mechanisms |
| | R.S.2 | Done | Authentication and authorization of the stakeholders MUST be enforced by the Network controller, e.g. through access and role-based lists for different levels of function granularities (overlay, customized access to service, QoS manipulation, etc.) | Authentication and authorization of stakeholders is enforced through the Security Manager component of the SSC. |
| | R.S.4 | Done | All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually. | Any interaction with the NFV Component must be done by an authorized party. Tokens/credentials must be distributed to other relevant components (e.g. Pattern Orchestrator) during integration. |

### 2.1.2 MIDDLEWARE FRAMEWORKS AND USE CASES

As already mentioned, the Middleware is responsible for implementing the IIoT policies (e.g., application requirements in terms of delay, minimum throughput, packet error rate tolerances, etc.), defined with Patterns, with its various frameworks. WP3 frameworks are designed and developed in separate tasks, and their integration is performed in T3.5. The SEMIoTICS use cases leverage these frameworks via their APIs to implement advanced functionalities; please refer to deliverable D2.5 for the full API diagram with all supported APIs from SEMIoTICS architectural components and frameworks. Figure 1 above shows these frameworks along with the respective use cases that leverage their functionalities and APIs. In more detail, these include:

- The **SDN** framework is responsible for network programmability and network virtualization. Its APIs allow the implementation of Virtual Tenant Networks (VTNs), traffic steering via SFCs and QoS provisioning via traffic shaping and prioritization of critical services. The SDN APIs are leveraged in Use Case 1 for QoS provisioning in critical infrastructure monitoring, in a wind park scenario.
- The **NFV** framework is responsible for the virtualization and orchestration of the Compute and Storage infrastructure. IIoT applications are implemented as chains of VNFs that can be managed by the NFV Orchestrator. NFV APIs allow individual VNFs can be scaled-out to meet increased demand via load

balancing, optimally placed and migrated at edge or cloud hypervisors to meet latency requirements. NFV is leveraged in UC2 and UC3 to implement local/edge clouds at customer premises which can host VNFs that implement latency critical functions, that have to be deployed on-site. Specifically, in UC2 VNFs implement security functions (i.e., a virtualized firewall and a DPI) for the healthcare use-case, while in UC3 VNFs are leveraged for sensor value aggregation, correlation and visualization.

- The **Semantic Interoperability** framework is leveraged across the board in SEMIoTICS, and is present in all Use Cases. Its APIs allow the automated discovery and bootstrapping of SEMIoTICS field devices, greatly simplifying device commissioning. Furthermore, brownfield (i.e., legacy) devices leverage the Semantic Mapping APIs for interoperability with the SEMIoTICS infrastructure.
- Finally, the **SPDI Pattern-driven reasoning,** enabled by Pattern Engines deployed at all layers of the IIoT infrastructure and which leverage the pattern language (see deliverable D4.1) to define properties for the connection between the NFV Management and Orchestration components and the NFV Infrastructure. With regard to network-level interfacing with SEMIoTICS, the pattern language defines and enforces (via the Pattern Engine) mechanisms that guarantee the establishment of E2E connectivity between different types of devices (e.g. SARA hubs, sensors, backend servers), actors (e.g. human operators, applications) and interaction types (e.g. maintenance, medical staff, simple user/patient), while monitoring that certain SPDI (and, additionally, QoS) properties are maintained. Due to its central role in SEMIoTICS, Pattern Engines are deployed in all use cases.

All aforementioned frameworks and APIs are deployed at the SEMIoTICS integration testbed, which is detailed in Section 5, to be tested and evaluated.

## 2.2 SEMIoTICS Implementation process

In SEMIoTICS, T3.5 is the main implementation task of WP3, which will deliver the SEMIoTICS Middleware in incremental releases. In the following sections, the software development and release processes are defined.

### 2.2.1 SEMIOTICS DEVELOPMENT AND RELEASE CYCLES

In the framework defined in T2.4, we have designed the SEMIoTICS architecture and defined the architectural components of each layer. Each architectural component is associated with a respective software module, and an owner is assigned. These software modules are implemented with an iterative process, which follows the concept of Continuous Integration (CI). This iterative development process is performed in cycles, with each cycle ending with a new software release. Each release cycle consists of the following phases, also illustrated in Figure 2, and is expected to last approximately 4 months:

1. **Feature planning**: The consortium agrees on the features that will be implemented in the next release. This might occur during a feature planning meeting. They compile all required mechanisms and interfaces in a high-level specification document, which also includes the test cases which will be executed during system verification. This phase requires approximately 1 month.

2. **Development:** With the specification document at hand, all required features are implemented by the responsible developers. Each partner is responsible for a certain number of architectural components, as defined in T2.4, and will have to implement all essential functionalities. Furthermore, appropriate testing will ensure that the developed components and feature sets perform as specified. Development requires 2 months.

3. **Integration**: After completion of the development phase, changes are integrated to the main SEMIoTICS codebase. Automated sanity tests are performed to rule-out regressions. The task requires 1-2 weeks.

4. **System testing**: The testing team deploys the new software release to the testbed and performs all the required system tests to validate that it runs as specified, and new modules and features correctly interoperate with the rest of the system. In cases of issues, they report back to the responsible developers, and depending on the required effort further development might occur to fix the issue or move the issues for resolution in upcoming releases. This phase requires 2-3 weeks.

5. **System release**: Eventually, the integrator generates all the release artefacts and documents and tags the current version of the software. In addition, a system release review meeting takes place to identify and discuss problems encountered during this release cycle.

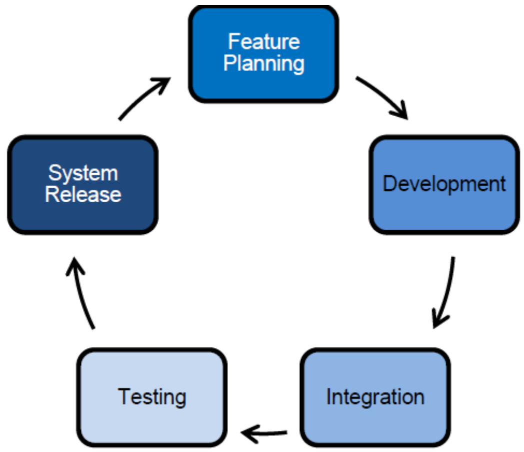



FIGURE 2: SEMIOTICS RELEASE CYCLE

The consortium decided on the following release schedule:

- On M17 we had the first software release, with the basic functionality of the SEMIoTICS NFV Cloud and SSC implemented.
- On M23 the second software release incorporated semantic bootstrapping and NFV Orchestration support, as well as advanced SSC functionalities with a Pattern Engine embedded into the ODL SDN controller.
- On M28 the third release will deliver the end-to-end SEMIoTICS architecture with support for pattern-driven functionality across layers.
- On M32 the final stable release of the SEMIoTICS framework will be delivered.

## 2.3 SEMIoTICS development workflow

SEMIoTICS has adopted the Git Distributed Version Control System (DVCS) for source code and asset management, as well as for monitoring the development process. We rely on a hosted solution from GitLab for the central SEMIoTICS repo, which is located at gitlab.com. We will refer to this repo as the *origin*, which is the standard Git terminology, and all SEMIoTICS partners have permissions to push and pull changes. Furthermore, developers can directly pull changes from other peers to form sub-teams, e.g., to collaboratively work on a new feature which will then be pushed to the to the origin repo.

### 2.3.1 SEMIOTICS GIT BRANCHES

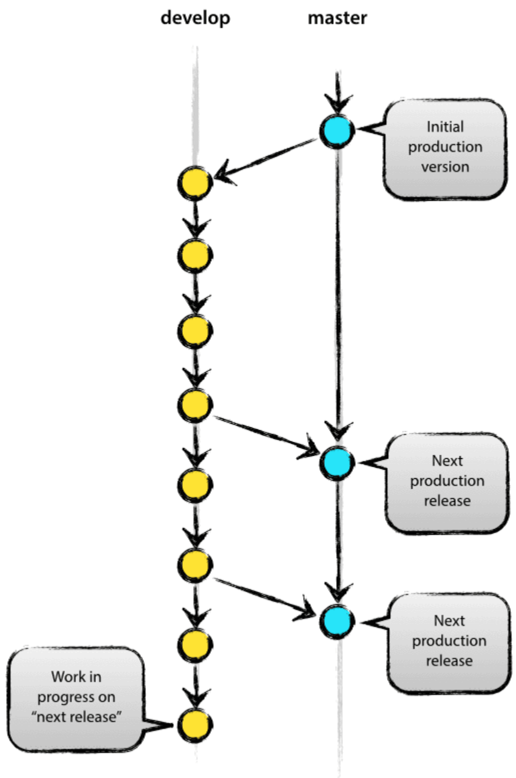



FIGURE 3: SEMIOTICS GIT REPOSITORY BRANCHES

The central SEMIoTICS repository holds two main branches, the *master* branch, and the *develop* branch. The master is generally considered to be the main branch, that reflects the latest stable software release. The master branch integrates all delivered development changes for the next release, so it can also be considered to be the "integration branch". When the source code in the develop branch reaches a stable point and is ready to be released, all the changes are merged back into master and then tagged with a release number.

### 2.3.2 CONTINUOUS INTEGRATION PIPELINE

A CI/CD pipeline is also part of GitLab, in the form of a web application with an API that stores its state in a database. It manages the project builds and provides a Graphical User Interface (GUI) which gives an easy to understand overview of the project development process. Most importantly, the CI pipeline is closely integrated with the core features of GitLab. The Gitlab CI pipeline is part of the SEMIoTICS testing framework with all required unit tests and integration tests. Tests are authored by the respective developers, or a separate testing team. Only if tests pass, the new code is committed to the source code repository.

Furthermore, the system performs nightly builds and in case of build failure notifies the responsible developers. The SEMIoTICS Continuous Integration processes also include the following, which are also accomplished via the GitLab system:

- A ticketing system to assign tasks and feature requests to partners
- A task planning system to assign features to future releases

# 3 SOFTWARE-DEFINED INTEGRATION OF IOT/IIOT DEVICES

This section describes the reference points between the NFV building blocks as well as the interfaces that the NFV exposes to interact with the Middleware, and indirectly with the underlying IoT/IIoT devices. Herein, the ETSI NFV architectural framework (ETSI, 2014a) is considered as a reference. Furthermore, it details the components of the SEMIoTICS SDN controller, which are0 responsible for the network integration and orchestration.

## 3.1 NFV MANO framework

In legacy networks Network Functions (NF), or Physical Network Functions (PNFs) are strictly related to the hardware they operate on. That is, switching, routing, firewalls and other kinds of NF are provided by specialized hardware that contains the appropriate compute, storage and network capabilities each NF uses. NFV decouples NF from hardware, realizing one or many NF as software on top of commercial-off-the-shelf (COTS) devices with sufficient compute, storage and network resources. The move towards NFV promises to provide the dynamicity required to satisfy heterogenous application requirements, but also to take the most advantage out of the infrastructure by satisfying each application's constraint on top of a single, shared hardware infrastructure.

The introduction of VNF is strongly dependent on SDN technologies, which in a similar manner have also achieved the decoupling of functionality from dedicated hardware by ways of separating the control and data planes. SDN is a necessary tool in NFV, mainly for realizing the interconnection of several VNF via virtual network overlays on top of a physical infrastructure. By leveraging SDN and NFV it is possible to interconnect blocks of functionality, i.e. VNFs or PNF, into chains tailored to provide a given Network Service (NS)[2], e.g. enforce security while accessing a Data Base (DB), placing embedded intelligence closer to the sensor/actuator, among others. Such NS are the result of VNF Forwarding Graphs (VNFFG), that when coupled with VTN allow NFV to support many NS to applications with heterogeneous requirements, effectively reducing OPEX/CAPEX relative to legacy networks.

The creation, instantiation, updating, and termination of NS is a new concept in networking, requiring the definition of new reference points (interfaces), functionality and entities. Moreover, the management of existing physical resources for virtualization, assignment of virtual resources to VNFs, lifecycle management of each VNF, and the realization of NS across a distributed set of physical resources impose new challenges to traditional networking. Efforts towards standardization in this regard have yielded ETSI's NFV Infrastructure (NFVI), which include the Virtualized Infrastructure Manager (VIM) and the NFV Orchestrator in the so-called Management and Orchestration (MANO) Framework.

The aforementioned components of the NFVI are to be described in this section, as well as the interaction among them to orchestrate NS and the role they play within the SEMIoTICS framework.

### 3.1.1 VIRTUALIZED INFRASTRUCTURE MANAGER

NFVI defines two Administrative Domains (ETSI, 2014b) namely the Infrastructure and Tenant domains. The former contemplates the physical infrastructure upon which virtualization is performed, and therefore application agnostic; while the latter makes use of virtualized resources to spawn VNFs and create NS. Unlike resource allocation in other virtualized environments, in NFVI requests simultaneously ask for compute, storage and network resources. Moreover, NS could be composed of VNFs with hardware affinity/anti-affinity or require specific latency/bandwidth constrains in virtual links connecting VNFs. Such demands occur dynamically, allocating or freeing resources that could then be used for other NS, e.g. scaling up VNF's compute.

A VIM lies in the Infrastructure Domain. It takes care of abstracting the physical resources of the NFVI and making them available as virtual resources for VNFs. This is achieved through the reference point ***Nf-Vi****,* which interconnects the VIM and NFVI (see Figure 4). It allows the VIM to acknowledge the physical infrastructure (compute, storage) as well as enabling communication with network controllers (SDN Controllers) to provide virtual network resources to NS. Even-though VIMs could well control all resources of the NFVI (compute, storage and network), they could also be specialized in handling only a certain type of NFVI resource (e.g. compute-only, storage-only, network-only) (ETSI, 2014b).

[2] NS could also be composed of a single VNF.

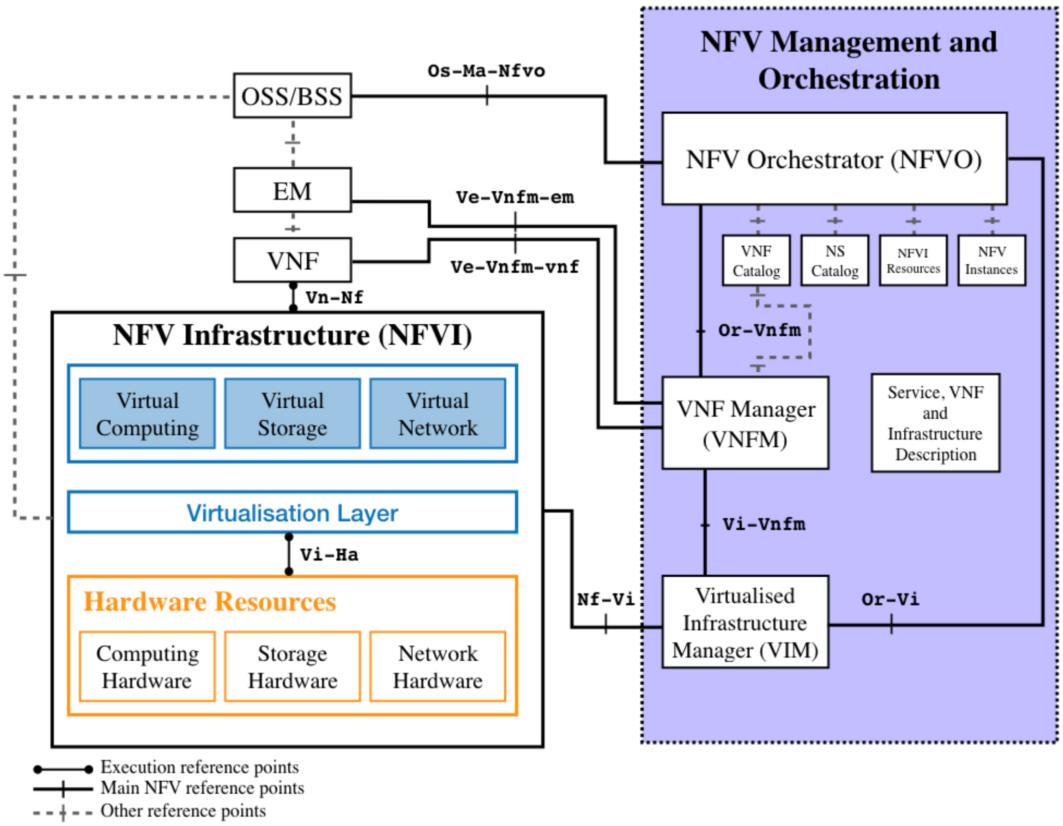



FIGURE 4: NFV REFERENCE ARCHITECTURAL FRAMEWORK

Beyond the already-mentioned functions carried on by the VIM, there are also the following:

- Orchestrate requests made to the NFVI from higher layers (NFVO), e.g. allocation/update/release/reclamation of resources.
- Keep an inventory of allocated virtual resources to physical resources.
- Ensure network/traffic control by maintaining virtual network assets, e.g. virtual links, networks, subnets, ports.
- Management of VNFFG by guaranteeing their compute, storage and network requirements.
- Management and reporting of virtualized resources utilization, capacity, and density (e.g. virtualized to physical resources ratio).
- Management of software resources (such as hypervisors and images), as well as discovery of capabilities of such resources.

As detailed in (ETSI, 2014b) other relevant VIM responsibilities within the NFVI network are:

- Provide "Network as a Service" northbound interface to the NFVO (realized via the ***Or-Vi*** reference point, see Figure 4).
- Abstract the various southbound interfaces (SBI) and network overlays mechanisms exposed by the NFVI network.
- Invoke SBI mechanisms of the underlying NFVI network.
- Establish connectivity by directly configuring forwarding instructions to network VNFs (e.g. vSwitches), or other VNFs not in the domain of an external network controller.

The above compose the network controller part of the VIM. Nevertheless, and as mentioned previously, the required network abstractions mechanisms and management may as well be left to an external network controller, which feeds of NFVI information via the defined reference points (***Nf-Vi***, see Figure 4). It is reasonable to assume the VIM as key part of the NFVI. Being the only NFV component interfacing with the physical infrastructure it exposes open and comprehensible APIs to higher layers, i.e. NFVO, so functions could trigger them to get relevant information from the physical as well as the virtualized infrastructure, and trigger actions upon such information, e.g. create a NS with the necessary resources.

In the SEMIoTICS framework, the physical NFVI is able to support virtualization as realised by the VIM. This allows the NFVO to instantiate VNFs subject to the available compute and storage resources, as well as interconnect such VNFs together via an external SEMIoTICS SDN controller. The following subsections describe relevant Northbound Interfaces (NBI) or APIs usually exposed by VIMs, i.e. OpenStack, which are used by the Resource Orchestration function in the NFVO in order to create the NS satisfying the requirements of the SEMIoTICS use cases (UC).

#### 3.1.1.1 COMPUTE

Compute services at the VIM not only are in charge of creating virtual servers (or containers) on top of physical machines, but also to provision bare metal nodes. In the case of OpenStack this is achieved by means of projects such as Ironic (OpenStack 2018a). The compute API for OpenStack is provided through the project Nova (OpenStack 2018b). It provides "*scalable, on demand, self-service access to compute resources*" through RESTful HTTP endpoints that can be triggered by any authorized entity. All content sent or received from the Compute API endpoints are in JavaScript Object Notation (JSON) format. As it is a text-based type, it allows developers to employ a wide range of tools in order to reach such APIs, easing automation.

The following is a non-exhaustive list of concepts related to the Compute service as well as the information they provide or actions they are able to execute through the corresponding API for SEMIoTICS UC (OpenStack 2018b):

- **Hosts**: physical machines that provide enough resources to spawn a Server. In SEMIoTICS, hosts conform the set of field level, network, and backend devices that together compose the NFVI. For instance, field level devices are assumed to provide enough compute resources to host VNFs realising local smart behaviour. Similarly, network level devices support VNFs for forwarding/routing/firewalling data to and from upper layers; and finally, backend/cloud servers have enough resources to host a wide variety of VNFs, e.g.: SCADA, Web applications and servers.
- **Server**: a virtual machine (VM) instance. In NFV it is often assumed that VNFs reside inside VMs or other type of virtualization container, such as LXC (Canonical, 2018). Some of the server status and actions reachable through the Compute API (OpenStack 2018c):
  - <u>Status:</u> ACTIVE, BUILD, DELETED, ERROR, SHUTOFF, SUSPENDED, among others.
  - <u>Actions:</u> Start/Stop, Reboot, Resize, Pause/Unpause, Suspend/Resume, Snapshot, Delete/Restore, Migrate/Live Migrate, among others.
    - Migration and live migration relate to moving the Server to another Host. Live Migration performs this action without powering off the Server, avoiding downtime.

  The ability to read the current status of Server and modify it, opens the way for dynamic (re)allocation of resources, specifically relevant as performance metrics from the underlying NFVI change in time. For SEMIoTICS this is of paramount importance, as it paves the way to optimize the end-to-end performance of network services in terms of e.g. latency or reliability.
- **Hypervisor**: the piece of computer software that creates and runs VMs. Hosts in each layer of the SEMIoTICS framework run a Hypervisor, which can be queried via the Compute API in order to obtain information regarding the Server, e.g. CPU, memory or other configuration.
- **Flavour**: virtual hardware configuration requested for a given Server, i.e. disk space, memory, vCPUs. Such configurations are onboarded prior to deployment, quantising the scaling factor of Servers e.g.: flavour small (1 vCPU), flavour medium (2 vCPUs), flavour big (4 vCPUs).
- **Image**: a collection of files used to create a Server, i.e. OS images. For SEMIoTICS, each UC component is assumed to run a preconfigured image tailored to its role, i.e. VNF. Such images are uploaded to the VIM for instantiation.
- **Volume**: a block storage device the Compute service could use as a permanent storage for a given Server.

- **Quotas and Limits**: upper bound on the resources a tenant could consume for the creation of Servers. SEMIoTICS employs such functionality to enforce an efficient sharing of the NFVI resources among the different UC.
- **Availability zones**: a grouping of host machines that can be used to control where a new server is created. As different SEMIoTICS UC require the placement of Servers at specific Hosts, this VIM capability allows the NFVO to instantiate VNFs at precisely the right locations in the NFVI.

#### 3.1.1.2 NETWORKING

VIMs are responsible for building virtual network overlays connecting VNFs, but also should expose or relay such information to other components. For instance, if an external network controller is assigned the task of managing connectivity between virtual endpoints, as in the case with the SEMIoTICS SDN Controller, the VIM should expose API endpoints where the necessary network information can be retrieved or modified. Furthermore, in the presence of a NFVO, Network as a Service (NaaS) APIs are expected.

OpenStack Neutron Networking (Denton, 2018) is an SDN controller which is part of the OpenStack networking project and provides the virtual networking resources expected in the SEMIoTICS Backend Cloud infrastructure (or NFVI), such as L2/L3 networking, security, resource management, QoS, virtual private networks (VPN), VTN, among others (OpenStack, 2018d). To configure such functionality or to retrieve logging information, functions are exposed through a set of RESTful HTTP APIs in JSON format. The following shows a non-exhaustive list providing a description of the functionality exposed through the Networking API (as shown in (OpenStack, 2018d):

- **L2 Networking**
    - <u>Networks</u>: list, shows details for, creates, updates and deletes networks. It provides a wide range of extensions capable of configuring several aspects of L2 networking, such as: network availability zones, port security, definition of QoS policies, VLAN trunks, among others.
    - <u>Ports</u>: list, shows details for, creates, updates and deletes ports. Ports are associated with Servers (VMs). They expose a similar set of extensions than the “Networks” mentioned above.
- **L3 Networking**
    - <u>Addresses</u>: list, shows details for, updates and deletes address scopes. Deals with the reservation of IPv4 addresses for Servers (Floating IPs), port forwarding, among others.
    - <u>Routers</u>: when enabled, it allows the forwarding of packets across internal subnets and applying NAT, so they can reach external networks through the appropriate gateway. Routers can be realized in a distributed manner (spanning all compute nodes of the NFVI) or using Router availability zones.
    - <u>Subnets</u>: lists, creates, shows details for, updates, and deletes subnet or subnet pools.
- **Security**
    - <u>Firewall as a Service (FWaaS)</u>: applies firewall rules to ingoing or outgoing traffic, creates and manages an ordered collections of firewall rules.
    - <u>Security groups</u>: lists, creates, shows information for, updates and deletes security groups. Such groups are used to classify types of traffic, allowing or prohibiting certain kind of network traffic through a set of predefined, but also user-defined rules.
    - <u>Virtual Tenant Networks</u> (VTNs). Operators can create multiple private (or Virtual Tenant) networks and can have control over the security policies, IP addresses, monitoring, and QoS.
    - <u>VPN as a Service (VPNaaS)</u>: enables tenants to extend their private networks across the public network infrastructure. Provided functionality includes:
        - Site-to-Site VPN.
        - IPSec using several types of encryption algorithms.
        - Tunnel or transport mode encapsulation.
        - Dead Peer Detection (DPD).
- **Others**
    - QoS bandwidth limiting rules.
        - With the ability to distinguish between egress or ingress traffic.
    - QoS Minimum bandwidth rules.
    - QoS Differentiated Service Code Point (DSCP).
    - Logging resources.

  - DHCP servers.

SEMIoTICS falls within the particular case where the delegation of NFVI networking control is relayed to an external SEMIoTICS SDN Controller. For such cases, Neutron exposes control tools via the Modular Layer 2 (ML2) north-bound plug-in (OpenDaylight 2018). This way, external controllers can manage the network flows traversing the NFVI via southbound interfaces, such as OVSDB.

#### 3.1.1.3 STORAGE

Block storage is common place in virtual environments. Such type of storage can be though similar to USB drives: you can attach one to a compute Server (VM), and then detach it when turning the Server off or destroying it. Particularly interesting is the fact that in a NFVI the storage and compute Hosts are separate. Despite such separation of physical hardware, VMs are exposed to users as if they were running on top of a single Node thanks to the virtual networking resources used by the VIM; allowing the NFVI to grow to massive scales, e.g. server farms.

VIMs such as OpenStack manage block storage through the Cinder project. As concisely put in (OpenStack, 2018e) “It virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device”. A non-exhaustive list of functionalities realised through the Storage API is shown below:

- Create, list, update, or delete volumes.
- Read volumes statuses:
  - Among such statuses are: creating, available, reserved, attaching, detaching, in-use, maintenance, deleting, error, backing-up, among others (OpenStack, OpenStack Docs: Block Storage, n.d.).
- Modify a volume:
  - Extend size, reset statuses, set metadata, attach/detach.
- Management of volumes: create or list volumes.
- Volume snapshots: creates point-in-time copies of the data a volume may contain.
- Volume transfer: transfer a volume from one user to another.
- Backups: full copy of a volume to an external service, as well as the restoration from such backup.
- Snapshots and Group Snapshots.
- Quotas and Limits: per tenant quotas and limits on storage resource allocation.

Compute, Networking and Storage resources are then allocated by the VIM according to requests made through the corresponding APIs. SEMIoTICS UC can be seen as NS, which in turn are the composition of a set of VNF that run within VMs with specific compute and storage resources that are connected in a predefined manner with network resources (SEMIoTICS SDN Controller) known to the VIM. Thereby, the proper allocation of computing, communication and storage resources, to run the chain of VNFs at the corresponding VMs, is fundamental to guarantee the desired performance of SEMIoTICS use cases. Namely, these performance metrics are related to latency or reliability.

All in all, SEMIoTICS UC can be considered complex NS, mostly due to their specific requirements, e.g. Host affinity/anti-affinity (e.g. smart device behaviour, embedded intelligence through patterns, VNFs at specific IoT gateways), specific bandwidth/delay requirements between VNF links, firewalls at the backend/cloud, and/or others. Such specifications are collected in NS descriptors (NSD), which in turn are composed of VNF descriptors (VNFD), and VNFFG descriptors (VNFFGD) that realize SFC according to the specifications contained in their respective descriptors. It is then the task of the NFVO to store/maintain such descriptors and interface with the VIM to realise the NS/VNF/VNFFG therein.

#### 3.1.1.4 TELEMETRY

To facilitate SEMIoTICS’ SPDI properties at NFV-component level, SEMIoTICS’ MANO framework provides a set of endpoints where authorised external entities (e.g. Pattern Orchestrator) may collect telemetry measures from diverse virtualised elements of the architecture, and trigger the orchestration of Network Services (NS) with modified parameters (i.e. modify a specific NS descriptor by pushing additional supported primitives at orchestration time). Figure 5 shows a summary of the NFV component’s API endpoints.

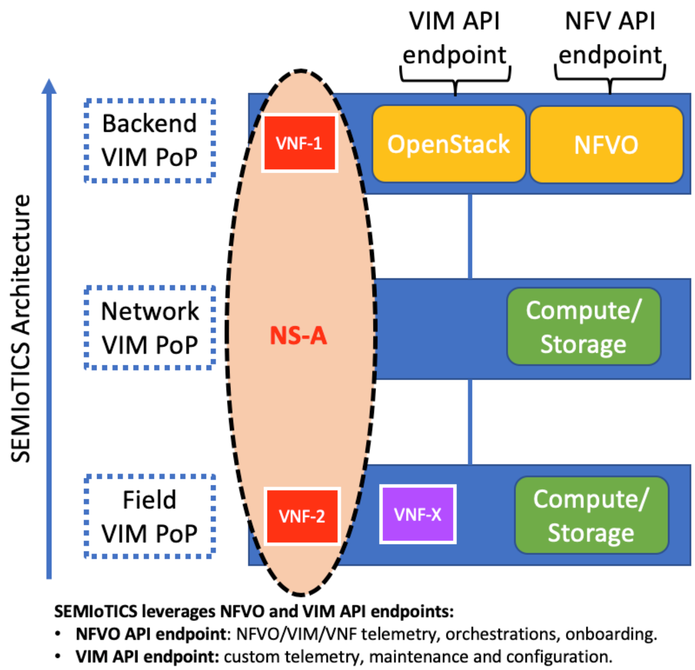



FIGURE 5: SUMMARY OF SEMIOTICS NFV COMPONENT'S API ENDPOINTS

In Figure 5, the VIM API endpoint serves authorised clients with different capabilities and information related to the NFVI. This endpoint is served by OpenStack services' APIs (e.g.: Nova (compute), Neutron (network), Ceilometer (metrics polling engine), Gnocchi (metrics database), etc.). The so-called NFV API endpoint is the one provided by the NFVO. As clarified at the bottom of Figure 5, this endpoint can be used for descriptor onboarding, NS orchestrations, VNF manual scaling out operations, and VNF metrics' collection (as specified in the VNF descriptor).

Metrics and their values are served by the Telemetry service at the Virtualized Infrastructure Manager (VIM), i.e. OpenStack. Such service is split across multiple projects, each one designed to provide a discrete service in the telemetry space (e.g. element polling (metric value retrieval), alarms, storage, etc). In the SEMIoTICS' VIM, the following services are active:

- **Ceilometer:** efficiently collects data (via a polling mechanism) of the OpenStack core components and VNFs (which can be used e.g. for resource tracking). Furthermore, it normalizes and transforms data produced by specified OpenStack services. Ceilometer is not a metric storage solution, but instead it is able to push data to a wide range of so-called *publishers*, which can store telemetry data. Gnocchi is one of such publishers.
- **Gnocchi:** is an externally managed project (non-OpenStack) whose goal is to provide a time-series resource indexing and metric storage. It provides scalable means for storing both short- and long-term data. Administrators may decide how long measures are stored, the reporting period, or other data transformations (e.g. gauge, means, Boolean transformation, etc.) by declaring different archive-policies for metrics.

SEMIoTICS-specific API endpoints are provided by an additional element inside the NFV component. Termed *NFV Proxy*, it serves as middleware between the NFV/VIM API endpoints and other components of the SEMIoTICS architecture. Its goal is to abstract the set of available APIs (e.g. NS orchestration, NS primitive modification, pulling of VIM metrics, manual VNF scale out operations, etc.) into pre-defined procedures tailored to satisfy SEMIoTICS's SPDI requirements. Figure 6 below shows the summary of SEMIoTICS NFV Component's API Endpoints, including with the NFV Proxy.

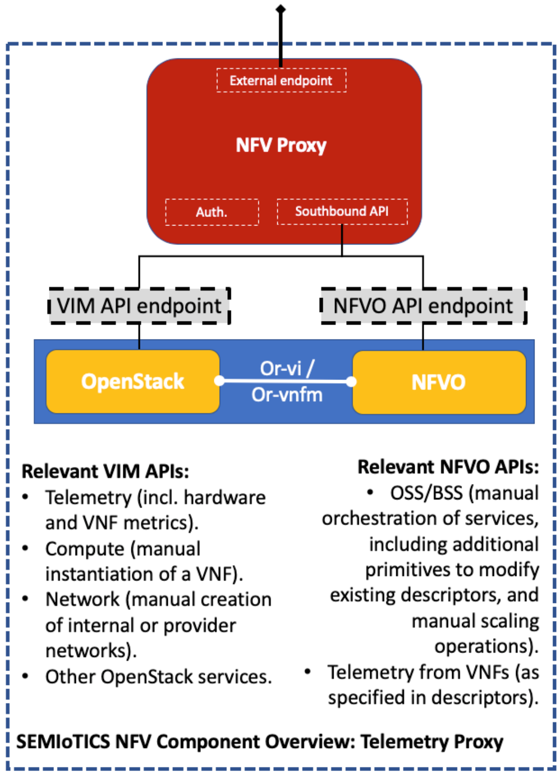



FIGURE 6: NFV PROXY

As can be observed in Figure 6, the NFV Proxy holds an Authentication (Auth.) module responsible for authenticating petitions coming from the External endpoint. Furthermore, authorised parties (e.g. Pattern Orchestrator) may trigger all the APIs provided by VIM and NFVO being redirected towards its Southbound API module (authentication is then performed by the corresponding components, that is, VIM and NFVO). Details on its implementation and functionality will be provided in Deliverable 3.8 as part of Task 3.2.

## 3.2 SEMIoTICS NFV Orchestration

Opensource MANO (OSM), adopted by SEMIoTICS for the implementation of the NFVO, is a project adopted by ETSI, in an initiative to develop an Open Source NFV MANO software stack aligned with ETSI NFV. Two of the key components of the ETSI NFV architectural framework are the NFV Orchestrator and VNF Manager, known as NFV MANO. Additional layers, such as service orchestration are also required for operators to enable true NFV services. Open Source software can facilitate the implementation of an ETSI aligned NFV architecture, provide practical and essential feedback to the ETSI ISG NFV and increase the likelihood of interoperability among NFV implementations. OSM supports descriptor files written in YAML, namely the VNFD and the NSD. The former defines the needed VNF resources in terms of compute resources and logical network connection points, the image that will be launched on the VM, as well as the auto-scale thresholds (e.g., scale-in, scale-out and cooldown period, minimum or maximum number of VNFs) based on the metrics that are being collected from the Telemetry service of the VIM. The latter is responsible for the connection point links, using virtual links, among the interconnected VNFs, mapping them on the physical networks provided by the VIM. In what follows, we detail how ETSI OSM is leveraged in SEMIoTICS to automate the initial VNF placement during Network Service onboarding, as well as to automate VNF scheduling.

### 3.2.1 SERVICE ONBOARDING AND VNF PLACEMENT SUBSYSTEM

In a real-world IIoT infrastructure it is vital to optimize the placement of VNFs also taking into account the networking perspective. This is essential for the efficient deployment of VNFs, such that their network-related QoS constraints are met. However, the OpenStack VIM lacks support for Service Function Chains and is agnostic of their underlying topology, hence placing VNFs individually, and risking the violation of their networking requirements (e.g., saturating hypervisor network interfaces) or QoS constraints (e.g., deploying a latency critical VNF at the cloud tier and not the edge tier). Thus, in SEMIoTICS VNF placement is implemented with an optimized procedure during Network Service (NS) Onboarding. For OSM to *onboard* (i.e., instantiate and configure) a network service (NS), its elements and runtime actions should be specified in the form of descriptors following an agreed-upon information model (IM). Even-though there are tools for generating such descriptors in a user-friendly manner[3], the configuration required for some SEMIoTICS' NS ask for manual configuration.

OSM NS are composed of one or several VNFs. In turn, VNFs are composed of one or several Virtual Deployment Units (VDU). Without loss of generality, one can think of VDUs as virtual machines (VM), requiring an Operating System (OS) image, processor, storage and network connectivity. VNF descriptors, or VNFd, should comply with the correspondent IM[4], that is, the composer of the NS should fill out at least the required elements of the VNFd/NSd in order for OSM to validate it and onboard it. The following Descriptor 1 shows an example of a VNFd in which the VDU will scale-out if a metric called ***cpu_utilization*** surpasses the ***scale-out-threshold*** threshold for ***threshold_time*** seconds. A scale-in operation is also considered, this time the aforementioned metric should be below the ***scale-in-threshold*** during a ***cooldown-time.*** It is also easy to spot the other relevant fields of the VNFd, such as ***image**, **vm-flavor**, **cloud-init-file*** (for day-0 configuration), ***max-instance-count*** (maximum number of VDUs to scale out), etc.

Notice that the VNFd domain is solely the VNF and the composing VDUs. How such elements interact with the rest of the infrastructure (e.g. network configuration) is detailed in the NSd. Descriptor 2 shows an example NSd for the VNFd in Descriptor 1. The relevant fields are highlighted, in short, they include ***vnfd-id-ref*** (link to a VNFd), ***vim-network-name*** (an actual network name defined at the VIM), ***vnfd-connection-point-ref*** (unique connection point identifier per VNF specified at referenced VNFd).

During onboarding by ETSI OSM, where the NS VNFFG is supplied by the NSd, a VNFFG embedding process is performed leveraging Neutron APIs, to assign VNFs to the core or edge tier based on their delay constraints. The Edge hypervisors (or MEC hosts) have a higher operational expenditure (OPEX) than the core tier hypervisors and hence a higher deployment cost which is reflected on a cost function. Thus, typically only a limited number of edges VNFs is deployed at the MEC:

1. The VNFFG embedding process starts from the services with the highest QoS. All VNFs in the VNFFG are traversed breadth-first, starting from the entry point where the UE connects.
2. If the latency constraints of the VNF links exceed the round-trip time to the Cloud tier, the VNF is assigned to the MEC. Otherwise, it is assigned to the Cloud tier.
3. If the MEC resources are exhausted, further deployment of VNFs is blocked, unless they can tolerate the increased latency associated with the Core tier deployment.

Template VNFd and NSd that are leveraged in the above procedure are generated using OSM's DevOps tools[5]. A descriptor verification tool (***validate_descriptor.py***) matches the content of a descriptor and the corresponding IM, highlighting possible errors in the configuration. Once reviewed and fixed, VNFd/NSd are packaged with the ***generate_descriptor_pkg.sh*** utility, and then onboarded to OSM using OSM client[6] or the GUI (see Figure 7).

[3] RFIT VNF Onboarding tool: https://riftio.com/vnf-package-generator/
[4] OSM Information Model: https://osm.etsi.org/wikipub/index.php/OSM_Information_Model
[5] OSM DevOps tools: https://osm.etsi.org/gitweb/?p=osm/devops.git;a=summary
[6] OSM Client: https://osm.etsi.org/wikipub/index.php/OSM_client

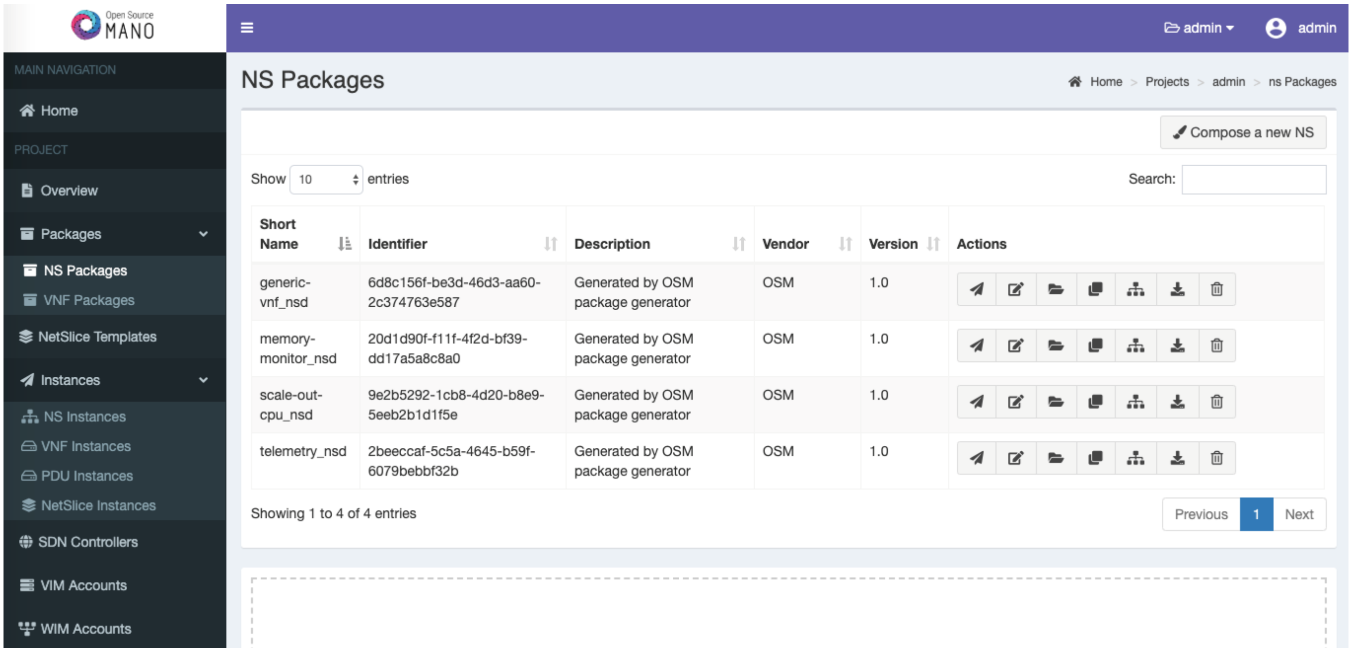



FIGURE 7: OSM ONBOARDED NSD EXAMPLE

```
vnfd:vnfd-catalog:
    vnfd:
    -   id: scale-out-cpu_vnfd
        name: scale-out-cpu_vnfd
        short-name: scale-out-cpu_vnfd
        description: Generated by OSM package generator
        vendor: SEMIoTICS
        version: '1.0'
        mgmt-interface:
            cp: vnf-cp0
        vdu:
        -   id: scale-out-cpu_vnfd-VM
            name: scale-out-cpu_vnfd-VM
            description: scale-out-cpu_vnfd-VM
            count: 1
            vm-flavor:
                vcpu-count: 1
                memory-mb: 1024
                storage-gb: 10
            image: 'ubuntu18-server'
            cloud-init-file: 'telemetry-user-data'
            interface:
            -   name: eth0
                type: EXTERNAL
                virtual-interface:
                    type: PARAVIRT
                external-connection-point-ref: vnf-cp0
            monitoring-param:
                - id: "metric_vdu1_memory"
                  nfvi-metric: "average_memory_utilization"
```

```
                - id: "metric_vdu1_cpu_util"
                  nfvi-metric: "cpu_utilization"
        scaling-group-descriptor:
        -   name: "scale_vdu_autoscale"
            min-instance-count: 0
            max-instance-count: 10
            scaling-policy:
            -   name: "scale_cpu_util_above_threshold"
                scaling-type: "automatic"
                threshold-time: 10
                cooldown-time: 180
                scaling-criteria:
                -   name: "scale_cpu_util_above_threshold"
                    scale-in-threshold: 20
                    scale-in-relational-operation: "LT"
                    scale-out-threshold: 60
                    scale-out-relational-operation: "GT"
                    vnf-monitoring-param-ref: "metric_vim_vnf1_cpu_util"
            vdu:
                -   vdu-id-ref: scale-out-cpu_vnfd-VM
                    count: 1
        monitoring-param:
        -   id: "metric_vim_vnf1_memory"
            name: "metric_vim_vnf1_memory"
            aggregation-type: AVERAGE
            vdu-monitoring-param:
                vdu-ref: "scale-out-cpu_vnfd-VM"
                vdu-monitoring-param-ref: "metric_vdu1_memory"
        -   id: "metric_vim_vnf1_cpu_util"
            name: "metric_vim_vnf1_cpu_util"
            aggregation-type: AVERAGE
            vdu-monitoring-param:
                vdu-ref: "scale-out-cpu_vnfd-VM"
                vdu-monitoring-param-ref: "metric_vdu1_cpu_util"
        connection-point:
        -   name: vnf-cp0
```

DESCRIPTOR 1 VNFD FOR A SCALING OUT VNF

```
nsd:nsd-catalog:
    nsd:
    -   id: scale-out-cpu_nsd
        name: scale-out-cpu_nsd
        short-name: scale-out-cpu_nsd
        description: Generated by OSM package generator
        vendor: SEMIoTICS
        version: '1.0'

        constituent-vnfd:
        -   member-vnf-index: 1
            vnfd-id-ref: scale-out-cpu_vnfd

        vld:
        -   id: scale-out-cpu_nsd_vld0
```

```
name: management
short-name: management
type: ELAN
mgmt-network: 'true'
vim-network-name: 'externalNet'
vnfd-connection-point-ref:
-   member-vnf-index-ref: 1
    vnfd-id-ref: scale-out-cpu_vnfd
    vnfd-connection-point-ref: vnf-cp0
```

DESCRIPTOR 2 NSD FOR A SCALING OUT VNF

### 3.2.2 VNF SCHEDULING SUBSYSTEM

In this section we discuss the role of the NFVO in the VNF lifecycle management, and detail the operation of the VNF scheduling subsystem. Its deployment and performance evaluation at the Testbed is detailed in section 5.3. In order to keep up in with the challenging cloud-native environments, where sub-second reaction times are sometimes required, fast online algorithms are needed. More specifically VNF scheduling is split in three phases, which are centrally controlled by the NFVO:

- The VNFFG embedding phase, detailed in Section 3.2.1, is executed once during service initialization and onboarding, to allocate VNFs to the MEC or Cloud hypervisors based on delay constraints.
- Service scale-out is performed periodically based on a user-defined cooldown period[7] and triggers a scheduling operation for all scaled-out VNFs. A fast online algorithm is devised to handle this operation.
- Service scale-in is also a periodic process, which erases VNF instances when the user demand decreases, to free up resources when they are not needed. We propose a live service migration step to be performed after each scale-in operation to further optimize the VNF placement.

VNF scheduling is based on a cost function, which takes into account the hypervisor resources consumed by the VNF (i.e., CPU, memory and disk size) as well as bandwidth costs to interconnect the VNFs in the VNFFG. These are provided by the Telemetry system, presented in Section 3.1.1.4. In general, the minimum scheduling cost is achieved when all VNFs of the same VNFFG are placed on the same hypervisor. It gradually increases as VNFs are placed on different hypervisors occupying network links for communication, while MEC hypervisors are generally assigned a higher cost than Cloud hypervisors.

VNF scheduling is an online problem, as VNFs are typically scaled-out and scaled-in within very fast time-frames, in the order of seconds, based on current traffic. Although many works solve an offline version of the problem, where the total number of VNFs is known during service bootstrapping, this assumption is not valid in modern cloud infrastructures. Assuming that the VNF assignment to the core or edge tier has been completed during the service bootstrapping phase (i.e., when the VNFs are onboarded at the NFVO and added to its internal database), an online scheduling algorithm will assign the VNF at a Cloud or MEC hypervisor with sufficient compute, memory and networking resources. We have implemented the following algorithm, which tries to first accommodate the highest cost VNFs, starting from the hosts with the highest available resources. The main algorithmic steps of the proposed **Algorithm 1** (see below) for scheduling scaled-out VNFs are explained as follows and they are generally performed after a predefined cooldown period has elapsed. Furthermore, the algorithm tries to accommodate higher priority VNFs via live migration actions of lower priority VNFs, while it tries to restore the balance of the system after a scale-in process.

**Input:**
HMECMax{N}: Total MEC capacity for each MEC hypervisor N
HCloudMax {M}: Total Cloud capacity for each Cloud hypervisor M
VNF{i,{{Type, Resources, Hypervisor}}} where Type in {HP,LP,LT},
Resources in {1..max(HCloudMax, HMECMax)},
Hypervisor in {MEC{N},Cloud{M}}
HMEC{N}: Available resources on MEC 1..N

[7] A cooldown period prevents excessive oscillation

HCloud {M}: Available resources on Cloud 1..M
Sort MEC{N}VNFs descending based on resource allocation VNF{i,2}
Sort Cloud{M} VNFs based on Resource allocation in descending order
Triggering Event e, where e in {scale-in, scale-out}
VNF{e}
**Output:** Hypervisor for VNF placement

```
if e is scale-out of VNF{e}
  if VNF{e, Hypervisor} is MEC{e}
    do
      if available resources on MEC{e}
        allocate VNF{e} on MEC{e}
        update MEC{e} resources
      else if VNF{e,Type} is LP &&
        VNF{e,Resources}<= max(HCloud)
          allocate incoming VNF{e} on max(HCloud)

          update max(HCloud)
      else if LP VNF exists on MEC{e}
        if resources allocated for LP on MEC{e} <= max(HCloud)
          live migrate the first LP MEC{e} on table VNF to
            max(HCloud)and flag it
                                                          update HMEC{e}
           sort VNF table
         end if
    else
        reject scale-out request
                                                      exit algorithm
    end if
    while (LP exist on MEC{e} && VNF{e} is not allocated)
  else if VNF{e, Hypervisor} is Cloud{e}
    if available resources on HCloud{e}
      allocate incoming VNF on Cloud{e}
      update HCloud{e} resources
    else if available resources on max(HCloud)
      allocate incoming VNF on max(HCloud)
      update max(HCloud) resources
    else
      reject scale-out request
                                                      exit algorithm
    end if
else if e is scale-in of VNF{e}
  if VNF{e, Hypervisor} is MEC{e}
    while flagged LP VNFs exist on Cloud &&
      HMEC{e}>=min(flagged VNF resources)
      live migrate the flagged LP VNF with the lowest resources on
        MEC{e}
      update HMEC{e}
    end while
  end if
end if
```

ALGORITHM 1. ONLINE VNF SCALE-OUT/SCALE-IN AND DYNAMIC LIVE-MIGRATION SCHEDULING.

## 3.3 SDN based integration and orchestration

SEMIoTICS SDN Controller is responsible for orchestration of field- and network-level switching devices. We assume an OpenFlow model where SDN Controller computes the network paths used to deploy the forwarding rules for both QoS-constrained and best-effort traffic. The SDN controller does so by parsing the end-points and the service flow requirements (e.g., on bandwidth, delay, fault-tolerance/availability) from the content of pattern specification message provided by the network administrator or higher-layer orchestration element (i.e., the Pattern Orchestrator in the SEMIoTICS architecture).

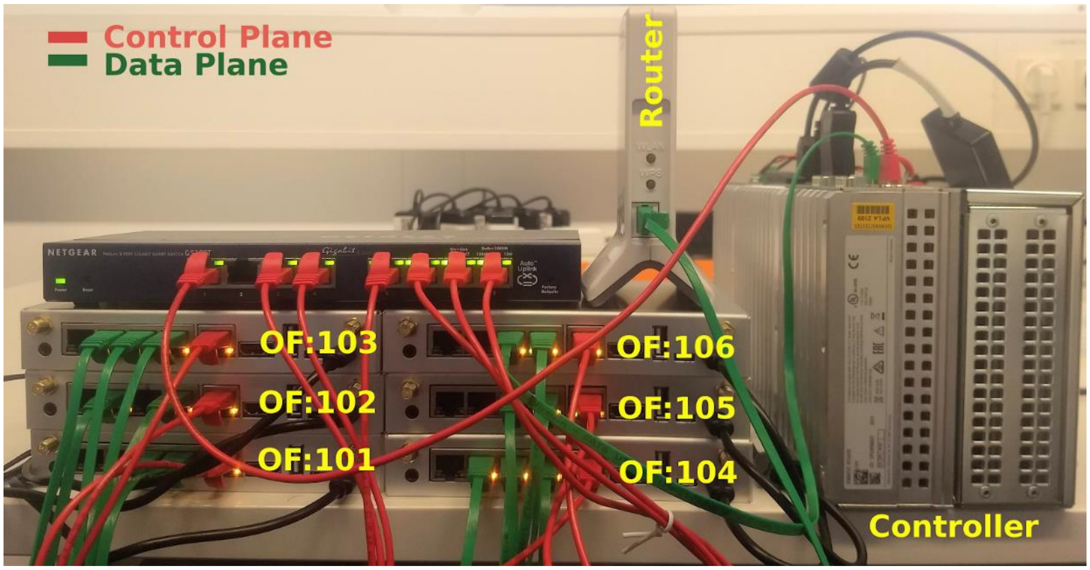



FIGURE 8: INITIAL VERSION OF THE SSC DEMONSTRATOR

The initial release of SEMIoTICS SDN Controller includes all controller components planned and described in D3.5 and D3.1. We omit the detailed description of the components and summarize their implementation level here instead. In specific cases, existing open-source software was modified and extended for the purpose of Use Case / Demonstrator implementations. The current release of the SEMIoTICS SDN Controller was demonstrated in the mid-term review, with the physical setup depicted in Figure 8 above. The per-component details are presented below, with exhaustive algorithmic and design decisions to be contained in D3.7:

- **Network Pattern Engine:** A component for evaluation, monitoring and adaptation of pattern instances (see deliverable D4.1) related to SPDI and QoS properties. In the initial implementation cycles, connectivity patterns providing for liveness of point-to-point network connections, bandwidth guarantees as well as active enforcement of QoS-constrained paths are supported by its Pattern Engine sub-component. Specifically, on acceptance of a QoS-constrained enforcement request, Pattern Engine interacts with the VTN Manager to evaluate the mapping of end-points that are to be connected, to the underlying VTN. If the end-points were, indeed specified in scope of the same VTN, Path Manager proceeds to evaluate the path request and embeds the path. The status of the pattern instance evaluates to True if the path was detected and has been configured successfully in the network, alternatively, the pattern instance evaluates to False. At the backend, the Pattern Orchestrator is correspondingly notified of the result of the implementation.

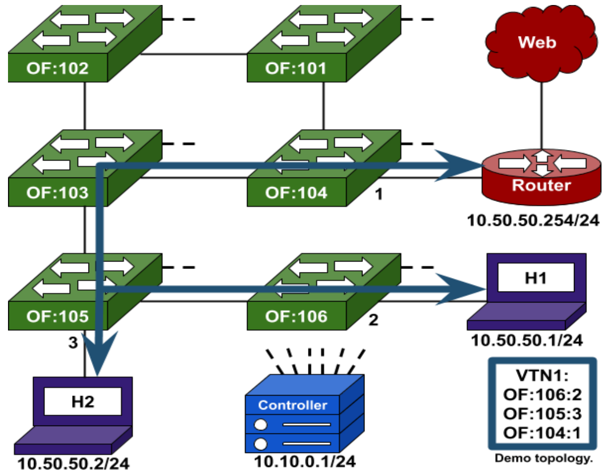



FIGURE 9: AN EXEMPLARY VIRTUAL TENANT NETWORK ENCOMPASSING THREE END-DEVICES

- **VTN Manager:** Responsible for assignment of virtual tenants and their admission in the existing infrastructure during network deployment time. I.e., a network administrator is in charge of specifying the tenants of its network, as well as the set of end-points / network ports behind which that tenant's devices are to be attached. The tenants are isolated and limited to communicating only with the end-points partaking in the same VTN. To this end, at runtime, VTN Manager proceeds with resource assignment for network requests only if the mapping of end-points is compatible with an existing admitted VTN. It ensures a separation of L2 traffic (i.e., ARP request broadcast propagation to ports assigned) in scope of a virtual tenant network. In initial implementation cycles, VTN Manager was adapted to support for enablement of best-effort traffic flows between any end-points connected dynamically during the runtime at network ports specified as end-points during the VTN specification. Thus, the manual effort of explicit pattern instance specifications for each basic infrastructural service can be omitted (e.g., communication between field and backend semantic components). Figure 9 depicts one such exemplary VTN established for the purpose of execution of mid-term demo demonstrator on Programmable SDN Connectivity Layer in IoT. The VTN encompasses three devices, of which one is the router gateway to public internet, used for field devices-backend cloud connectivity.
- **Path Manager:** Main network path computation engine of the SDN Controller, responsible for identification of nodes and ports combined into a path that fulfils the pattern requirements (e.g., on fault-tolerance or bandwidth/delay constraints). The module was unchanged during the initial implementation cycles of SEMIoTICS.
- **Resource Manager:** Provides Path Manager with a resource view of the network (i.e., the available topology resources, port speed, no. of queues metrics etc.). Compared to the existing VirtuWind open-source solution, the module was unchanged during the initial implementation cycles of SEMIoTICS.
- **Security Manager:** The security component of the controller responsible for administration of tenants. The module was unchanged during the initial implementation cycles of SEMIoTICS.
- **SFC Manager:** Used in enforcement of Service Function Chains for overlaying VIM, given the ordering and IP addresses of nodes that are to be traversed by a tenant's traffic. The module was unchanged during the initial implementation cycles of SEMIoTICS.
- **Registry Handler (a component of the Clustering Manager):** Used in state-keeping of other component's knowledge base, as well as for its strong consistent replication across the SDN controller instances for the purpose of fault-tolerance and high-availability. Registry Handler is used without changes in SEMIoTICS. Additionally, Byzantine Fault Tolerant operation for multi-controller decision-making was enabled to allow for tolerating Byzantine faults (e.g., malicious controller decisions or transient/bug issues i.e., due to software aging). Support for BFT was, however, not implemented in OpenDaylight-based SEMIoTICS SDN Controller (SSC), but is evaluated in a separate python-based implementation, due to an extreme effort of related necessary changes in OpenDaylight-based controllers, if BFT support was to be provided there. Additional details on the design and evaluation of the prototypical BFT design will, however, be provided in deliverable D3.7.

- **Bootstrapping Manager:** Used in initial flow configuration of just-connected switches, so to allow for seamless interaction with IoT devices (i.e., to enable flow rules for propagation of unmatched application packets up to the controller for the purposes of ARP-based end-device discovery). Bootstrapping Manager was additionally extended to support for automated establishment of in-band control plane in iterative manner. While the automated in-band functionality was already provided by the OSS VirtuWind controller version, it was extended with a more efficient approach that does not rely on Spanning Tree protocol and thus offers more robust and less complex realization. The design and evaluation aspects of the approach will be contained in upcoming deliverable D3.7.

# 4 SEMANTIC BOOTSTRAPPING AND INTEROPERABILITY

In this section, standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic interoperability are detailed, that are developed in tasks T3.3 and T3.4 respectively. These mechanisms form the basis of the semantic bootstrapping and Interoperability framework, i.e., a significant part of the SEMIoTICS field-level middleware.

## 4.1 Semantic bootstrapping and interoperability framework

As detailed in Deliverable D3.3, SEMIoTICS provides standardized semantic models for IIoT applications, that form the basis of the semantic bootstrapping and interoperability framework. These models harmonize data models from existing automation systems and integrate them with standard IIoT information models. In this context, deliverable D3.3 provides semantics that aims to make field devices interoperable with new IoT devices. Second, it helps to expose capabilities of field devices in a uniform manner by an IIoT gateway. Semantics at this level is thus a key enabler for bootstrapping and easier integration of devices in an IIoT system, as well as a facilitator for creation of new applications. Current automation systems are fully integrated vertical systems. They are efficient, but inflexible. Once engineered and operational, they cannot be changed easily. For example, it is not straightforward to plug a new device into a running system and expect to be functional with respect to an already engineered system. Or it is not effortless to develop an added value service for an existing automation system. In both cases the reason is a know-how contained by experts, but not explicitly represented in machine-interpretable form.

In order to enable creation of new IIoT applications we need to explicitly represent this knowledge, thereby expressing capabilities of field devices in machine-interpretable form. The following use case describes problems found in the current vertically integrated automation systems and sketches the role of semantics in IIoT in order to amend these problems.

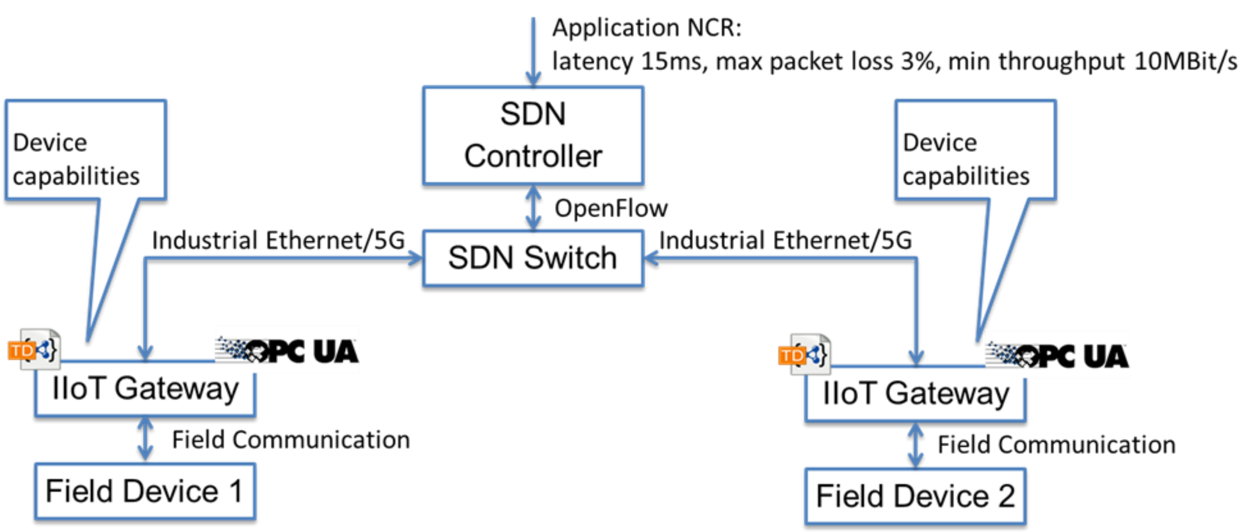



FIGURE 10: SEMANTIC-BASED ENGINEERING AND NETWORKING

Figure 10 depicts an example industrial application, which processes data from Field Device 1 and Field Device 2. In addition, the application imposes certain QoS requirements, which are here expressed as a network constraint rule (NCR). Based on this example application we will explain the role of semantics for interfacing SEMIoTICS field level devices (as the scope of Deliverable 3.3). Let us suppose that Field Device 1 and Field Device 2 are heterogeneous in terms of protocol they communicate, and data they exchange. In order to enable an application to process data from these two devices, we first need to enable a common application protocol. Second, we need to provide a common data model. Finally, we need to provide a common semantic model, which will describe interaction patterns and capabilities of device. Only then, it will be possible for an application developer to discover field devices based on their capabilities they provide, and to put them into a semantically-correct interaction. Further, this enables the developer, as well as machines to understand the data that is produced or consumed by devices. It allows semantic validation of this data or automatically match-make the devices capabilities with the requirements of an application. All these features are useful when a new device is plugged into an existing IIoT system and needs to support an old or a new application, or a malfunctioning device needs to be replaced with the new device etc.

There are two approaches that are most prominent: the first is based on W3C Web of Things Thing Description[8]; he second is based on a prominent industrial standard OPC-UA[9]. In the scope of the first version of Deliverable 3.3 our focus is on the first approach.

In general, the mission of W3C Web of Thing (WoT) is to counter the fragmentation of the IoT. That is, device from different ecosystems become interoperable under a common application layer, provided by WoTs. This should be achieved similar to Web, which has provided a unified application layer to Internet. To this end, W3C WoT standardization group has identified four building blocks:

- First, the Thing Description (TD) describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity.
- Second, the accompanying Protocol Binding Templates[10] enable a TD to be adapted to the specific protocol usage across the different standards.
- Third, the Scripting API[11] describes a programming interface representing the WoT Interface that allows scripts run on a Thing. These scripts can be used to discover and consume other Things (via their TDs), and to expose Things characterized by their capabilities (WoT Interaction Patterns).
- Finally, Security and Privacy Considerations[12] is the fourth building block, which provides guidance for the design and deployment of a secure WoT system.

In the scope of the work in SEMIoTICS, we mainly focus on the first building block, while the second and the third building blocks will be used in our implementation as well.

The WoT TD can be considered as the “index.html” page for Things. It contains semantic metadata describing the Thing itself (e.g. name, location, application context, and software and hardware versions); the offered interface in the form of interaction patterns (i.e., Properties, Actions, and Events); the data model used in messages; and relations to other Things expressed through annotated Web Links [RFC8288]. In the following, we provide a short description of TD basic interaction patterns.

TD Properties expose internal state of a Thing that can be directly read or (optionally) written. Typical examples of Properties are configuration parameters, sensor readings, and set-points that control actuators through Thing-internal logic (e.g., a set-point for the temperature of a thermostat). TD Properties may also be observable. In this case they push the new state to registered subscribers, following best effort mechanisms (e.g. CoAP Observe).

TD Actions enable invocation of Thing’s functions. These functions manipulate the internal state of Thing in a way different from setting Properties. Examples are changing internal state that is hidden, i.e., not exposed as a Property; changing multiple Properties with a single Action; or changing long-running processes (i.e., time is needed to complete the process, and a Property can be used to check the process, e.g., check the state or cancel it during the execution). Actions interaction pattern can also be used to abstract RPC-like calls of existing platforms.

TD Events are raised in order to notify state changes, alarms or streams of values that are sent asynchronously to the subscriber. Unlike Properties, which can be called, TD Events are pushed to subscribers. Events may be triggered as result of conditioned state changes in a Thing. Events are different from observable Properties in that their data cannot be accessed at any time, but only when a notification is emitted by the Thing.

The TD with its presented interaction model is typically enriched with external semantic models (ontologies). TD imports additional Linked Data vocabularies in order to give semantic meaning to its constructs. For example, a TD may have a Property. In order to specify what is the type of that Property, what data it produces,

[8] https://w3c.github.io/wot-thing-description/
[9] https://opcfoundation.org/about/opc-technologies/opc-ua/
[10] https://www.w3.org/TR/wot-binding-templates/
[11] https://www.w3.org/TR/wot-scripting-api/
[12] https://www.w3.org/TR/wot-security/

in which range the data is, what is the measurement unit, what Thing's capability this Property belongs, and so forth, we use external semantic models. A common semantic model to be used with TD is iot.schema.org.

iot.schema.org is an extension of well-known schema.org that is used to annotate Web pages. iot.schema.org provides similar concept for annotations of IoT Things. iot.schema.org features three levels for semantic annotations: Capabilities, Interactions, and Data. A Capability represents a Thing's trait. It usually consists of a set of Interactions. Interactions are semantically aligned to Interaction Patterns from W3C WoT TD. Finally, Data specifies all information about the data that a Thing provides or consumes via its Interactions.

## 4.2 Network Level Semantic Interoperability framework

The SEMIoTICS framework facilitates the deployment of network services and provide seamless connectivity with all its layers and IoT applications, as is the aim of Task 3.4. To achieve that, the project employs SPDI pattern-driven mechanisms that guarantees network level semantic interoperability for various components of SEMIoTICS. Specifically, the following considerations are made (for a more detailed list, please refer to deliverable D3.4):

1. Regarding the interfacing of IT & Cloud infrastructures, to support Nf-Vi, Os-Ma-Nfvo and interfaces for NS management, the NFV reference architectural framework along with the Nf-Vi, and Os-Ma-Nfvo points;
2. Regarding the IoT Platforms, to support Publish/Subscribe Context Broker, Context Producer and Context Consumer by defining and ensuring communication between them, via a different platform (i.e. FIWARE);
3. Regarding the network level of SEMIoTICS itself, to support the different needs of the 3 major use cases.
4. Finally, regarding IoT applications, to support flows between multiple IoT applications, distributed on multiple devices (e.g. between applications of a wind turbine)

Additional considerations must also be identified and guaranteed by the Pattern Engine to facilitate complex interactions, of the above components such as:

- **Cross-Platform:** This covers applications or services access resources from multiple platforms though common interfaces. Further, it includes different instances of SEMIoTICS platform and/or SEMIoTICS to 3rd party IoT platforms (e.g. FIWARE, MindSphere), enabling an application deployed on one platform (e.g., an IIoT wind turbine status monitoring application aggregating information from pertinent sensors) to collect data from other platforms that process related data.
- **Cross-Layer:** This includes communication between entities that are deployed at different-non-adjacent layers of the SEMIoTICS framework, such as cloud to edge or application to network.
- **Cross-Application:** This includes communication between applications or services with applications of different domains or verticals. Such a communication means that an application could potentially gather data about environmental conditions and traffic, to propose the least polluted routes to patients with breathing issues.
- **Higher-level services:** These services, are enabled by exposed interfaces, to orchestrate existing deployments, applications, and the associated services, to provide value-added services, such as providing wind turbine failure predictions or energy demand predictions (to fine-tune energy output) from data aggregated across associated services, enabling effective predictions even for stakeholders/deployments that do not have the breadth of historical data or computational capabilities to extract this knowledge. (e.g. provide specific services to third party entities).

To support the above, two basic properties have been ensured across the deployment that also affected the design of then networking interfaces:

- **Platform-scale independence**, allowing the integration of resources from platforms at different scale. More specifically: at the Cloud/IoT backend level, platforms can host high volumes of data from a vast number of devices; field-level deployments (e.g., fog) interact with nearby devices in the field and maintain information in a constraint spatial scope; device level platforms (e.g. at the IoT gateway level) have direct communication with the things, managing small amounts of data. In this context, in the SEMIoTICS framework an application should be able to uniformly aggregate information for the different scale platforms (e.g. collect wind turbine status values for a specific area via cloud or minimally processed data via a platform at field).

- **Platform independence**, allowing the integration of distinct platforms that implement the same functionality, like an IIoT wind turbine status monitoring in different wind parks. The platforms may utilize different equipment and techniques to monitor the wind turbines (e.g. legacy wired sensors attached to smart gateway or newer wireless sensors); a single application at the backend should be able to interface with all instances in a uniform manner without requiring any changes.

The vision of such a heterogeneous and flexible deployment is sketched in Figure 11.

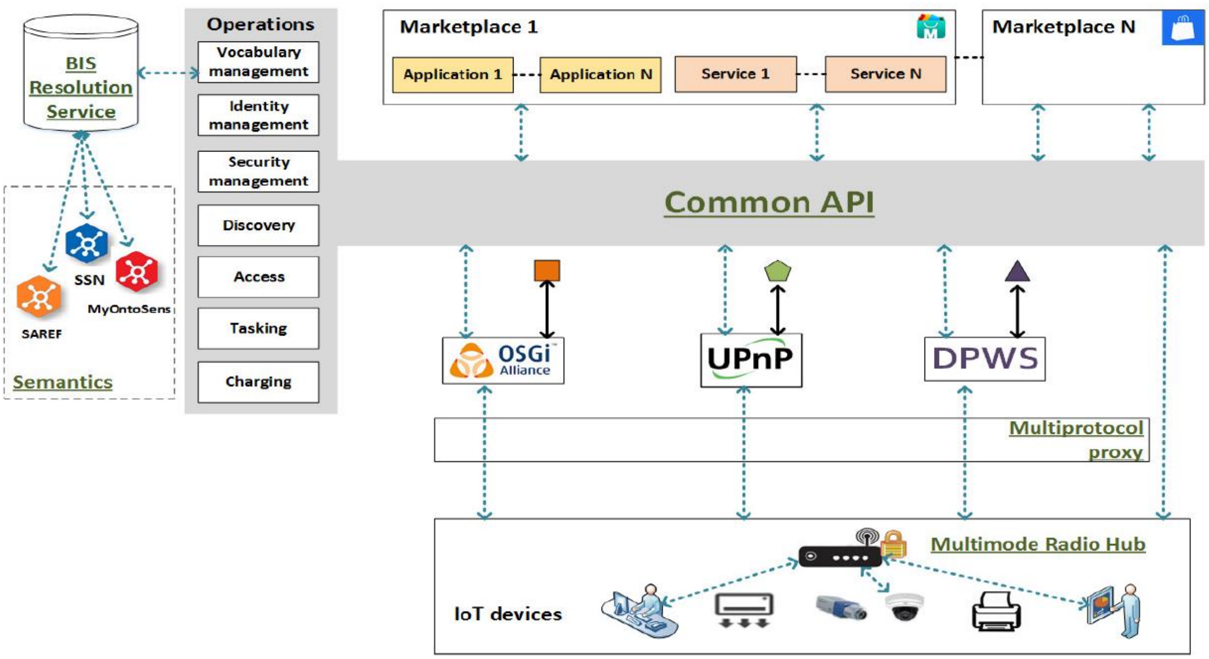



FIGURE 11: SEMIOTICS FRAMEWORK, INTEROPERABILITY ACROSS ALL 4 LEVELS

In all of the above cases, the ability to specify the desired Security, Property, Dependability and Interoperability properties should be provided, in order to additionally enable the SEMIoTICS SPDI pattern-driven approach that is at the core of the framework. The related considerations that are described in detail in D3.4 ("Network-level Semantic Interoperability"), whereby a network interface is specified for the SEMIoTICS SDN controllers (exposed through the Pattern Engine module integrated into the controller), allowing the definition of such requirements through the purpose-defined SEMIoTICS pattern language. The design and specification of said language is detailed in D4.1 ("SEMIoTICS SPDI Patterns (first draft)").

Through this pattern-driven approach, enabled via the deployment of Pattern Engines at the network (but also its interaction with Pattern Engines deployed at the field and backend layers), **IT & Cloud infrastructures** can leverage the pattern language to define properties for the essential connection between the NFV Management and Orchestration components and the NFV Infrastructure. Additionally, the Pattern Engine facilitates the communication with the North Bound Interface via the Os-Ma-Nfvo endpoint.

Considering the **IoT Platforms,** Network interoperability with other platforms was considered in the design phase of the Pattern Engine as it is an essential part of SEMIoTICS. Current activities in the context of Task 3.4 focus on ensuring that the network-level semantics of the Pattern Engine are compatible with different IoT frameworks/platforms (e.g. FIWARE), their Context Brokers and Producers (e.g. sensors) and Consumers (e.g. a context-based application).

With regard to **network-level interfacing with SEMIoTICS,** the pattern language defines and enforces (via the Pattern Engine) mechanisms that guarantee the establishment of E2E connectivity between different types of devices (e.g. SARA hubs, sensors, backend servers), actors (e.g. human operators, applications) and interaction types (e.g. maintenance, medical staff, simple user/patient), while monitoring that certain SPDI (and QoS) properties are maintained.

Additionally, the Pattern Engine enables various more complex interactions such as cross platform (e.g. cloud apps <-> private cloud), cross layer interactions (e.g. field devices <->backend), cross application (e.g. SDN controller <-> remote management service) or interactions with higher level services (e.g. Third-party entities). To support this functionality, interoperability mechanisms (e.g., the semantic brokers already present in the SEMIoTICS framework) need to ensure that all the devices support the required protocols (e.g. MQTT, HTTP, etc.) for bootstrapping, discovery and registration operations and that they fulfil these actions also in different layers of the framework. (e.g. cloud interfacing with the IoT sensing gateway).

As to the **IoT applications,** the pattern language defines and guarantees via the Pattern Engine the communication between various IoT devices through their interfaces. The interaction with edge devices is further assured. Finally, using pattern-based operations SEMIoTICS translates high-level application QoS constraints to network-level QoS constraints.

For more information on the pattern-driven NBI in the context of the end-to-end semantic interoperability provided by SEMIoTICS please refer to deliverable D3.4, while more details will be presented in D4.4 ("Semantic Interoperability Mechanisms for IoT (first draft)", as part of the Task 4.4 ("End-to-End Semantic Interoperability) efforts, where said interface is a key enabler.

### 4.2.1 IMPLEMENTATION ASPECTS

#### 4.2.1.1 ADOPTED TECHNOLOGIES

In the context of implementing the network-level semantic interoperability, an identification and description of key enabling technologies that further advance the interoperability of SEMIoTICS, such as network protocols and data formats that can be used for the communication from field devices to the backend cloud, was carried out. As a result, the following established protocols have been adopted:

- **Hypertext Transfer Protocol (HTTP) – Representational State Transfer (REST):**
  - HTTP/1.1. (i.e. the most commonly accepted version of this protocol) is the fundamental client-server protocol used for the Web.
  - REST is a distinguished architecture style used for developing of web services. With the rapid success of IoT the combination of HTTP & REST offers very easy ways to create, read, update and delete data, making it essential for SEMIoTICS.
- **Yet Another Next Generation (YANG)** is a data modeling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF), NETCONF remote procedure calls, and NETCONF notifications. A YANG module defines a hierarchy of data that can be used for NETCONF-based operations, including configuration, state data, Remote Procedure Calls (RPCs), and notifications. This allows a complete description of all data sent between a NETCONF client and server.
- **Advanced Message Queuing Protocol (AMQP),** is an open standard protocol following the publish-subscribe paradigm, aimed to offer interoperability between a large diverse set of applications and systems, regardless of their internal designs.
- **Constrained Application Protocol (CoAP),** is designed by the Constrained RESTful Environments (CoRE) with recent versions using a like publish-subscribe approach, to provide HTTP REST capabilities for constrained devices with limited processing resources, such as IoT devices.
- **and Message Queuing Telemetry Transport (MQTT),** is another protocol that follows the publish-subscribe paradigm. It is especially efficient and lightweight, designed for constrained devices and non-optimal connectivity conditions, such as low bandwidth and high latency.

Employed data formats involve:

- **Extensible Markup Language (XML),** is markup language made for encoding data in a format that is both human-readable and machine-readable.
- **JavaScript Object Notation (JSON),** is a lightweight open-standard file format based on a portion of JavaScript, made to transmit data objects using human-readable text (that can be easily parsed and produced by a machine).
- **SensorThings,** is an Open Geospatial Consortium (OGC) standard that provides an open and unified framework used by IoT sensing devices, data, and applications to communicate over the Web.

#### 4.2.1.2 NETWORK INTERFACING

To facilitate the use of the SPDI-driven network services, the Pattern Engine embedded into the ODL SDN controller (see Figure 12) exposes a rich REST-based interface which devices, services and applications across layers can consume.

As per SEMIoTICS architecture definition, the majority of interactions at the SDN Controller's exposed NBI are consumed by the overarching Pattern Orchestrator. Indeed, Pattern Orchestrator leverages the REST-based northbound interface of the controller to describe the pattern requirements initiated at the higher-layer recipe definition. In the initial SDN controller development, the Pattern Schema describing the structure of the networking-related pattern, is intentionally be kept open and extensible to support the most diverse types of connectivity-related patterns possible.

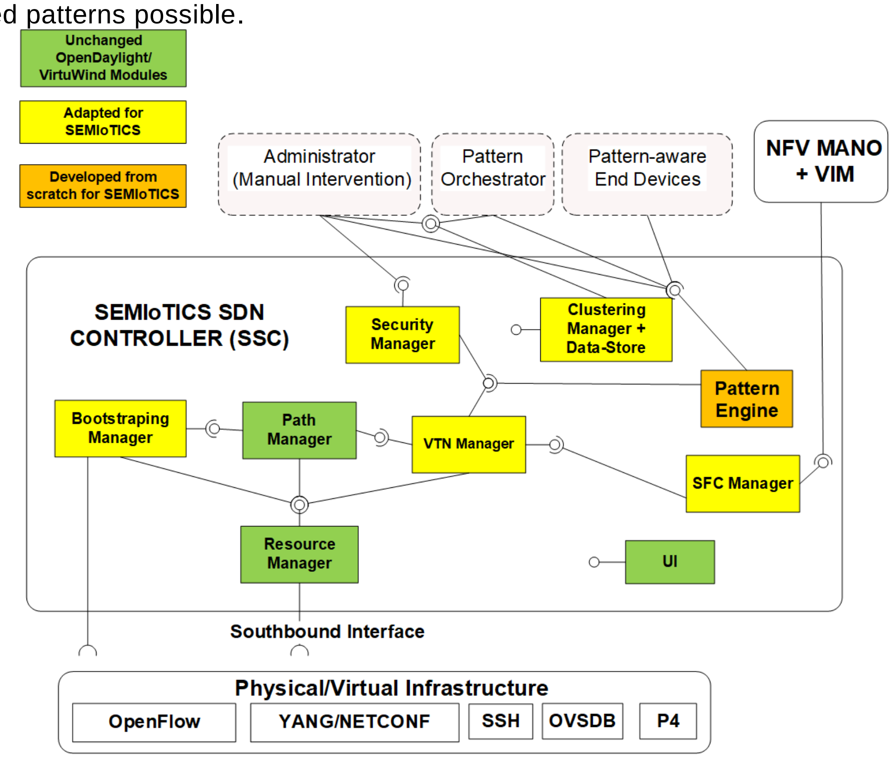



FIGURE 12: THE PATTERN-DRIVEN NBI (IN ORANGE) WITHIN THE SEMIOTICS SDN CONTROLLER'S PATTERN MODULE

This Northbound interface exposed by the Network Pattern Engine is specified as follows:

```
rpc insertRule {
                input {
                       leaf RuleName {
                           type string;
                           description "Rule Name";
                       }
                       leaf Rule {
                           type string;
                           description "Add a new Rule or update an existing";
                       }

                      }
                 output {
                         leaf RuleResponse {
```

```
                            type string;
                        }

                    }
                }
rpc removeRule {
                input{
                       leaf Name {
                           type string;
                     description "Remove an existing Rule";
                      }

                }
                 output{
                        leaf DeletionResponse {
                                                            type string;
                         }

                        }
                }
rpc getRule {
             input{
                   leaf Name {
                                              type string;
                       description "Retrieve an existing Rule by its name";
                    }
                   }
              output{
                     leaf ruleName {
                                          type string;
                     }
                     leaf rule {
                                                 type string;
                     }
                    }
             }
rpc getRuleStatus {
                   input{
                         leaf RuleName {
                             type string;
                             description "Get the status of an existing Rule";
                          }
                    }
                    output{
                           leaf RuleResponse{
                               type boolean;
                           }
                          }
                    }
rpc addFact {
             input{
                   leaf recipe_id{
                       type string;
```

```
                    }
                     leaf fact_id{
                              type string;
                          }
                          leaf fact_from{
                              type string;
                          }
                          leaf fact_message{
                              type string;
                          }
                          leaf fact_type{
                              type string;
                          }

                      }
              output{
                          leaf factResponse{
                              type string;
                          }

                      }
                  }

 rpc factUpdate {
               input{
                      leaf recipe_id{
                          type string;
                          }
                      leaf fact_id{
                          type string;
                          }
                          leaf fact_from{
                              type string;
                          }
                          leaf fact_message{
                              type string;
                          }
                          leaf fact_type{
                              type string;
                          }

                      }
                      output{
                          leaf factResponse{
                              type string;
                          }

                      }
                  }

                  rpc factStatus {
                      input{
                             leaf recipe_id{
```

```
                                        type string;
                                    }
                                    leaf fact_id{
                                        type string;
                                    }
                                    leaf fact_from{
                                        type string;
                                    }
                                    leaf fact_message{
                                        type string;
                                    }
                                    leaf fact_type{
                                        type string;
                                    }

                            }
                            output{
                                leaf factStatusResponse{
                                   type boolean;
                                   }
                                leaf factMessageResponse{
                                                  type string;
                                                  }
                            }
                        }
rpc factRemove {
                            input{
                                leaf recipe_id{
                                               type string;
                                }
                                leaf fact_id{
                                                type string;
                                }

                                leaf fact_type{
                                               type string;
                                 }

                            }
                            output{
                                leaf factResponse{
                                    type string;
                                }

                            }
                        }
```

Additional implementation details about the semantics of the patterns and the specification of network-level properties can be found in deliverables D3.4 and D4.1, while their updates will be provided in D3.10 ("Network-level Semantic Interoperability (final)") and D4.8 ("SEMIoTICS SPDI Patterns (final)"), i.e. the respective final versions of said deliverables.

### 4.2.2 USE CASE EXAMPLE

A demonstration scenario that relies on the SEMIoTICS pattern-driven network interface and its capabilities was designed and developed around Use Case 1, i.e. industrial IoT environments, and more specifically oil leakage detection in wind turbines through video monitoring. This was also demonstrated during the Mid-Term Review. The overarching aim of the scenario is to distribute a complex application (composed of multiple tasks) to a network of IoT/Edge device and specify constraints (through patterns) on the network / orchestration. In this context, the developed scenario also leverages user-friendly design and deployment of IoT orchestrations through a custom-built, distributed version of Node-RED[13]. The two key research innovation of the scenario and associated demonstration relate to: 1) True distribution of application flows over multiple devices and representing the network perspective in Node-RED, and; 2) Automated enforcement of network / orchestration constraints by defining them as SEMIoTICS patterns.

In terms of the actual setup, it involves transmission of video between two Raspberry Pi credit-card sized embedded devices (from “piA” to “piB”), coordinated by Node-RED running on a Nanobox (industrial PC), while monitoring of QoS constraints with patterns. This setup is depicted in Figure 13.

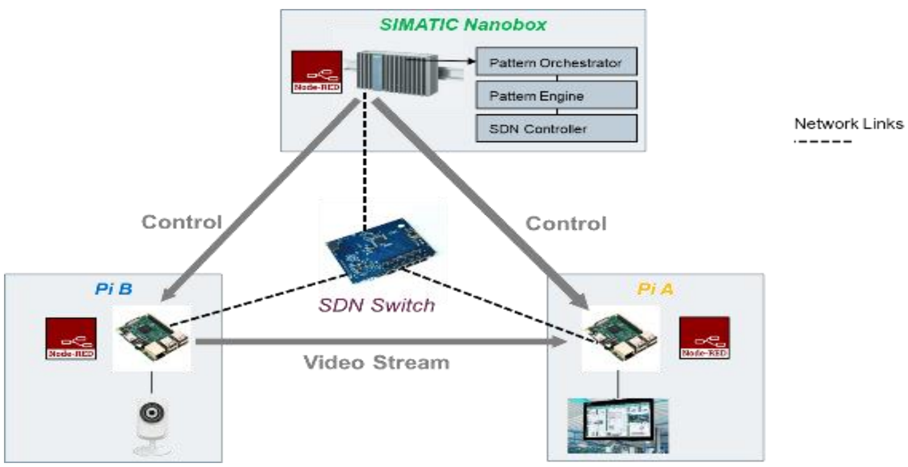



FIGURE 13: PATTERN-ENABLED IOT ORCHESTRATIONS LEVERAGING THE PATTERN-DRIVEN NETWORK INTERFACE

In the above, other than the user-friendly, graphical interface and distributed nature of defining the IoT orchestrations involved (including where / on which devices parts of a flow are deployed), we also want to define SPDI and QOS between these deployments (see Figure 14 and Figure 15).

[13] https://nodered.org/

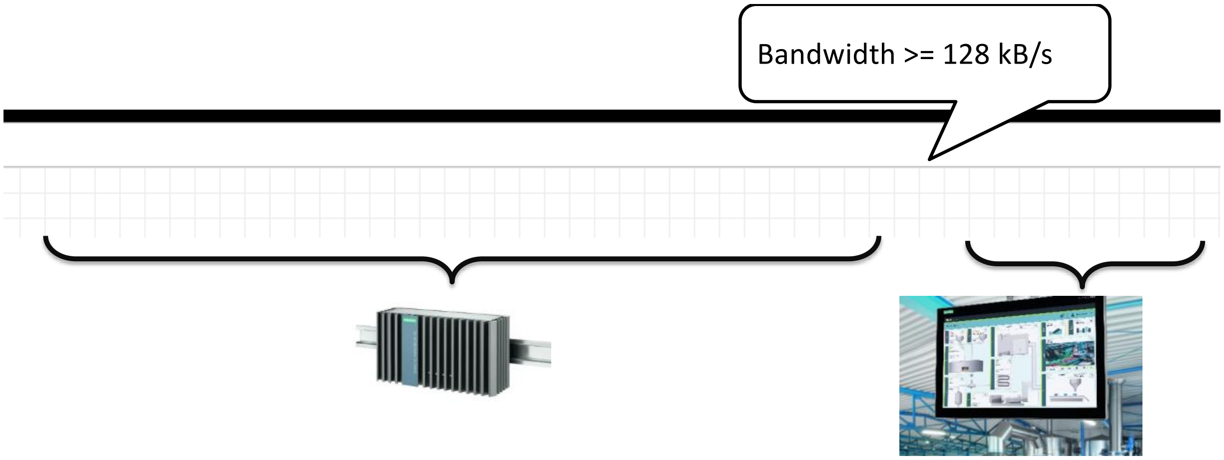



FIGURE 14. GRAPHICAL IOT ORCHESTRATION DEFINITION

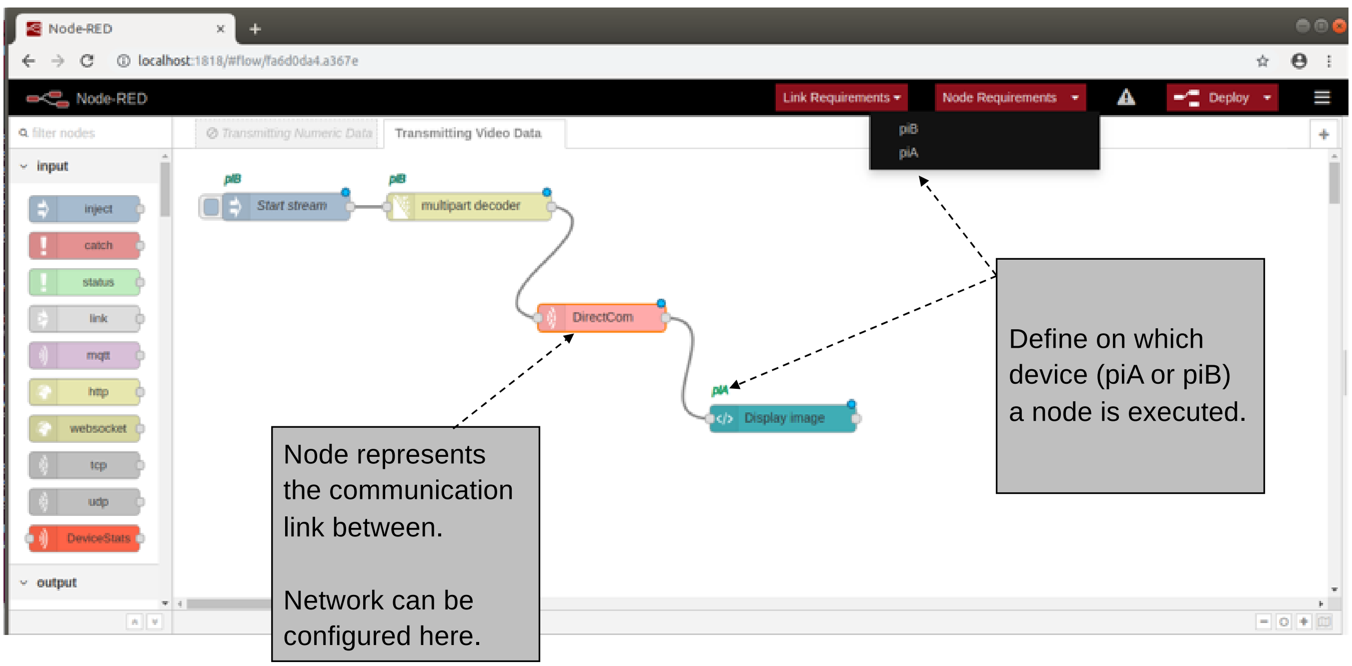



FIGURE 15: THE CUSTOMISED NODE-RED GUI AND SCENARIO ORCHESTRATION DEFINITION

Focusing on the network aspects, while maintaining the high level abstractions needed for user-friendliness, a “Network Link” node enables direct communication between distributed Node-RED instances. Said “Network Link” node enables definition of QoS constraints (e.g., minimum bandwidth, latency) and the whole orchestration specification (a “Recipe”) and the QoS constraints are translated into the SEMIoTICS pattern language and sent to Pattern Orchestrator. From the latter, the information is relayed to the network (SDN) Pattern Engine. A high-level view of this process is shown in Figure 16.

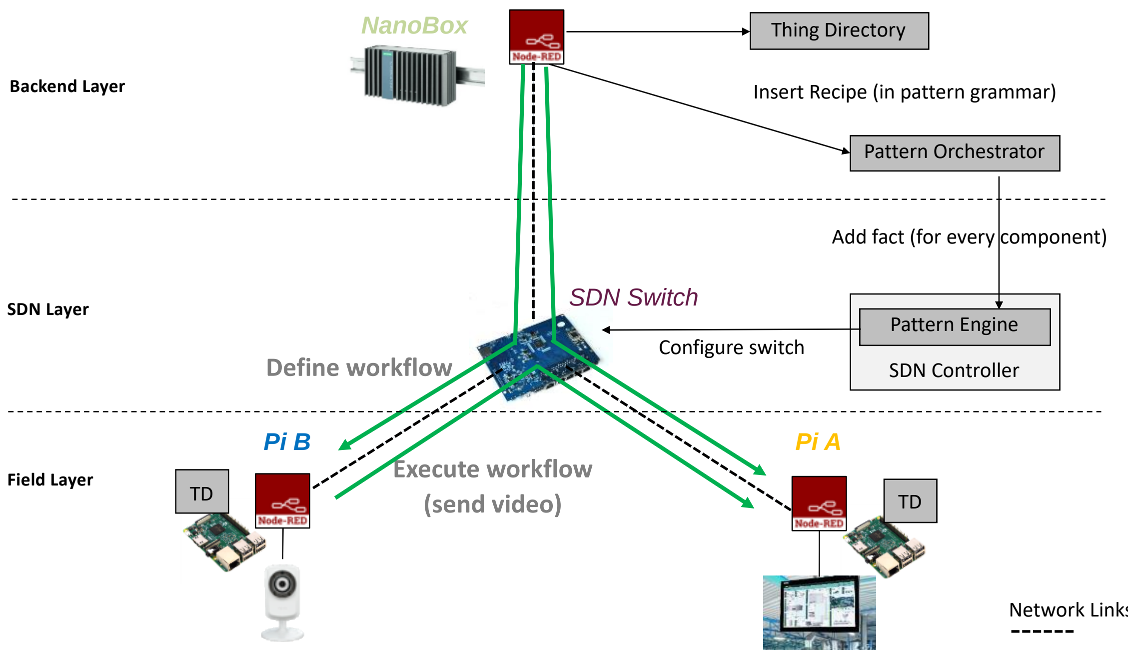



FIGURE 16: HIGH LEVEL VIEW OF SCENARIO IMPLEMENTATION SEQUENCE AND INVOLVED COMPONENTS

## 4.3 Integration of Brownfield devices

During the bootstrapping process a new device is registered and integrated in the SEMIoTICS platform. Moreover, the functionality of the device is semantically described and made discoverable, a scenario that was demonstrated during the Mid-term review. The device is also exposed throughout a common interface. All these steps are handled by SEMIoTICS IoT Gateway, and known as the process of *semantics-based bootstrapping and interfacing of field devices*. Figure 17 depicts this process graphically. The figure distinguishes this process for brown-field devices (left-hand side) and green-field devices (right-hand side). Green-field devices are devices that already have a Web-based RESTful interface, and are described by W3C Thing Description (TD). The brown-field class of devices comprise of all other devices that yet need to be made accessible over a Web-based RESTful interface and described by TD.

As Figure 17 shows, the focus until Mid-Term Review was on the right-hand side. This part has been implemented. In a demonstrated scenario a new device (an IP camera) has been plugged into a network where SEMIoTICS IoT Gateway operates. The gateway scans the network, discovers the new device, stores its TD in the Knowledge Repository (Local Thing Description Directory in Figure **18**) and provide a common interface for the device. So bootstrapped device is than ready to be used for new applications.

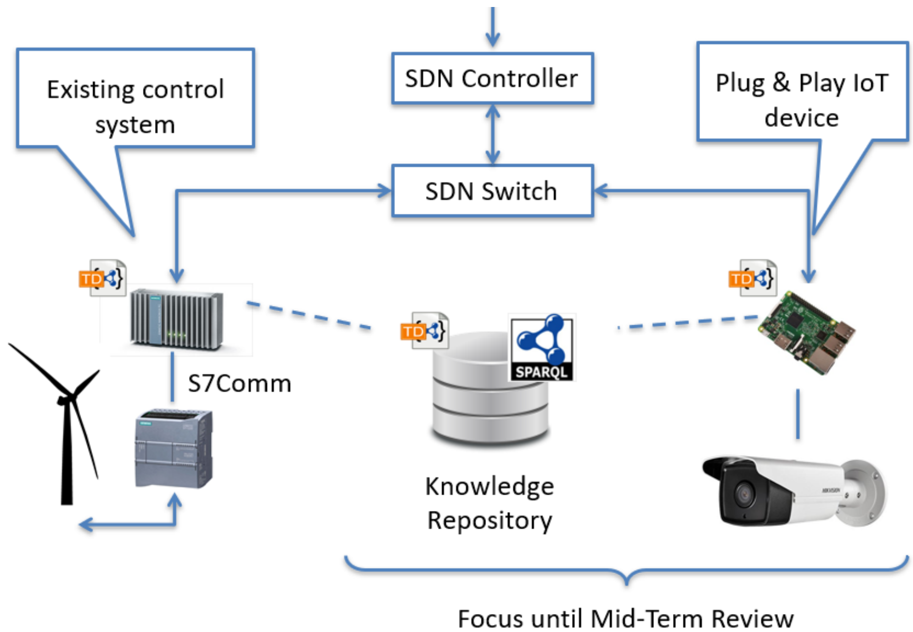



FIGURE 17: SEMANTICS-BASED BOOTSTRAPPING AND INTERFACING OF FIELD DEVICES

Figure **18** shows SEMIoTICS IoT Gateway with all its components. In comparison to the previous version of this gateway (as reported in Deliverable D3.3), Figure **18** introduces one additional component "Semantic Edge Platform".

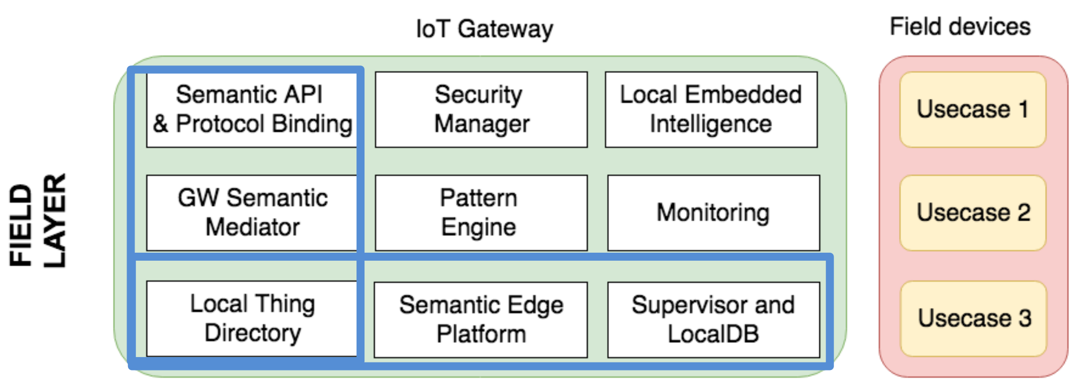



FIGURE 18: SEMIOTICS IOT GATEWAY

Semantic Edge Platform (SME), shown in Figure 19, has multiple purposes in the SEMIoTICS architecture. It provides a convenient user interface for configuring SEMIoTICS IoT Gateway. Further, SME enables a convenient development environment for creating new Apps with a newly bootstrapped device. Finally, it provides a mechanism to semantically annotate brownfield devices. The implementation of SME is based on Node-RED[14] tool.

On the right-hand side of Figure 19 we have extended Node-RED tool so that it enables the bootstrapping process in SEMIoTICS. In this implementation, one can define an IP network range, which the gateway will use when scanning for new devices. Further on, a device that is supposed to be bootstrapped can be selected and the process of bootstrapping can be initiated. Once the process if completed, nodes that can be used to interact with the device will appear on the left-hand side of the tool, see Figure 19. Each node represents one function of the device (an interaction pattern in the device's interface). Such nodes represent a convenient way

[14] https://nodered.org/

to interface a device, and to be used later on in Recipe-based applications. Figure 19 shows a test flow for a newly bootstrapped camera. Similar flows can be instantiated from Recipes in order to speed up the process of creating new applications.

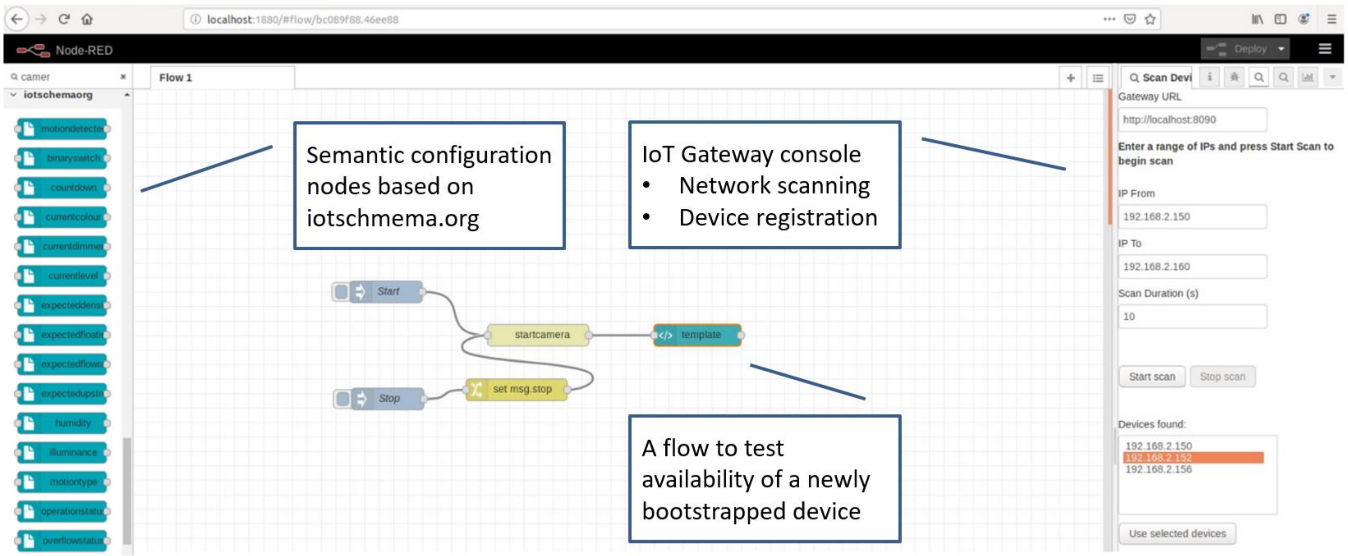



**FIGURE 19: SEMANTIC EDGE PLATFORM: DEVICE- SCANNING, CONFIGURATION, AND EXPOSURE FOR APPS**

Figure 20 depicts Local Thing Directory, which runs in SEMIoTICS IoT Gateway. As already mentioned, the gateway stores a Thing Description of each bootstrapped device in this directory. The directory provides a sematic query interface for discovering all present devices. Using this interface, it is possible to search for devices that have certain capabilities. It is assumed that both Thing Descriptions and semantic queries are annotated with iotschema.org[15].

[15] http://iotschema.org/docs/full.html

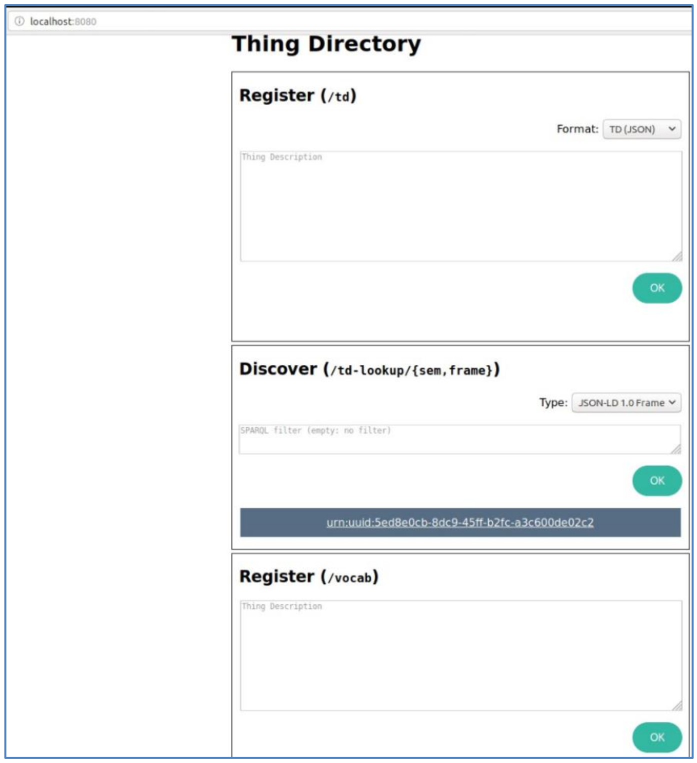



FIGURE 20: DEVICE DISCOVERY VIA SEMANTIC SEARCH IN LOCAL THING DESCRIPTION DIRECTORY

So far, in this section we have described the current state of the implementation of SEMIoTICS IoT Gateway and the process of bootstrapping a device. After the Mid-Term project review, we are continuing with the implementation related to the brown-field integration, see Figure 17.

# 5 DEPLOYMENT AND EVALUATION OF THE MIDDLEWARE AT THE SEMIOTICS TESTBED

## 5.1 SEMIoTICS Integration testbed

In this section we present the architecture and physical infrastructure of the SEMIoTICS integration testbed, which is used for the deployment and validation of the SEMIoTICS Middleware and its frameworks (see Section 2.1). These are deployed in the SDN/NFV Orchestration layer, detailed in Section 5.1.1, and the Field layer, detailed in 5.1.2, which are addressed by WP3 in the SEMIoTICS architecture. Use-cases will build their own testbeds and showcase more advanced scenarios, using a subset of the Middleware (Section 2.1.2). In what follows, the term **"NFV Cloud"** (or just Cloud) is used to refer to the virtualized infrastructure (or NFVI) which includes the VIM, as well as the Cloud and Edge hypervisors. The latter, also termed as MEC hosts, are typically deployed in closer proximity to the Field layer as shown in Figure 21 to minimize latency, following the MEC paradigm[16]. A preliminary version of this testbed was demonstrated at the EUCNC 2018 exhibition and its upgraded version at EuCNC 2019. The physical infrastructure of the SEMIoTICS integration testbed currently includes the following hardware components and is constantly upgraded:

- One 6-core 64-bit server with 32 GB RAM hosts the OpenStack Controller and Network services, related to Management, Orchestration and SDN control.
- One 4-core 64-bit server with 32 GB RAM hosts the ETSI OSM NFVO management services.
- Two 6-core 64-bit servers with 32 GB RAM act as the Compute Nodes, or Cloud hypervisors, that host all IIoT services and VNFs in dedicated Virtual Machines (VMs).
- One 4-core 64-bit server with 8 GB RAM acts as a resource constrained MEC node that hosts Edge VNFs.
- One Odroid C2 Single-Board Computer (SBCs) acts as the Field layer Virtualized IoT gateway. An 802.15.4 radio module is employed to interconnect Field devices (smart sensors) with the gateway.
- Field layer smart sensors that transmit temperature, humidity, light intensity and vibration values wirelessly over 802.15.4 and BLE. Smart Light actuators are also used for demonstration purposes.
- SDN access switches are employed at the Network layer, to implement the SDN Data plane.

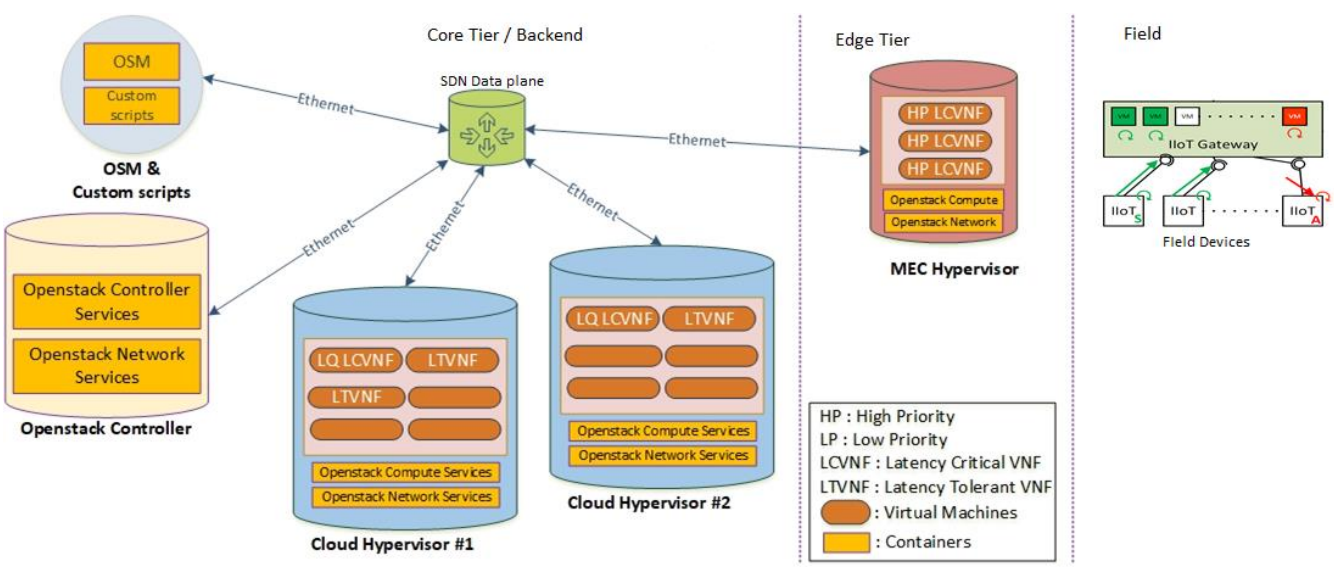



FIGURE 21: SEMIOTICS TESTBED PHYSICAL INFRASTRUCTURE

### 5.1.1 SDN/NFV ORCHESTRATION LAYER

[16] https://www.etsi.org/technologies/multi-access-edge-computing

The SEMIoTICS integration testbed leverages the OpenStack ecosystem (see section 3.1.2) as well as an ETSI MANO stack. OpenStack is a complex software framework with multiple components that handle security and authentication, VM image storage, VM instantiation and termination, etc. In our testbed, a Controller node hosts all OpenStack services in Linux Containers. Linux Containers (LXD) is an emerging virtualization solution which allows services to run almost to the "bare metal" with minimal performance penalties, but with the requirement that they share the same kernel with the host (in this case the Controller node). The following OpenStack services are deployed in our Controller:

- **Glance** stores the VNF (or VM) images in its local filesystem
- **Keystone** acts as the identity service, keeping track of OpenStack users and their respective permissions (e.g., admin, user, etc.)
- **MySQL** stores configuration options in a master database
- **Neutron** is the OpenStack networking layer, which handles connectivity among VMs and applications. It is responsible for deploying end-to-end slices and virtual networks among VNFs that can physically reside in different physical servers
- **openstack-dashboard** implements the OpenStack Horizon GUI which allows us to manage our network and VMs with an easy to use GUI.
- **Nova** is the OpenStack hypervisor service. OpenStack Nova employs KVM (i.e., Kernel-based Virtual Machine) technology to natively execute multiple VMs at a host operating system.
- **RabbitMQ-server** implements a fast message bus that allows individual OpenStack services to communicate and exchange information.

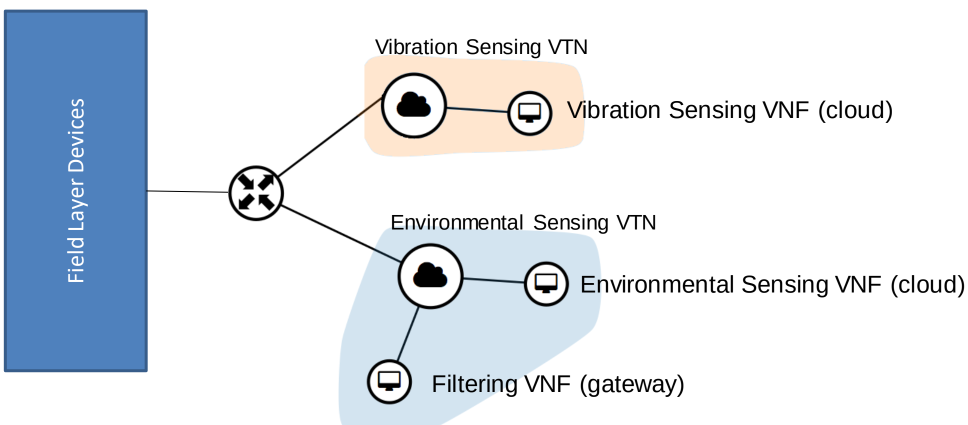



FIGURE 22: IIOT SERVICES AND VIRTUAL TENANT NETWORKS EXAMPLE

IIoT services **related to smart monitoring and actuation are** implemented in the form of VNFs, that are managed by an ETSI compliant MANO stack, which is detailed in Sections 3.1 and 3.2. An example scenario with two such services, each in their own VTN, is shown in Figure 22. The MANO stack, whose central element is ETSI OSM, i.e., the NFVO, handles the automatic deployment and lifecycle management of services, based on performance KPIs from the Telemetry system (detailed in Section 3.1.2.4), without requiring a system administrator's input. Moreover, VNFs can be individually scaled, i.e., multiple instances can be deployed to meet user demand and migrated to a different hypervisor for optimization purposes. For example, to meet service KPIs, a VNF may have to be moved to a hypervisor with a lower CPU load, or higher networking capacity. VNF migration is a relatively complex procedure and care should be taken not to cause downtime. Specifically, there are two modes of operation for VNF migration:

- Legacy mode involves shutting down and then restarting the VM that hosts the VNF in a different hypervisor.
- Live migration mode involves running both instances (in the old and new hypervisor) in parallel while the migration is performed, and only migrating RAM contents as a final step. This mode causes minimal service disruption.

### 5.1.2 FIELD LAYER

Our testbed Field layer includes a virtualized IIoT gateway that interconnects a set of sensors and actuators with the backend cloud. Our IoT gateway supports KVM virtualization, enabling us to push VNFs down to the gateway tier. This allows services with ultra-low latency requirements to be pushed in very close proximity to the IIoT devices, hence minimizing latency. The relatively modest resources available at the gateway, which is implemented with an Odroid 64-bit ARM-based Single-Board Computer (SBC), means that it must be used for a minimum number of VNFs with low processing needs. Furthermore, ARM64 support is still in its early stages at the OpenStack ecosystem and is only reliably supported in a small set of 64-bit SBCs. Hence, significant effort was required to deploy the OpenStack Nova hypervisor to our Odroid C2 SBC:

- We had to compile and install the Open vSwitch Kernel module, which was missing from the Odroid Kernel, after appropriate modifications for compatibility purposes.
- We had to manually modify the OpenStack installation scripts, which failed due to missing features in the ARM platform (e.g., due the lack of a PCI bus)
- We had to remove GRE tunneling support from OpenStack (and restrict it to VLAN and VXLAN tunnels) as the respective GRE Kernel module was missing from the Odroid C2 Kernel
- We had to add a second Ethernet interface via a Gigabit USB-to-Ethernet adapter which serves as the provider network, and is capped to ~300 Mbps as Odroid lacks support for USB 3.

Nevertheless, after the successful deployment of OpenStack Nova at the Odroid C2, it has worked very reliably and allows the virtualization of even low cost IIoT gateways, with the same MANO stack also leveraged by the SEMIoTICS Orchestration layer.

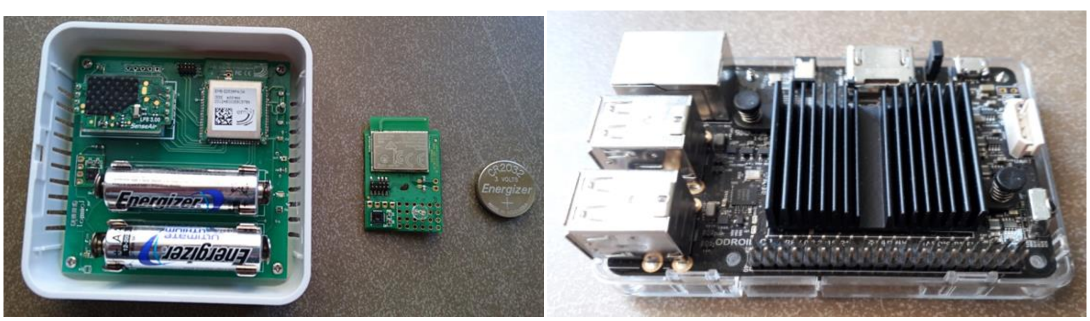


FIGURE 23: ARM-BASED IIOT GATEWAY AND FIELD DEVICES

For the field-layer smart sensors, we employ custom-designed battery operated 802.15.4 and BLE devices that perform periodic measurement of CO2, Temperature, Vibration and Light (Lux) values. Sensor values are encapsulated in IPv6 packets and transmitted to the IIoT gateway via MQTT. The actuators are commercial Philips Hue Smart Lights that are connected to the IIoT gateway via a Hue bridge. The Sensors and Actuators are communicating with the respective VNFs, that are hosted at the Cloud or IIoT gateway hypervisors.

Furthermore, the integration testbed leverages Semantic models, presented in Section 4.1, to annotate data that is exchanged between things, as well as to describe capabilities of things in a machine interpretable format. Our gateway serves as a semantic mediator in the task of integrating semantics of brownfield industrial devices and IoT things, as detailed in Section 4.2. More specifically, at the input, the gateway accepts data from diverse field devices. At the output, it provides an API to access semantically-annotaded data along with descriptions of capabilities of connected devices. The API is based on the W3C WoT upcoming standard, and things are specified in the WoT TD format. TD is semantically annotated with iot.schema.org, as it has been thoroughly described in Deliverable 3.3 and Section 4.1.

```
{
  "@context": [ "http://www.w3.org/ns/td",
               {"iot": "http://iotschema.org/"} ],
  "@type" : [
      "Thing", "iot:LightControl", "iot:BinarySwitchControl"
```

```
    ],
    "id": "urn:dev:wot:lamp",
    "name": "WirelessLamp",
    "description" : "WirelessLamp uses JSON-LD 1.1 serialization",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
      "status" : {
        "@type" : "iot:SwitchStatus",
        "type": "string",
        "forms": [{
           "href": mqtt://192.168.1.11:1883/house/lamp/status,
           "mediaType": "application/json"}]
      }
    },
    "actions": {
       "toggle" : {
       "@type" : "iot:ToggleAction",
       "forms": [{
          "href": mqtt://192.168.1.11:1883/house/lamp/toggle,
          "mediaType": "application/json"}]
       }
    },
    "events":{
      "overheating":{
        "@type" : "iot:TemperatureAlarm",
        "data": {"type": "string"},
        "forms": [{
            "href": "mqtt://192.168.1.11:1883/house/lamp/oh",
            "subprotocol": "longpoll"
        }]
      }
    }
  }
```

Figure 24 THING DESCRIPTION ANNOTATED WITH IOT.SCHEMA.ORG

For verification purposes, in our testbed, we deployed the Smart Light as a Thing that is automatically registered in the database with the reception of an MQTT availability message, as soon as it connects to the network. In detail, a listener at the IIoT gateway receives the availability MQTT message "ON" and retrieves the thing description from the local database, as seen in Figure 24. The result of the discovery is shown in the Thingweb Directory immediately, as seen in Figure 25. Thus, the TD is registered at the thing directory that allows searching for a Thing based on its metadata, properties, actions or events. In Figure 26, we show the JSON format of the TD and the address that it has been given to the thing by the Thingweb directory. Through this platform is also possible to update the TD and even generate a servient based on a discovered thing.

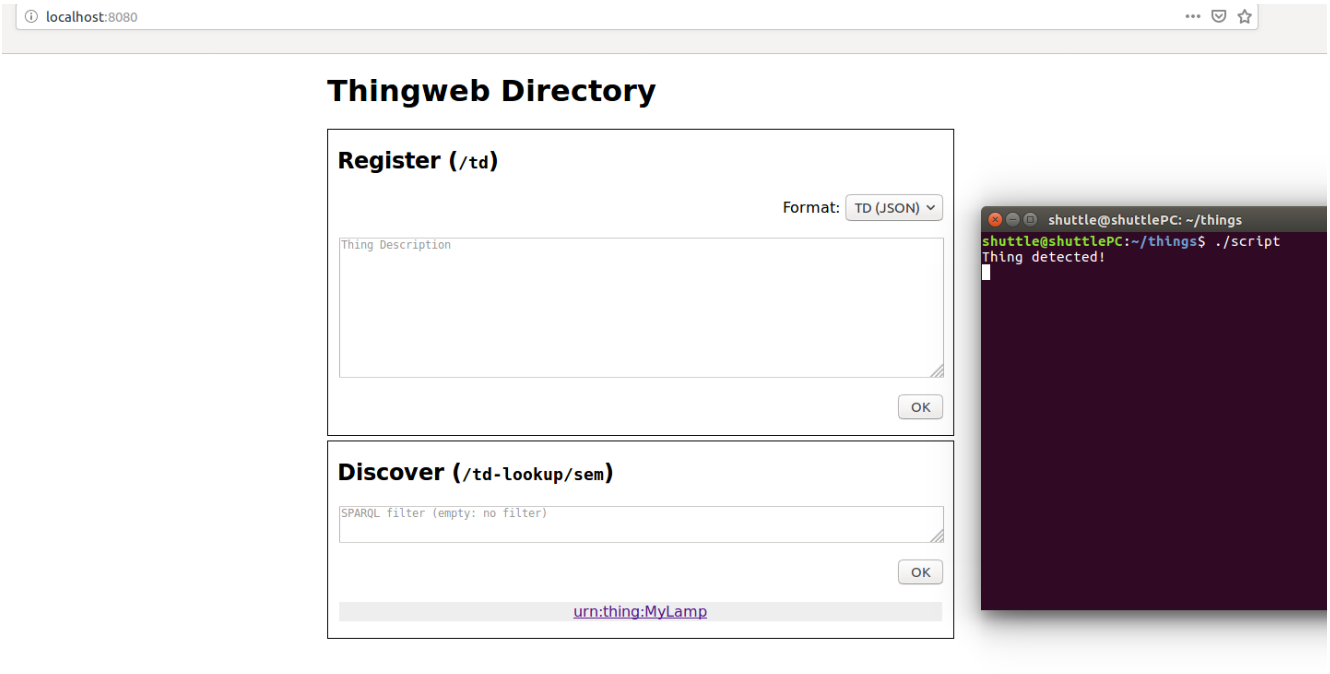



FIGURE 25: THING DISCOVERY

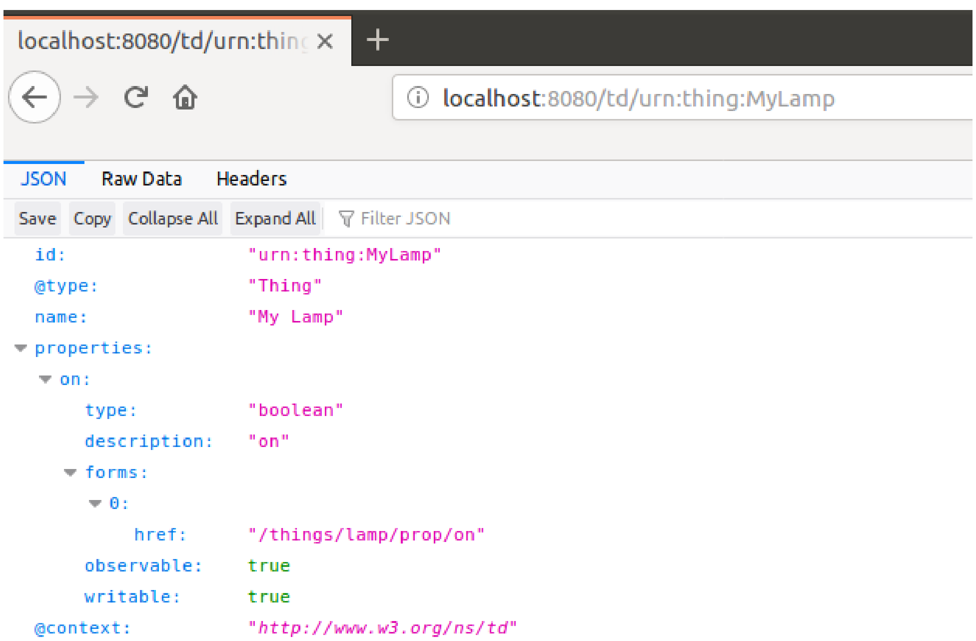



FIGURE 26: TD OF THE WIRELESS SMART LIGHT

### 5.1.3 MID-TERM REVIEW DEMO

Part of the SEMIoTICS testbed was demonstrated during the Mid-Term review, showcasing NFV functionality. In this scenario, a sensing VNF was performing real-time vibration analysis on a mini rack, with an on-board cooling fan. The vibration analysis was performed with a field device using a LIS2DH 3-axis “femto” accelerometer supplied by ST (the same is used in Use Case 3). This scenario emulates a real-world data center with multiple rack-mounted servers and mission critical ventilation systems that have to be monitored in real time, and excess vibration is an indication of impeding malfunction. A second filtering VNF, deployed at the virtualized IoT gateway, was responsible for implementing a moving average filter on the vibration measurement, suppressing noise and therefore compressing the information that had to be transmitted from the gateway to the cloud layer. This demonstrated the capabilities of the SEMIoTICS architecture in relying on local analytics functions to remove some of the burden from the cloud hypervisors, and prevent bottlenecks at the network layer. The following figure shows the testbed setup used during the MTR. Finally, an actuation VNF was responsible for controlling a (virtual) smart light, based on the readings of a field layer light sensor. This VNF was also deployed at the IoT gateway, to benefit from a reduced latency and hence we were able to demonstrate instantaneous reaction of the smart light, as a response to changes to the measured Lux value.

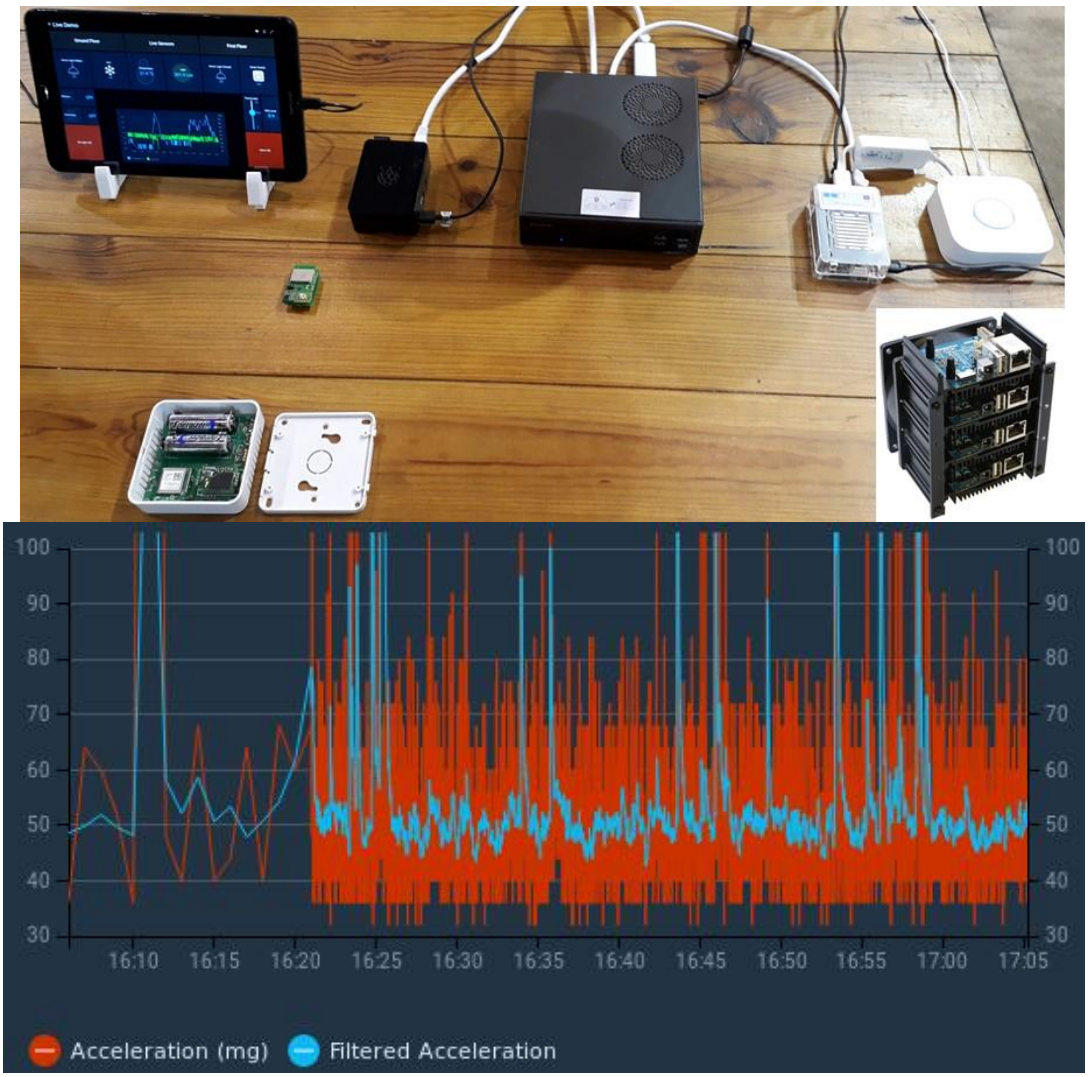



**FIGURE 27: MID-TERM REVIEW DEMO PLATFORM AND MEASUREMENTS**

## 5.2 Slicing implementation and verification

Network slicing (P. Mekikis, 2019) is a typical SDN use case, which involves reserving resources for critical applications (e.g., critical infrastructure monitoring) such that they are offered performance guarantees related to throughput, latency, and packet error rate.

### 5.2.1 SLICING IMPLEMENTATION

The SEMIoTICS reference architecture includes SDN switches at its Network layer, that interconnect Field Layer IIoT gateways. SDN switches in SEMIoTICS are implemented with Open vSwitch (OvS), a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. The OvS switches are controlled by Neutron, which exposes control APIs via the Modular Layer 2 (ML2) north-bound plug-in and supports a wide variety of Layer 2 technologies, including OvS (see Section 3.1.2.2). End-to-end slicing is implemented by leveraging the ML2 API to communicate QoS requirements to the relevant SDN switches that lie at the VTN data path. QoS rules are stored at the OvS database and applied to the OvS switch ports. Specifically, the QoS model supported by OvS, shown in Figure 28, includes three QoS rules, that are used to manage the network ports' priority queues, and implement traffic shaping:

- DSCP marking of packets for traffic prioritization
- Bandwidth limits to prevent interface saturation
- Minimum bandwidth guarantees for bandwidth reservation

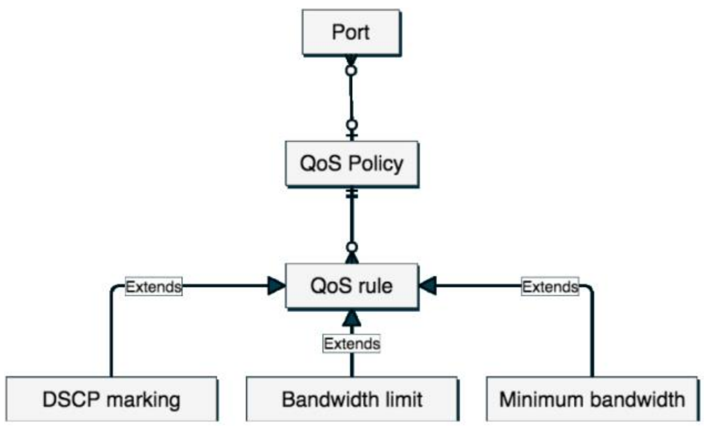



FIGURE 28: NETWORKING LAYER QOS

From the 3 QoS policies supported, bandwidth guarantee is the most critical for Industrial IoT networks that often need strict delay and throughput assurances (e.g., for infrastructure monitoring and smart actuation use cases). Furthermore, it is important to ensure that hypervisor interfaces and not just OvS interfaces are not over-subscribed and saturated when new VNFs are deployed. This is currently ensured by an additional Verification and Live Migration step. Overall, service deployment involves the following steps:

1. A VNFD file is supplied to the VNF Manager with service metadata and requirements.
2. The VNF Manager instantiates the VNF, which is automatically placed at a Cloud hypervisor.
3. A verification step checks if the hypervisor interface was over-subscribed
4. If the verification fails, perform a Live Migration of the VNF to a cloud hypervisor with sufficient networking resources and go to step 3.
5. An end-to-end slice is deployed based on service requirements leveraging Neutron APIs.

### 5.2.2 SLICING VERIFICATION AND EXPERIMENTAL RESULTS

In this section, the testbed is evaluated in terms of its ability to guarantee bandwidth reservations in Tenant Networks with slicing, as well as the effectiveness of Live Migration in optimizing VM placement. Finally, the suitability of a virtualized IIoT gateway, which is capable of hosting VNFs, for industrial and haptic applications is also evaluated. In all our experiments, the traffic was generated with the D-ITG traffic generator which can generate TCP traffic with various profiles, e.g., Pareto, Exponential, etc., as well as write trace files. Moreover, a Smart Sensing and an Actuation VNF were deployed, each in a dedicated Tenant Network, that compete for testbed resources.

#### 5.2.2.1 TENANT NETWORK SLICING

In this experiment, we measured the maximum throughput that could be sustained between the two VNFs, both hosted at the Cloud hypervisors, and a client device which was connected at the Field layer. At first, the link capacity, which is 1 Gbps, is equally shared by the two VNFs, as shown in Figure 29. At time t=11s the Neutron API is employed to setup an end-to-end Network Slice for VNF2, with a dedicated throughput of 700 Mbps. Figure 29 shows that the measured throughput of both VNFs changes instantaneously to 700 Mbps for VNF2 and 300 Mbps for VNF1. This was achieved with successful bandwidth reservation at the hypervisor network interface, as well as at the SDN switch output port where the client device is connected.

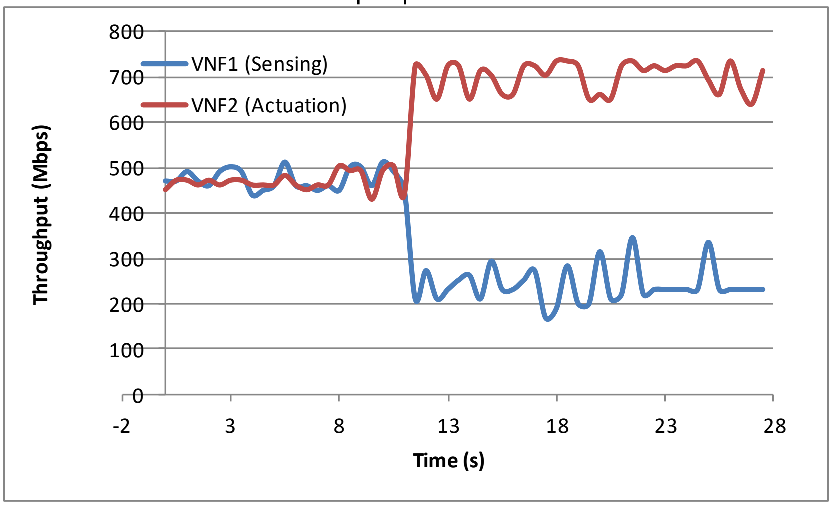



FIGURE 29: THROUGHPUT MEASUREMENT VS. TIME FOR VNF1, VNF2

#### 5.2.2.2 VNF PACKET DELAY

In terms of resource usage, slicing is a relatively expensive solution, and hence often reserved only for the most critical services. An alternative solution to afford low latencies to delay-sensitive services is to place them directly at the IIoT gateway. This way, they bypass the Network Layer and its potential bottleneck, and can directly communicate with Field Layer devices. In the following experiment, the Round-Trip Time (RTT) of packets transmitted from the actuation VNF to the Hue bridge is measured, when it is placed at a Cloud hypervisor, or directly at the virtualized IIoT gateway. The RTT of the local cloud is also compared to the cloud service provided by the smart light vendor. In both cases background traffic with an Exponential traffic profile is also generated, with a Load that varies from 0 (no background traffic) to 0.8 (severe congestion). The measured packet delay of the actuation VNF, when hosted at the Local or Remote cloud or at the Gateway is plotted in Figure 30. We conclude that sub-millisecond latencies are achievable for services hosted directly at the IIoT Gateway, which are unaffected by network congestion. Therefore, given that uRLLC is crucial for the manufacturing process, we show that our platform can attain sub-millisecond end-to-end communication, proving the suitability of our platform for tactile internet industrial applications. This is also possible for local cloud services, as long as the link load is less than 0.5, which can be achieved with dedicated slices. However, as shown in Figure 30, even when slicing is employed, queueing delay of Exponential traffic increases noticeably when input load exceeds 50%. Hence, a dedicated slice typically uses up twice the bandwidth required on average and is therefore considered an expensive solution. Finally, Remote Cloud solutions should be avoided for delay sensitive services, as they are subject to significantly higher latencies.

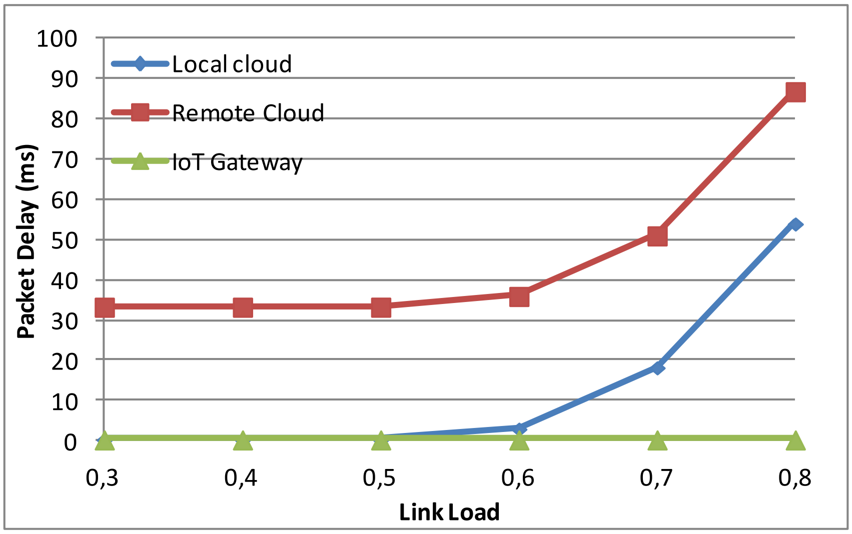



FIGURE 30: PACKET DELAY VS. LOAD FOR DIFFERENT VNF PLACEMENT OPTIONS

#### 5.2.2.3 VM MIGRATION

In our last experiment, we explore whether VM migration is an efficient mechanism for the optimal placement of VNFs. Specifically, we test the service disruption caused when VMs are migrated to a different hypervisor at the NFV Cloud. Figure 31 shows how the throughput measurement of the two VNFs in 0.1 second intervals, when measured from a Field layer client device. The migration time was found comparable is both cases, as in our testbed it is dominated by the copying of Virtual Hard Disk of the VMs. However, in the case of Legacy migration a service disruption of around 8.5 seconds was measured, while services and TCP connections would terminate and need to be restarted. On the other hand, Live Migration caused no service disruption and was only noticeable by a small drop in the measured throughput, which dropped by 40% for a duration of less than 0.5 seconds.

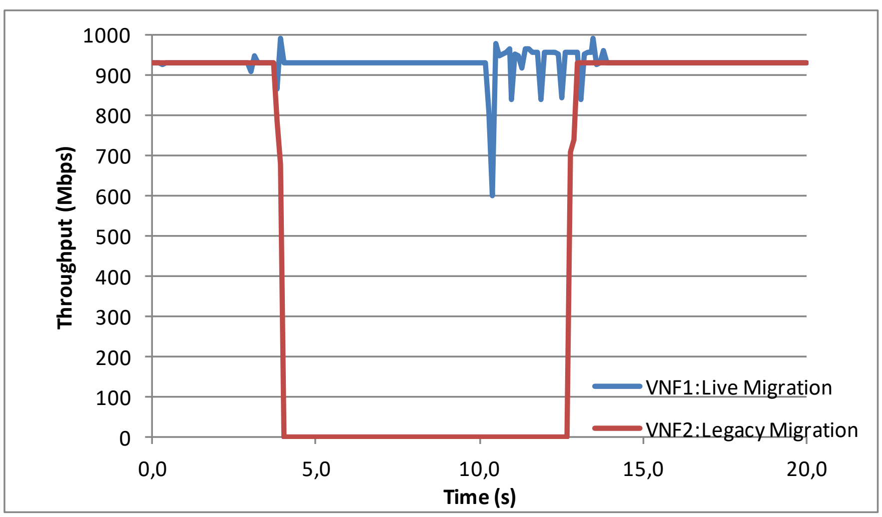



FIGURE 31: THROUGHPUT VS. TIME FOR LIVE AND LEGACY MIGRATION

## 5.3 Experimental evaluation of the NFV Orchestration subsystems

In order to demonstrate the potential of NFV Orchestration introduced in Section 3.2, we implemented an experimental setup, as detailed in (I. Sargiannis, 2019), leveraging the SEMIoTICS integration testbed described in Section 5.1. In the following, we first provide an experimental setup and, then, we evaluate the performance of the NFV Orchestration subsystems, presented in Section 3.2. In our testbed setup the MEC/Edge hypervisor has a maximum capacity of HMECMax=3 and one Cloud hypervisor with HCloudMax=6, and $VNF_i${Resources}=1. We distinguish 3 VNF types, based on delay constraints: Latency-critical VNFs (LCVNFs), which are sensitive to latency, and latency-tolerant VNFs (LTVNFs) that can tolerate a higher degree of delay. Accordingly, the IIoT applications can be classified intro three categories: i) Real-time applications, consisting of high priority LCVNFs (HP LCVNFs), ii) Near real-time applications, consisting of low priority LCVNFs (LP LCVNFs), and iii) Non real-time applications that consist of LTVNFs. In terms of QoS, the SLA for the HP LCVNF is set at 100ms, for the LP LCVNF at 200ms while for the LTVNF the latency is irrelevant as the transmission is asynchronous. The scale-out threshold is set at 90% CPU utilization, the scale-in at 30% and the cooldown period at 180 seconds. Since we assume exponential service time on the LCVNF service, as soon as the CPU utilization exceeds the 91% threshold, the response time violates the SLA, so the scale-out process will take place prior to this violation.

### 5.3.1 SERVICE ONBOARDING & AUTOSCALING

On the first experiment, illustrated in Figure 33, we validate and demonstrate the Network Service (NS) onboarding process detailed in Section 3.2.1, with the goal to provide an optimal placement which results in maximizing the served requests. More specifically, we assume two chained VNFs, one HP LCVNF and one LTVNF and we show that there are three VNF placement methods: i) all VNFs deployed to the Cloud (Figure 33-a), ii) the LCVNFs deployed on the MEC and the LTVNFs on the Cloud (Figure 33-b), and iii) all VNFs deployed to the MEC (Figure 33-c). We reject the first solution as the Service Level Agreement (SLA) is being violated, because the HP LCVNF cannot tolerate the increased latency imposed by the MEC-Cloud link. According to the onboarding algorithm, the initial placement is performed based on latency constrains, i.e., the HP LCVNFs are allocated on the Edge tier, while the LTVNFS are allocated on the Core tier. After the initial placement, the HP LCVNF is hosted on the MEC ($VNF_1${HP,1, MEC}), while the LTVNF is hosted on the Cloud ($VNF_2${LT,1, Cloud}). This is the optimal solution as, in case of increased traffic, the HP LCVNF can scale-out twice until the MEC resources are depleted (HMEC=0) and serve more request (Figure 33-e). Finally, in the third deployment method, where everything is deployed on the Edge tier, the HP LCVNF can scale-out only

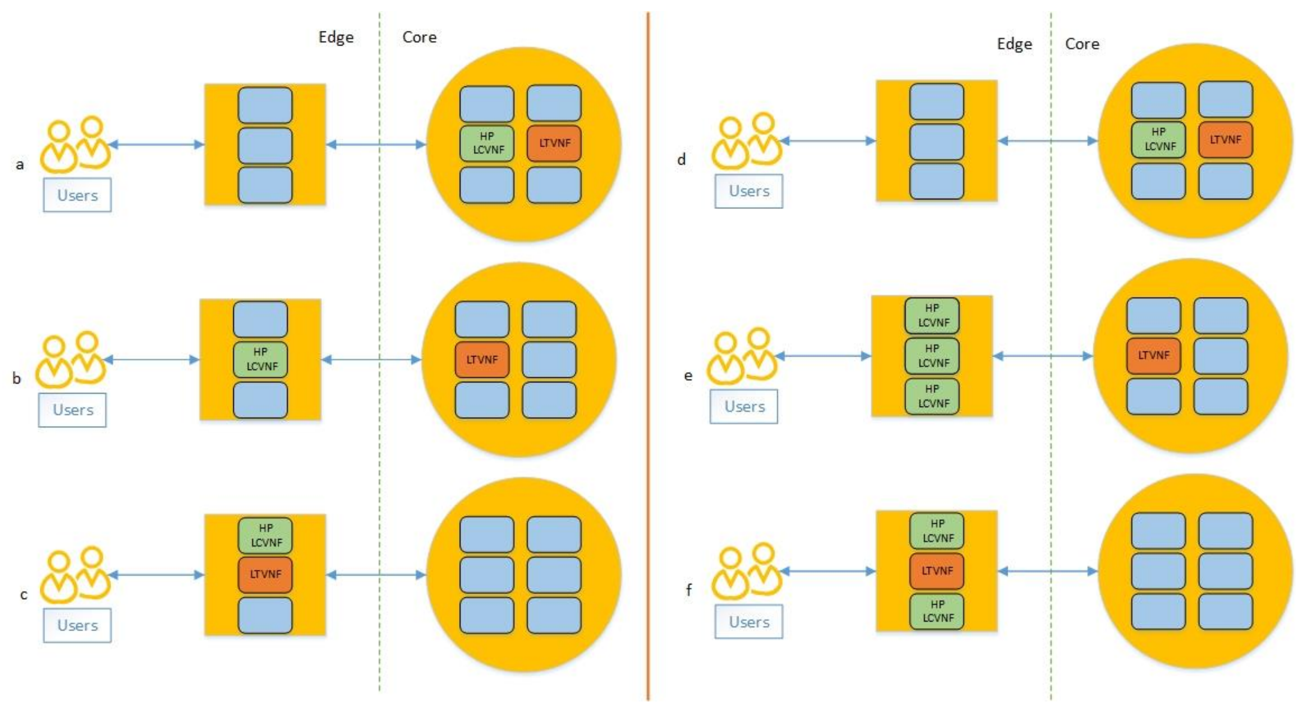



FIGURE 32: VNF INITIAL PLACEMENT (LEFT) AND PLACEMENT AFTER AUTO-SCALE FUNCTIONALITY (RIGHT)

once (Figure 33-f) and the MEC resources are depleted (HMEC=0), since there is one LTVNF deployed on the MEC ($VNF_2${LT,1,MEC}).

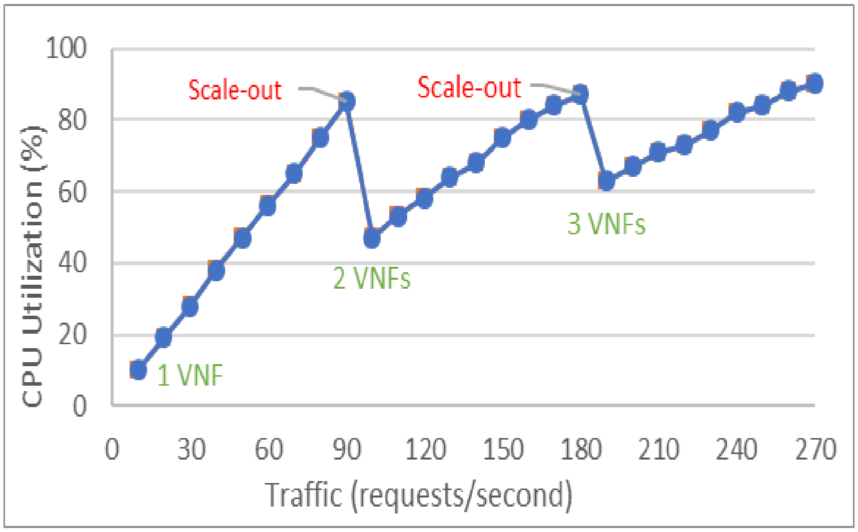



FIGURE 33: SCALE-OUT PROCESS TO ACCOMMODATE INCREASED INCOMING TRAFFIC

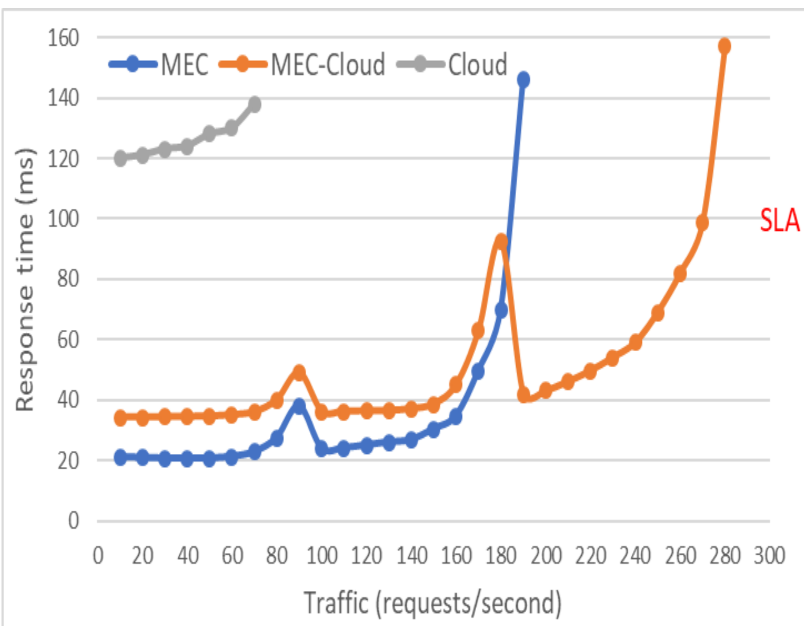



FIGURE 34: RESPONSE TIME OVER TRAFFIC FOR THE DIFFERENT DEPLOYMENT SCENARIOS

To illustrate, Figure 34 shows the scale-out process. We start with one VNF and, as the traffic increases the CPU utilization of the VNF increases accordingly. When it reaches the CPU utilization threshold at 90%, it is scaled-out and a second VNF is being instantiated. In order to equally distribute the traffic between the two VNFs, we deploy a load balancer VM with a round robin balancing policy. Hence, each VNF has approximately 45% CPU utilization when the new VNF is instantiated. While the traffic is further increased, another scale-out event is triggered and a third VNF is instantiated, with the load balancer distributing the incoming requests to three VNFs. This results in a 60% CPU utilization by the time the third VNF is instantiated. With the autoscaling feature, we can accommodate more requests, compared with the legacy monolithic deployments that do not support such feature.

In Figure 35, the response time of the VNFs is depicted depending on their placement. From this figure, we can observe that if all the VNFs are deployed on the Cloud, no further investigation is performed as this deployment method violates the SLA (over 100 ms). For the MEC-Cloud placement method. i.e., VNFs are placed between the MEC and the Cloud, the system will be able to support up to 3 LCVNFs on the MEC in order to serve up to 270 requests/second without violation of the SLA. Finally, while the third deployment method has improved response time due to the elimination of the link for the communication of the HP LCVNF with the LTVNF (they are hosted on the same hypervisor), the total requests/second that can serve are limited up to 180, due to the fact that the MEC resources quota has been reached.

### 5.3.2 ONLINE VNF SCHEDULING

In the second experiment, depicted in Figure 36, we demonstrate how the scheduling subsystem with live migration support can be employed to support more requests when LCVNFs with different priorities are competing for the same MEC resources, without disrupting the low priority latency critical service availability. In this scenario, the Network service onboarding subsystem allocates both VNFs on the MEC side (Figure 36-a) (VNF1{HP,1, MEC}, VNF2{LP,1, MEC}). While the requests for the VNF1 are increasing, the CPU utilization increases as well, resulting in VNF1 scale-out (VNF3{HP,1, MEC}). When a second scale-out (VNF4{HP,1,MEC}) takes place, the MEC resources have been depleted (HMEC=0), triggering the scheduling algorithm to: i) live migrate the LP LCVNF to the Cloud (VNF2{LP,1,Cloud}), as depicted in Figure 36-b, and ii) place the scaled-out HP LCVNF (VNF4) on the MEC (Figure 36-c). When the traffic on the HP LCVNF is decreased, a scale-in (erasure of VNF4) occurs and the LP LCVNF (VNF2) is migrated back to its original hypervisor (Figure 36-d).

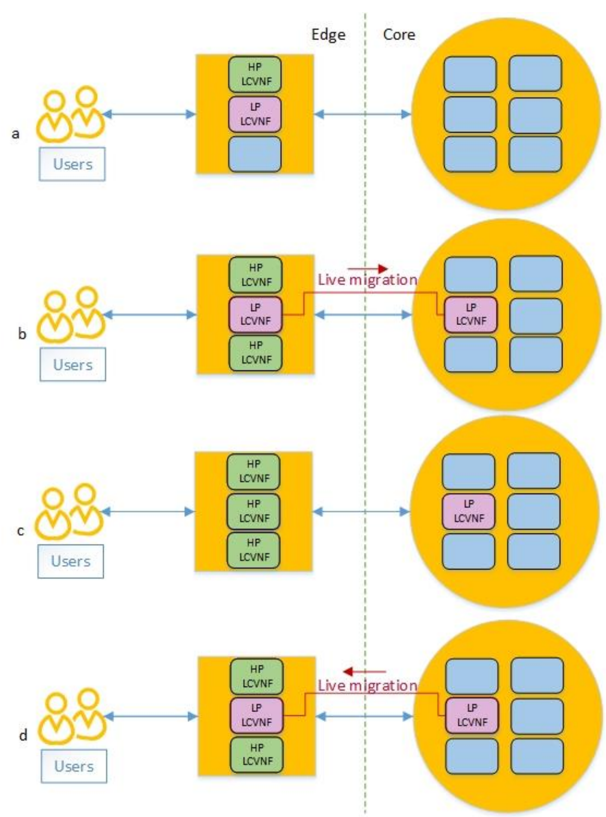



FIGURE 35: LIVE MIGRATION TO ACCOMMODATE MORE HP LCVNF ON THE EDGE

In Figure 37, we evaluate the response time versus the time in minutes. As it can be observed, the requests for the HP LCVNF are increased over time while the requests for the LP LCVNF are stable. As the HP needs to scale-out at minute 85, the script commands the VIM to live migrate the LP LCVNF from the MEC to the cloud, thus freeing up resources for the scale-out of the HP LCVNF. The live migration process, at the minute 85, lasts 28 seconds, for a VM with 1 vCPU, 512MB RAM and 3GB local storage, while no service interruption was observed. It can be noticed that during the live migration process, we notice a slightly increased response time for the LP LCV NF that is not violating the SLA neither during nor after the migration has been completed. Finally, when the scale-in action occurs at the minute 145, the LP LCV NF is migrated back to its original hypervisor.

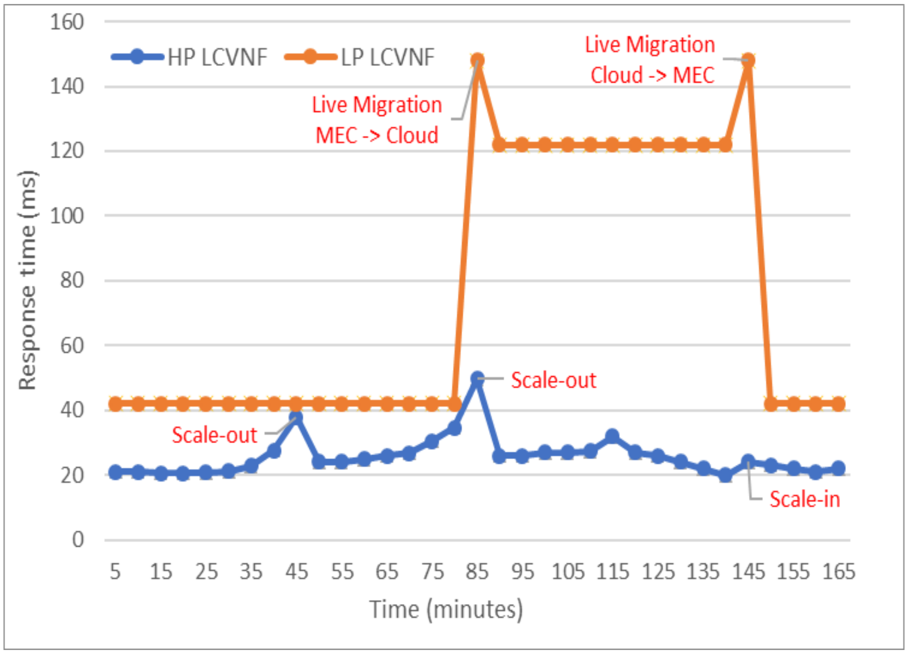



FIGURE 36: RESPONSE TIME PRE AND POST MIGRATION

# 6 CONCLUSIONS AND FUTURE WORK

This deliverable, being the second output of Task 3.5, provides an update on the design and implementation of the Field-level middleware and networking toolbox of SEMIoTICS, for giving access to sensor data via semantically annotated interfaces over multiple messaging protocols.

In this deliverable we discussed how concepts, like NFV, SDN, semantic bootstrapping and interoperability can be leveraged by IIoT networks to increase their reliability, flexibility, and performance. Furthermore, we contributed the architectural design of the SEMIoTICS field-level middleware and explained how it is used in the use cases. The SEMIoTICS integration testbed was also presented, which implement an end-to-end IIoT SDN/NFV architecture, complete with the local cloud, SDN networking and Field layers that demonstrate smart actuation, monitoring and analytics functionalities. Standardized semantic models for IIoT applications and SPDI pattern-driven mechanisms that guarantee network-level semantic interoperability were detailed. These form the basis of the semantic bootstrapping and Interoperability framework, which is a significant part of the SEMIoTICS field-level middleware. Finally, we contributed experimental results regarding the deployment of IIoT applications on top of virtualized infrastructure. In one scenario we achieved sub-millisecond latencies for services hosted directly at the IIoT Gateway, which are unaffected by network congestion. And we showed how the Orchestration subsystems can adapt to user demand, automating service placement, migration, scale-out and load balancing.

In the third and final cycle of this deliverable (which will be provided as “D3.11 – Field-level middleware & networking toolbox (final)”), even more emphasis will be placed on the pattern-driven functionality, and specifically on the interconnection with the NFV Management and Orchestration components via the Os-Ma-Nfvo endpoint and the SSC via its Pattern Engine NBI. Hence, we will be able to showcase policy-driven adaptation and orchestration of the SDN and NFV Infrastructure. Furthermore, the semantics, patterns and specifications of network-level properties, to be included in D3.10 ((“Network-level Semantic Interoperability (final)”), will also be implemented as part of the Pattern-driven NBI in the Field-Level Middleware. The pattern-driven NBIs among MANO entities will also ensure seamless interoperability among different entities of the Backend Cloud.

After validation of the above at the integration testbed, each new feature implemented at the Field-layer Middleware will be made available to use cases, to implement their advanced scenarios and functionalities.

# 7 REFERENCES

ETSI, 2014a Architectural Framework (ETSI GS NFV 002 V1.2.1). Available: https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf. [Accessed November 2018].

ETSI, 2014b Management and Orchestration (ETSI GS NFV-MAN 001), December 2014b. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf. [Accessed November 2018].

ETSI OSM. Available: https://www.etsi.org/technologies/nfv/open-source-mano. [Accessed October 2019]

P. Mekikis et al., 2019, NFV-enabled Experimental Platform for 5G Tactile Internet Support in Industrial Environments, IEEE Transactions on Industrial Informatics

OpenStack 2018a, OpenStack Ironic Project: Bare metal provisioning. Available: https://wiki.openstack.org/wiki/Ironic

OpenStack 2018b, Compute API. Available: https://developer.openstack.org/api-guide/compute/.

Canonical, 2018, Linux Containers. Available: https://linuxcontainers.org/.

OpenStack 2018c, OpenStack Docs: Server concepts. Available: https://developer.openstack.org/api-guide/compute/server_concepts.html.

I. Sariggiannis et al., 2019, Online VNF Lifecycle Management in a MEC-enabled 5G IoT Architecture, IEEE Internet of Things Journal

J. Denton, 2018, Learning OpenStack Networking (Neutron), Second Edition, Birmingham: Packt Publishing Ltd.

OpenStack, 2018d, OpenStack Docs: Networking API v2. Available: https://developer.openstack.org/api-ref/network/v2/.

OpenDaylight 2018, OpenStack and OpenDaylight. Available: https://wiki.opendaylight.org/view/OpenStack_and_OpenDaylight.