SEMIoTICS

Deliverable D5.9: Demonstration and Validation of IWPC - Energy Use Case (Cycle 2)

Deliverable release date	29.12.2020
Authors	1. Darko Anicic, Arne Bröring, Ermin Sakic (SAG) 2. Ulrich Hansen (BWC) 3. Manolis Michalodimitrakis, Konstantinos Georgopoulos, Georgios Tsirantontakis (FORTH) 4. Kostas Ramantas (IQU)
Responsible person	Ermin Sakic (SAG)
Reviewed by	Ermin Sakic, Vivek Kulkarni (SAG), Nikolaos Petroulakis (FORTH), Konstantinos Fysarakis (STS), Mirko Falchetto (ST-I)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Christos Verikoukis, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

Acronyms Table	
Acronym	**Definition**
API	Application Programming Interface
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
Hz	Hertz
IIoT	Industrial IoT
IoT	Internet of Things
Kbps	Kilobits / Second
KPI	Key Performance Indicator
MDSP	MindSphere
MQTT	Message Queuing Telemetry Transport
NFV	Network Function Virtualization
OEDK	Open Edge Device Kit
PE	Pattern Engine
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software Defined Networking
SSC	SEMIoTICS SDN Controller
UC1	Use Case 1
VM	Virtual Machine
W3C	World Wide Web Consortium
WoT	Web of Things

1 INTRODUCTION

1.1 Overview

Use Case 1 (UC1 hereafter) focuses on the demonstration of the SEMIoTICS framework in a power generation environment. UC1 demonstration features four Sub Use Cases, focused on local edge analytics for processing unstructured data, interoperability with private and public cloud environment for use case of classic analytics of structured data and field layer monitoring, as well as the interoperability within the field layer in the utility network itself. The orchestration processes and industrial operation occur over an SDN capable of providing the necessary Quality of Service (QoS) guarantees to deployed network flows. Summarized, UC1 highlights the following benefits of using the SEMIoTICS framework:

- Usage of SEMIoTICS for simplified definition of the business logic, distribution, and instantiation of applications in an edge device in the field layer;
- Usage of SEMIoTICS to achieve semantic interoperability between a greenfield device executing the business logic and a legacy brownfield environment connected to a model wind turbine utilizing a common semantic abstraction layer;
- Usage of SEMIoTICS to achieve efficient "plug and play" on-boarding of a new field layer IIoT devices;
- Usage of SEMIoTICS to integrate field layer devices in interoperable manner with private and public clouds;
- Usage of integrated SEMIoTICS orchestration to deploy network services with deterministic QoS properties and;
- Usage of SEMIOTICS to guarantee Security (Data Integrity and Confidentiality), and Connectivity properties in the system, using the pattern-based framework.

Considering the above and being the final output of Task 5.4 ("Demonstration and validation of IWPC-Energy scenario"), this deliverable detail the business motivation behind the final set of Sub Use cases, as well as their final implementation state.

1.2 D5.9 Content updates compared to D5.4

Compared to D5.4, the previous deliverable of T5.4, which provided the initial description of the conceptual goals of UC1, and the intermediate implementation of the SEMIoTICS components supporting the use case, D5.9 is updated and contain the most recent and final status of the testbed integration. In more detail:

- Section 2 was updated with the latest information, content, and photos of the integrated Use Case 1 demonstrator. Additional visual presentation and execution is presented in the accompanying demonstrator video
- Section 3 now provides a separate and up-to-date description of the networking approach adopted throughout each of the four sub Use-Cases
- Sections 4 to 7 were updated to contain the latest and final description of their implementations. i.e. they are the four Sub Use Case Sections defining the UC1
- Section 8 now includes the description of requirements and KPIs addressed by the Sub Use Cases, comprising their achievement status and the methodology of evaluation
- Section 9 now succinctly describes obstacles observed during the UC1 development and integration

2 USE CASE DESCRIPTION

2.1 Overview of the Use Case

The overall success criteria of UC1 is to demonstrate the coexistence of a highly integrated control system and an agile IIoT ecosystem based on the SEMIoTICS framework; which in return allows service providers to deploy new value-added services faster and provide effective access to data from both an existing 'brownfield' control system and a newly deployed 'greenfield' sensor network. The demonstrator of UC1 in the lab trial addresses some of the common challenges in extending the capabilities of an existing control system in a brownfield environment in the Wind Energy domain.

Control systems for industrial processes are in general terms heavily embedded in nature. Extending the capabilities of such a system typically involves engagement with component manufacturer or service integrator to fuse-in the new functionality that must be tested to validate the new component does not void the integrity and the intended behaviour of the control system. Most of these control systems expose certain high-level machine-readable interfaces for seamless integration with other systems, which is not enough to take full advantage of the value that can be generated.

The SEMIoTICS approach in UC1 provides an end-to-end ecosystem allowing service integrators and service operators to build and deploy new value-added services, interfacing directly with the existing assets.

Overall, SEMIoTICS UC1 demonstrates four concrete Sub Use cases in a lab trial showing the IIoT field devices and applications:

- Sub Use Case 1: Audio Processing Demo – Detection of loose objects
- Sub Use Case 2: Video Processing Demo – Oil/Grease leakage detection
- Sub Use Case 3: Inclination Measurement of wind turbine tower
- Sub Use Case 4: Availability Monitoring of Field Orchestration Layer devices

The physical components involved in the developed demonstrator are visualized in Figure 1, and comprise:

a) A wind turbine model with a rotating piece
b) a SIMATIC S7 programmable logic controller (PLC) used to control the wind turbine rotor
c) the greenfield sensory devices (microphone, camera, inclinometer), attached either directly to a Nanobox device (see h)) or to a Raspberry Pi field layer device
d) six OpenFlow 1.3 SDN switches
e) the SEMIoTICS SDN Controller (SSC) hosted on a Siemens Microbox Industrial PC (IPC)
f) the backend components of the SEMIoTICS platform hosted on a Siemens NanoBox Industrial PC
g) the video and audio analytics software processes hosted on the two IPCs
h) the integrated SEMIoTICS Gateway solution hosted on another IPC
i) the LTE Gateway for enabling internet access and connectivity to remote clouds
j) a management switch for enabling out-of-band control channel over SDN switches and visualization on an external device
k) external software application counterparts in the MindSphere cloud and a private cloud at IQU's premises

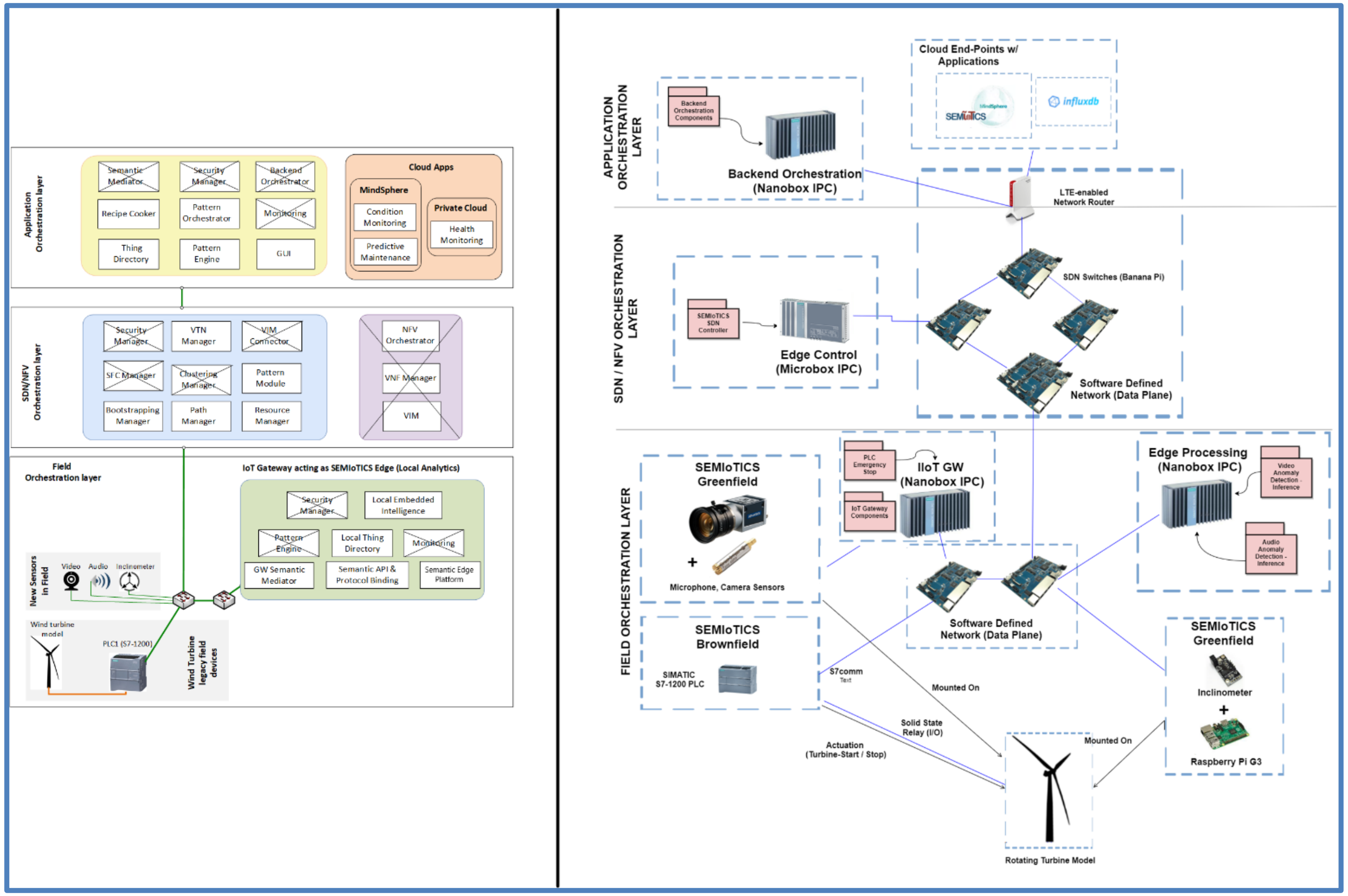


Figure 1: Hardware setup of the demo bag

The components of the SEMIoTICS platform deployed in UC1 are portrayed on the left in Figure 1 and include the:

- Recipe Cooker for orchestration of business logic
- Pattern Orchestrator for translation of recipe descriptions into Security and Connectivity patterns
- The Pattern Engine in the Application Orchestration Layer for ensuring Security constraints in the private cloud for the purpose of enabling the Sub Use Case 4
- The Pattern Engine in the SDN/NFV Orchestration Layer for ensuring Connectivity constraints in the local network for the purpose of enabling the Sub Use Cases 1 and 2
- Global and Local Thing Directory components in Application Orchestration Layer and Field Orchestration Layer, respectively. Thing Directory instances are used for local and global registration of field layer devices, their capabilities, and exposure of these to the backend Recipe Cooker
- The backend GUI in the Application Orchestration Layer for visualization of the state of Pattern requirements and Thing Directories
- The majority of sub-components of the SEMIoTICS SDN Controller for enabling end-to-end connectivity with delay, bandwidth, and redundancy constraints for point-to-point network connections
- The majority of IoT Gateway components for enabling local embedded intelligence, semantic mediation between brownfield and greenfield field components and exposure of their capabilities and actions to external entities using Semantic API and Protocol bindings

The exact deployment mapping of SEMIoTICS platform components to the underlying hardware resources and layers of SEMIoTICS architecture is depicted in Figure 2.

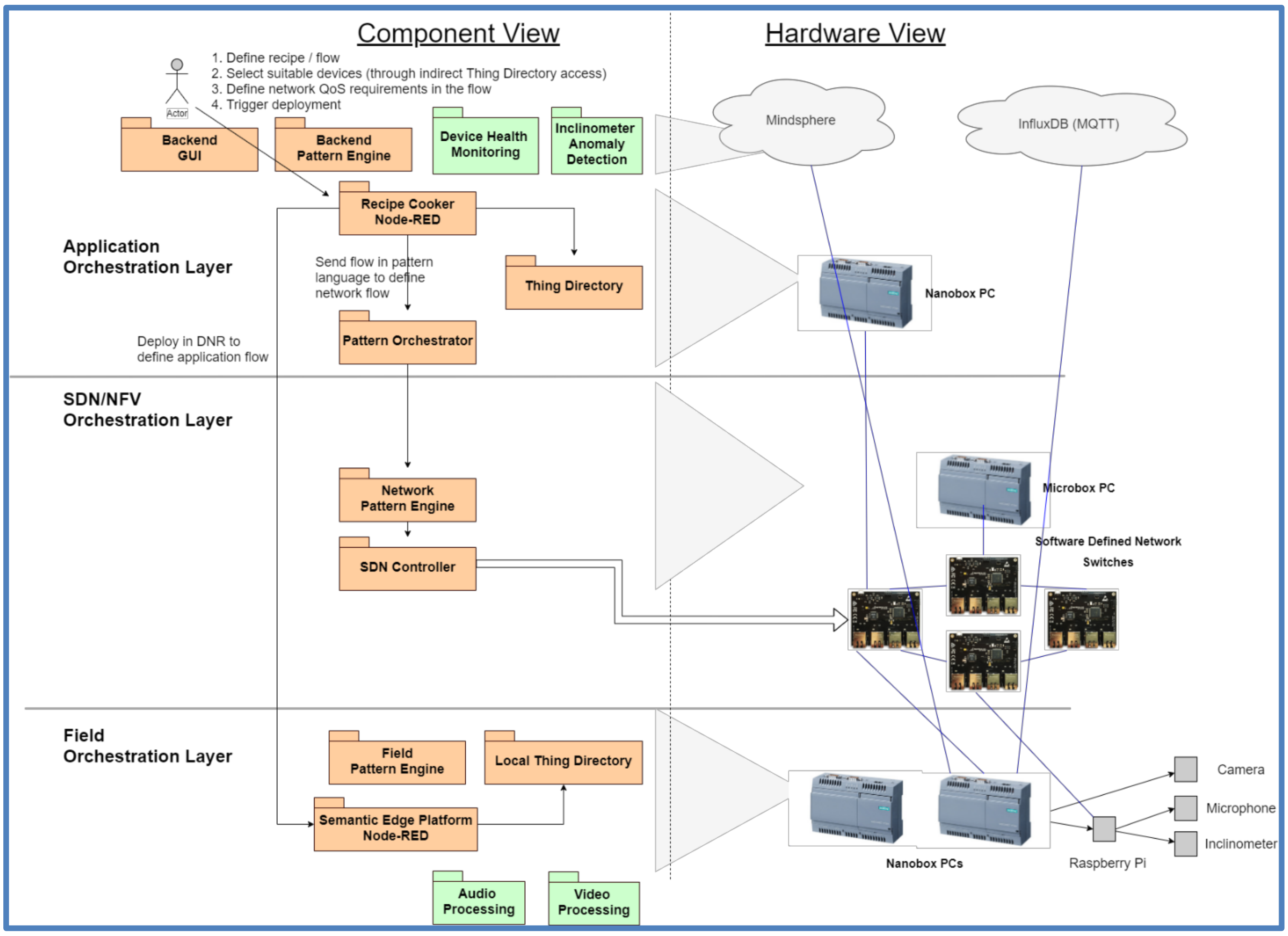


FIGURE 2: OVERVIEW OF KEY COMPONENTS IN USE CASE 1

Finally, Figure 3 shows the final deployment of the hardware components and the separation of SEMIoTICS layers in the transportable demo case that was created for this purpose.

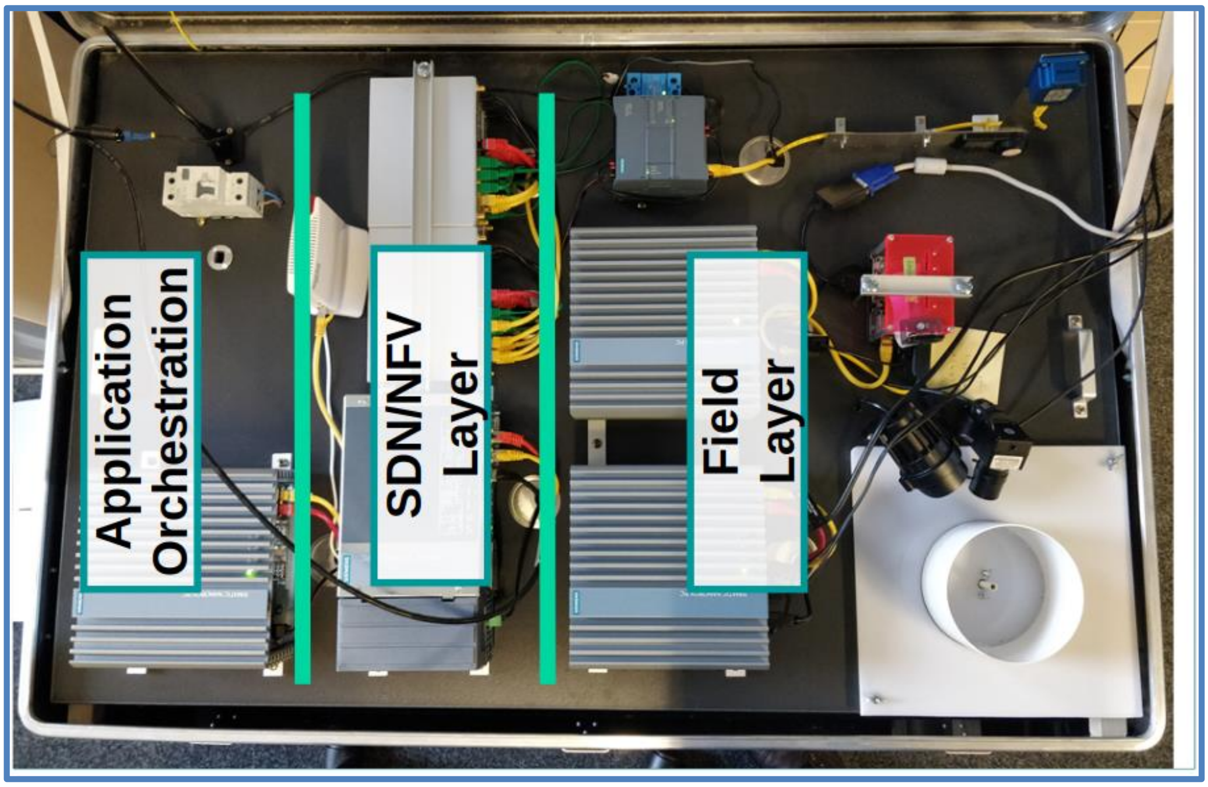


FIGURE 3: FINAL DEPLOYMENT OF THE DEMONSTRATOR IN THE TRANSPORTABLE DEMONSTRATOR CASE

2.2 Main Objectives of Use Case 1

Current wind turbine controllers in a wind park control network are typically highly integrated operating systems, which follow a long process of thorough development, testing, pre-qualification, and finally deployment in the real world. As a result of this long process, introduction of new features, addition of new sensors, actuators and related advancements require several months or even years to be fully matured and become operational in the field. To improve this situation and shorten this development cycle, the UC1 achieves the following objectives:

1) **Local predictive analytics for structured data, smart behavior and monitoring:** We deploy a localized edge to achieve semi-autonomous IIoT behavior where a minimal amount of data is transferred between the wind turbine and the remote control center, while the majority of pre-processing, analytics and triggering of the actuation steps happen at the ‘edge’ of the system, close to the actual turbine. In Sub Use Case 1 and 2 we demonstrate the sensing of unstructured data (i.e., audio and video respectively), processing the data in real-time, and taking an immediate action to prevent any damage to the parts of the turbine. Additionally, a federation, based on a semantic data model and capabilities, is formed between the IIoT Gateway and the legacy control system in Sub Use Cases 1, 2 and 3, to enable the use of data exposed by the sensors connected to the legacy control system. This allows the operators to leverage the existing computing resources to make use of data exposed by sensors connected to an existing brownfield infrastructure, in an automated manner.

2) **Multi-tenant capable network service establishment:** A wind park network typically comprises multiple stakeholders requiring access to a common network infrastructure. Nevertheless, providing reliable and Quality of Service (QoS)-constrained packet transmission for critical IoT services, in conjunction with non-real-time and bandwidth-heavy services, necessitates service differentiation. The SSC support this case. In Sub Use Cases 1 and 2, the users of the system (i.e., the operators) communicate their network requirements to the SSC by means of a simple service specification interface (Recipe Cooker) that is

eventually propagated to the SSC for embedding. Thus, all accepted stakeholders in the system are isolated with regards to the network resource utilization and are guaranteed their requested service invariants.

3) **Semantic-based engineering:** In a motivating scenario, an engineer wants to add a new IIoT device (e.g., a sensor, actuator) or replace an existing one with minimal effort. To achieve this, the device should be bootstrapped in a semi-automated fashion. Further, it should expose its functionality over an IIoT Gateway via a machine-interpretable interface so that the engineer can develop a new service or application with that device. UC1 demonstrates that, leveraging SEMIoTICS, the integration and configuration of a new IIoT device in an automation system of a wind park requires very low effort. For this purpose, the greenfield IIoT devices are equipped with an associated semantic description, which support the process of device discovery, as well as commissioning, engineering and re-engineering of automation functions, within an ecosystem of heterogeneous devices and services. The SEMIoTICS semantic models, which enable these device descriptions, are based on standardized semantics such W3C Web of Things[1] (Thing Description), as well as on the IoT domain semantics created by iot.schema.org[2]. SEMIoTICS, thus, supports creation of IIoT applications in a cost-effective manner. The brownfield devices, on the other hand, are similarly supported with their semantics models and exposed by means of a combination of an intermediate semantic API and the semantic binding.

4) **Semantics-based application creation** – Going beyond integration and configuration supported by the "semantic-based engineering" (described above), an application developer needs to be supported in connecting and composing services and devices to Edge-level or Cloud-level applications. To this end, the following aspects are considered:

    a. In order to facilitate quicker development of applications, the Recipe Cooker allows for compositions by the developer as templates (or "recipes") that describe how logical links are established and what kind of devices are required in these compositions.

    b. As part of these definitions of compositions, the developer needs to be able to define Security, Integrity, Connectivity, Privacy or Dependability criteria on the connections for the logical links of the distributed application. These criteria are semantically described and are interpretable by the Pattern Engine instances prior to the deployment of the application to evaluate whether the infrastructure can meet the requirements of the application. Thus, with the seamless integration of specification of pattern requirements with the application workflow design, the developer is capable of both orchestrating the available resources to deploy and execute the application, but also automatically enabling, validating and monitoring additional security and connectivity invariants from a single interface.

2.3 UC1 Sub Use Cases

Prior to addressing the details of the individual Sub Use cases within UC1, a summary of each is provided below.

Sub Use Case 1 - Audio Processing Demo – Detection of Loose Objects: Sub Use Case 1 demonstrates the integration of a distributed audio processing function with the remainder of the SEMIoTICS framework. The analytics function detects a loose object in the wind turbine by processing a continuous stream of unstructured audio data, issuing an emergency stop command to the wind turbine if an anomaly is detected while the wind turbine is in operation. Numerous greenfield and brownfield field layer devices are involved in the operation. Furthermore, the analytics function is realized by deploying an application to an edge device using orchestration introduced by SEMIoTICS and involves other components of the framework, namely the SSC and the IIoT Gateway, to guarantee the network invariants and expose audio state to external applications, respectively.

[1] https://www.w3.org/WoT/WG/
[2] Currently available from: http://iotschema.org/

Sub Use Case 2 - Video Processing Demo – Oil/grease leakage detection: This sub use case focuses on demonstrating video processing capabilities deployed at the field layer, to detect a grease leakage from a main bearing in a wind turbine. This is achieved by processing a stream of unstructured video data and, as above, issuing an emergency stop command to the wind turbine if a severe grease leakage is detected. The capability is realized by deploying an application to multiple edge devices installed at the field layer, by mean of several components of the SEMIoTICS framework to provide the needed supporting capabilities.

Sub Use Case 3 – Inclination Measurement of the Tower: An application deployed in an external IoT platform, i.e., MindSphere, interfaces with a SEMIoTICS-enabled wind park and consumes the inclination data measured by an inclinometer in the field layer, along with other operational data from the simulated wind turbine environment, for visualization and manual decision making. The cloud application furthermore periodically evaluates the reported inclination values and triggers an alarm if a set threshold is exceeded, enabling the operator to act. Only raw data from the inclinometer is being transferred to the cloud and all business logic for decision making is being defined in the cloud application.

Sub Use Case 4 – Field Devices Availability Monitoring: The final business application is being deployed in a private cloud environment with the purpose of monitoring the deployed inclinometer and other greenfield sensors in the system for availability purposes. As the inclinometer sensor is not natively integrated with the existing control and instrumentation system of the wind turbine, it is not being monitored by the native turbine monitoring system. Instead, a small agent deployed at the field layer continuously monitors the availability of the deployed inclinometer sensor and forwards a heartbeat signal to the private cloud for monitoring purposes. In this Sub Use Case only availability information of the sensor is being transmitted to the private cloud, whereas all operational data recorded is being transmitted to the public cloud application as in the Sub Use Case 3. Finally, the Confidentiality and Integrity (and thus Security) of data transmitted to the private cloud is guaranteed with the use of pattern framework.

3 REALIZATION OF THE SDN-BASED CONNECTIVITY

The components of the SEMIoTICS SDN Controller (SSC) designed and developed in Work Package 3 are, for reader's reference, depicted in Figure 4. While most of the presented components were deployed in the SEMIoTICS demonstrators, the SEMIoTICS Use Cases 1 and Use Case 2 focus on the functionalities of only a subset of these.

In the final Use Case 1 prototype we deploy the SSC for two purposes:

- To allow for simplified deployment of the infrastructural services, using the SDN controller as a flow rule installation proxy (internet access, connectivity among the industrial devices);
- To allow for establishment of QoS-enabled end-to-end path configurations requiring minimal user intervention – network requests are specified as part of application description using a Recipe template in the Recipe Cooker.

To establish the QoS constrained network flows, the UC1 relies on interaction between Recipe Cooker, the Pattern Orchestrator, and the Pattern Engine of the SSC to provide for the facts (i.e., the knowledge base) used in the instantiation of QoS connectivity services.

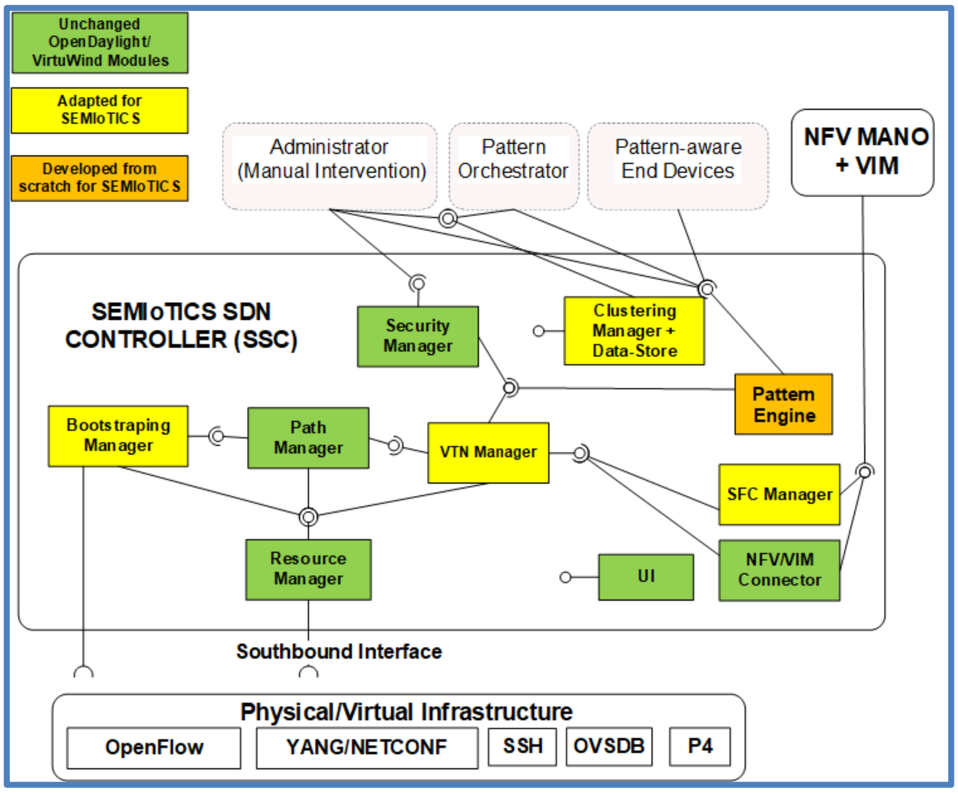

Unchanged OpenDaylight/ VirtuWind Modules
Adapted for SEMIoTICS
Developed from scratch for SEMIoTICS
Administrator (Manual Intervention)
Pattern Orchestrator
Pattern-aware End Devices
NFV MANO + VIM
SEMIoTICS SDN CONTROLLER (SSC)
Security Manager
Clustering Manager + Data-Store
Pattern Engine
Bootstraping Manager
Path Manager
VTN Manager
SFC Manager
Resource Manager
UI
NFV/VIM Connector
Southbound Interface
Physical/Virtual Infrastructure
OpenFlow
YANG/NETCONF
SSH
OVSDB
P4

FIGURE 4: COMPONENTS OF THE SEMIOTICS SDN CONTROLLER (SSC)

Use Case 1 deploys a number of industrial end-devices interconnected by a Layer-2 network comprising 6 OpenFlow switches. The network stack is based on OpenFlow 1.3.1 switch instances, deployed using the kernel-space forwarding daemon of Open vSwitch and the corresponding OpenFlow agent. The physical switches are 1 Gbps Banana PI R1 hardware devices. Each switch is equipped with 4xRJ45 interfaces. Figure 5 and Figure 6 showcase the networking deployment in the UC1 transportable testbed, and the close-up of the deployed data plane, respectively. In addition to the SSC and switches, we also deploy an LTE router attached to switch OF:104, acting as the gateway to enable reachability of backend layer services. The SSC, including the Pattern Engine (SDN PE), is deployed on the SIMATIC IPC427E industrial PC running Linux, equipped with a recent Intel i7 Processor and 16 GB of DDR4 RAM.

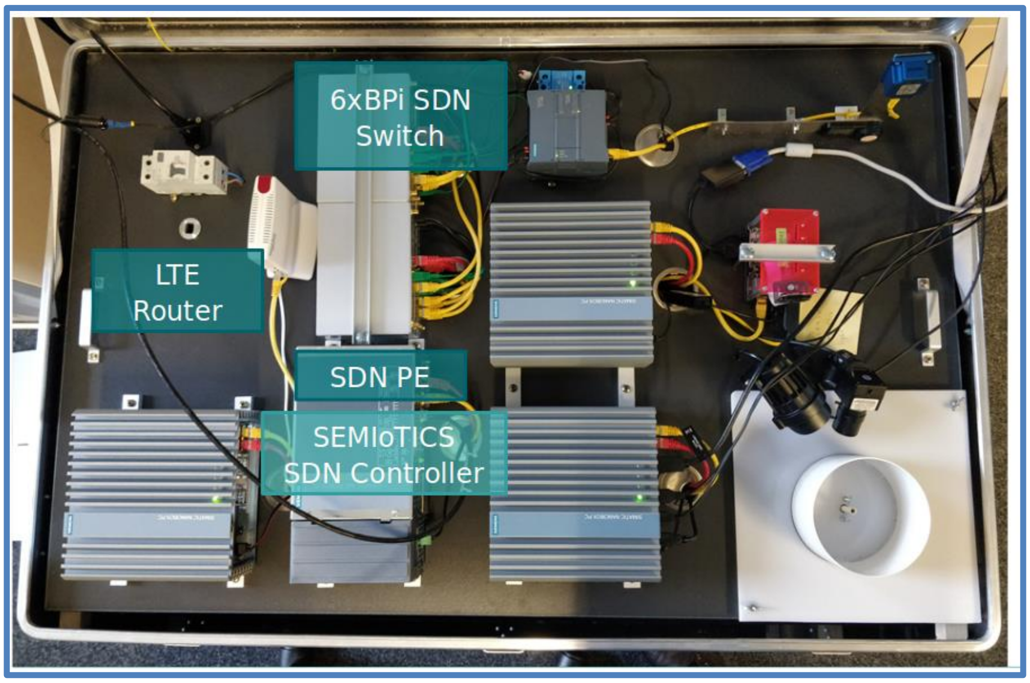


FIGURE 5: NETWORKING COMPONENTS IN THE UC1 TRANSPORTABLE TESTBED

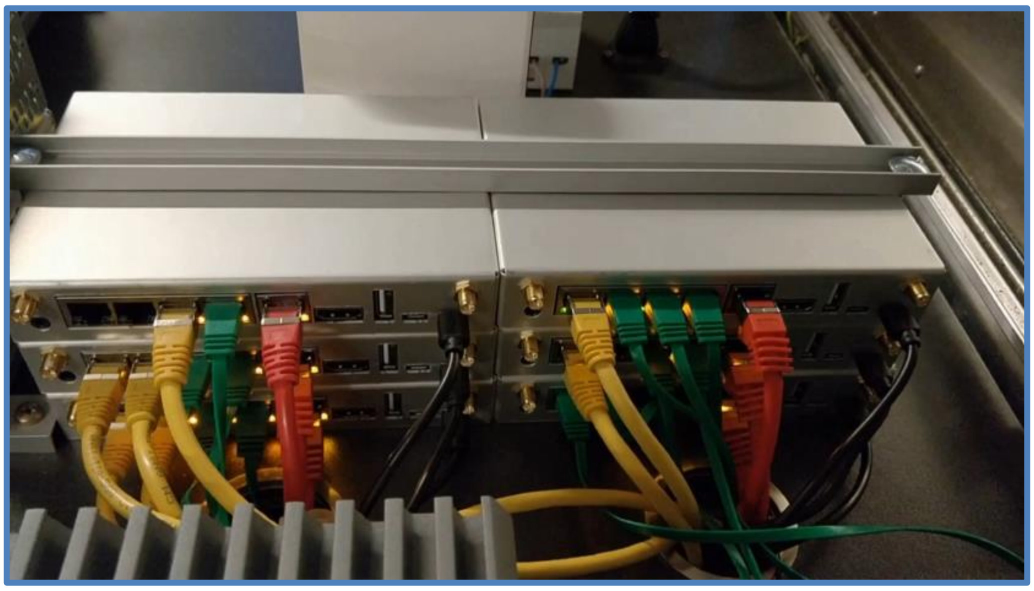


FIGURE 6: CLOSE-UP OF THE SDN SWITCHES INTERCONNECTED IN THE UC1 TOPOLOGY

The SDN was used to route a subset of flows that required only the Best-Effort connectivity. These connections are thus considered ‘infrastructural’ services, as they necessitate only partial bandwidth without time-slot allocations and guarantees to transfer the data. The examples of these are:

- Interconnection of all SEMIoTICS components: Recipe Cooker ⇔ Pattern Orchestrator, Pattern Orchestrator ⇔ SSC, IIoT Gateway ⇔ LTE ⇔ InfluxDB / MindSphere, Recipe Cooker ⇔ Distributed Node-RED workers etc.

- Interconnection of networking services: DNS resolve, SSC ⇔ SDN Switches control channel;

Additionally, the networking infrastructure hosts the QoS-constrained network flows used to interconnect the sensory and actuation devices of Sub Use Case 1 and Sub Use Case 2. The aim is to provide for per-packet delay guarantees in transmission of the mission-critical camera and audio information that are used to infer the decision about the need to stop the turbine at the receiver.

3.1 SDN Bootstrapping

After the initial bootup of network devices, the switches first discover the SSC. The switches are configured with the IP address of the SSC and the port on which the SSC is listening for new OpenFlow connections. Indeed, the controller/port configurations in the Open vSwitch implementation persists after reboots, hence a single-time configuration was necessary to achieve this functionality. Bootstrapping Manager can alternatively deploy the DHCP server and provide the switches with automatically derived IPv4 addresses but we opted for manual IPv4 configuration of management interfaces of the switches for lowered complexity of the UC1 demonstration.

Both the switches' and the SSC's management interfaces are thus configured in the same IP subnet. After the switches have initiated an OpenFlow session to the controller, the Bootstrapping Manager initiates the bootstrapping procedure in the newly instantiated switches, i.e., by provisioning them with default flow rules used for control channel communication. Figure 7 showcases the SSC's UI containing resulting discovered topology (including deployed endpoints). Access to the UI and all other REST-enabled functions of the SSC requires HTTPS digest authentication.

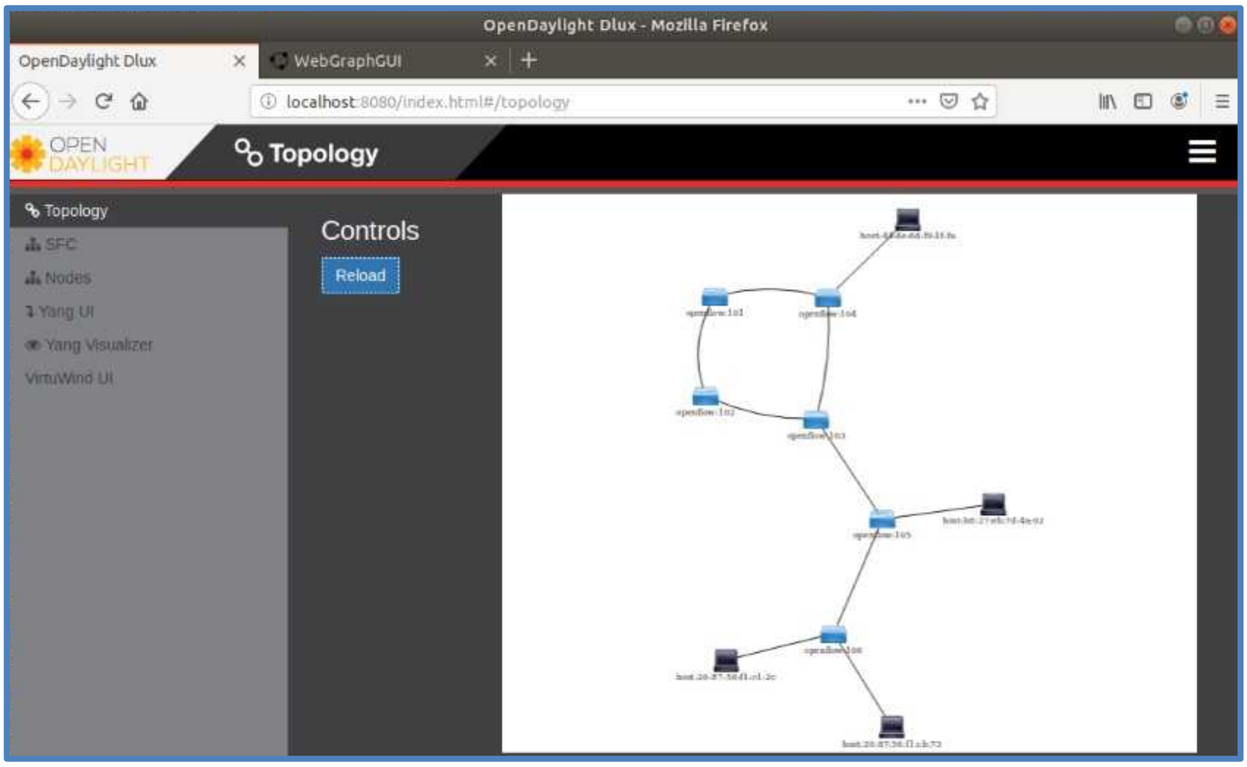


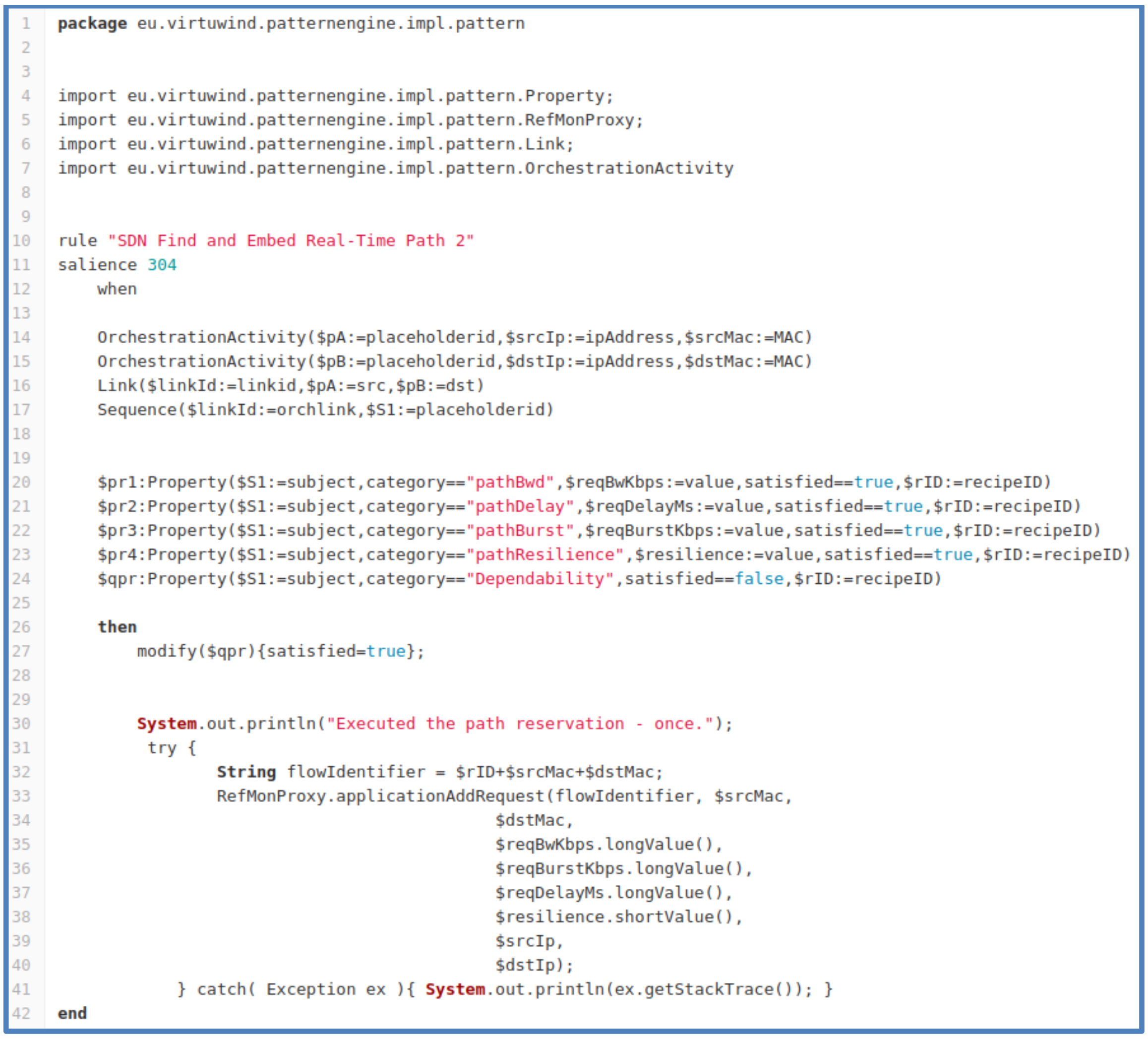
FIGURE 7: THE DISCOVERED TOPOLOGY AND END-DEVICES IN SSC AFTER CONNECTING THREE END-POINTS, BOOTING UP THE SWITCHES AND THEIR SUCCESSFUL ESTABLISHMENT OF OPENFLOW SESSIONS WITH THE SSC

3.2 Flow Establishment

To establish default infrastructure communication flows, SDN switches propagate any unmatched packets (including the ARP requests) to the SSC for further handling. The SSC compares the source and destination nodes with the contents of the discovered topology in the Bootstrapping Manager, and accordingly computes a route between the two endpoints in the Path Manager. Prior to notifying the Path Manager of the route request, the Bootstrapping Manager interacts with the VTN Manager to evaluate the mapping of endpoints to the default VTN (pre-provisioned). Namely, in at least one of the provisioned VTNs, the egress ports of the switches on which the source/destination end-nodes are connected must have been pre-registered. Thus, isolation of critical from best-effort services can be guaranteed on all communication layers. If the end-devices can be found registered in a VTN, the Path Manager handles the route requests and notifies the Resource Manager of the identified path and the flow rules to be installed in the switches. The Resource Manager enforces the installation of the best-effort flows. Best-effort flows in UC1 are always mapped to Queue 0, i.e., the lowest priority queue used by the Strict Priority scheduler implemented by the SDN switches.

To establish the QoS-enabled connectivity between declared endpoints of the evaluated test scenario, we rely on instantiation of the relevant Dependability pattern instances using the SSC's Pattern Engine. The Pattern Engine interacts with the VTN Manager to evaluate the mapping of endpoints to the default VTN (pre-provisioned) and, if satisfied, notifies the Path Manager module of the path request. The Path Manager then computes the corresponding path fulfilling the QoS constraints and notifies the Resource Manager of the flow rules, as well as the associated queue mapping for each hop on the computed path. Thus, an end-to-end path is established in the Sub Use Cases 1 & 2 with deterministic queueing decided individually for each hop on the path. Resource Manager finally installs the flow rules, and the end-point connectivity is enabled.

The corresponding QoS pattern used in implementation of the QoS-enabled service path interconnecting the sensors and actuators is described in Drools syntax in Figure 8, per the pattern rule specification approach detailed in deliverable D4.8.



```
1   package eu.virtuwind.patternengine.impl.pattern
2
3
4   import eu.virtuwind.patternengine.impl.pattern.Property;
5   import eu.virtuwind.patternengine.impl.pattern.RefMonProxy;
6   import eu.virtuwind.patternengine.impl.pattern.Link;
7   import eu.virtuwind.patternengine.impl.pattern.OrchestrationActivity
8
9
10  rule "SDN Find and Embed Real-Time Path 2"
11  salience 304
12      when
13
14      OrchestrationActivity($pA:=placeholderid,$srcIp:=ipAddress,$srcMac:=MAC)
15      OrchestrationActivity($pB:=placeholderid,$dstIp:=ipAddress,$dstMac:=MAC)
16      Link($linkId:=linkid,$pA:=src,$pB:=dst)
17      Sequence($linkId:=orchlink,$S1:=placeholderid)
18
19
20      $pr1:Property($S1:=subject,category=="pathBwd",$reqBwKbps:=value,satisfied==true,$rID:=recipeID)
21      $pr2:Property($S1:=subject,category=="pathDelay",$reqDelayMs:=value,satisfied==true,$rID:=recipeID)
22      $pr3:Property($S1:=subject,category=="pathBurst",$reqBurstKbps:=value,satisfied==true,$rID:=recipeID)
23      $pr4:Property($S1:=subject,category=="pathResilience",$resilience:=value,satisfied==true,$rID:=recipeID)
24      $qpr:Property($S1:=subject,category=="Dependability",satisfied==false,$rID:=recipeID)
25
26      then
27          modify($qpr){satisfied=true};
28
29
30          System.out.println("Executed the path reservation - once.");
31           try {
32                  String flowIdentifier = $rID+$srcMac+$dstMac;
33                  RefMonProxy.applicationAddRequest(flowIdentifier, $srcMac,
34                                                    $dstMac,
35                                                    $reqBwKbps.longValue(),
36                                                    $reqBurstKbps.longValue(),
37                                                    $reqDelayMs.longValue(),
38                                                    $resilience.shortValue(),
39                                                    $srcIp,
40                                                    $dstIp);
41              } catch( Exception ex ){ System.out.println(ex.getStackTrace()); }
42  end
```

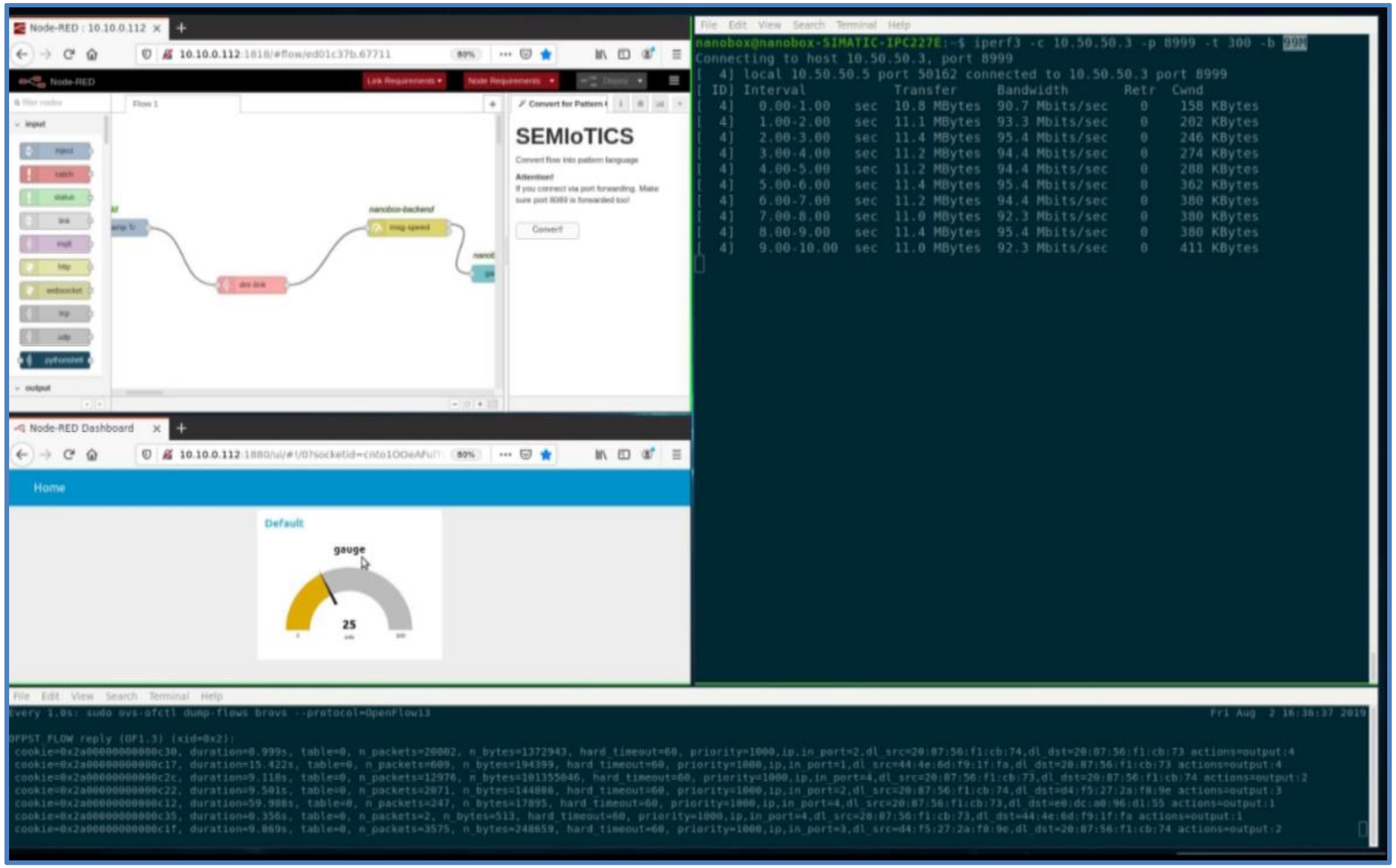
FIGURE 8: DEFINITION OF THE DEPENDABILITY PATTERN USED IN NETWORK SERVICE INSTANTIATION

Referring to the Drool definition provided in Figure 8, the request properties considered in the instantiation of the network service are:

- Line 32: The request identifier;
- Line 20: The required bandwidth share in Kilobits per Second (Kbps);
- Line 21: The requested end-to-end delay requirement in milliseconds;
- Line 22: The input traffic burst in max. Kbps;
- Line 14: The source MAC address;
- Line 15: The destination MAC Address;
- Line 23: The requirement for resilient path establishment.

After establishment of the above flow, the referenced endpoints (with given MAC addresses as identifiers in the flow rules) are guaranteed the requested QoS requirements (bandwidth and delay). This is assuming that that the input traffic arrivals are shaped as per promised maximal traffic burst and sending rate and do not exceed the requested rate.

The full sequence of steps taken starting from specification of individual fields of the request in the Recipe Cooker to embedding of flow rules and automated per-flow / per-hop queue mapping by the SSC are visualized and explained further in our demonstration video (ref. Figure 9).



```
nanobox@nanobox-SIMATIC-IPC227E:~$ iperf3 -c 10.50.50.3 -p 8999 -t 300 -b 99M
Connecting to host 10.50.50.3, port 8999
[  4] local 10.50.50.5 port 50162 connected to 10.50.50.3 port 8999
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  10.8 MBytes  90.7 Mbits/sec    0    158 KBytes
[  4]   1.00-2.00   sec  11.1 MBytes  93.3 Mbits/sec    0    202 KBytes
[  4]   2.00-3.00   sec  11.4 MBytes  95.4 Mbits/sec    0    246 KBytes
[  4]   3.00-4.00   sec  11.2 MBytes  94.4 Mbits/sec    0    274 KBytes
[  4]   4.00-5.00   sec  11.2 MBytes  94.4 Mbits/sec    0    288 KBytes
[  4]   5.00-6.00   sec  11.4 MBytes  95.4 Mbits/sec    0    362 KBytes
[  4]   6.00-7.00   sec  11.2 MBytes  94.4 Mbits/sec    0    380 KBytes
[  4]   7.00-8.00   sec  11.0 MBytes  92.3 Mbits/sec    0    380 KBytes
[  4]   8.00-9.00   sec  11.4 MBytes  95.4 Mbits/sec    0    380 KBytes
[  4]   9.00-10.00  sec  11.0 MBytes  92.3 Mbits/sec    0    411 KBytes
```

```
Every 1.0s: sudo ovs-ofctl dump-flows brovs --protocol=OpenFlow13                                  Fri Aug  2 16:36:37 2019

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2a00000000000c30, duration=8.999s, table=0, n_packets=20002, n_bytes=1372943, hard_timeout=60, priority=1000,ip,in_port=2,dl_src=20:87:56:f1:cb:74,dl_dst=20:87:56:f1:cb:73 actions=output:4
 cookie=0x2a00000000000c17, duration=15.422s, table=0, n_packets=609, n_bytes=194399, hard_timeout=60, priority=1000,ip,in_port=1,dl_src=44:4e:6d:f9:1f:fa,dl_dst=20:87:56:f1:cb:73 actions=output:4
 cookie=0x2a00000000000c2c, duration=9.118s, table=0, n_packets=12976, n_bytes=101355046, hard_timeout=60, priority=1000,ip,in_port=4,dl_src=20:87:56:f1:cb:73,dl_dst=20:87:56:f1:cb:74 actions=output:2
 cookie=0x2a00000000000c22, duration=9.501s, table=0, n_packets=2071, n_bytes=144086, hard_timeout=60, priority=1000,ip,in_port=2,dl_src=20:87:56:f1:cb:74,dl_dst=d4:f5:27:2a:f8:9e actions=output:3
 cookie=0x2a00000000000c12, duration=59.988s, table=0, n_packets=247, n_bytes=17895, hard_timeout=60, priority=1000,ip,in_port=4,dl_src=20:87:56:f1:cb:73,dl_dst=e0:dc:a0:96:d1:55 actions=output:1
 cookie=0x2a00000000000c35, duration=0.356s, table=0, n_packets=2, n_bytes=513, hard_timeout=60, priority=1000,ip,in_port=4,dl_src=20:87:56:f1:cb:73,dl_dst=44:4e:6d:f9:1f:fa actions=output:1
 cookie=0x2a00000000000c1f, duration=9.869s, table=0, n_packets=3575, n_bytes=248659, hard_timeout=60, priority=1000,ip,in_port=3,dl_src=d4:f5:27:2a:f8:9e,dl_dst=20:87:56:f1:cb:74 actions=output:2
```

FIGURE 9: EMBEDDING OF QOS-ENHANCING TRAFFIC FLOWS USING RECIPE COOKER, PATTERN ORCHESTRATOR AND SSC. THE ENHANCED QOS IS PROVIDED FOR INTERCONNECTING SENSORY AND INFERENCE DEVICES DEPLOYED IN THE SUB USE CASE 1 (REF. UC1 DEMONSTRATION VIDEO)

4 SUB USE CASE 1: AUDIO PROCESSING DEMO – DETECTION OF LOOSE OBJECTS

4.1 Scope and Objectives

The first sub use case focuses on demonstrating analytical capabilities deployed at the field layer to detect a loose object in the wind turbine by processing a stream of unstructured audio data and issuing an emergency stop command to the wind turbine if an anomaly is detected while the wind turbine is in operation. The capability is realized by deploying an application to an edge device installed at the field layer utilizing various components of the SEMIoTICS framework, as will be detailed below. The analytical capability is implemented, using an approach based on machine learning techniques to detect an abnormal change in the audio stream.

The system, when deployed in real life, is expected to be in proximity of the wind turbine's hub as shown in Figure 3. The hub is the casted mechanical structure at the very front of the wind turbine, where the three blades are fastened to.

The business objective is to detect a foreign object in hub during operation and stop the wind turbine before any severe damage occurs to the internal parts of the hub.
The hub contains several critical hydraulic and electrical systems responsible for controlling the individual blades' angle in the wind. This system is known as the pitch control system and is in constant rotation while the wind turbine is in operation. If parts of this critical system are damaged, causing the embedded wind turbine controller to detect an error in the pitch control system, the wind turbine will stop automatically with an error and will be unable to produce electricity until it has been manually inspected and repaired.
Said foreign objects can be loose internal mechanical parts to the hub or the three blades, that have been shook loose during operation or seldom forgotten hand tools by wind turbine technicians. Examples of such objects are broken and/or loose bolts in the interface between the hub and the blade bearings, or even plastic covers or access covers to the blades.

Depending on the severity of the damage caused to the internal system by such a loose object, the cost of bringing the turbine back into operation can be very costly in terms of direct expenditure related to the repair work, but also lost revenue, as the wind turbine is unable to produce until repaired. The latter is likely to account for the largest part of the cost.

The scope of the demonstrator focuses on showcasing the following SEMIoTICS advancements:

- The use of select SEMIoTICS components to define the business logic, distribute and to instantiate the application in an edge device at the field layer

- To demonstrate the semantic interoperability between a greenfield device executing the business logic and a simulated legacy brownfield environment connected to a model wind turbine utilizing a common semantic abstraction layer

- The implementation of the business logic focusing on the use of unsupervised machine learning to detect an anomaly in the stream of audio. The demonstrator does not focus on extending the capabilities of the business application required for adoption by industry.

4.2 Interaction with SEMIoTICS Framework and Components

Successful deployment of the business application, for detecting a loose object in the wind turbine's hub relies on several components provided by the SEMIoTICS framework along with components specifically implemented for enabling the demonstration of the use case. An overview of the components and their use is provided in Table 1.

TABLE 1: INVOLVED SEMIOTICS COMPONENTS IN SUB USE CASE

Component	**SEMIoTICS component**	**Architecture reference**	**Description of use**
Semantic Edge Platform	✓	Field Layer	• Provides capabilities for initial device bootstrapping and discovery • Hosts Local Thing Directory service • Provides API for interaction with other devices enrolled in the SEMIoTICS framework deployment
Semantic API and Protocol Binding	✓	Field Layer	• Processing capabilities for interacting with brownfield environments, e.g. the legacy control system
Edge device		Field Layer	• Raspberry Pi device with an embedded microphone for capturing the audio • Anomaly detection process, based on unsupervised machine learning • Execution of the business logic
Legacy Control System		Field Layer	• Simulation of Wind Turbine Control System based on Power Logic Controllers and an operational and controllable wind turbine model
Recipe Cooker	✓	Backend Layer	• Recipe Cooker framework based on Node Red is used to define and deploy the business logic to the field device in a recipe style format
Pattern Orchestrator	✓	Backend Layer	• Used to translate the recipe (business logic request) into a set of patterns comprising Connectivity requirements.
Network Pattern Engine	✓	SDN/NFV Layer	• Used to trigger embedding of networking services in the SDN via associated SDN Controller components, under consideration of delay and bandwidth constraints stemming from Connectivity pattern properties.
SDN Controller	✓	SDN/NFV Layer	• SDN Controller sub-modules, such as Path Finding, Registry Handler, and Resource Manager are used to compute the network paths necessary to fulfil the Connectivity pattern constraints, refer to and keep track of reserved network resources, as well as to embed the service via external configuration of SDN switches.

Global Thing Directory	✓	Backend Layer	• Orchestration of devices and capabilities registered with the Local Thing Directory • Information made available to the Recipe Cooker through an API

Figure 10 shows the sequence diagram of the detection of loose objects. As a prerequisite, the field device (Raspberry Pi with the microphone attached) must be successfully bootstrapped, connected to the communication network and on boarded in the SEMIoTICS eco-system. The application is defined in the Recipe Cooker and the business logic is deployed to the field device.

Audio processing: captures and processes sound recorded by the microphone. The audio process bridges between the physical and digital world: capturing raw sound, amplifies the signal in hardware and converts the analog signal to a digital format. The digitized raw audio signal is then used by the detection process.

Detection process: relying on Fast Fourier Transformation (utilizing Python library Librosa[3]) of the unstructured audio signal source, the detection process creates a structured pattern from the audio signal and compares it with those known to the neural network. The machine learning model has been pre-trained with a finite number of audio samples from a normal operational environment. If an anomaly in the audio signal is detected – indicating a detection of a loose object in the wind turbine – the process will forward an event notification to the action process.

Action process: this process is being triggered whenever the detection process detects an anomaly. The process will first determine the running state of the wind turbine. If the turbine is not in an operational mode (e.g., it is in service mode or not operational for other reasons), it will simply ignore the anomaly detected. If, however, the wind turbine is in operational mode and producing electricity when the anomaly is detected, the process assumes the anomaly to be a legitimate reason for concern and will immediately issue a stop command to the wind turbine.

Semantic Edge Platform: it provides an API that enables the action process to interact with another device enrolled to the SEMIoTICS deployment. The API exposes enrolled devices as Device Nodes, along with their associated capabilities, as sourced from the devices' Thing Descriptions. It ultimately enables the application to interact with the legacy control system.

Semantic API and protocol binding: this component is responsible for the translation between Web of Things protocol and the last mile connection to the simulated brownfield control system, that is using a legacy industrial protocol. It provides a uniform semantic API for all connected devices. This service is the primary enabler to bridge greenfield edge technology with existing brownfield installations. When the stop command has been received by the action process, the service will forward the stop command to the wind turbine via the industrial communication protocol that is supported by the (simulated, in this case) wind turbine control system.

Wind turbine control system: the autonomous simulation of a wind turbine control system. It will take the action to stop the turbine when the command is received. The simulated control system exposes the operational state of the wind turbine, which is used by the action process to determine the authenticity of the anomaly detected.

[3] https://pypi.org/project/librosa/

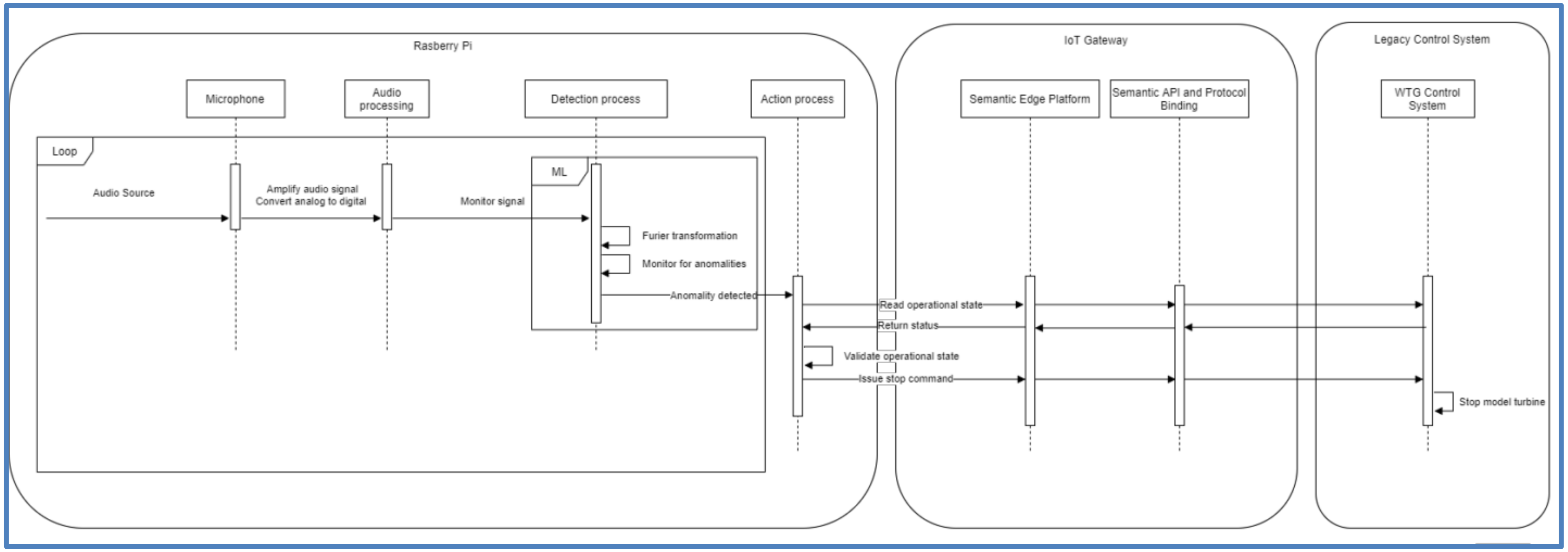


FIGURE 10: OVERVIEW OF DETECTION OF LOOSE OBJECT APPLICATION (UC1 - SUB UC1)

4.3 Setup Testbed and Integration

A prerequisite for this sub use case to work is that all systems are operational and connected to the SEMIoTICS framework deployment, also including the Smart Audio Processing device that must be connected and registered with the Thing Directory.

Furthermore, the capabilities of the legacy control system are loaded to the Thing Directory and the protocol binding service is operational. These Thing capabilities are replicated to the Global Thing Directory so that services can find them. The Backend GUI can visualize the set of capabilities of involved field layer devices based on the content of the global Thing Directory. The IoT Gateway is operational with excess capacity to execute the application. Finally, the Recipe Cooker framework is operational and the application flow for the audio fault detection is already setup.

Figure 11 shows the deployment setup of Sub Use Case 1 broken down to actual machines and installed software components on the right.

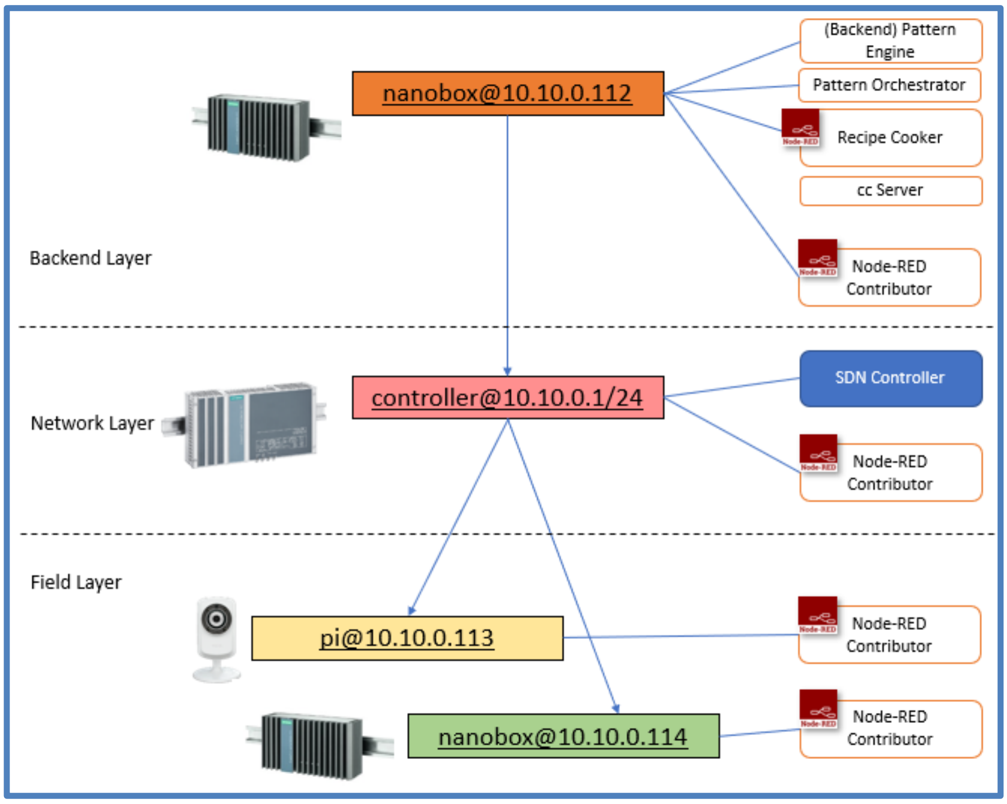


FIGURE 11: SUB UC1 DEPLOYMENT SETUP

5 SUB USE CASE 2: VIDEO PROCESSING DEMO – OIL/GREASE LEAKAGE DETECTION

5.1 Scope and Objectives

The second sub use case focuses on demonstrating analytical capabilities deployed at the field layer to detect grease leakage from the main bearing in a wind turbine, by processing a stream of unstructured video data and issuing an emergency stop command to the wind turbine if a severe grease leakage is confidently detected. The capability is realized by deploying an application to multiple edge device installed at the field layer, also leveraging various components of the SEMIoTICS framework.

The analytical capability is implemented is using an approach based on Machine Learning techniques to detect grease leakages in the video feed images. A large set of training images has been created using a small-scale bearing mock-up that was created for this purpose. This set was used to train the model before deploying the business logic to the field.

The business objective is to prevent an uncontrolled grease leakage from the main bearing by monitoring and eventually stopping the wind turbine as a preventive maintenance action.

Main bearings on large, modern wind turbines are designed to consume and lose some grease during normal operation. Thus, they are fitted with automating greasing systems that ensure the bearing is re-greased with a regular interval. Dependent on the implementation by the wind turbine manufacturer, the bearings are re-greased either at fixed time intervals or ad-hoc by use of more advanced monitoring techniques involving continuously monitoring the pressure inside the bearings (if a lower limit threshold is exceeded the system will re-grease the bearing).

In this context, while grease leakage is expected, larger grease leakages could indicate an issue with the seal of the bearing that needs repair or replacement. An early detection of the issue enables the operator to monitor the development of the leakage and plan appropriate measures to address the issue before stopping the turbine becomes as necessity. A stopped turbine means loss of revenue for the operator.
Since some industrial implementations do not have an automated re-greasing and monitoring system put in place, early detection is critical to avoid mechanical failure.

The scope of the demonstrator focuses on showcasing the following SEMIoTICS advancements:

- The use of select SEMIoTICS components to define the business logic, distribute, and to instantiate the application in an edge device at the field layer
- To demonstrate the semantic interoperability between a greenfield device executing the business logic and a simulated legacy brownfield environment connected to a model wind turbine utilizing a common semantic abstraction layer
- The implementation of the business logic focusing on the use of supervised machine learning to detect a leak in a mock-up of a bearing. The demonstrator does not focus on extending the capabilities of the business application required for adoption by industry.

5.2 Interaction with SEMIoTICS Framework and Components

Successful deployment of the business application, for detecting a loose object in the wind turbines' hub, relies on several components provided by the SEMIoTICS framework along with components implemented for enabling the demonstrating of the use case specifically. An overview of the components and their use is provided in Table 2.

TABLE 2: INVOLVED SEMIOTICS COMPONENTS IN SUB USE CASE 2

Component	**SEMIoTICS component**	**Architecture reference**	**Description of use**
Semantic Edge Platform	✓	Field Layer	• Provides capabilities for initial device bootstrapping and discovery • Hosts Local Thing Directory service • Provides API for interaction with other devices enrolled in the SEMIoTICS framework deployment
Semantic API and Protocol Binding	✓	Field Layer	• Processing capabilities for interacting with brownfield environments; e.g., the legacy control system
Edge device		Field Layer	• Raspberry Pi device with an external camera connected for capturing the video stream • Anomaly detection process, based on machine learning • Execution of the business logic
Legacy Control System		Field Layer	• Simulation of Wind Turbine Control System based on Power Logic Controllers and an operational and controllable wind turbine model
Recipe Cooker	✓	Backend Layer	• Recipe Cooker framework based on Node Red is used to define and deploy the business logic to the field device in a recipe style format
Global Thing Directory	✓	Backend Layer	• Orchestration of devices and capabilities registered with the Local Thing Directory • Information made available to the Recipe Cooker through an API
Pattern Orchestrator	✓	Backend Layer	• Used to translate the recipe (business logic request) into a set of patterns comprising Connectivity requirements.
Network Pattern Engine	✓	SDN/NFV Layer	• Used to trigger embedding of networking services in the SDN via remaining SDN Controller components, under consideration of delay and bandwidth constraints stemming from Connectivity pattern properties.
SDN Controller	✓	SDN/NFV Layer	• SDN Controller sub-modules, such as Path Finding, Registry Handler, and Resource Manager are used to compute the network paths necessary to fulfil the Connectivity pattern

			constraints, refer to and keep track of reserved network resources, as well as to embed the service via external configuration of SDN switches.

Figure 12 shows an overview of the business logic view for grease leakage detection. As a prerequisite, the field devices (consists of Raspberry Pi with the camera attached and another NanoBox running the anomaly detection application) need to have been successfully bootstrapped, connected to the communication network and on boarded in the SEMIoTICS eco-system. Following that, the application is defined in the Recipe Cooker and the business logic is deployed to the field device.

Transcoder: captures the video feed from the camera and encodes the signal to reduce the size of the video when transmitted via the SDN network to the detection process. The latter is hosted and executed on a different device (i.e., the Microbox in Figure 1) in this demonstrator.

Detection process: processes the encoded video feed, providing it to the machine learning model with the purpose of detecting an anomaly. The machine learning model has been pre-trained with a finite number of image samples from a simulated environment, enabling the model to determine when a grease leakage occurs with a high accuracy. If the model determines - with significant confidence - that significant leakage occurs, the process forwards an event notification to the action process.

Action process: this process is being triggered whenever the detection process detects an anomaly. The process immediately issues a stop command to the wind turbine.

Semantic Edge Platform: this component provides an API that enables the action process to interact with another device enrolled to the SEMIoTICS deployment. The API exposes enrolled devices as Device Nodes, along with their associated capabilities sourced from Thing Descriptions. It ultimately enables the application to interact with the legacy control system.

Semantic API and protocol binding: it is responsible for the translation between Web of Things protocol and the last mile connection to the simulated brownfield control system, that is using a legacy industrial protocol. It provides a uniform semantic API for all connected devices. This service is the primary enabler to bridge greenfield edge technology with existing brownfield installations. When the stop command has been received by the action process, the service forwards the stop command to the model wind turbine via the industrial communication protocol that is supported by the (simulated, in this case) wind turbine control system.

Wind turbine control system: the autonomous simulation of a wind turbine control system. It will take the action to stop the model turbine when the command is received.

5.3 Setup Testbed and Integration

A prerequisite for this sub use case to work is that all systems are operational and connected to the SEMIoTICS framework deployment, also including the video processing unit that must be connected and registered with the Thing Directory. The capabilities of the legacy control system are registered at the Thing Directory. The protocol binding service is operational, and the thing capabilities are replicated at the Global Thing Directory. Furthermore, the IoT Gateway is operational with excess capacity to execute the application. Finally, the Recipe Cooker framework is operational, and the application flow is designed and ready to be deployed.

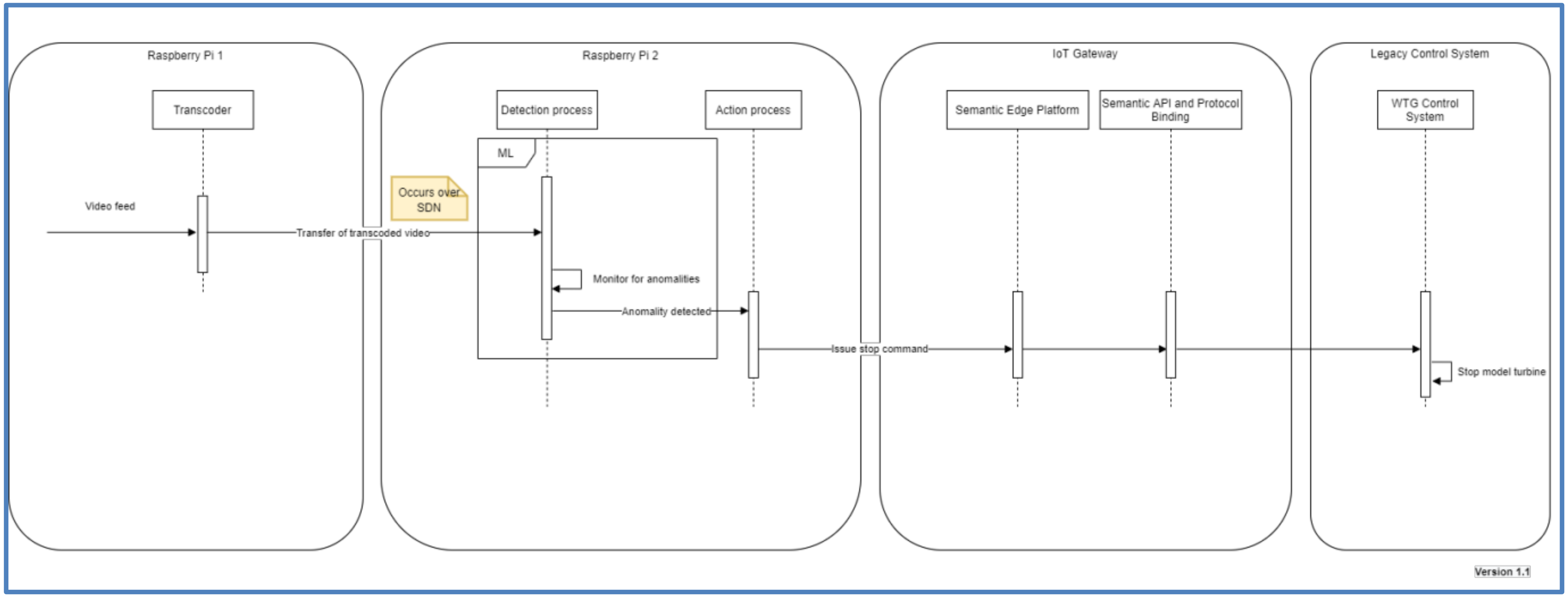


FIGURE 12: OVERVIEW OF GREASE LEAKAGE DETECTION APPLICATION (UC1 - SUB UC2)

6 SUB USE CASE 3: INCLINATION MEASUREMENT OF TOWER

6.1 Scope and Objectives

The third sub use case focuses on “plug and play” onboarding capabilities provided by the SEMIoTICS framework, demonstrating integration of a third-party inclination sensor for deployment in a wind turbine. An inclination sensor is used to measure the orientation angle of an object in relation to the gravitational field of the earth.

Furthermore, it focuses on publishing selected data to both the public and private cloud environments, demonstrating the use of various SEMIoTICS components for interoperability from field layer to cloud. Two business applications deployed in the cloud and making use of said data were created for this purpose.
One business application, deployed in the public cloud, consumes the measured inclination data along with other operational data from the simulated model wind turbine environment, for visualization and decision-making purposes. The cloud application monitors the inclination values transmitted and enables the operator to act if a set threshold is exceeded. Only raw data from the inclinometer is transferred to the cloud and all business logic for decision making is defined in the cloud application.

The second business application is deployed in the private cloud environment with the purpose of monitoring the health of field devices, i.e., the inclinometer sensor. As the inclinometer sensor is not natively integrated with the existing control and instrumentation system of the wind turbine, it is not being monitored by it. The part of the application, which is deployed at the field layer, continuously monitors the availability of the deployed inclinometer sensor and forwards a heartbeat signal to the private cloud. In this example, only availability information of the sensor is transmitted to the private cloud, whereas all operational inclination data recorded is transmitted to the public cloud application.

The business objective is to enable continuous monitoring of the structural health of the wind turbine tower, by continuously monitoring the X/Y sway of the tower. It is common in the wind industry to assess and offer lifetime extending upgrade packages for brownfield installations when they are nearing the end of their original design life, e.g., by performing extended service, inspections and upgrades that would extend the lifetime of a wind turbine; for example, from 25 to 30 years.

Steel towers lose structural strength over time due to fatigue caused by the cycling loads, when the turbine is both in stand-still and in operational mode, therefore it is important to monitor the condition of the tower when the structure is nearing the end of its designed life.

The tubular steel tower of the wind turbine is not easy to or economically viable to exchange for such an upgrade. Inclination sensors can then be fitted at certain locations inside the tower, to continuously monitor the sway of the tower in relation to the current operational and weather conditions, to aid structural engineers to determine to health of the structure.

The scope of the demonstrator focuses on showcasing the following SEMIoTICS advancements:

- The use of select SEMIoTICS components to define the business logic, distribute and to instantiate the application in an edge device at the field layer
- To demonstrate the efficient “plug and play” onboarding of a new field layer IIoT device utilizing the SEMIoTICS framework
- The integration of field layer devices and exposure of data to both private and public cloud.

- Sample business applications, deployed at the field layer, in public and private cloud environment, to monitor the inclination of the wind turbines tower and the availability of the inclinometer sensor.

An overview of the components and their use is provided in Table 3.

TABLE 3: INVOLVED SEMIOTICS COMPONENTS IN SUB USE CASE 3

Component	**SEMIoTICS component**	**Architecture reference**	**Description of use**
Semantic Edge Platform	✓	Field Layer	• Provides capabilities for initial device bootstrapping and discovery • Hosts Local Thing Directory service • Provides API for interaction between SEMIoTICS IoT Gateway and other devices enrolled to SEMIoTICS architecture
Semantic API and Protocol Binding	✓	Field Layer	• Implements field integration with brownfield and greenfield devices • Implements protocol bindings for field devices • Provides a unified semantic interface for field devices
GW Semantic Mediator	✓	Field Layer	• Translates one semantic model into another one, e.g., an existing metadata of brownfield device can be transformed into W3C Thing Description or a Thing Description can be translated into the MindSphere Asset model
Local Thing Directory	✓	Field Layer	• Provides API to manage W3C Thing Descriptions for field devices, e.g., create, read, update, delete, and query • Synchronizes the content (TDs) with Global Thing Directory
Edge device		Field Layer	• Raspberry Pi device with an external inclinometer connected for measuring the inclination of a wind turbine tower • W3C Web of Thing servient, which provides field data and a Thing Description for this (greenfield) device
Legacy Control System		Field Layer	• Simulation of Wind Turbine Control System based on Power Logic Controllers and an operational and controllable wind turbine model (brownfield device)
Public Cloud	✓	Application Layer	• Cloud where inclination filed data from wind turbine park is gathered and processed, implemented with Siemens MindSphere • A host for Condition Monitoring and Predictive Maintenance application
Private Cloud	✓	Application Layer	• Cloud implemented with InfluxDB • A host for Health Monitoring application

6.2 Interaction with SEMIoTICS Framework and Components

The primary components used are the Semantic Edge Platform, IoT Gateway API, the network layer and the data forward application running on the SEMIoTICS IoT Gateway. Furthermore, the connectivity to the cloud is demonstrated. An application to compute and visualize the incoming flow of data from the inclination sensor is defined in MindSphere. The Semantic Edge Platform visualizes the set of and capabilities of newly detected field layer devices, based on the content of Local / Global Thing Directory.

Figure 13 shows an overview of the inclination measurement Sub Use Case, covering all interactions (from the inclinometer bootstrapping process, to the process of use of data in two Clouds). An operator installs an inclinometer in the tower and plugs it in the communication network where IIoT Gateway operates. GW Semantic Mediator discovers and registers the device, using the Thing Description of the device. It stores the description in the Local and Global Thing Directories. Further on, it exposes the device over the gateway's API and generates a Device Node object used to interact with the device (based on information from Thing Description).

In order to integrate brownfield devices, the GW Semantic Mediator first generates a Thing Description (TD). Brownfield devices are different as they originate from very different domains. For this purpose, in this use case we consider a wind turbine and a Siemens controller which controls the turbine. The GW Semantic Mediator reuses existing metadata for the controller (which can be exported from Siemens engineering tool) to create a TD for that brownfield device (see Deliverable 3.9, Section 4.1.2). The GW Semantic Mediator's role, in general, is to mediate and map one form of semantic information of a field device into another form. In this case it maps brownfield metadata of the controller into the W3C Thing Description format. The Mediator is also used in the Cloud integration, i.e., to map W3C Thing Description into the MindSphere Asset model, which is a data model used by said Siemens cloud application.

Once the bootstrapping process is complete, the device can be used in an application. Concerning this sub use-case we have provided two applications, one deployed in MindSphere Cloud and another one in a private Cloud (refer to Sub Use Case 4 as well). We have also provided an Edge application that puts the inclinometer data in a data format suitable for both of the Cloud applications and, in addition, creates device health -related data (e.g., heart-beat data stating that the inclinometer is alive). The Edge application publishes this data to both Clouds. The MindSphere Cloud requires that the inclinometer data is sent to the Cloud over a special agent (MindSphere Agent). The agent is integrated by using Open Edge Device Kit (OEDK)[4]. Device-health data is sent to Private Cloud directly (with no agent) over MQTT.

[4] https://support.industry.siemens.com/cs/document/109766079/open-edge-device-kit-(oedk)?dti=0&lc=en-YE

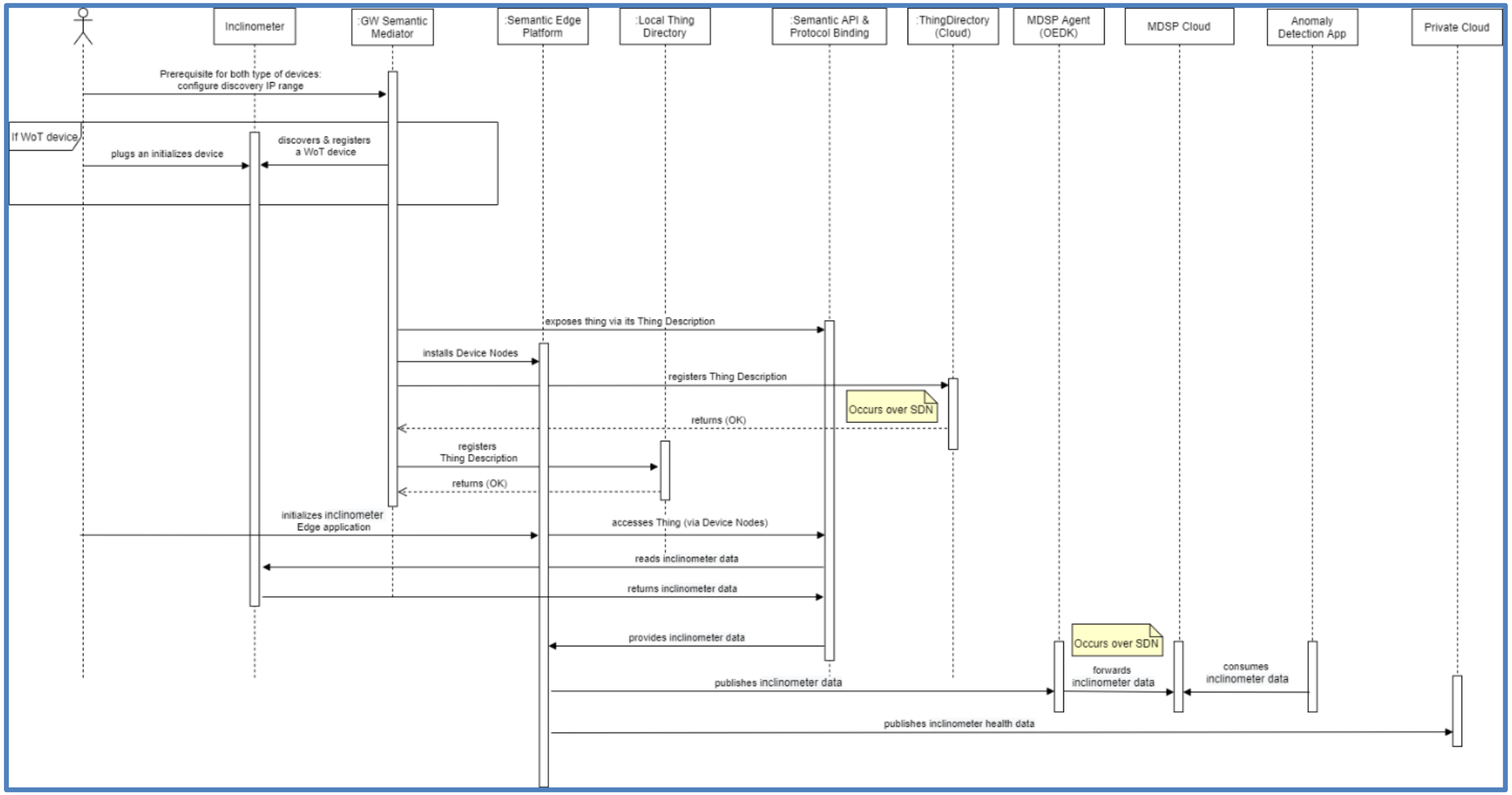

Inclinometer
:GW Semantic Mediator
:Semantic Edge Platform
:Local Thing Directory
:Semantic API & Protocol Binding
:ThingDirectory (Cloud)
MDSP Agent (OEDK)
MDSP Cloud
Anomaly Detection App
Private Cloud
Prerequisite for both type of devices: configure discovery IP range
If WoT device
plugs an initializes device
discovers & registers a WoT device
exposes thing via its Thing Description
installs Device Nodes
registers Thing Description
Occurs over SDN
returns (OK)
registers Thing Description
returns (OK)
initializes inclinometer Edge application
accesses Thing (via Device Nodes)
reads inclinometer data
returns inclinometer data
provides inclinometer data
Occurs over SDN
forwards inclinometer data
consumes inclinometer data
publishes inclinometer data
publishes inclinometer health data

FIGURE 13: OVERVIEW OF INCLINATION MEASUREMENT OF TOWER (UC1 - SUB UC3)

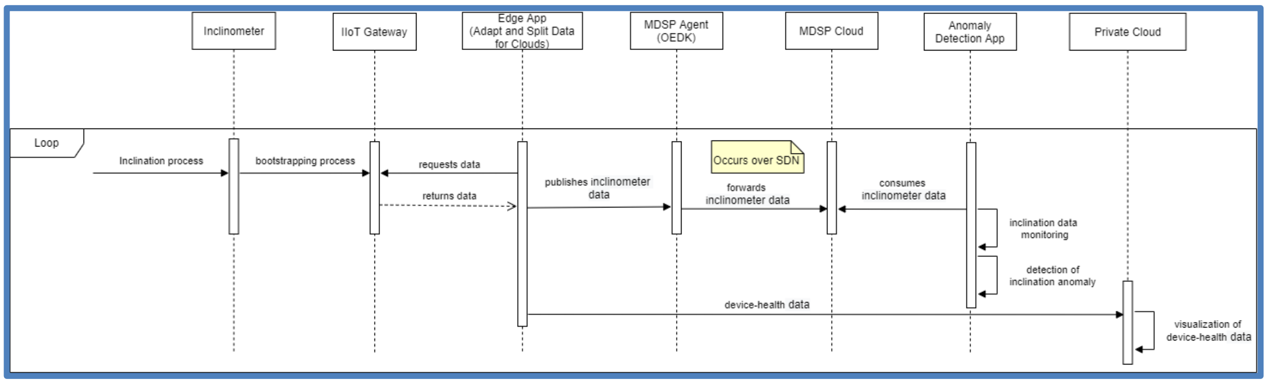

Inclinometer
IIoT Gateway
Edge App (Adapt and Split Data for Clouds)
MDSP Agent (OEDK)
MDSP Cloud
Anomaly Detection App
Private Cloud
Loop
Inclination process
bootstrapping process
requests data
returns data
publishes inclinometer data
Occurs over SDN
forwards inclinometer data
consumes inclinometer data
inclination data monitoring
detection of inclination anomaly
device-health data
visualization of device-health data

FIGURE 14: APPLICATIONS RELATED TO INCLINATION MEASUREMENT OF TOWER

Figure 14 provide details on the two applications that rely on inclinometer data. The user initializes both applications over the Semantic Edge Platform (IIoT Gateway), per the sequence shown in Figure 13. The first application continuously monitors the sway of the tower and detects anomalies (the tower inclination exceeds the allowed threshold). This application is deployed in the MindSphere Cloud. The second application monitors the health status of the inclinometer and visualizes this data. The application is deployed in the private Cloud.

6.3 Setup Testbed and Integration

This subsection describes the testbed setup and integration results of the third sub use case. In this test we successfully on-boarded an inclinometer sensor via the SEMIoTICS IIoT Gateway and visualized its data in the MindSphere Cloud. The test succeeded via a communication enabled by the SEMIoTICS SDN network.

Figure 15 shows the final deployment of the demonstrator with components relevant for Sub Use Case 3 marked in red. The inclinometer is a greenfield device that is added to brown-field installations with the goal of continuously monitoring the condition and health of the wind turbine tower. The inclinometer (the blue sensor in Figure 15) is integrated in the overall automation system with minimal effort, i.e., in plug-and-play manner. Apart from the inclinometer, Figure 15 also shows the other field devices, e.g., a camera and a microphone (the latter used in the previous Sub Use Case), as well as the Siemens S7 controller, i.e., the brownfield device.

The SEMIoTICS IIoT Gateway on-boards the inclinometer. This process includes discovery of the newly plugged device, registration of its semantic meta data (Thing Description) with the Local and Global Thing Directory, and the exposure of the sensor's data over the unified (WoT) API. The IIoT Gateway and the newly plugged device are not directly connected. The gateway scans the network and discovers the device. Based on Thing Description, the gateway is enabled to perform the onboarding procedure in an automated fashion.

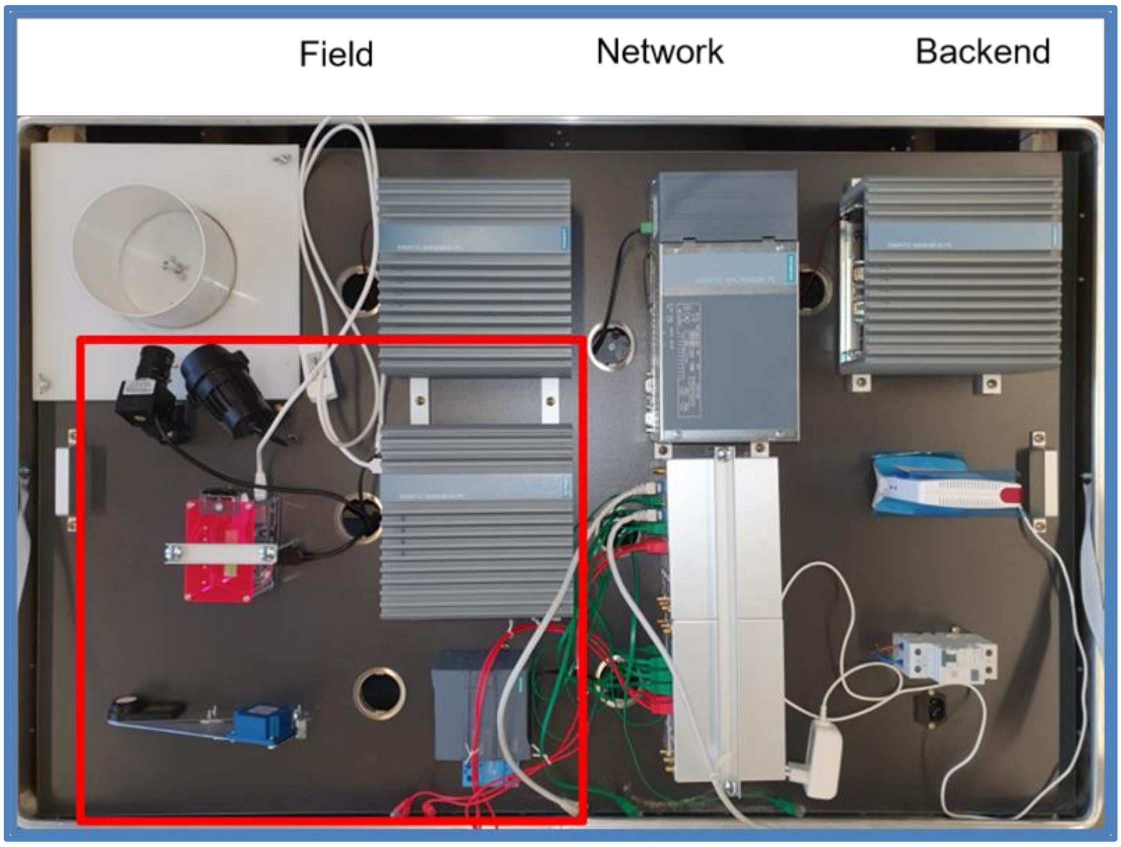


FIGURE 15: HARDWARE FOR INCLINOMETER SUB USE CASE

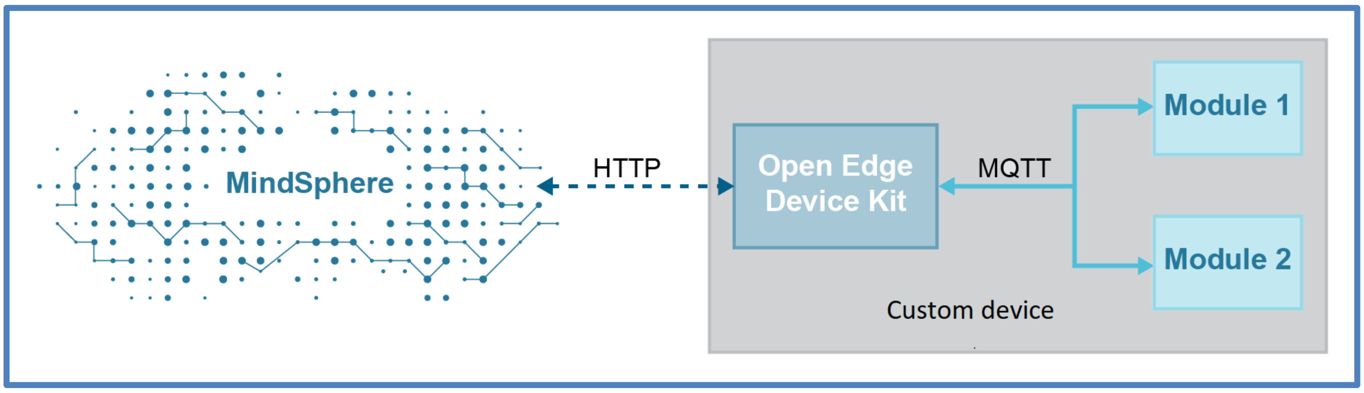
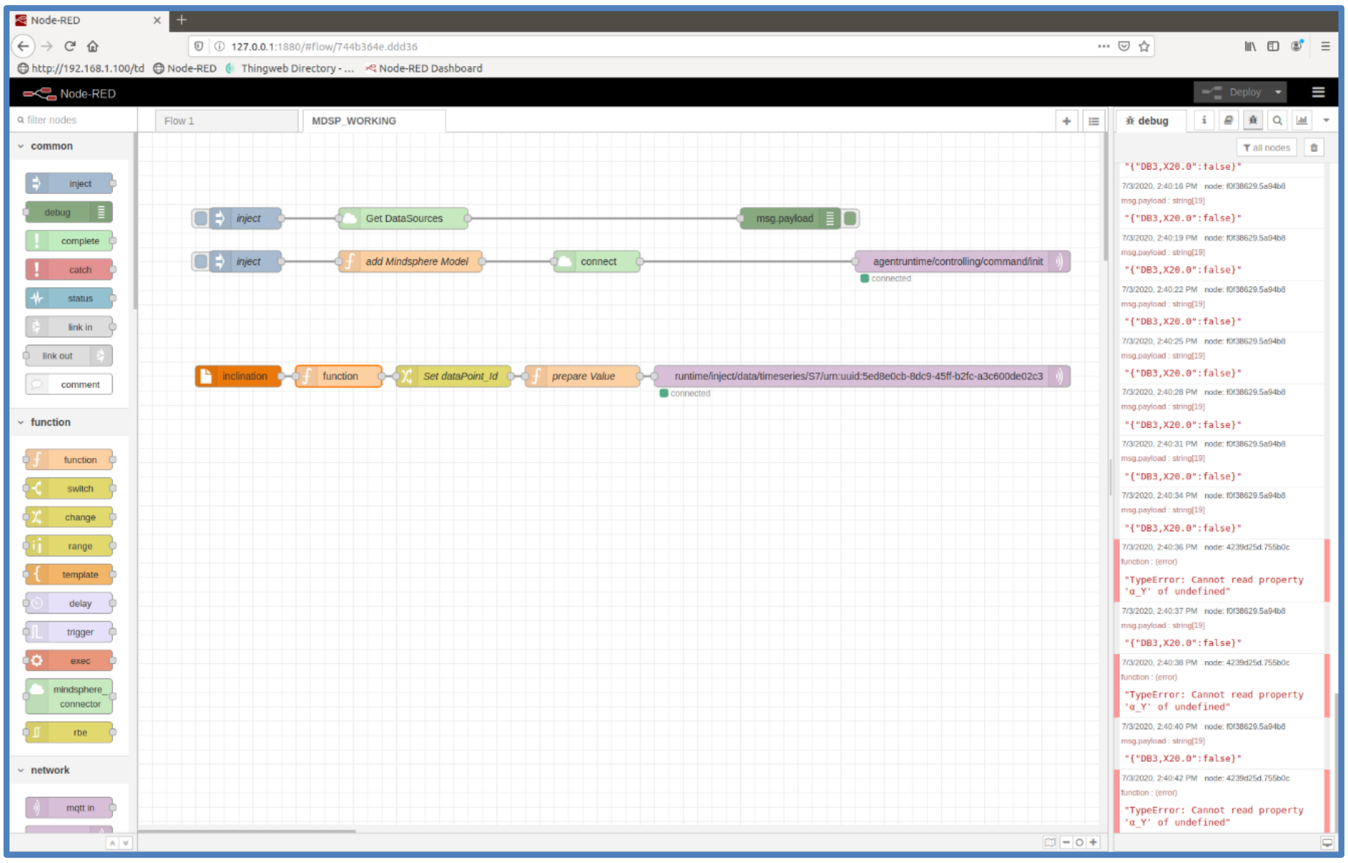


FIGURE 16: INTEGRATION OF IIOT GATEWAY WITH MINDSPHERE CLOUD

The IIoT Gateway integrates field-level data and eases development of Edge applications. The data of the inclinometer sensor in Sub Use Case 3 is processed and visualized in the public MindSphere Cloud (MindSphere Cloud App). Therefore, we have extended IIoT Gateway to meet this requirement. Open Edge Device Kit (OEDK)[5] has been used for this purpose. Figure 16 depicts how a custom device (on the Edge level) may communicate with the MindSphere Cloud over OEDK. For example, the inclinometer data can be sent to OEDK via MQTT protocol and further from OEDK to MindSphere Cloud over HTTP (in fact over the secure HTTPS protocol). This communication pattern works for any field device that is registered by the IIoT Gateway.

[5] OEDK: https://developer.mindsphere.io/resources/openedge-devicekit/index.html

The data of an on-boarded field device can be forwarded to the MindSphere Cloud via the Semantic Edge Platform[6]. The latter provides a convenient user interface for interacting with field devices, the IIoT Gateway, and the Cloud (via OEDK). Figure 17 shows a Node-RED flow for testing Sub Use Case 3, i.e., the provision of inclinometer (field-level) data via gateway to the Cloud (MDSP Cloud).



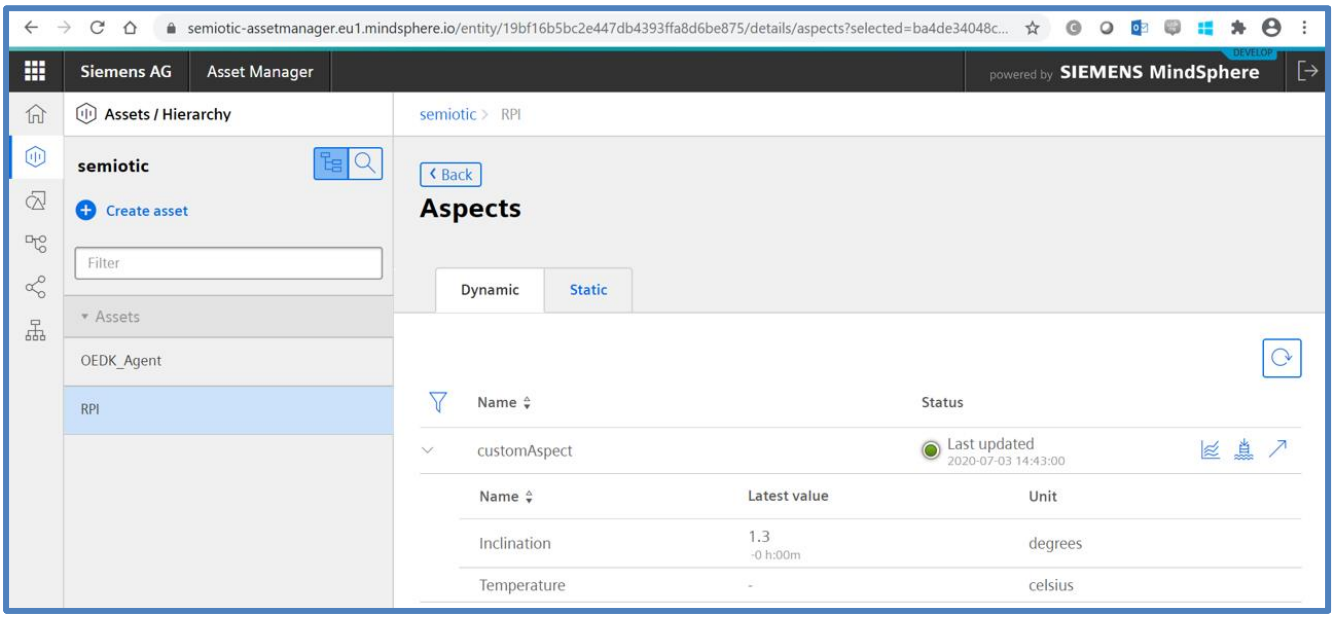
```
"{"DB3,X20.0":false}"
7/3/2020, 2:40:16 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:19 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:22 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:25 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:28 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:31 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:34 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:36 PM   node: 4239d25d.755b0c
function : (error)
"TypeError: Cannot read property
'α_Y' of undefined"
7/3/2020, 2:40:37 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:38 PM   node: 4239d25d.755b0c
function : (error)
"TypeError: Cannot read property
'α_Y' of undefined"
7/3/2020, 2:40:40 PM   node: f0f38629.5a94b8
msg.payload : string[19]
"{"DB3,X20.0":false}"
7/3/2020, 2:40:42 PM   node: 4239d25d.755b0c
function : (error)
"TypeError: Cannot read property
'α_Y' of undefined"
```

FIGURE 17: PROVISION OF INCLINOMETER DATA IN MINDSPHERE CLOUD OVER SEMANTIC EDGE PLATFORM

In general, the Cloud has the capability to integrate all kinds of data from the field. Thousands of inclinometers can be connected from wind farms to MindSphere (together with other types of sensors, e.g., for audio, video etc.). This brings up the question how to interpret all these different kinds of data in the Cloud. To this end, MindSphere requires a user to create an information model (meta-data)[7] for an asset before the data of the asset can be sent to the Cloud.

IIoT Gateway has the goal to provide the semantic integration of field-level data. Thus, the semantic meta-data, which the gateway integrates and provides at the Edge level, is used in the Cloud integration too. GW Semantic Mediator, which is a component of the IIoT Gateway, automatically creates the MindSphere information model based on a Thing Description. Figure 18 shows an automatically created MindSphere information model based on Thing Description for inclinometer and a temperature sensor.

[6] Semantic Edge Platform is a SEMIoTICS extension of Node-RED, see deliverables D3.3 and D3.9.
[7] https://documentation.mindsphere.io/resources/html/asset-manager/en-US/113537583883.html


semiotic-assetmanager.eu1.mindsphere.io/entity/19bf16b5bc2e447db4393ffa8d6be875/details/aspects?selected=ba4de34048c...
Siemens AG
Asset Manager
powered by SIEMENS MindSphere
Assets / Hierarchy
semiotic > RPI
semiotic
Create asset
Filter
Assets
OEDK_Agent
RPI
Back
Aspects
Dynamic
Static

Name	Status
customAspect	Last updated 2020-07-03 14:43:00

Name	Latest value	Unit
Inclination	1.3 -0 h:00m	degrees
Temperature	-	celsius

FIGURE 18: IIOT GATEWAY AUTOMATICALLY CREATES MINDSPHERE INFORMATION MODEL BASED ON W3C THING DESCRIPTION

Figure 19 shows a successful test, i.e., field data of the inclinometer is available in Cloud. The test demonstrates the integration of a field device, the IIoT Gateway, the SEMIoTICS SDN network, and the MindSphere Cloud. With this infrastructure in place, a SEMIoTICS user can, through simple interaction via the Semantic Edge Platform, onboard a field device and access its data in MindSphere.

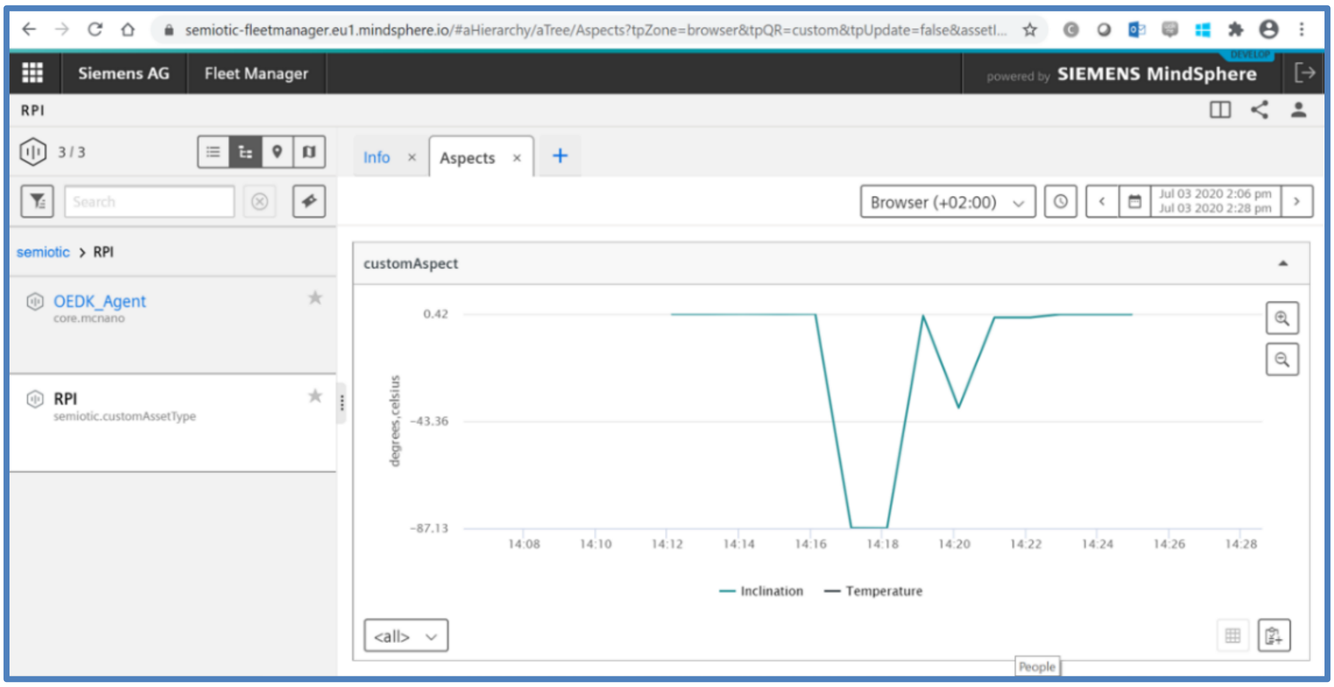


FIGURE 19: DATA OF INCLINOMETER AVAILABLE IN MINDSPHERE CLOUD

Finally, Figure 20 shows the Cloud app (MDSP App), which consumes the inclinometer data and triggers an alert when the data exceed a predefined threshold. The left-hand side of Figure 20 shows a list of on-boarded assets in the MindSphere Cloud. In the context of this demonstration, each of them represents a wind park. For the selected asset in Munich (Perlach 31) we see locations of four wind turbines (Figure 20). Furthermore, we see a time series data from our inclinometer. In the right-bottom part of Figure 20 we see the inclinometer sensor and the sensed movement around its axes. The application monitors the stream of inclinometer data and, for a user-defined threshold, triggers an alert. An engineer first gets a warning. If the problem does not get solved and the condition of the tower gets worse, then the user gets an alert. The stop signal is sent to SEMIoTICS IIoT Gateway, which in turn issues a command to stop the turbine.

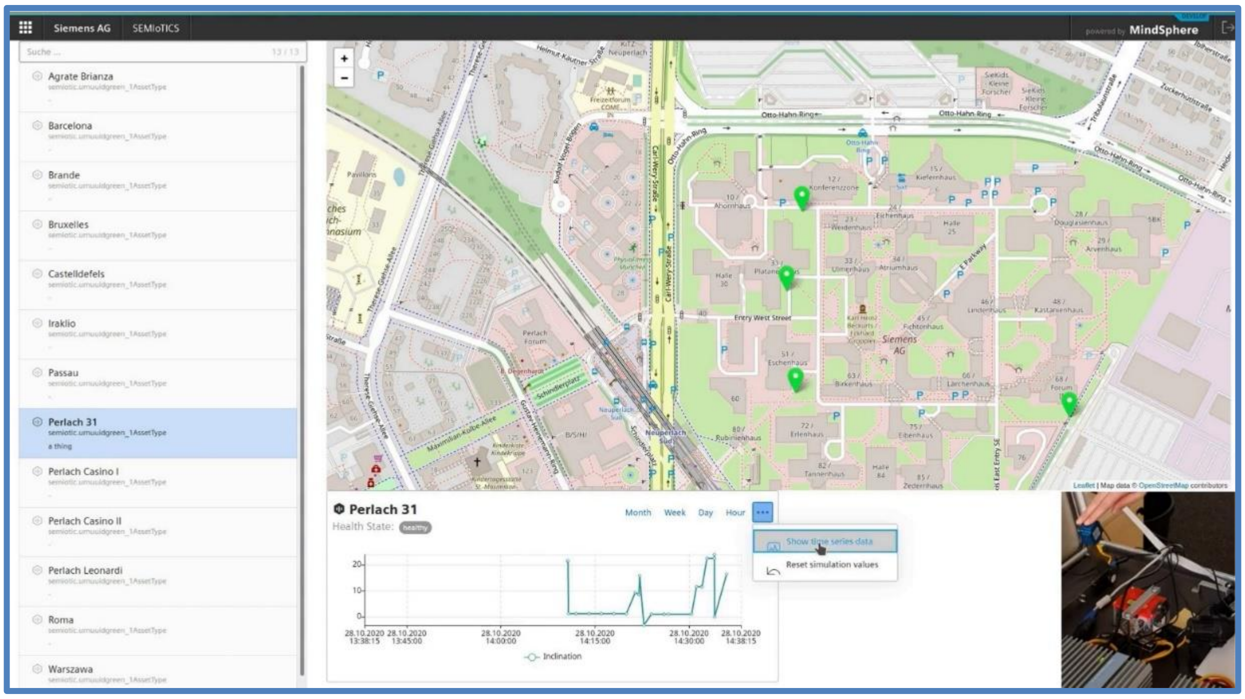


FIGURE 20: CONDITION MONITORING AND PREDICTIVE MAINTENANCE APPLICATION IN MINDSPHERE

7 SUB USE CASE 4: FIELD DEVICES AVAILABILITY MONITORING

7.1 Scope and Objectives

Continuous availability of field devices, such as sensors and actuators, enrolled to the SEMIoTICS eco-system, is a prerequisite for critical applications – whether instantiated in the cloud or at the edge level.

In this scenario, it's foreseen that the integrator or operator of the wind farm is responsible for ensuring the availability and security of the field devices enrolled, whereas the applications deployed that rely on said devices to generate value, is defined by other actors, e.g., business intelligence experts, data scientists, or external applications that are instantiated on license from an application marketplace.

In this context, the fourth Sub Use Case focuses on demonstrating interoperability and seamless integration with an 'availability monitoring application', defined and instantiated in the private cloud space, simulating a third-party integrator responsible for the availability monitoring of field devices enrolled to the SEMIoTICS eco-system.

All field devices used throughout the UC1 demonstrator are being monitored and the availability status is displayed in a global user interface hosted in the private cloud. Additionally, in order to guarantee that data in-transit is not accessible nor modifiable by unauthorized parties, i.e., to guarantee confidentiality and integrity of said data, the involved components are forced to use encryption when it is detected that encryption is not used. As the field devices are registered to an MQTT Broker, as per Figure 21, enforcing encryption is possible by interacting with the Broker.

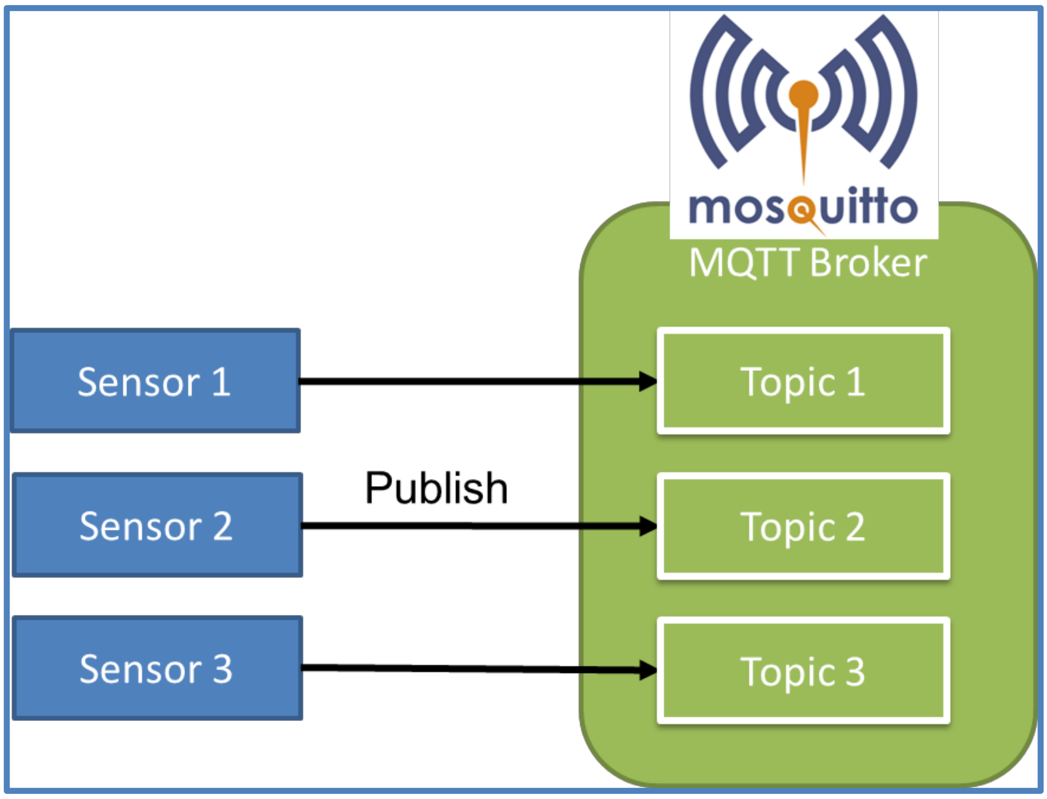


FIGURE 21: INTERACTION OF FIELD DEVICES IN SUB USE CASE 3

7.2 Interaction with SEMIoTICS Framework and Components

The components of the SEMIoTICS framework that are used in this sub use case are highlighted in Table 4. The backend components include the Recipe Cooker, the Pattern Orchestrator, and the Pattern Engine, with the SEMIoTICS GUI at the application layer. While initially a Pattern Engine was also deployed at the field layer, it was finally considered not crucial for the execution of the specific scenario and its role was adopted by the Pattern Engine deployed at the backend. The deployment of a lightweight Pattern Engine at the field layer is showcased in the context of UC3.

The Recipe Cooker is used to relay to the Pattern Orchestrator a specific recipe which contains the sensors, the MQTT Broker and the requirement for encrypted communication between them.

The Pattern Orchestrator, in turn, is responsible for automated configuration, coordination, and management of the different patterns needed to implement the Recipe, as well as their deployment to the various Pattern Engines. To achieve this, the Pattern Orchestrator processes received recipes; the result of the said processing enables the Pattern Orchestrator to choose which Pattern Engine should receive the corresponding Drools facts. Additionally this processing includes the generation of the REST endpoint required by the SEMIoTICS GUI to visualize the status of the relevant SPDI properties.

TABLE 4: SEMIOTICS COMPONENTS USED IN SUB USE CASE 4

Component	**SEMIoTICS component**	**Architecture reference**	**Description of use**
Recipe Cooker	✓	Application Layer	• Recipe Cooker framework based on Node Red is used to define and deploy the business logic to the Gateway and Backend Pattern Engines using Pattern Orchestrator as a proxy.
Pattern Orchestrator	✓	Backend Layer	• Pattern Orchestrator is used to translate the recipe (business logic request) into a set of patterns comprising Security requirements.
Backend Pattern Engine	✓	Backend Layer	• Pattern Engine at the backend layer is used to toggle encryption on the MQTT broker in the public cloud based on triggered Security pattern by the Pattern Orchestrator.
Backend GUI	✓	Backend Layer	• Used to visualize the SPDI properties' status, as received by the Pattern Orchestrator, including the adaptation of patterns applied to health monitoring connection.

The Pattern Engine at the backend layer is used for gathering all the facts from the Pattern Orchestrator regardless which layer are referred to. This enables the Pattern Engine at the SEMIoTICS Backend to have a complete view of the current status of the SPDI properties across layers. All the application-related pattern rules are also hosted at this engine; specifically in this Sub Use Case, this includes the encryption rules that must be enforced to protect all interactions with the MQTT Broker. Finally, the GUI is used for visualizing the status of the SPDI properties through the Pattern Orchestrator's corresponding REST endpoint.

Figure 22 shows the sequence of interactions involved in the current scenario and the messages exchanged between components in order to carry out the functionality comprising the Sub Use Case.

The Recipe Cooker initiates the flow by sending the recipe to the Pattern Orchestrator. As already described, the recipe contains the field devices, the MQTT Broker and the requirement for encryption between the communication of the sensors and the MQTT Broker. The Pattern Orchestrator, after processing the recipe, translates the recipe "ingredients" to Drools facts and sends them to the Backend Pattern Engine. For every fact the Pattern Engine receives, it tries to match it to a rule, triggering the associated reasoning process.

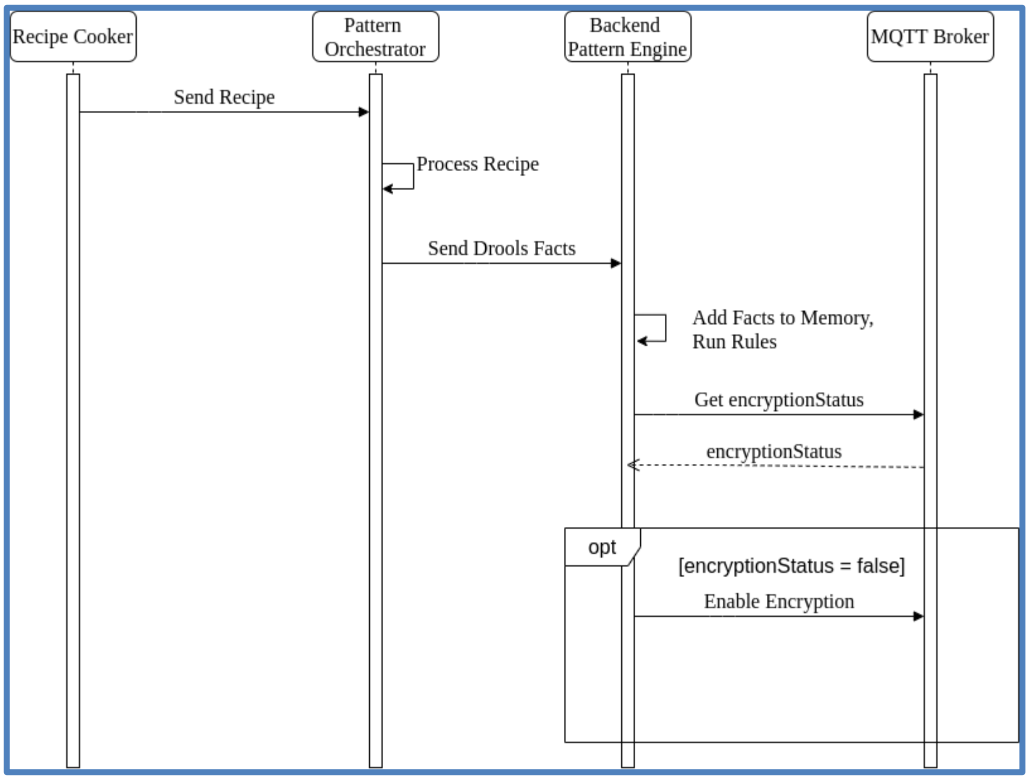
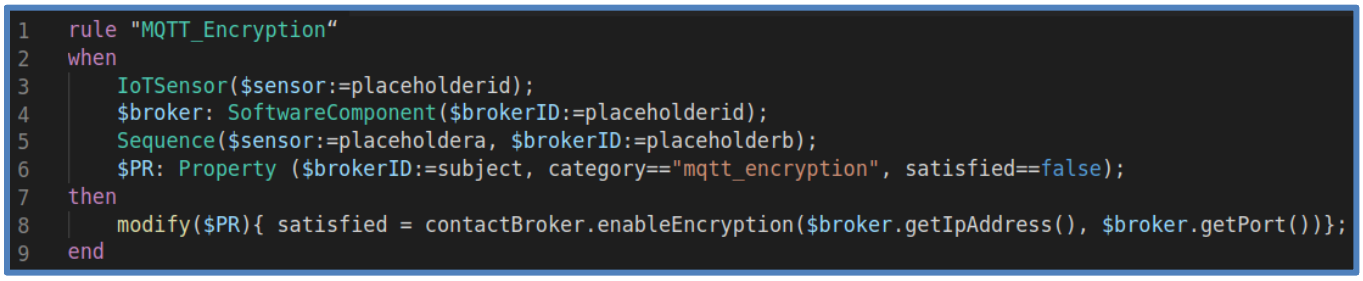


FIGURE 22: SEQUENCE DIAGRAM OF THE SECURITY ASPECT (UC1 - SUB UC4)

As soon as all the necessary facts are received, the encryption rule is triggered as per Figure 23.



```
rule "MQTT_Encryption“
when
    IoTSensor($sensor:=placeholderid);
    $broker: SoftwareComponent($brokerID:=placeholderid);
    Sequence($sensor:=placeholdera, $brokerID:=placeholderb);
    $PR: Property ($brokerID:=subject, category=="mqtt_encryption", satisfied==false);
then
    modify($PR){ satisfied = contactBroker.enableEncryption($broker.getIpAddress(), $broker.getPort())};
end
```

FIGURE 23: MQTT ENCRYPTION RULE

The **when** part of the rule specifies:

1. An IoT sensor.
2. A software component that represents the MQTT Broker.
3. A sequence of the sensor and the MQTT Broker indicating their interaction.
4. The orchestration property that can be guaranteed through the application of the pattern, i.e., the encryption property in this case ($PR).

The **then** part triggers adaptation actions by contacting the Broker and forcing him to use encryption.

For more details on the pattern specification and reasoning approach followed in SEMIoTICS, we refer the reader to deliverable D4.8.

7.3 Setup Testbed and Integration

A testbed was set up to integrate and test the security aspects of the sub use case detailed above and to implement the sequence diagram shown in Figure 22. The technologies used to implement this include Java, Maven and Spring Boot for the development and creation of the Pattern Engine. Moreover, as detailed in deliverable D4.8, the Pattern Engine integrates the Drools Business Rules Management System (BRMS) solution to implement its reasoning capabilities. Finally, the Eclipse Mosquitto was extended with a restful service based also in Java, Maven and Spring Boot.

Initially in FORTH's premises the Pattern Orchestrator, Backend Pattern Engine and MQTT Broker are deployed in VMs. An Intel NUC (Figure 24) is used with 32GB RAM, 500GB storage and a CPU i7-6770HQ 2.60GHz with 8 cores. The Proxmox Virtual Environment is installed in the Intel NUC which hosts the following VMs:

- Pattern Orchestrator VM
  - 10 GB storage
  - 4 GB RAM
  - 4 CPU cores.
  - Ubuntu 18.04.4 LTS operating system
- Pattern Engine VM
  - 10 GB storage
  - 4 GB RAM
  - 4 CPU cores.
  - Ubuntu 18.04.4 LTS operating system
- MQTT Broker VM
  - 10 GB storage
  - 2 GB RAM
  - 4 CPU cores.
  - Ubuntu 18.04.4 LTS operating system

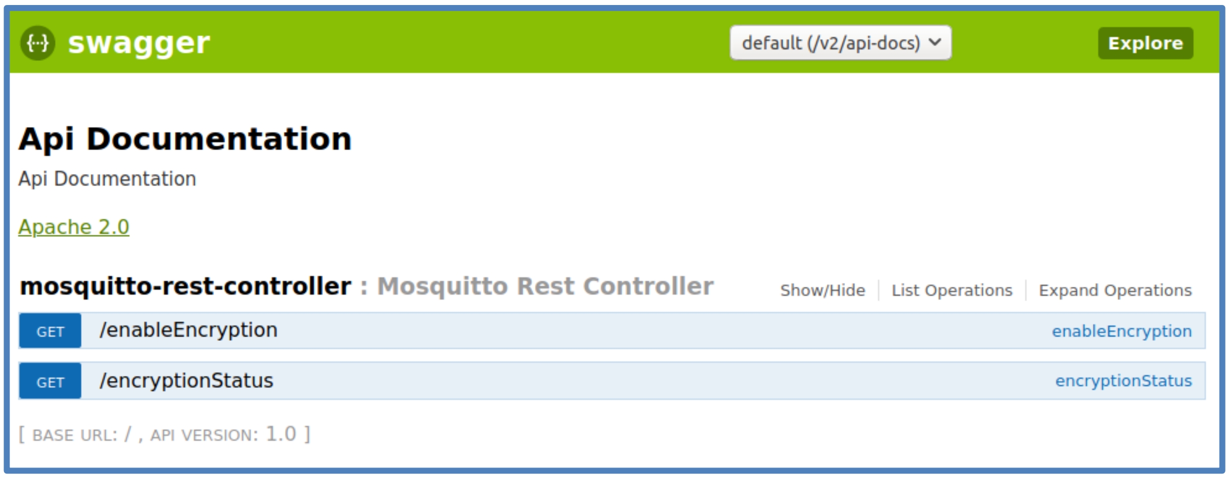
FIGURE 24: THE INTEL NUC

After all the local testing was successful, the Pattern Orchestrator and the Pattern Engine were transferred in Siemens premises, in the form of Docker images inside the NanoBox (Figure 1), to create the UC1 integrated testbed.

7.4 Validation

The above testbed allowed validating the functionality of the components by implementing the sequence diagram in Figure 22. Figure 25 shows the two REST endpoints implemented on the Broker. In the configuration of the MQTT Broker, all the related information for incoming connections such as port, protocol certificates etc., is grouped and the terminology used to describe that group is "listener". The "enableEncryption" alters the configuration of the Broker in such a manner that it will include certificates for the available listener for incoming connections, whereas the "encryptionStatus" verifies whether such certificates are present in the configuration.

Additionally, the REST endpoints are protected with self-signed certificates generated at the Broker side to guarantee encrypted communication. At the Backend Pattern Engine, the appropriate certificate is also installed in order to trust the Broker, thus fulfilling the requirement R.GSP.4.



Api Documentation

Api Documentation

Apache 2.0

mosquitto-rest-controller : Mosquitto Rest Controller

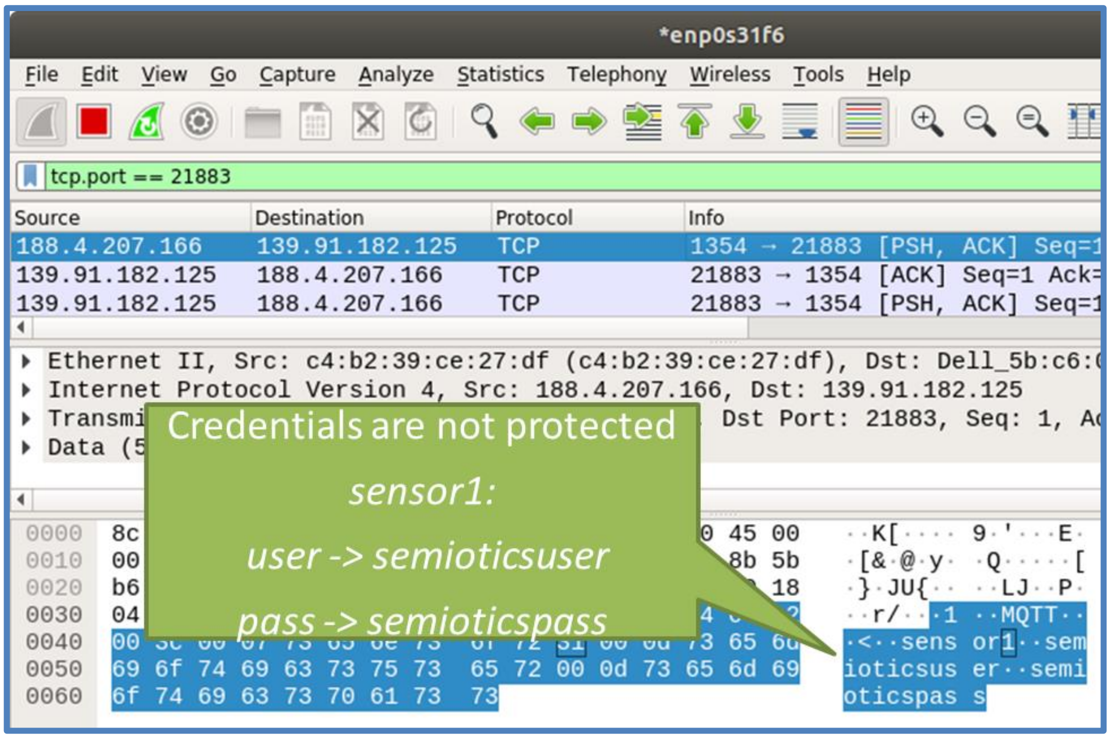
GET /enableEncryption enableEncryption

GET /encryptionStatus encryptionStatus

[BASE URL: / , API VERSION: 1.0]

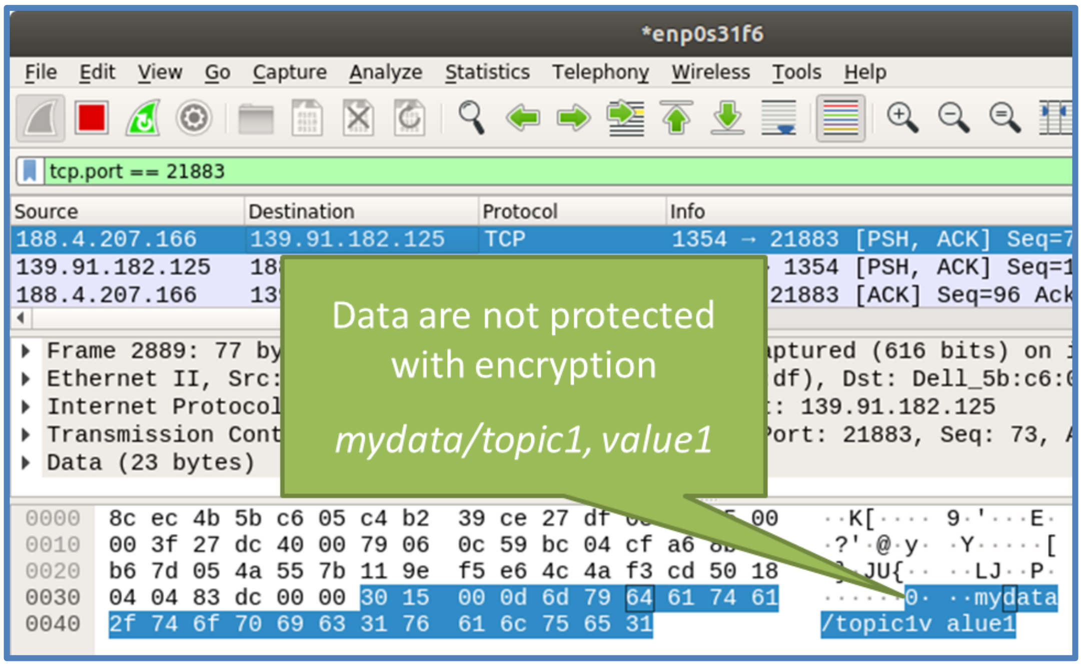
FIGURE 25: MQTT BROKER REST APIS

Figure 26 and Figure 27 shows the capture of packets from the Broker side before encryption is enabled. At this point all the communication is unencrypted, thus making the credentials exchange as well as all other data exchanges readable for passive attackers listening in on the connection.


*enp0s31f6
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Credentials are not protected
sensor1:
user -> semioticsuser
pass -> semioticspass

```
tcp.port == 21883

Source            Destination       Protocol   Info
188.4.207.166     139.91.182.125    TCP        1354 → 21883 [PSH, ACK] Seq=1
139.91.182.125    188.4.207.166     TCP        21883 → 1354 [ACK] Seq=1 Ack=
139.91.182.125    188.4.207.166     TCP        21883 → 1354 [PSH, ACK] Seq=1

▸ Ethernet II, Src: c4:b2:39:ce:27:df (c4:b2:39:ce:27:df), Dst: Dell_5b:c6:
▸ Internet Protocol Version 4, Src: 188.4.207.166, Dst: 139.91.182.125
▸ Transmi                                  Dst Port: 21883, Seq: 1, Ac
▸ Data (5

0000  8c                                        0 45 00    ··K[····  9·'···E·
0010  00                                        8b 5b    ·[&·@·y·  ·Q·····[
0020  b6                                           18    ·}·JU{··  ··LJ··P·
0030  04                                        4                ··r/···1  ··MQTT··
0040  00                       31 00 0d 73 65 6d    ·<··sens or1··sem
0050  69 6f 74 69 63 73 75 73  65 72 00 0d 73 65 6d 69    ioticsus er··semi
0060  6f 74 69 63 73 70 61 73  73                         oticspas s
```

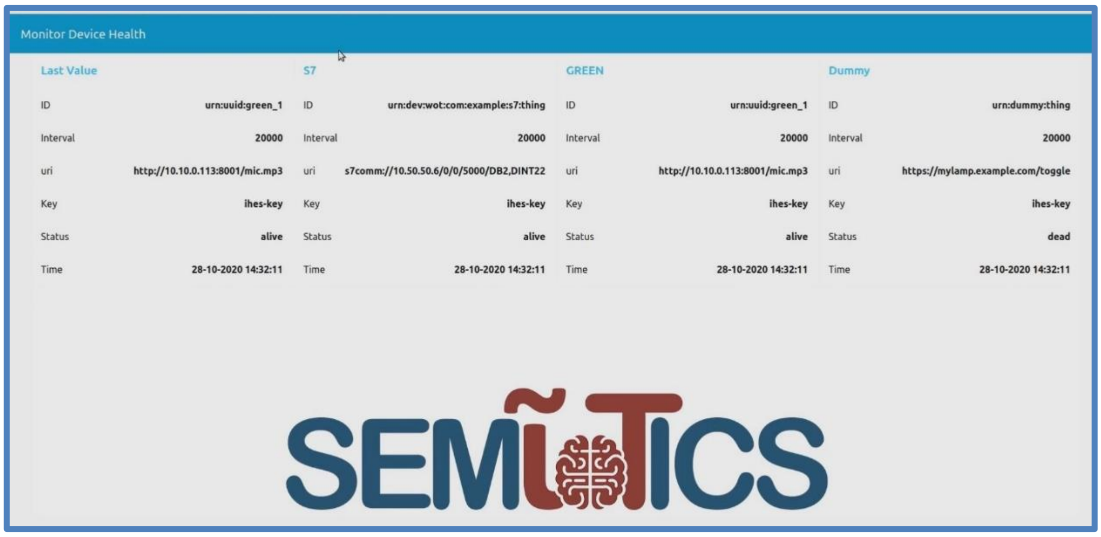
FIGURE 26: CREDENTIALS EXCHANGE


*enp0s31f6
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Data are not protected
with encryption
mydata/topic1, value1

```
tcp.port == 21883

Source            Destination       Protocol   Info
188.4.207.166     139.91.182.125    TCP        1354 → 21883 [PSH, ACK] Seq=7
139.91.182.125    18                           1354 [PSH, ACK] Seq=1
188.4.207.166     13                           21883 [ACK] Seq=96 Ack

▸ Frame 2889: 77 by                   ptured (616 bits) on i
▸ Ethernet II, Src:                   df), Dst: Dell_5b:c6:
▸ Internet Protocol                   : 139.91.182.125
▸ Transmission Cont                   Port: 21883, Seq: 73, A
▸ Data (23 bytes)

0000  8c ec 4b 5b c6 05 c4 b2  39 ce 27 df          00    ··K[····  9·'···E·
0010  00 3f 27 dc 40 00 79 06  0c 59 bc 04 cf a6          ·?'·@·y·  ·Y·····[
0020  b6 7d 05 4a 55 7b 11 9e  f5 e6 4c 4a f3 cd 50 18       JU{··  ··LJ··P·
0030  04 04 83 dc 00 00 30 15  00 0d 6d 79 64 61 74 61    ······0·  ··mydata
0040  2f 74 6f 70 69 63 31 76  61 6c 75 65 31             /topic1v alue1
```

FIGURE 27: DATA EXCHANGE

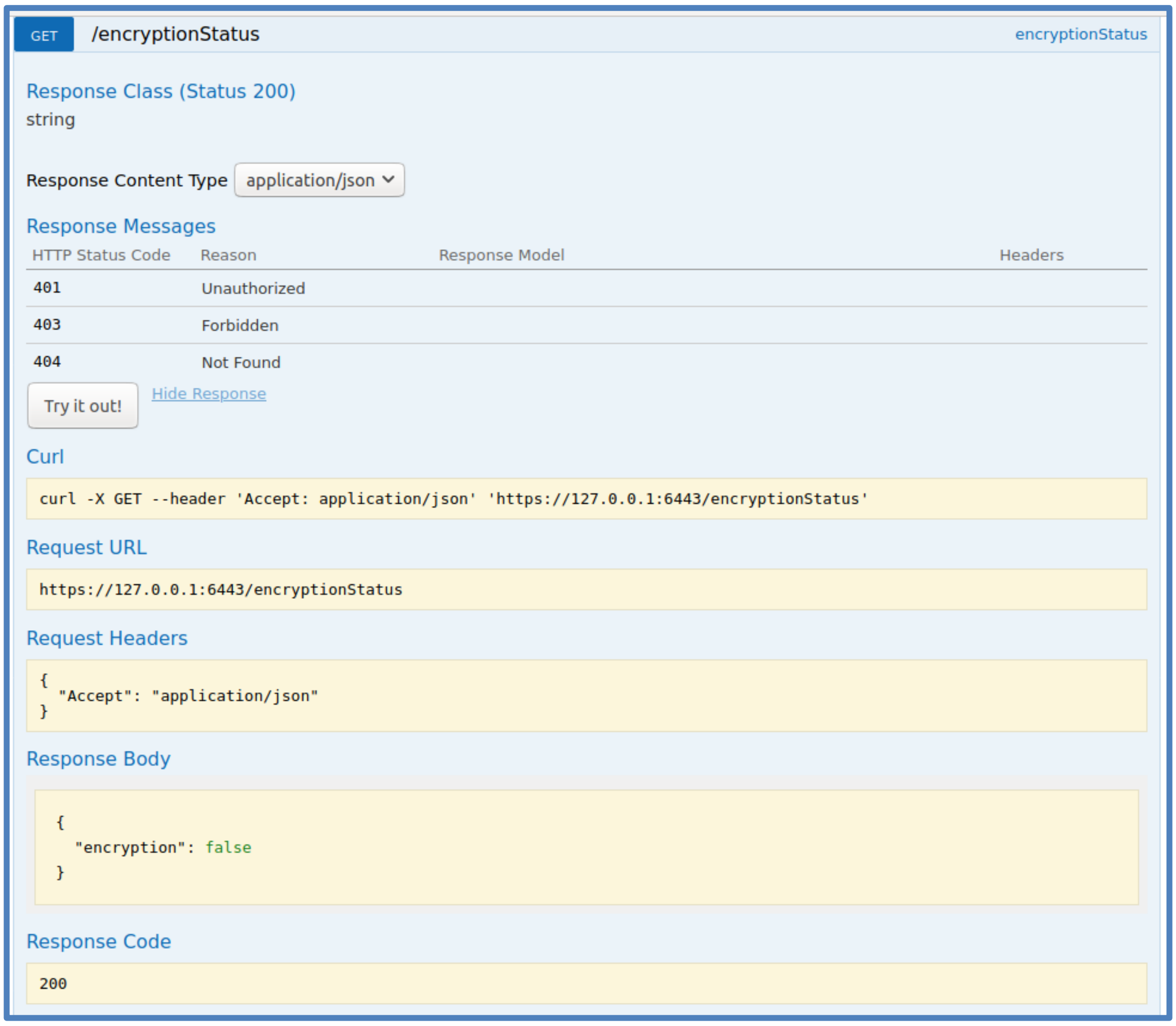
As data is published to the MQTT Broker, an InfluxDB is populated with this data. We can see the latest data for each of the three sensors in Figure 28, as well as the latest value inserted to the InfluxDB. This data, among others, includes information regarding the availability of the sensors with the field “status” and the timestamp of the latest value with the field “Time”.


Monitor Device Health

Last Value		S7		GREEN		Dummy	
ID	urn:uuid:green_1	ID	urn:dev:wot:com:example:s7:thing	ID	urn:uuid:green_1	ID	urn:dummy:thing
Interval	20000	Interval	20000	Interval	20000	Interval	20000
uri	http://10.10.0.113:8001/mic.mp3	uri	s7comm://10.50.50.6/0/0/5000/DB2,DINT22	uri	http://10.10.0.113:8001/mic.mp3	uri	https://mylamp.example.com/toggle
Key	ihes-key	Key	ihes-key	Key	ihes-key	Key	ihes-key
Status	alive	Status	alive	Status	alive	Status	dead
Time	28-10-2020 14:32:11	Time	28-10-2020 14:32:11	Time	28-10-2020 14:32:11	Time	28-10-2020 14:32:11

FIGURE 28 GUI FOR INFLUXDB DATA

Figure 29 shows the REST “encryptionStatus” that gives a response *encryption: false*.



Response Class (Status 200)
string

Response Content Type application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X GET --header 'Accept: application/json' 'https://127.0.0.1:6443/encryptionStatus'
```

Request URL

```
https://127.0.0.1:6443/encryptionStatus
```

Request Headers

```
{
  "Accept": "application/json"
}
```

Response Body

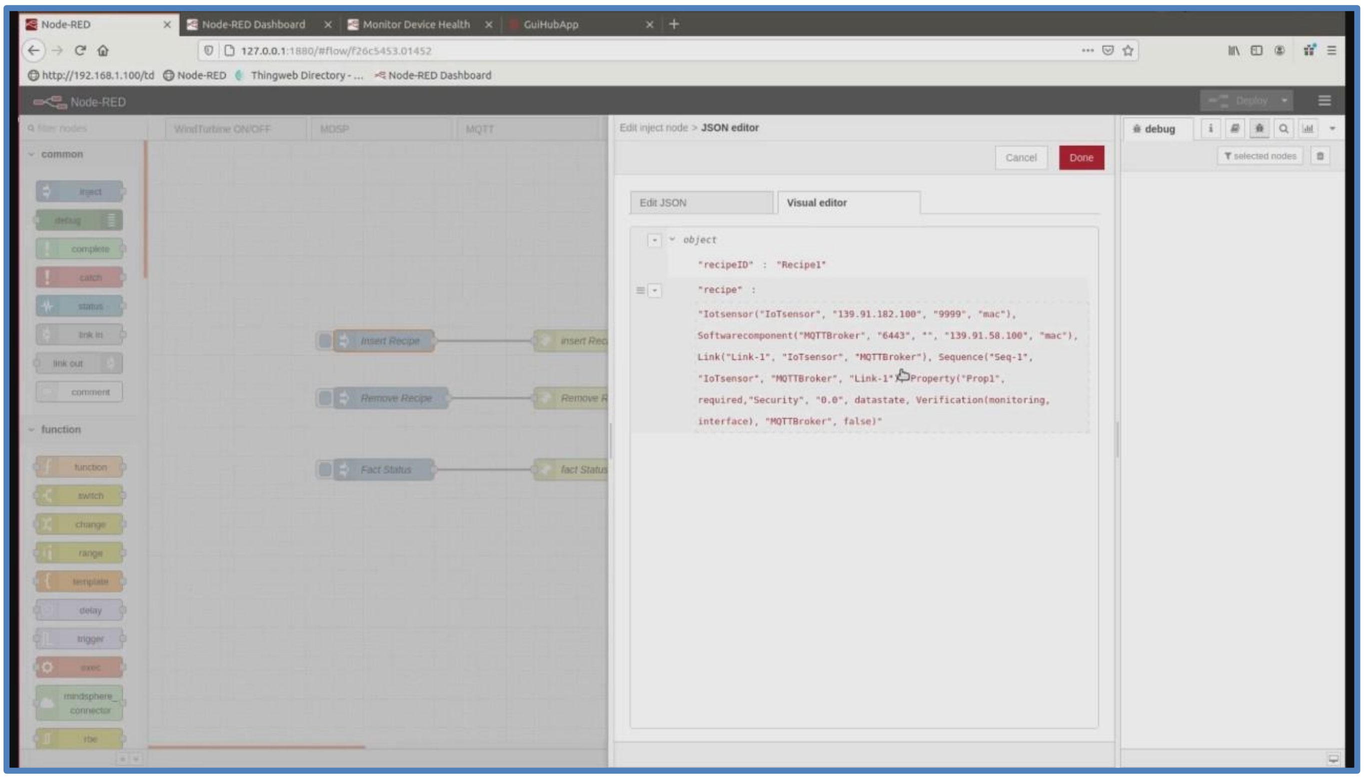
```
{
  "encryption": false
}
```

Response Code

```
200
```

FIGURE 29: ENCRYPTION STATUS BEFORE PATTERN REASONING

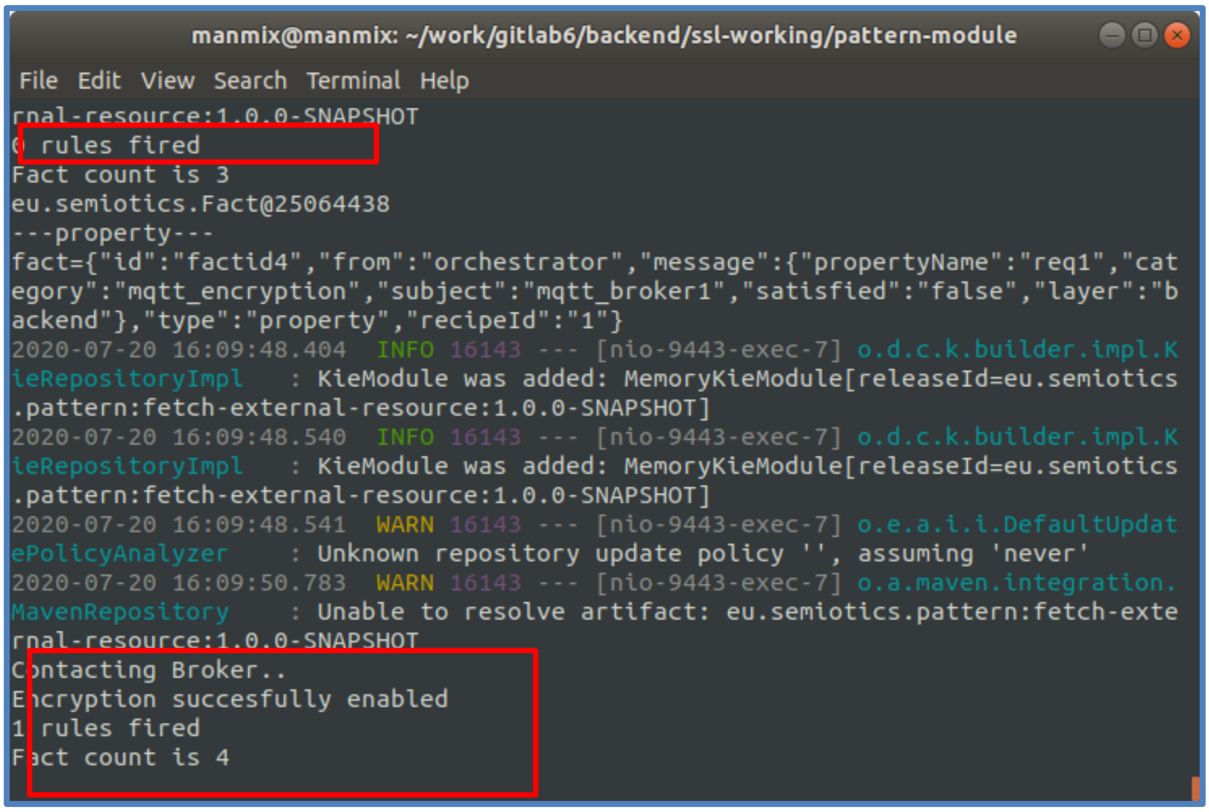
As mentioned, we want to enforce secure communications on our sensors; so, we create a new recipe with the “Security” Property on the Broker. This recipe is sent to the Pattern Orchestrator via an HTTP request, using the Recipe Cooker interface, as shown in Figure 30.



```
object
    "recipeID" : "Recipe1"
    "recipe" :
    "Iotsensor("IoTsensor", "139.91.182.100", "9999", "mac"),
    Softwarecomponent("MQTTBroker", "6443", "", "139.91.58.100", "mac"),
    Link("Link-1", "IoTsensor", "MQTTBroker"), Sequence("Seq-1",
    "IoTsensor", "MQTTBroker", "Link-1"), Property("Prop1",
    required,"Security", "0.0", datastate, Verification(monitoring,
    interface), "MQTTBroker", false)"
```

FIGURE 30: SECURE BROKER COMMUNICATION RECIPE SPECIFICATION

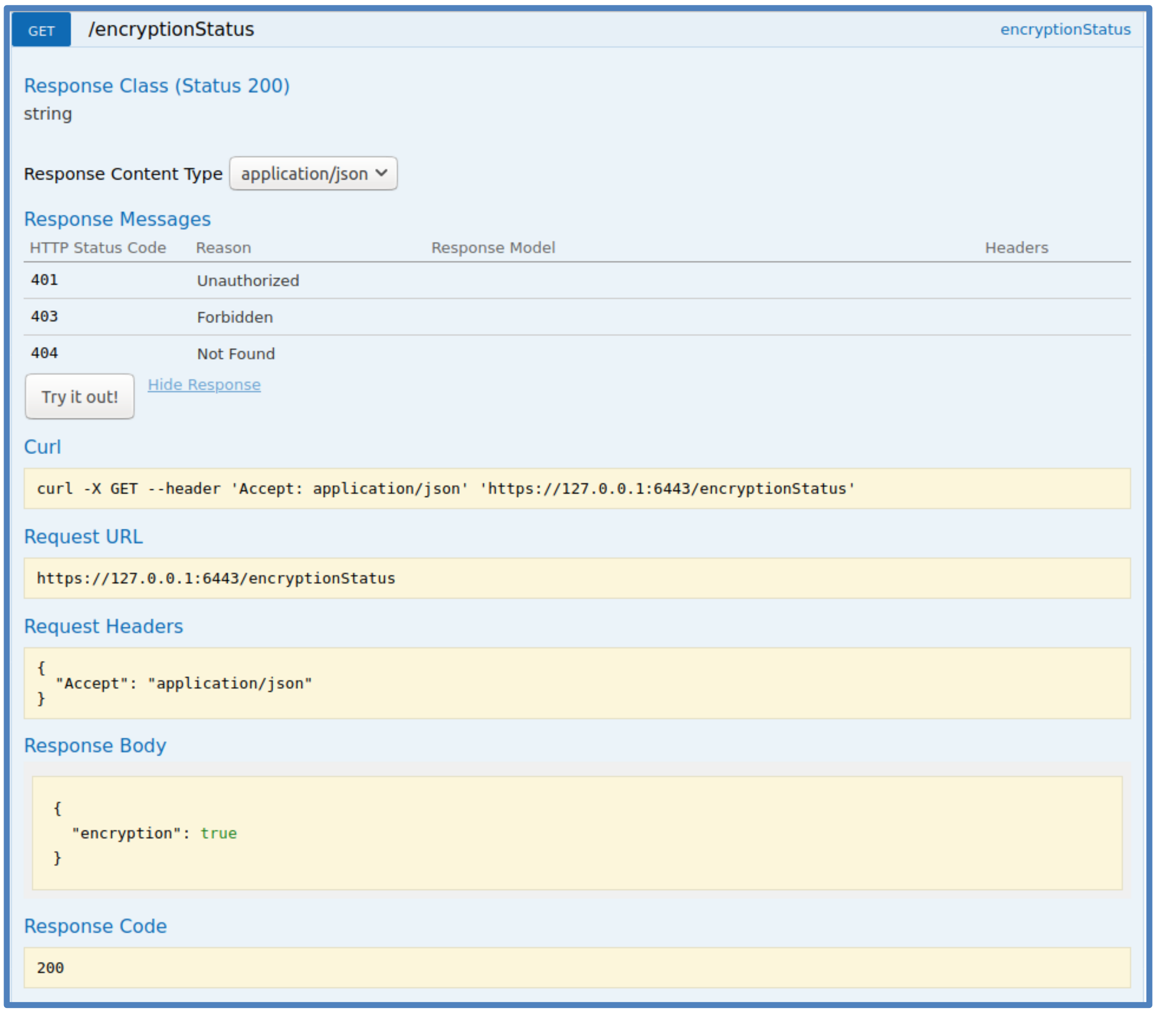
While the facts inserted to the Backend Pattern Engine do no satisfy any rule, the output of the Pattern Engine is “0 rules fired” as we see in Figure 31. As soon the final required fact is inserted, the rules are triggered, and the Pattern Engine initiates communication with the MQTT Broker. After that, the application on the Broker side makes the appropriate changes to the configuration and encryption is enabled. The response of the MQTT Broker, stating that the encryption is enabled, is also shown in Figure 31.


manmix@manmix: ~/work/gitlab6/backend/ssl-working/pattern-module
File Edit View Search Terminal Help

```
rnal-resource:1.0.0-SNAPSHOT
0 rules fired
Fact count is 3
eu.semiotics.Fact@25064438
---property---
fact={"id":"factid4","from":"orchestrator","message":{"propertyName":"req1","cat
egory":"mqtt_encryption","subject":"mqtt_broker1","satisfied":"false","layer":"b
ackend"},"type":"property","recipeId":"1"}
2020-07-20 16:09:48.404  INFO 16143 --- [nio-9443-exec-7] o.d.c.k.builder.impl.K
ieRepositoryImpl    : KieModule was added: MemoryKieModule[releaseId=eu.semiotics
.pattern:fetch-external-resource:1.0.0-SNAPSHOT]
2020-07-20 16:09:48.540  INFO 16143 --- [nio-9443-exec-7] o.d.c.k.builder.impl.K
ieRepositoryImpl    : KieModule was added: MemoryKieModule[releaseId=eu.semiotics
.pattern:fetch-external-resource:1.0.0-SNAPSHOT]
2020-07-20 16:09:48.541  WARN 16143 --- [nio-9443-exec-7] o.e.a.i.i.DefaultUpdat
ePolicyAnalyzer     : Unknown repository update policy '', assuming 'never'
2020-07-20 16:09:50.783  WARN 16143 --- [nio-9443-exec-7] o.a.maven.integration.
MavenRepository     : Unable to resolve artifact: eu.semiotics.pattern:fetch-exte
rnal-resource:1.0.0-SNAPSHOT
Contacting Broker..
Encryption succesfully enabled
1 rules fired
Fact count is 4
```

FIGURE 31: PATTERN ENGINE OUTPUT

As a confirmation, when we call the REST */encryptionStatus* again (Figure 32), we get a different response, this time indicating that the encryption is enabled on the MQTT Broker.



Response Class (Status 200)
string

Response Content Type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X GET --header 'Accept: application/json' 'https://127.0.0.1:6443/encryptionStatus'
```

Request URL

```
https://127.0.0.1:6443/encryptionStatus
```

Request Headers

```
{
  "Accept": "application/json"
}
```

Response Body

```
{
  "encryption": true
}
```

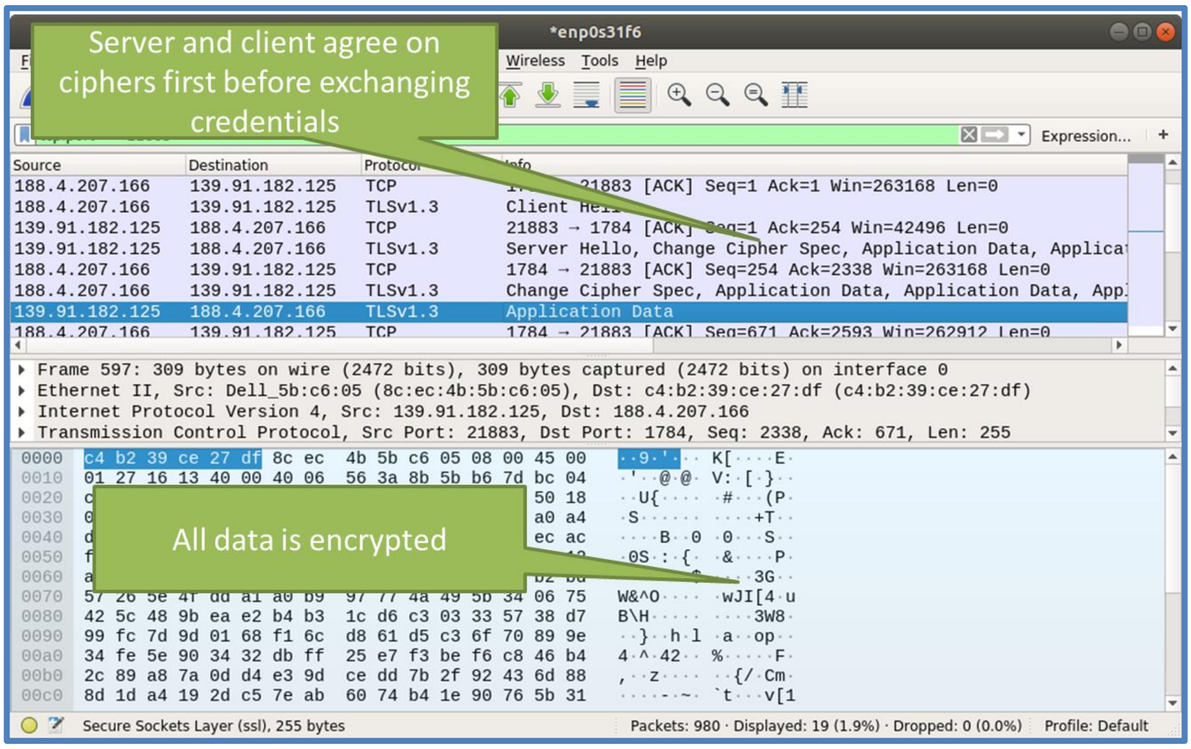
Response Code

```
200
```

FIGURE 32: ENCRYPTION STATUS AFTER PATTERN REASONING

At this point the connection between the sensors and the MQTT Broker does not work anymore because it does not use the encryption. As soon as we configure them to use encryption the communication is restored.

We further verify the security of communications with the capture of packets from the Broker side after the encryption is enabled. At this point all the communication is encrypted with TLS1.3, making the credentials exchange as well as any data exchange unreadable, as per Figure 33.


*enp0s31f6
Wireless Tools Help
Expression...
Server and client agree on ciphers first before exchanging credentials
All data is encrypted

```
Source          Destination     Protocol    Info
188.4.207.166   139.91.182.125  TCP         21883 [ACK] Seq=1 Ack=1 Win=263168 Len=0
188.4.207.166   139.91.182.125  TLSv1.3     Client Hel
139.91.182.125  188.4.207.166   TCP         21883 → 1784 [ACK] Seq=1 Ack=254 Win=42496 Len=0
139.91.182.125  188.4.207.166   TLSv1.3     Server Hello, Change Cipher Spec, Application Data, Applica
188.4.207.166   139.91.182.125  TCP         1784 → 21883 [ACK] Seq=254 Ack=2338 Win=263168 Len=0
188.4.207.166   139.91.182.125  TLSv1.3     Change Cipher Spec, Application Data, Application Data, App
139.91.182.125  188.4.207.166   TLSv1.3     Application Data
188.4.207.166   139.91.182.125  TCP         1784 → 21883 [ACK] Seq=671 Ack=2593 Win=262912 Len=0

▸ Frame 597: 309 bytes on wire (2472 bits), 309 bytes captured (2472 bits) on interface 0
▸ Ethernet II, Src: Dell_5b:c6:05 (8c:ec:4b:5b:c6:05), Dst: c4:b2:39:ce:27:df (c4:b2:39:ce:27:df)
▸ Internet Protocol Version 4, Src: 139.91.182.125, Dst: 188.4.207.166
▸ Transmission Control Protocol, Src Port: 21883, Dst Port: 1784, Seq: 2338, Ack: 671, Len: 255

0000  c4 b2 39 ce 27 df 8c ec  4b 5b c6 05 08 00 45 00   ··9·'···  K[····E·
0010  01 27 16 13 40 00 40 06  56 3a 8b 5b b6 7d bc 04   ·'··@·@·  V:·[·}··
0020  c                                      50 18   ··U{····  ·#···(P·
0030  0                                      a0 a4   ·S······  ····+T··
0040  d                                      ec ac   ····B··0  ·0···S··
0050  f                                              ·0S·:·{·  ·&····P·
0060  a                                                        ····3G··
0070  57 26 5e 4f dd a1 a0 b9  97 77 4a 49 5b 34 06 75   W&^O····  ·wJI[4·u
0080  42 5c 48 9b ea e2 b4 b3  1c d6 c3 03 33 57 38 d7   B\H·····  ····3W8·
0090  99 fc 7d 9d 01 68 f1 6c  d8 61 d5 c3 6f 70 89 9e   ··}··h·l  ·a··op··
00a0  34 fe 5e 90 34 32 db ff  25 e7 f3 be f6 c8 46 b4   4·^·42··  %·····F·
00b0  2c 89 a8 7a 0d d4 e3 9d  ce dd 7b 2f 92 43 6d 88   ,··z····  ··{/·Cm·
00c0  8d 1d a4 19 2d c5 7e ab  60 74 b4 1e 90 76 5b 31   ····-·~·  `t···v[1

Secure Sockets Layer (ssl), 255 bytes      Packets: 980 · Displayed: 19 (1.9%) · Dropped: 0 (0.0%)      Profile: Default
```

FIGURE 33: DATA EXCHANGE AFTER REASONING

In the SEMIoTICS GUI we also see that in the SPDI pattern monitoring view there is the Security Pattern Satisfied at the Backend Layer.

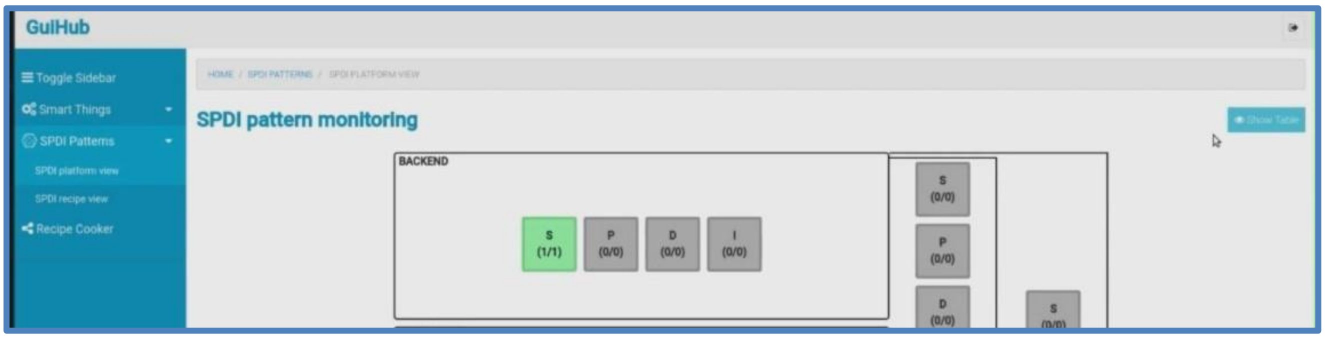


FIGURE 34: GUIHUB GRAPHICAL INTERFACE

8 OVERALL KPIS AND REQUIREMENTS VALIDATION

This section aims to provide a summary of requirements and KPIs relevant for evaluation in Use Case 1. The four presented Sub Use Cases, together with the remaining Use Cases 2 & 3 aim to address and validate the requirements presented in Table 5, while the relevant KPIs are presented in Table 6.

Further information on use case -specific requirements related to Use Case 1 are provided in D2.3. The original methodology of KPI evaluation contained in D5.1 was updated where required and is mostly summarized here for brevity.

TABLE 5: REQUIREMENTS PLACED ON USE CASE 1 AS PER D2.3

Req-ID	Description	Reference	Fulfilment
R.UC1.1, R.GP.1 (from D2.3)	Automatic establishment of networking setup MUST be performed to establish End-to-end connectivity between different stakeholders	Sections on Sub Use Cases 1 and 2 and SDN-enabled connectivity. Deliverables D3.1/D3.7.	This requirement is fulfilled and demonstrated by the interaction of Pattern Engine and Path Finding and reservation components in the SSC.
R.UC1.2	Automatic establishment of computing environment MUST be performed in IIoT Gateway for the minimum operation of the IIoT devices through network controller based on SDN/NFV	Sections on Sub Use Cases 1 and 2. Deliverables D3.2/D3.8.	Dynamic orchestration of computing resources is the focus of Use Case 2, SFC being the main highlighted benefit of the approach. Nevertheless, computing resources in UC1 are orchestrated by the Recipe Cooker when deploying machine learning workloads on NanoBox devices.
R.UC1.3	There MUST be enabled the definition of network QoS on application-level and automated translation into SDN controller configurations.	Section on SDN-enabled connectivity. Deliverables D3.1/D3.7.	This requirement is fulfilled and demonstrated by the interaction of Pattern Engine, Path Finding and Resource Manager components in the SSC, in Sub Use Cases 1 and 2.
R.UC1.4	Network resource isolation MUST be performed for guaranteed Service properties – i.e. reliability, delay and bandwidth constraints.	Section on SDN-enabled connectivity. Deliverables D3.1/D3.7.	This requirement is fulfilled and demonstrated by the Path Finding / Resource Manager modules of SSC, in Sub Use Cases 1 and 2.
R.UC1.8	Semantic and robust bootstrapping/registration of IIoT sensors and actuators with IIoT gateway MUST be supported.	Section on Sub Use Case 3. Deliverables D3.3/D3.9.	This requirement is fulfilled and demonstrated by the SEMIoTICS components of the IIoT Gateway and is demonstrated in Sub Use Cases 1, 2, 3.
R.UC1.9	Semantic interaction between use-case specific application on IIoT Gateway and legacy turbine control system MUST be supported.	Sections on Sub Use Cases 1, 2, and 3.	This requirement is fulfilled and demonstrated by the SEMIoTICS components of the IIoT Gateway and is demonstrated in Sub Use Cases 1 and 2 - automated turbine rotor stopping (using a brownfield PLC) based on greenfield sensor inputs.

R.UC1.10	Local analytical capability of IIoT Gateway to run machine learning algorithms (e.g. specific to 2 specific Sub Use cases)	Sections on Sub Use Cases 1 and 2, KPI evaluation (below).	This requirement is fulfilled and demonstrated by the video and audio analytics applications demonstrated in Sub Use Cases 1 and 2.
R.UC1.11	Device composition and application creation SHALL be supported through template approach.	Sections on Sub Use Cases 1, 2, and 4.	This requirement is fulfilled and demonstrated by the Recipe Cooker as Recipe (Template) orchestration tool and is demonstrated in Sub Use Cases 1, 2 and 4.
R.UC1.12	Standardized semantic models for semantic-based engineering and IIoT applications MUST be utilized.	Sections on Sub Use Cases 1, 2, and 3 (especially). Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by the integration of Thing descriptions (WoT), i.e., by semantic modelling in IIoT Gateway is demonstrated in Sub Use Cases 1, 2 and 3.
R.UC1.13	Middleware functionality MUST be supported on IIoT gateway, to deal with termination of IIoT sensors, signal processing and termination of interfaces to legacy systems to provide prioritization and QoS for IIoT applications.	Section on Sub Use Cases 3 and 4. Deliverables D3.3/D3.9. Section on SDN-enabled connectivity and D3.1/D3.7	The requirement of sensor flow termination is fulfilled and demonstrated by the scenario of information splitting in the IIoT Gateway and is demonstrated in Sub Use Cases 3 and 4. QoS and prioritization are demonstrated in Sub Use Cases 1 and 2.
R.UC1.5	Fail-over and highly available network management SHALL be performed in the face of either controller or data-plane failures.	Deliverables D3.1/D3.7.	This requirement is fulfilled, demonstrated and evaluated in a demonstrator and paper presented at ACM SIGCOMM 2019. Due to numerous other highlights of UC1, it has been excluded from demonstration.
R.UC1.6	Decisions made by unreliable, i.e. faulty or malicious SDN controllers, SHALL be identified and excluded.	Deliverables D3.1/D3.7.	This requirement is fulfilled, demonstrated and evaluated in a demonstrator and paper presented at ACM SIGCOMM 2019. Due to numerous other highlights of UC1, it has been excluded from demonstration.
R.UC1.7	The operation of the SDN control SHALL be scalable to cater for a massive IoT device integration and large-scale request handling in the SDN controller(s) using a (near-) optimal IoT client – SDN controller assignment procedure.	Deliverables D3.1/D3.7.	This requirement is fulfilled, demonstrated and evaluated in a demonstrator and papers presented at IEEE GLOBECOM 2019 and ACM SIGCOMM 2019. Due to numerous other highlights of UC1, it has been excluded from demonstration.
R.GP.2	Scalable infrastructure due to the fast-paced growth of IoT devices	Sections on Sub Use Cases 1, 2, 3 and 4.	This requirement is fulfilled and demonstrated in Sub Use Cases 1, 2, 3 and 4 by IoT orchestration via Recipe Cooker in Node-RED environment.
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication)	Sections on Sub Use Cases 1 and 2 and SDN-enabled connectivity. Deliverables D3.1/D3.4/D3.7/D3.10.	This requirement is fulfilled and demonstrated by the interaction of Pattern Engine, Path Finding and Resource Manager

			components in the SSC, in Sub Use Cases 1 and 2.
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern	Sections on SDN-enabled connectivity and Sub Use Cases 1 and 2. Deliverables D3.1/D3.7.	This requirement is fulfilled and demonstrated by Dependability (Connectivity) and Security pattern in Sub Use Cases 1 and 4.
R.GP.5	Interaction between SDN controller and IoT backend cloud through a dedicated interface (called northbound software interface)	Sections on Sub Use Cases 1 and 2 and SDN-enabled connectivity. Deliverables D3.1/D3.4/D3.7/D3.10.	This requirement is fulfilled and demonstrated by interaction between Pattern Orchestrator in backend and SSC.
R.GP.6	Interaction between SDN controller and network nodes (e.g. switches, routers or IoT Gateways) through dedicated interface (called southbound software interface)	Section on SDN-enabled connectivity. Deliverables D3.1//D3.7.	This requirement is fulfilled and demonstrated by SSC controller in Sub Use Cases 1, 2, 3 and 4.
R.BC.16	Prediction mechanism based on the data stored in the cloud	Section on Sub Use Case 3. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by Sub Use Case 3 via MindSphere – condition monitoring and predictive maintenance Apps.
R.BC.18	The backend layer must feature SPDI pattern reasoning embedded intelligence capabilities	Section on Sub Use Case 4. Deliverables D4.1/D4.8.	This requirement is fulfilled by the deployment and integration of pattern orchestration and reasoning capabilities at the backend (Pattern Orchestrator & Backend Pattern Engine), as showcased in sub-use case 4.
R.BC.19	The backend layer should feature pattern-driven cross-layer orchestration capabilities	Sections on Sub Use Cases 1, 2 and 4. Deliverables D4.1/D4.8.	This requirement is fulfilled and demonstrated by Recipe Cooker component implemented in Node-RED.
R.NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities	Section on SDN-enabled connectivity. Deliverables D3.1/D3.7.	This requirement is fulfilled and demonstrated by interaction of Pattern Engine in SSC and Pattern Orchestrator.
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation	Section on SDN-enabled connectivity. Deliverables D3.1/D3.7 & deliverables D4.1/D4.8.	This requirement is fulfilled by the integration of the SDN Pattern Engine within the SSC and demonstrated in UCs relying on the reasoning capabilities of the SSC.
R.FD.1	Field devices SHOULD be able to get data from the environment through sensors (sensors).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components in UC1 (i.e., camera, microphone and inclinometer sensors).
R.FD.2	Field devices SHOULD be able to process data in near real time (process units).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components in UC1 (real-time anomaly detection and actuation).
R.FD.3	Field devices SHOULD be able to control (at least) a mechanism / system (actuators).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components in UC1 (ref. legacy PLC controller).

R.FD.5	Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by semantic integration of brownfield / greenfield devices in the IIoT Gateway in UC1.
R.FD.6	Field devices MUST interoperate using a standard communication protocol like REST APIs, COAP, MQTT.	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components in UC1 (ref. MQTT broker and REST API exposure).
R.FD.7	Field devices MUST use standardize interoperable message format (e.g. JSON, etc.).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components in UC1.
R.FD.8	Field devices MUST support secure bootstrapping / registration protocol.	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by semantic bootstrapping in IIoT Gateway components.
R.FD.9	Field devices MUST be able to communicate with the IIoT Gateway / other architectural components.	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by IIoT Gateway components. This is possible by semantic integration of greenfield and brownfield devices (S7/MQTT/REST HTTP protocol support) .
R.FD.10	Field devices SHOULD minimize data traffic.	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by Local Analytics component in IIoT Gateway (ref. KPI-3.1.1 description below for further information).
R.FD.12	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).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by the inclinometer greenfield IoT device.
R.FD.13	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).	Sections on Sub Use Cases 3 and 4. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by field layer devices (S7 protocol for interacting with the PLC).
R.FD.15	The field layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation	Sections on Sub Use Cases 1 and 2. Deliverables D3.3 and D3.9.	This requirement is fulfilled and demonstrated by the audio and video anomaly detection applications.
R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.	Sections on Sub Use Case 4.	This requirement is fulfilled and demonstrated in tandem with SEMIoTICS' pattern-based reasoning and adaptation SPDI capabilities, enabling TLS in Sub Use Case 4
R.P.1	The collection of raw data MUST be minimized	Sections on Sub Use Cases 1 and 2.	This requirement is fulfilled by the Pattern Engines in all layers

			because no raw data is collected. Additionally, Sub Use Cases 1 & 2 demonstrate inference in field layer thus resulting in decrease of information transfer outside the local wind park.
R.P.3	Storage of data MUST be minimized.	Sections on Sub Use Cases 1 and 2.	This requirement is fulfilled by the Pattern Engines in all layers because no raw data is collected. Additionally, Sub Use Cases 1 & 2 demonstrate immediate inference in field layer thus resulting in decrease of the total information storage in the local wind park.
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised	N/A.	This requirement is fulfilled natively by Recipe Cooker, SSC, Pattern Orchestrator and IIoT Gateway that log all information of operation.
R.GSP.4	Platforms, e.g. cloud platform and sensor, SHALL be trusted.	Sections on Sub Use Case 4.	This requirement is fulfilled and demonstrated by the Backend Pattern Engine capable of enabling secure interactions of MQTT broker and MQTT clients on-demand.

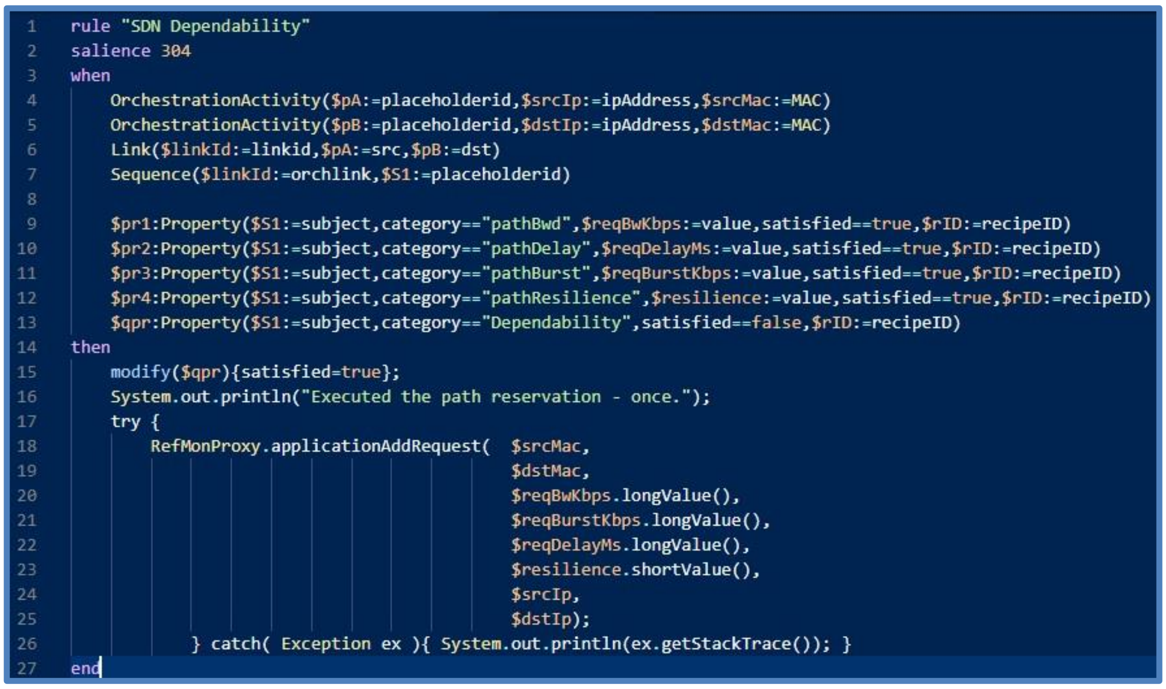
TABLE 6: KPIS VALIDATED / MEASURED IN USE CASE 1

KPI-ID	Name
KPI-1.1	Number of SPDI Patterns
KPI-2.1	Semantic descriptions for 6 types of smart objects
KPI-2.2	Data type mapping and ontology alignment
KPI-2.3	Semantic interoperability with 3 IoT platforms
KPI-3.1.1	Generating monitoring strategies in the 3 targeted IoT platforms
KPI-3.1.3	Performing predictive monitoring with an average accuracy of 80%
KPI-4.1	Delivery of lightweight ML algorithms
KPI-4.2	Delivery of mechanisms with adaptation time of 15ms
KPI-4.6	Development of new security mechanisms/controls
KPI-6.1	Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%
KPI-6.3	Delivery of 3 prototypes of IIoT/IoT applications
KPI-7.1	Provision the SEMIoTICS building blocks

8.1 KPI-1.1: Number of SPDI Patterns

This KPI is satisfied through the delivery of the final set of SEMIoTICS SPDI patterns that well exceed the initially specified KPI of 36 patterns, as detailed in the deliverable D4.8 – “SEMIoTICS SPDI Patterns (final)”.

In the context of UC1, the Dependability and Security patterns were used in sub use case 1 and sub use case 4, respectively. The Dependability pattern (Figure 35) was modified and was preinstalled in the SDN Pattern Engine while the Security pattern (Figure 37) was part of the Backend Pattern Engine.



```
rule "SDN Dependability"
salience 304
when
    OrchestrationActivity($pA:=placeholderid,$srcIp:=ipAddress,$srcMac:=MAC)
    OrchestrationActivity($pB:=placeholderid,$dstIp:=ipAddress,$dstMac:=MAC)
    Link($linkId:=linkid,$pA:=src,$pB:=dst)
    Sequence($linkId:=orchlink,$S1:=placeholderid)

    $pr1:Property($S1:=subject,category=="pathBwd",$reqBwKbps:=value,satisfied==true,$rID:=recipeID)
    $pr2:Property($S1:=subject,category=="pathDelay",$reqDelayMs:=value,satisfied==true,$rID:=recipeID)
    $pr3:Property($S1:=subject,category=="pathBurst",$reqBurstKbps:=value,satisfied==true,$rID:=recipeID)
    $pr4:Property($S1:=subject,category=="pathResilience",$resilience:=value,satisfied==true,$rID:=recipeID)
    $qpr:Property($S1:=subject,category=="Dependability",satisfied==false,$rID:=recipeID)
then
    modify($qpr){satisfied=true};
    System.out.println("Executed the path reservation - once.");
    try {
        RefMonProxy.applicationAddRequest(  $srcMac,
                                            $dstMac,
                                            $reqBwKbps.longValue(),
                                            $reqBurstKbps.longValue(),
                                            $reqDelayMs.longValue(),
                                            $resilience.shortValue(),
                                            $srcIp,
                                            $dstIp);
          } catch( Exception ex ){ System.out.println(ex.getStackTrace()); }
end
```

FIGURE 35: DEPENDABILITY PATTERN AT THE SDN

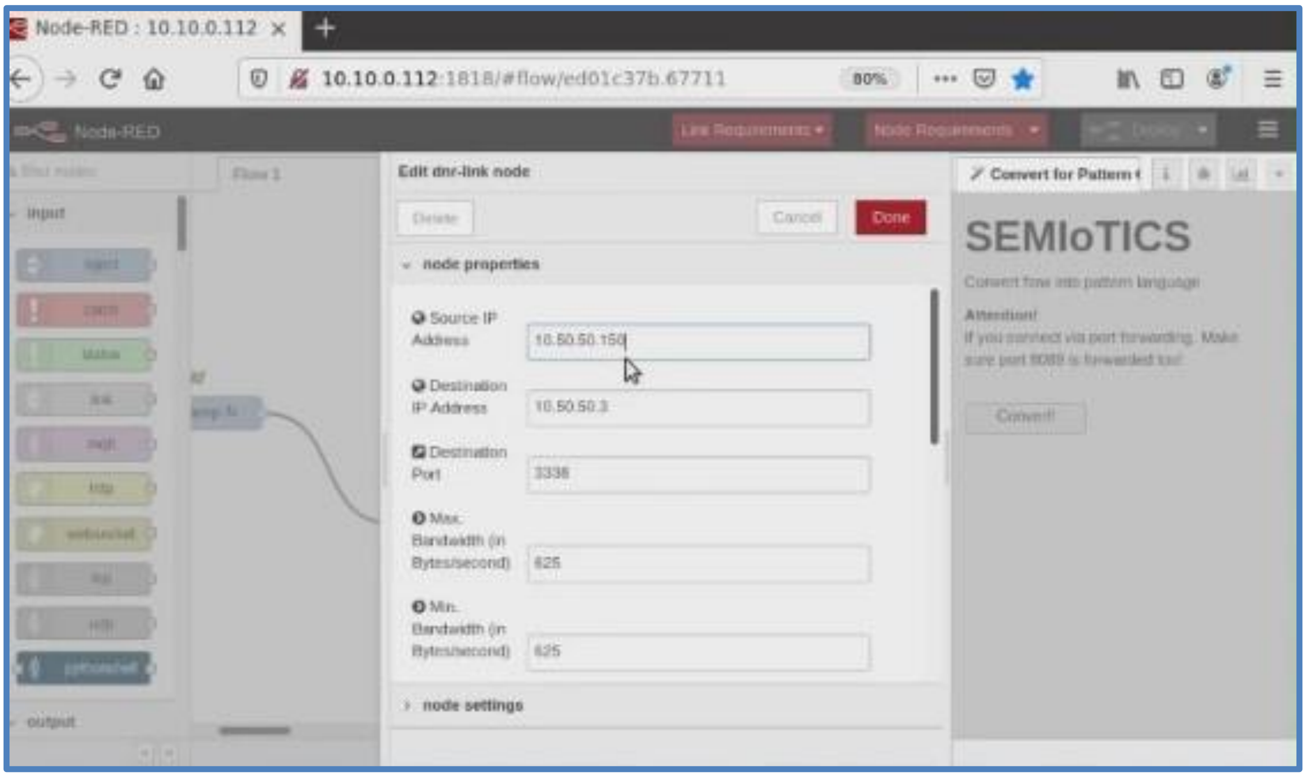
The Dependability pattern is modified in such a manner to include network QoS properties.

The **when** part of the rule specifies:

1. Two OrchestrationActivities that act as the two endpoints. The OrchestrationActivity is a parent class for Host, IoTSensor, IoTGateway etc, which enables the rule to be more abstract without the need to specify a new rule for every class.
2. A Link that represents the connection between the OrchestrationActivities.
3. A sequence of the OrchestrsatonActivities indicating their interaction.
4. Four QoS properties
5. The orchestration property that can be guaranteed through the application of the pattern, i.e., the dependability property in this case ($qpr).

The **then** part triggers the enforcement of the QoS properties which uses the SDN controller to generate in the switches per flow mapping configurations.

The QoS properties are provided by the Recipe Cooker (Figure 36) and are forwarded via the Pattern Orchestrator to the appropriate SEMIoTICS layer and the corresponding Pattern Engine.



Edit dnr-link node

Delete | Cancel | Done

node properties

Source IP Address: 10.50.50.150
Destination IP Address: 10.50.50.3
Destination Port: 3338
Max. Bandwidth (in Bytes/second): 625
Min. Bandwidth (in Bytes/second): 625

node settings

FIGURE 36: RECIPE COOKER QOS MAPPING

The Security pattern is a combination of three rules in the Backend Pattern Engine, Security decomposition, MQTT_Encryption and Security_verification.

Security decomposition:

The **when** part of the rule specifies:

1. A software component that represents the MQTT Broker.
2. The orchestration property that can be guaranteed through the application of the pattern, i.e., the security property in this case ($PR).

The **then** part creates a new property that we need to verify and that is the MQTT_Encryption of the software component.

MQTT_Encryption:

The **when** part of the rule specifies:

1. An IoT sensor.
2. A software component that represents the MQTT Broker.
3. A sequence of the sensor and the MQTT Broker indicating their interaction.
4. The orchestration property that can be guaranteed through the application of the pattern, i.e., the encryption property in this case ($PR).

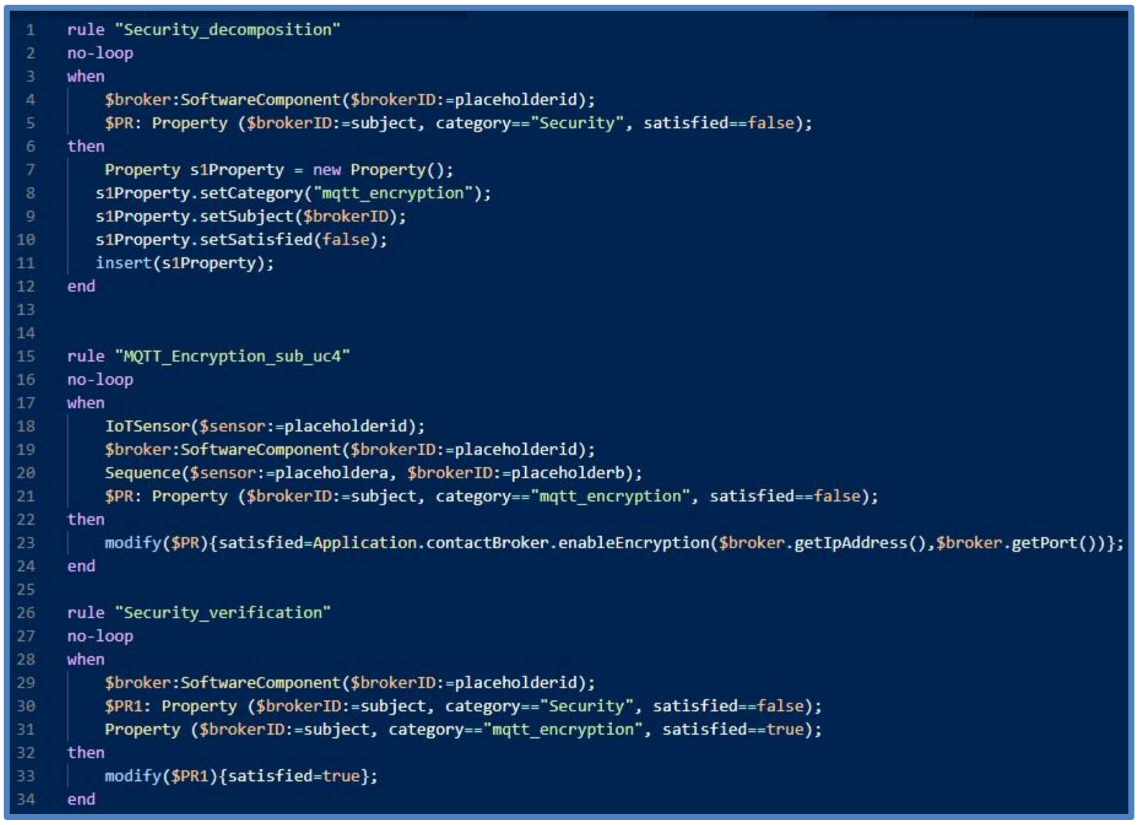
The **then** part triggers adaptation actions by contacting the Broker and forcing him to use encryption.

Security_verification

The **when** part of the rule specifies:

1. A software component that represents the MQTT Broker.
2. The orchestration property that can be guaranteed through the application of the pattern, i.e., the security property in this case ($PR1).
3. The property of MQTT_Encryption that must already be satisfied.

The `then` part modifies the status of the orchestration property to state that the property holds.



```
rule "Security_decomposition"
no-loop
when
    $broker:SoftwareComponent($brokerID:=placeholderid);
    $PR: Property ($brokerID:=subject, category=="Security", satisfied==false);
then
    Property s1Property = new Property();
   s1Property.setCategory("mqtt_encryption");
   s1Property.setSubject($brokerID);
   s1Property.setSatisfied(false);
   insert(s1Property);
end


rule "MQTT_Encryption_sub_uc4"
no-loop
when
    IoTSensor($sensor:=placeholderid);
    $broker:SoftwareComponent($brokerID:=placeholderid);
    Sequence($sensor:=placeholdera, $brokerID:=placeholderb);
    $PR: Property ($brokerID:=subject, category=="mqtt_encryption", satisfied==false);
then
    modify($PR){satisfied=Application.contactBroker.enableEncryption($broker.getIpAddress(),$broker.getPort())};
end

rule "Security_verification"
no-loop
when
    $broker:SoftwareComponent($brokerID:=placeholderid);
    $PR1: Property ($brokerID:=subject, category=="Security", satisfied==false);
    Property ($brokerID:=subject, category=="mqtt_encryption", satisfied==true);
then
    modify($PR1){satisfied=true};
end
```

FIGURE 37: SECURITY PATTERN AT THE BACKEND PATTERN ENGINE

In both the Dependability and the Security pattern, the recipe that will invoke the triggering of the rules is originating from the Recipe Cooker. The Pattern Orchestrator receives the translated recipe to SEMIoTICS pattern language and forwards the ingredients of the recipe as facts to the appropriate pattern engine for reasoning. The visualization of the pattern status occurs in the GUI component where the information is queried in standard intervals from the Pattern Orchestrator. More information on this process can be found in deliverable D4.8.

8.2 KPI-2.1 Semantic descriptions for 6 types of smart objects

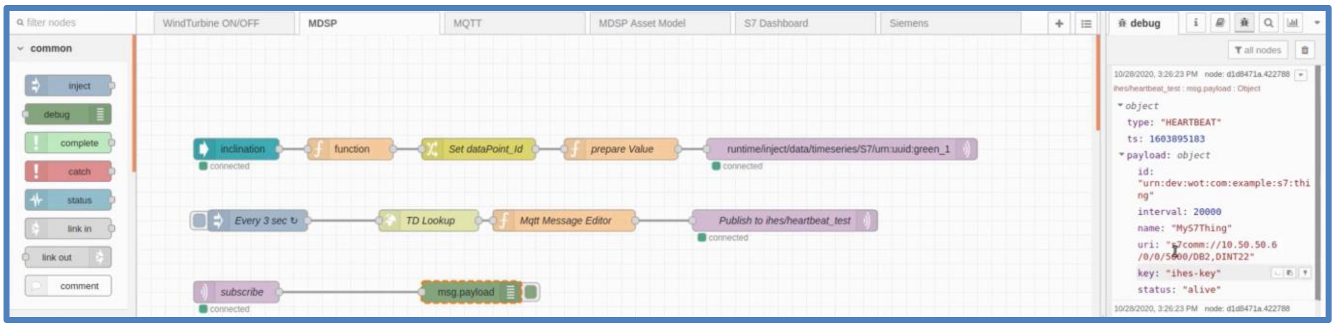
This KPI is satisfied as, within the project, semantic descriptions for six types of smart objects have been created. We have used W3C Thing Description (TD) standard for this purpose.

The Thing Descriptions which were created in SEMIoTICS include: (i) inclinometer, (ii) wind turbine, (iii) camera, (iv) microphone, (v) temperature sensing device, (vi) energy monitoring device, and (vii) multi-sensor node.

Two Thing Descriptions are provided in this document as examples, i.e., a TD for the wind turbine (a legacy device used in Use-Case 1) is shown in Appendix 10.1, and a TD for the inclinometer (a greenfield device used in Use-Case 1) is presented in Appendix 10.2.

In Deliverable D3.9 we provided information explaining how a TD is generated by SEMIoTICS IoT Gateway. Further on, we described how a TD can be used for creating applications. Thing Description is a cornerstone for making a field device programmatically accessible. The gateway will create a Device Node in Semantic Edge Platform for each interaction pattern from a TD. Figure 38 shows how an inclination

Device Node has been created from a TD provided in Appendix 10.2, and how this node has been used in an application to monitor Field Devices Availability (see Section 7).



```
10/28/2020, 3:26:23 PM   node: d1d8471a.422788
ihes/heartbeat_test : msg.payload : Object
▾object
  type: "HEARTBEAT"
  ts: 1603895183
 ▾payload: object
    id:
    "urn:dev:wot:com:example:s7:thi
    ng"
    interval: 20000
    name: "MyS7Thing"
    uri: "s7comm://10.50.50.6
    /0/0/5000/DB2,DINT22"
    key: "ihes-key"
    status: "alive"
10/28/2020, 3:26:23 PM   node: d1d8471a.422788
```

FIGURE 38: FIELD DEVICES AVAILABILITY MONITORING

8.3 KPI-2.2: Data type mapping and ontology alignment

The aim of this Use Case is the IIoT integration in Wind Park Control Network providing value-added services. Specifically, one of its parts refers to taking local action on sensing and analyzing structured data to find the inclination of a steel tower.

However, at runtime, a possible change in resource availability may require an adaptation mechanism for the dynamic recompositing of the flow components, without any change of the system behavior. We assume that the above flow has an Inclinometer component. If, for a reason, the Inclinometer becomes unavailable, another component from the IoT repository with the same functionality is selected to replace the one that has become unavailable. In this case, interoperability of the link between the replaced component and the connected flow component should be checked before the final decision for the replacement to ensure the initial target of the system. In particular, a potential conflict that needs to be addressed in case of the Inclinometer is the different unit of measurement of angle (degrees, radians), which is used by the two components (the replaced and the connected flow component).

The solution in this interoperability adaptation issue is with the participation of the Backend Semantic Validator as a core component. It examines the interoperability between the corresponding sensors and tries to ensure the interaction between them creating Adaptor Nodes. Figure 39 highlights and explains said methodology in the specific Use Case scenario.

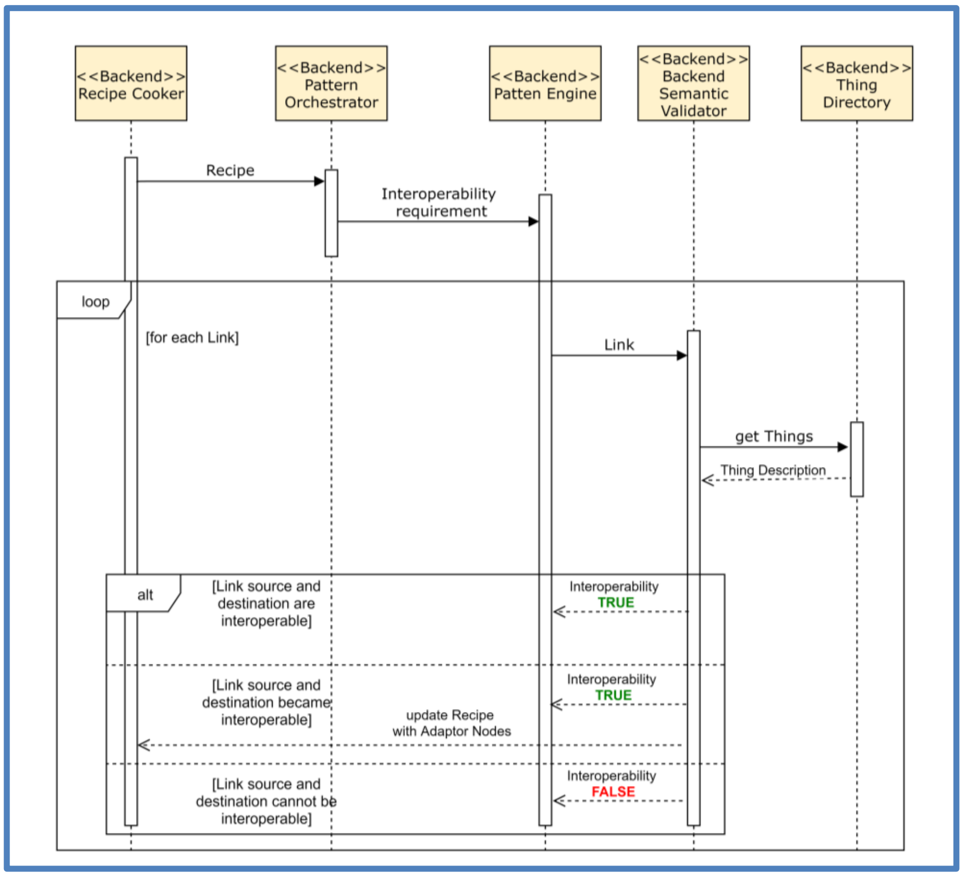

<<Backend>> Recipe Cooker
<<Backend>> Pattern Orchestrator
<<Backend>> Patten Engine
<<Backend>> Backend Semantic Validator
<<Backend>> Thing Directory
Recipe
Interoperability requirement
loop
[for each Link]
Link
get Things
Thing Description
alt
[Link source and destination are interoperable]
Interoperability TRUE
[Link source and destination became interoperable]
Interoperability TRUE
update Recipe with Adaptor Nodes
[Link source and destination cannot be interoperable]
Interoperability FALSE

FIGURE 39: DATA MAPPING APPROACH

Even though it was conceptually described in D4.11 Backend Semantic Validator is not presented in the UC1 due to focusing on other aspects of SEMIoTICS architecture. Nevertheless, the Backend Semantic Validator is presented in UC2.

In order to ensure interoperability from the application definition all the way through to the execution at runtime, we have implemented four levels of abstraction and accordingly three steps of transformation between them. These steps are indicated in below Figure 40.

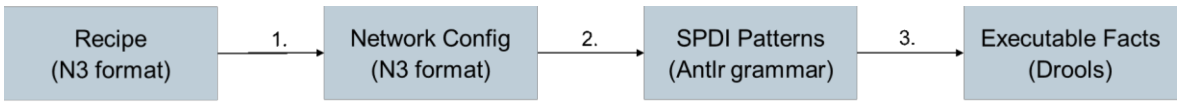

Recipe (N3 format)
1.
Network Config (N3 format)
2.
SPDI Patterns (Antlr grammar)
3.
Executable Facts (Drools)

FIGURE 40: TRANSLATIONS FROM RECIPES TO EXECUTABLE RULES

Recipes are designed in the Recipe Cooker component and serialized in the N3 language. Then, we have implemented various N3-based rules that can transform the Recipe into a network configuration. An overview of these 2 models (Recipe and Network model) and their transformation is shown in Figure 41.

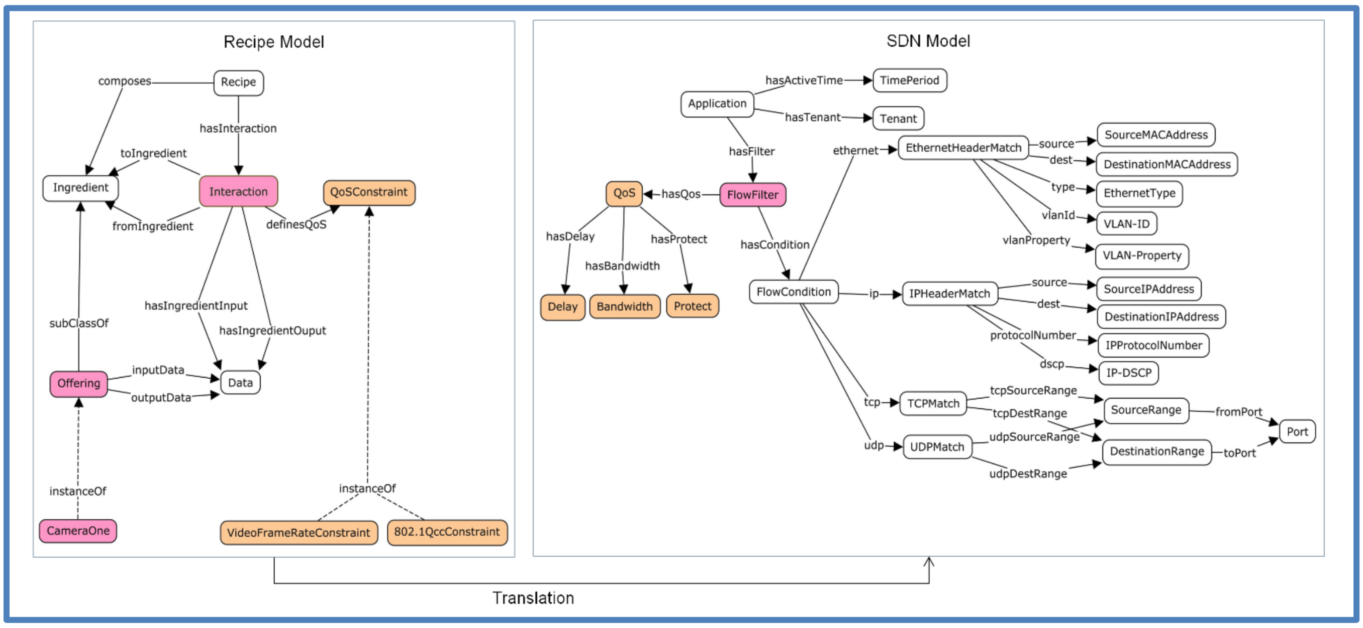


FIGURE 41: TRANSFORMATION FROM RECIPE TO NETWORK CONFIGURATION

In Table 7 a few semantics are presented related to SDN patterns. In the LHS, the components which constitute the topology of the pattern are defined. Different facts such as hosts, and links are included in the list. Moreover, the Property represents the constraints of the topology and the required property. In the RHS, the pattern provides the solution by inserting, modifying, updating or retracting facts from the knowledge base which will also update the inventory list in the controller. Each component is converted through the respective Java class to an understandable format to the Pattern Engine.

TABLE 7: PATTERN RULE SEMANTICS

Type	**Syntax**	**Description**
rule	rule "name"	name of the rule
Left Hand Side (LSH)		
when	**Network Pattern Elements (Facts)**	
	Host (mac, activityAddress)	match network nodes such as switches and hosts
	Link (srcId, destId)	match links between source and destination nodes
	Property (subject, category, satisfied)	match requirements of pattern such as subject (e.g. host), property category (e.g. Dependability) and satisfied (e.g. true/false)
	Conditional Elements	
	==	match conditions
	contains	contains object (logical)
	not	not match (logical)
	!=	not match (arithmetic)
Right Hand Side (RSH)		
then	**Actions**	

	modify (\$fact)\{pro=pro'\}	modify knowledge base fact
	retract (\$fact)	retract knowledge base fact
	insert (new Fact ())	insert knowledge base fact
	update (\$fact)	update knowledge base fact
	Java commands	other Java language syntax

8.4 KPI-2.3: Semantic interoperability with 3 IoT platforms

The semantic interoperability between the IIoT Gateway and two IoT platforms has been achieved. These are the public MindSphere Cloud and the private InfluxDB Cloud. Integration and interoperability with a third Cloud were not required by this use case. Thus, we focused on tighter integration and interoperability between the Field and Edge as well as between the two Cloud deployments. Each of these Clouds runs a UC-specific application. Figure 14 depicts details about the two applications, which are based on the data from the inclinometer. The user initializes both applications over Semantic Edge Platform (IIoT Gateway) as shown in Figure 13. The first application continuously monitors the sway of the tower and detects anomaly. This application is deployed in the MindSphere Cloud. The second application monitors the health status of the inclinometer and visualizes this data. The application is deployed in a private InfluxDB Cloud. Both applications use the data from the Field level, integrated by the Gateway at the Edge level, and pushed to two different Clouds, see Figure 42.

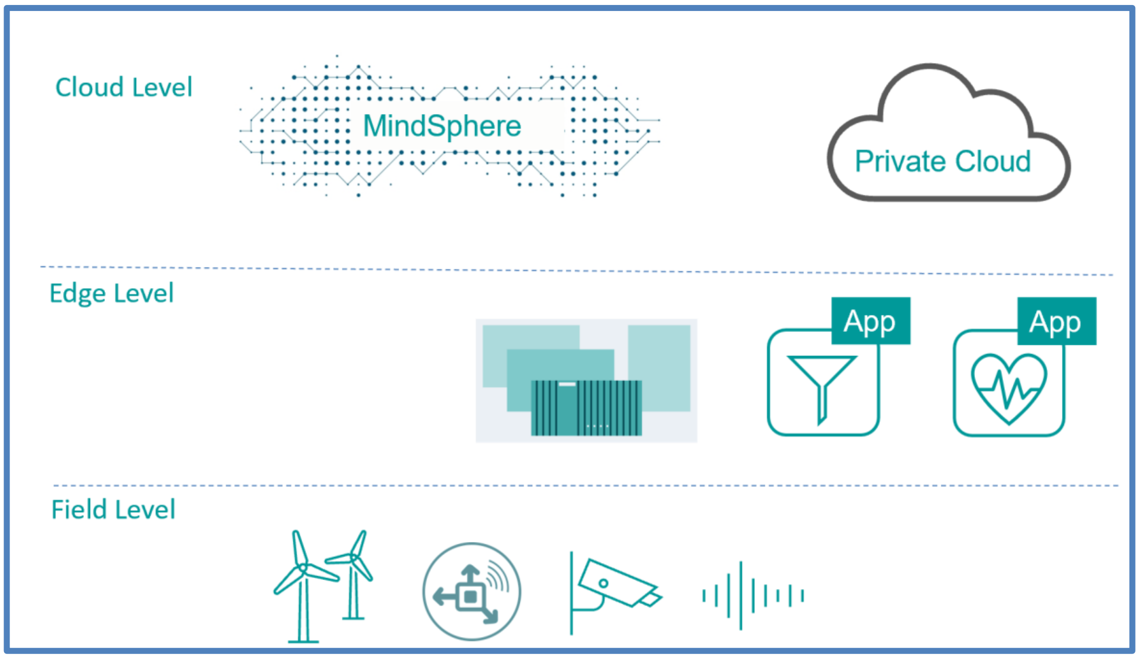


FIGURE 42: SEMANTIC INTEROPERABILITY ACROSS FIELD, EDGE AND TWO CLOUD PLATFORMS

In the following we summarize how the semantic interoperability has been achieved (further details are provided in Deliverable D3.9). The cornerstone for the semantic interoperability is based on the W3C Thing Description standard. Each field device is semantically described with a Thing Description (TD). The IoT Gateway generates a TD for brownfield device, whereas greenfield devices already have a TD available. Based on these TDs, the gateway provides a unified API at the Edge level. This has been realized with a standardized W3C Web of Things API. By making all field devices accessible over a unified and standardized API and describing them with standardized semantics, we have achieved the full interoperability at the Edge level.

In order to provide interoperability with the Cloud layer, our approach in SEMIoTICS is to reuse semantics from Edge in the Cloud. This means, that Thing Descriptions provided at the Edge has been used to form a Cloud-level semantics. The functionality of GW Semantic Mediator in the IoT Gateway has been extended for this purpose.

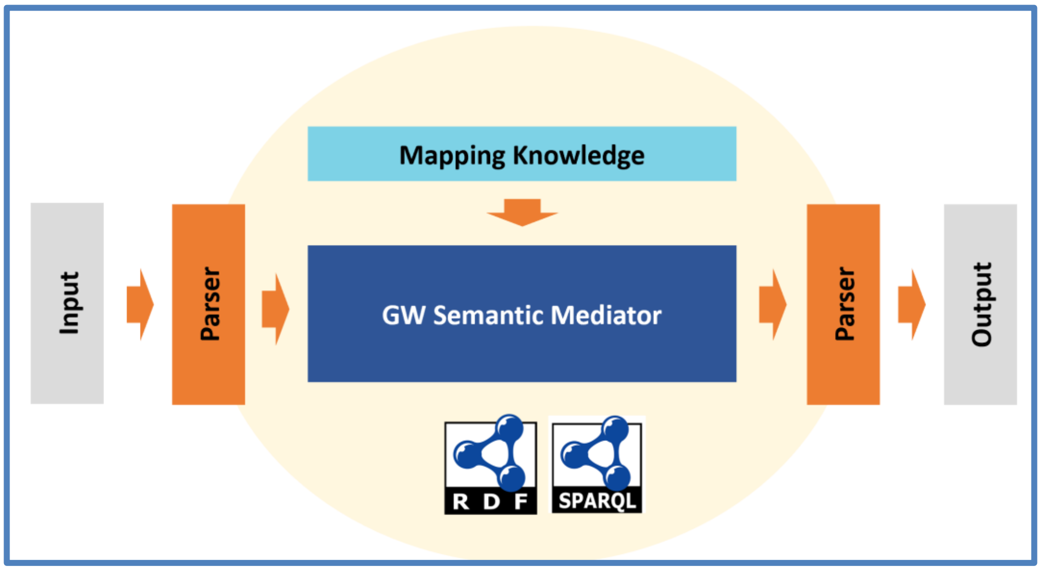


FIGURE 43: SEMANTIC MODELS TRANSFORMATION WITH GW SEMANTIC MEDIATOR

The GW Semantic Mediator accepts one semantic model as input and converts it into another semantic model, based on a mapping knowledge; see Figure 43. For example, in order to provide data of the inclinometer in the public MindSphere Cloud, we have to generate a MindSphere Asset model for the inclinometer. The GW Semantic Mediator automates this task. It takes a TD for the inclinometer (see Appendix 10.2) and automatically produces a valid MindSphere Asset model (see Appendix 0). The MindSphere model is generated with the same semantics as specified in the initial TD at the Edge. The two models differ in syntax and data formats though, e.g., TD is provided in JSONLD, whereas the MindSphere accepts the JSON format. The automated mapping is ensured based on Mapping Knowledge, see Figure 43. The Mapping Knowledge explains how to map one semantic model into another one. It is a declarative way to express the mapping rules. This knowledge can be created once for certain input/output models and used many times afterwards for an automated integration between Edge and Cloud. In Appendix 10.4 we show the Mapping Knowledge we created to map W3C Thing Description into Siemens MindSphere Asset model.

8.5 KPI-3.1.1: Generating monitoring strategies

Monitoring the infrastructure, in this use case the wind turbine, was mainly done via a microphone and a video camera. The strategies for handling the gathered data from those smart sensors are described below.

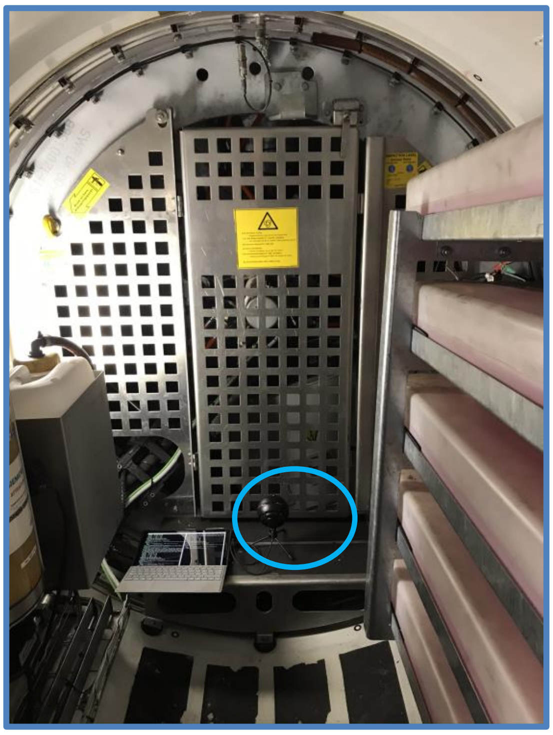


FIGURE 44: MICROPHONE (AT THE BOTTOM OF THE DOOR) INSTALLED IN WIND TURBINE NACELLE

Figure 44 shows the microphone mounted within the housing of a wind turbine. This Siemens NanoBox on the field layer, accesses the microphone and captures the audio data. The recorded audio is sliced into samples of predefined length. The audio samples are then transformed into images using Fourier transformation. These images are then fed into the AI model for inference to identify whether it is an anomaly (e.g., loose objects making tumbling noise) or a regular sound. These images of the audio are then sent via the network to the NanoBox on the backend layer, which is used to do the AI inference on the received data.

The concrete application flow of the audio processing that is setup in the Recipe Cooker is shown in Figure 45. The distribution of the implemented IoT applications over the two NanoBoxes on field and backend layer is also visible in their workflow, which is realized with Node-RED. On the left, we see the components deployed on the field NanoBox, and on the right the components deployed on the backend NanoBox. Such distribution of the application workflow on two (or more) edge devices has the advantage that the computational load can be shared, or processes can be parallelized. The workflow shown in Figure 45 comprises the following steps (each step is marked with a red square in the figure):

1. The audio data is accessed.
2. The recorded sample is played to the user.
3. The recorded audio sample is transformed into an image via Fourier transformation, which is then send over the network.
4. The image files are received.
5. The inference is computed.
6. Finally, the exceedance over a specified loss threshold indicates an anomaly, so a loose object in the turbine. Then, the PLC controller is triggered to stop the turbine.

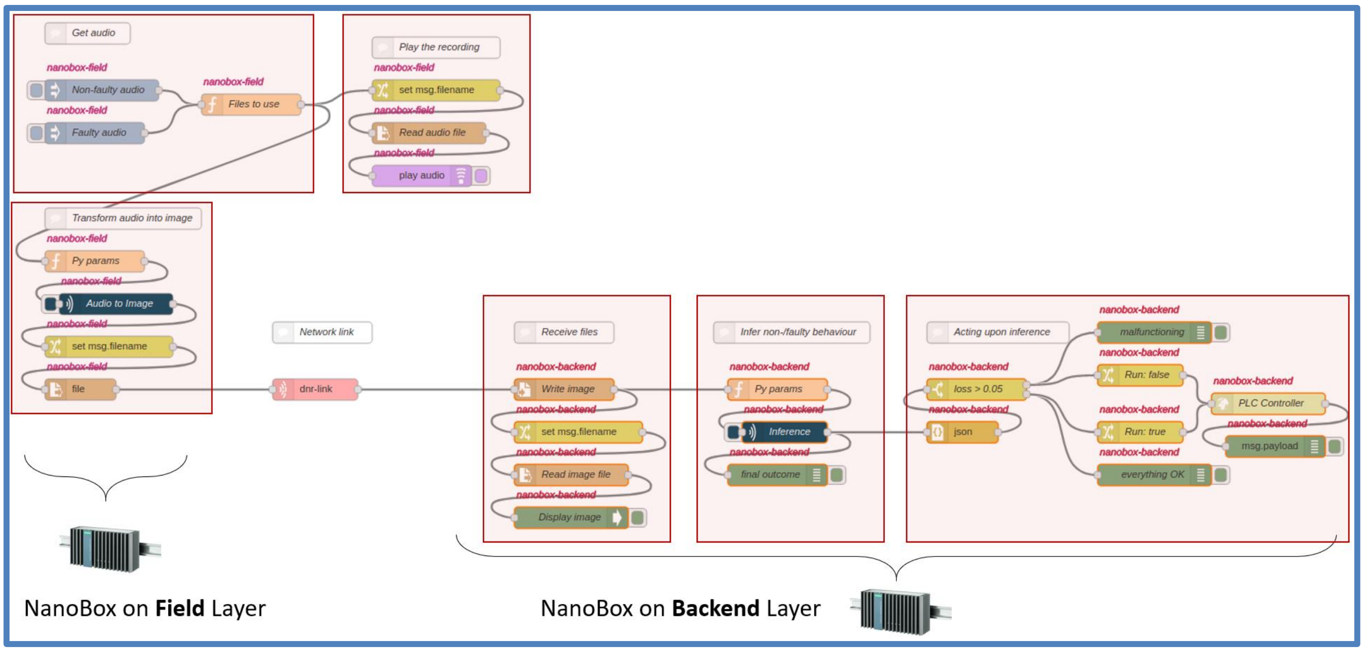


FIGURE 45: FLOW OF THE AUDIO PROCESSOR

Figure 46 shows example images resulting from Fourier Transformation of audio samples: on the left is a normal sound sample transformed into an image and on the right is an abnormal sound sample.

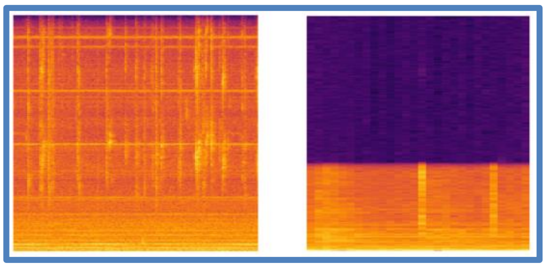


FIGURE 46: AUDIO FILES AFTER FOURIER TRANSFORMATION

Figure 47 shows the IoT application workflow of this video analytics case. On the left, we see the components deployed on the NanoBox on the field layer, and on the right the components deployed on the backend NanoBox. The workflow steps are as follows (shown in red squares in the figure):

1. The video analytics workflow starts by accessing and preparing the data.
2. In a second step it is sent over the network.
3. Where the data is received in this block and displayed through this node.
4. Feed into the AI inference here, which is captured in a Python script.
5. While the last step reacts to the inference output: the loss. If the loss exceeds a specified threshold, it means that grease has been detected, and the Wind turbine is switched off through contacting the PLC controller.

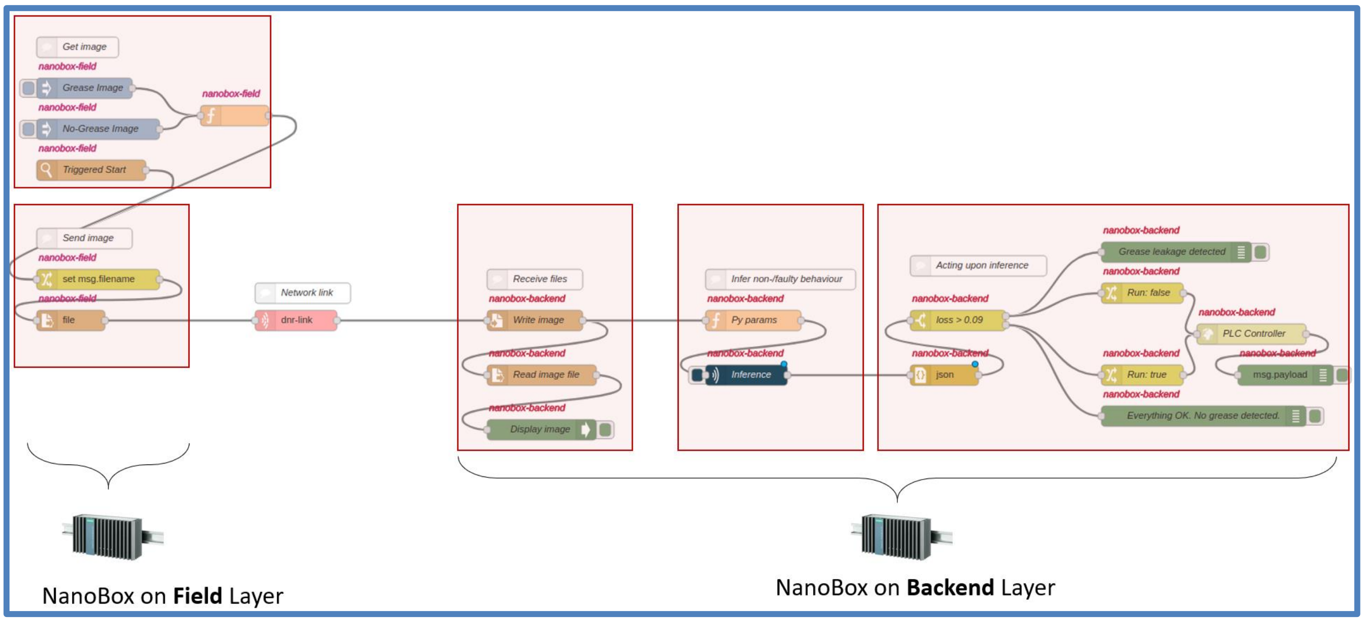


FIGURE 47: APPLICATION FLOW FOR THE VIDEO PROCESSING AND ANALYSIS

Compared to a conventional approach where raw data from the microphone deployed in Sub Use Case 1: Audio Processing Demo and the video camera deployed in Sub Use Case 2: Video Processing Demo respectively would be transmitted over a wide area and processed in either a private or public compute environment, the deployment of the application to the field layer for processing in an edge computing node, effectively eliminates the need to transmit the high-volume data off site.

The application deployed in Sub Use Case 1: Audio Processing Demo exploits a continuous 705 kilobits per second (kbps) stream of audio data (16-bit mono stream sampled at 44.1 KHz) from the microphone deployed equivalent every second. For a single deployment, the daily bandwidth saved by processing the data at the edge, can be seen in Equation 1.

$$Kilobytes\ per\ second\ KB/s = \frac{Kilobits\ per\ second\ (kbps)}{8}$$

$$88.125\ KB/s = \frac{705kbps}{8}$$

$$Daily\ bandwidth\ saved\ 7{,}6\ GB = 88.125\ kB/s\ x\ 86400\ seconds$$

EQUATION 1: DAILY BANDWIDTH SAVED IN SUB USE CASE 1 AUDIO PROCESSING DEMO

SUB USE CASE 1 AUDIO PROCESSING DEMO

Whereas the application deployed in Sub Use Case 2: Video Processing Demo only samples the image every 10 seconds. In the Sub Use Case 2 we sample a 1920 x 1080 (2MP) JPG compressed image that is downloaded directly from the video camera. The average size of the sampled image is 420 Kbyte. Thus, the daily bandwidth saved can be seen in Equation 2.

$$Megabyte\ MB = \ Kilobytes\ \div\ 1024$$

$$3.6288\ GB = 8640\ samples * 420KB\ (per\ sample)$$

$$Daily\ bandwidth\ saved\ is\ 3.62\ GB$$

EQUATION 2: DAILY BANDWIDTH SAVED IN SUB USE CASE 2 VIDEO PROCESSING

8.6 KPI-3.1.3: Performing predictive monitoring with an average accuracy of 80%

In D4.10 we reported already the creation of data sets for the test and validation of the developed AI algorithms for monitoring grease leakages within the wind turbine using video analytics. The final validation accuracy has been 99% (see Figure 48). For the validation data set, the aggregated model (federated model) was used for the calculation.

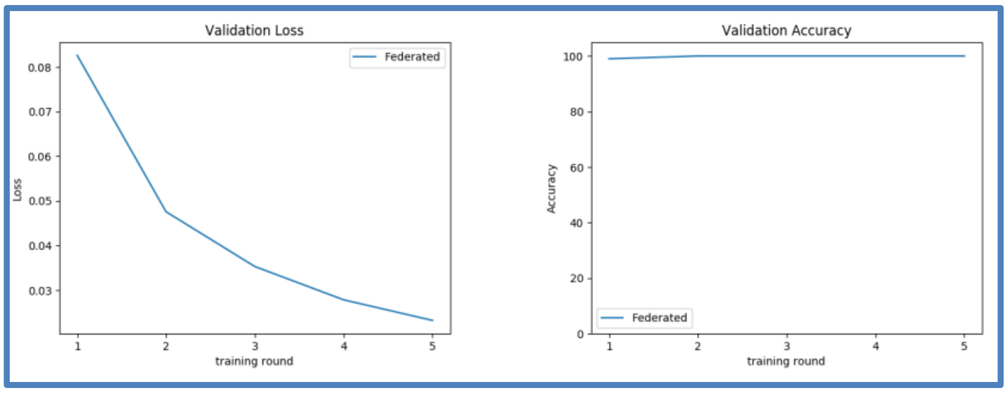


FIGURE 48: VALIDATION LOSS AND ACCURACY OF AGGREGATED MODEL FOR VIDEO ANALYSIS

As described in D4.10, we used Federated Learning to train the AI models. For the evaluation, the training was distributed among the three workers Alice, Bob, and Charlie. To minimize the runtime for this test case, all workers and the Orchestrator were started on the same device (and operating system). The data was transferred via the WebSocket protocol but is not really sent over a network. Only the loopback interface was used for the transmission since it is not important for the results of error and accuracy. A total of 5 training rounds were executed with a learning rate of 0.001. These two hyper parameters have a decisive influence on the training. The difficulty is to find the optimal values for the hyperparameters.

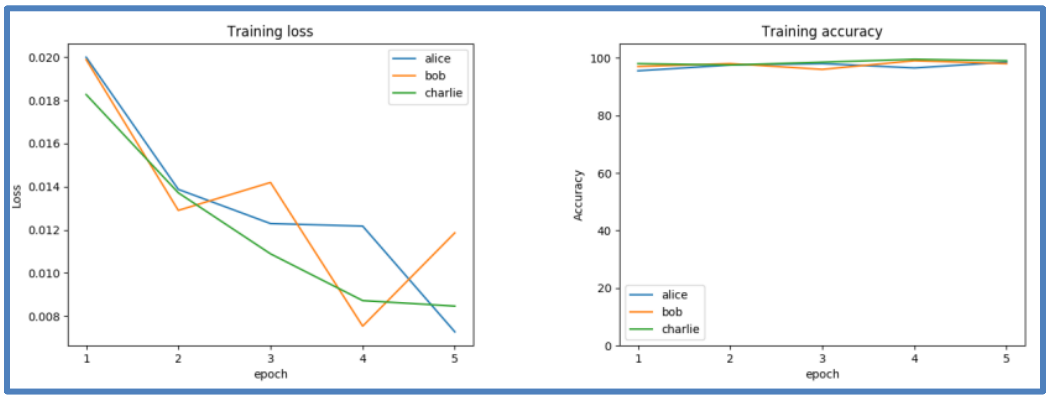


FIGURE 49: ERROR AND ACCURACY OF WORKER MODELS FOR VIDEO ANALYSIS

Figure 49 shows errors and accuracy of the training records of the three workers. Please note that each worker has his own model and training data set. Since the pre-trained model MobileNetV2 was used, the error after the first epoch is already very low and the accuracy is very high. The accuracy is between 98% and 99% for all workers after the training process.

In the following, we show how well the trained model works during operation. With the help of the library Matplotlib these images are graphically displayed together with the probabilities predicted by the model.

Figure 50 shows a screenshot of example images with their probabilities. The model has correctly classified all images. On closer analysis, it is noticeable that with an increasing amount of oil, the probabilities also increase. This means that the more oil there is, the more reliable the algorithm is. This is a further indication that the algorithm is working correctly. For example, in the second image in the first row, which shows a lot of oil, the algorithm is very secure at 98%. On the other hand, the third image in the third row has a comparatively low probability of 83%, which is due to the significantly lower oil content.

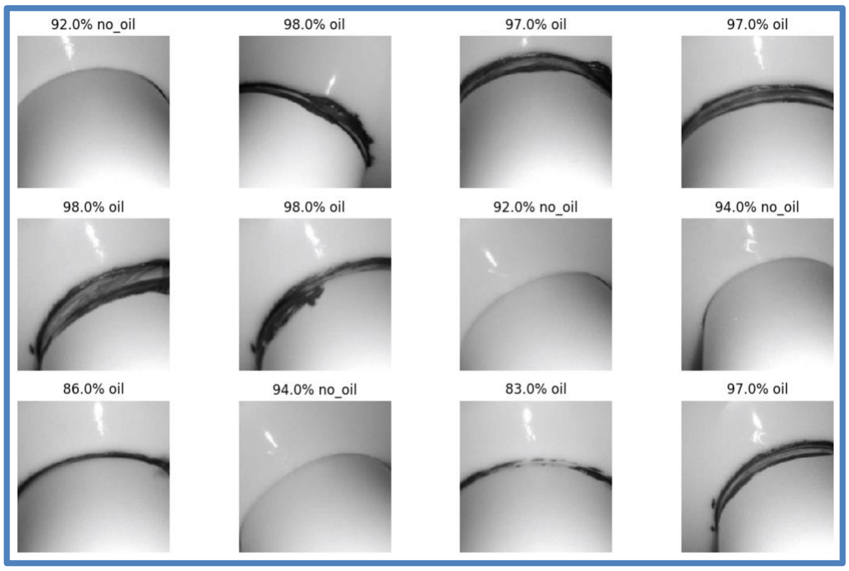


FIGURE 50: CLASSIFICATION AFTER TRAINING WITH FEDERATED LEARNING

8.7 KPI-4.1: Delivery of lightweight ML algorithms

The efficiency of the developed Federated Learning algorithm (see D4.10) for the communication network was analyzed using Wireshark. The used bandwidth was measured in both directions: between workers and orchestrator. Table 8 below shows the results.

TABLE 8: TRANSMITTED DATA BETWEEN ORCHESTRATOR AND WORKER

Direction	Size of model	Traffic
Orchestrator to Worker	8,7 MB	18,6 MB
Worker for Orchestrator	8,7 MB	18,6 MB

The amount of data transferred is approximately the same in both directions. The largest part of the transmission is determined by the size of the model. The reason for the considerably larger amount of data during transmission is therefore the conversion to the hexadecimal representation.

There is significant potential for improvement in the PySyft implementation. The implementation allows both the feature extract and the fine-tuning mode for transfer learning. By default, the latter is used because the pre-trained MobileNetV2 already delivers good results. Therefore, only the weights of the last shift are optimized during training. It would be more efficient to transfer only this layer and not the whole model. A possible solution is to split the model into two models. The first model (frozen_model) consists of all so-called frozen layer These are the layers whose parameters cannot be changed. In this case, these are all but the last layer. This model only needs to be transferred to the workers once. The second model (trainable_model) has the trainable layers, in this case only the last layer.

The differences between MobileNetV2 and frozen_model is only in the kilobyte range and can therefore be neglected. The file size of the trainable_model is much smaller with **11 KB**. If only this model were to be sent from the second training round onwards, an enormous amount of bandwidth could be saved.

In case of the audio analysis, to autonomously identify faults in the data captured by the microphone, we used an AutoEncoder, which is a special kind of neural network, that can be used to detect anomalies in data. The model of an AutoEncoder (Figure 51) has a bottleneck design, consisting of an Encoder (that takes in the input data which is compressed) and a decoder (that decompresses the data again to re-produce the input data). The output of the AutoEncoder has an intentional information reduction, which we can quantify through the loss. The loss is smaller when the uncompressed output is more similar to the input data.

Now, if the input data is an outlier, e.g., a fault situation such as an image that suddenly shows black dots of leaked grease, it diverts from the norm it was trained on (which would be images without any grease dots) and therefore it shows a higher loss. This fact can be used here to identify the faulty situations.

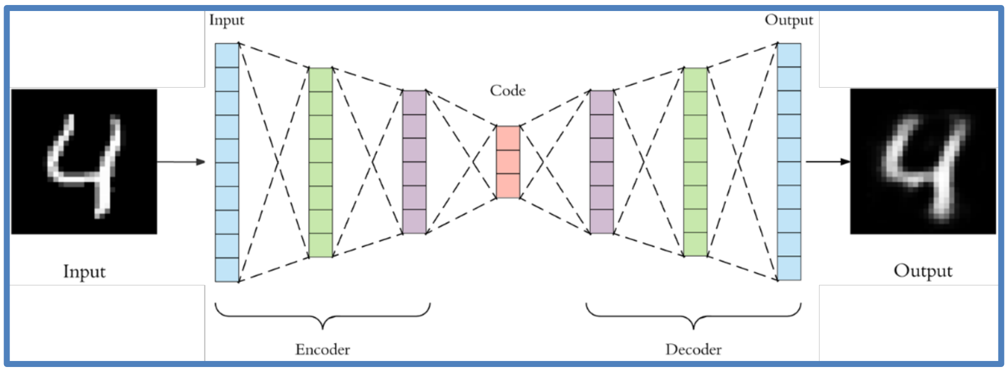


FIGURE 51: AUTOENCODER PRINCIPLE[8]

In order to train the AutoEncoder for the audio analytics for detection of loose objects within the turbine, we utilized the microphone within the turbine to record 24 hours of audio under normal conditions, which was cut in samples of 10 seconds of audio. These samples are then transformed from the time into the frequency domain using a Fourier Transformation (Figure 52). We did this to produce 200 images from such audio samples. These images were then used to train the AutoEncoder.

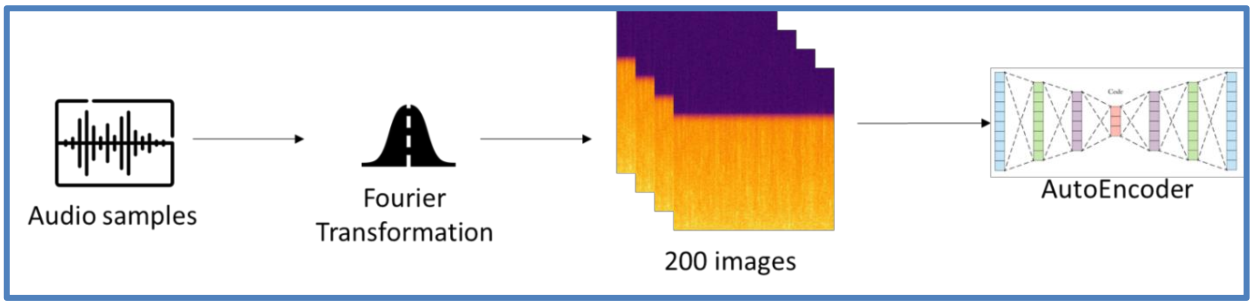


FIGURE 52: AUDIO ANALYTICS FOR LOOSE OBJECT DETECTION

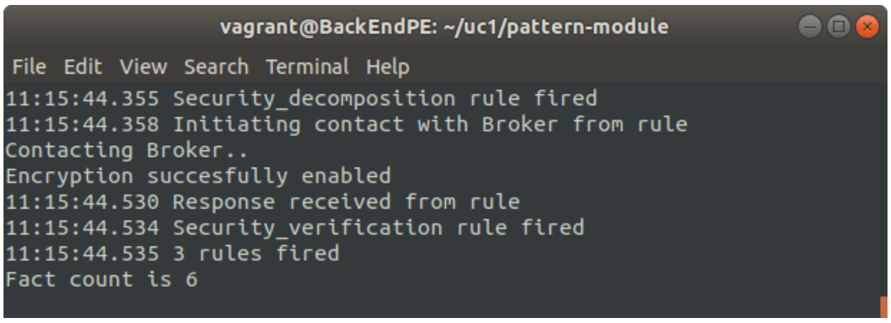
The model size of the AutoEncoder is about 108 MB so that would create a lot of overhead for the network. For the future, the goal is to implement a technique for reducing the size of the exchanged model, for example by compressing the model, or by transmitting the gradients that have changed significantly compared to the previous round, based on a threshold. Afterwards, the exchange of the partial model should be evaluated and compared with the one that suggests the exchange of the entire model. The

[8] Based on: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

comparison should be in terms of accuracy and training loss, as it is very important to maintain the same accuracy.

8.8 KPI-4.2: Delivery of mechanisms with adaptation time of 15ms

For the Sub Use Case 4, the encryption between the sensors and the Broker is enabled during runtime by the Pattern Engine. This is accomplished with the triggering of 3 Drools rules as mentioned in Section 8.1.


vagrant@BackEndPE: ~/uc1/pattern-module
File Edit View Search Terminal Help

```
11:15:44.355 Security_decomposition rule fired
11:15:44.358 Initiating contact with Broker from rule
Contacting Broker..
Encryption succesfully enabled
11:15:44.530 Response received from rule
11:15:44.534 Security_verification rule fired
11:15:44.535 3 rules fired
Fact count is 6
```

FIGURE 53: MEASUREMENTS OF RULES EXECUTION

The first rule is for the decomposition of the security property and the last is for the verification of the said property. The middle rule will attempt to contact the broker through internet to access the public cloud and enforce the encryption. The duration of the above process can be split in two parts. The first part includes the duration of the tasks occurring inside the Pattern Engine, while the second part includes the duration of the tasks occurring on the Broker side which is located in the cloud. In order to measure the duration of the adaptation mechanism we calculate the duration of the tasks occurring at the Pattern Engine.

We can see from Figure 53, and in more detail in Figure 54 that the first rule is triggered at 11:15:44.355 and the second rule is triggered at 11:15:44.358. This means that the time needed to trigger the first rule until the time that the engine-initiated contact with Broker took 3ms. Likewise, we see that we receive the response from the public cloud at 11:15:44:530 and all the rules have ended at 11:15:44.535. This adds 5ms more to the initial duration resulting to a total of 8ms thus satisfying the limits set in KPI 4.2.

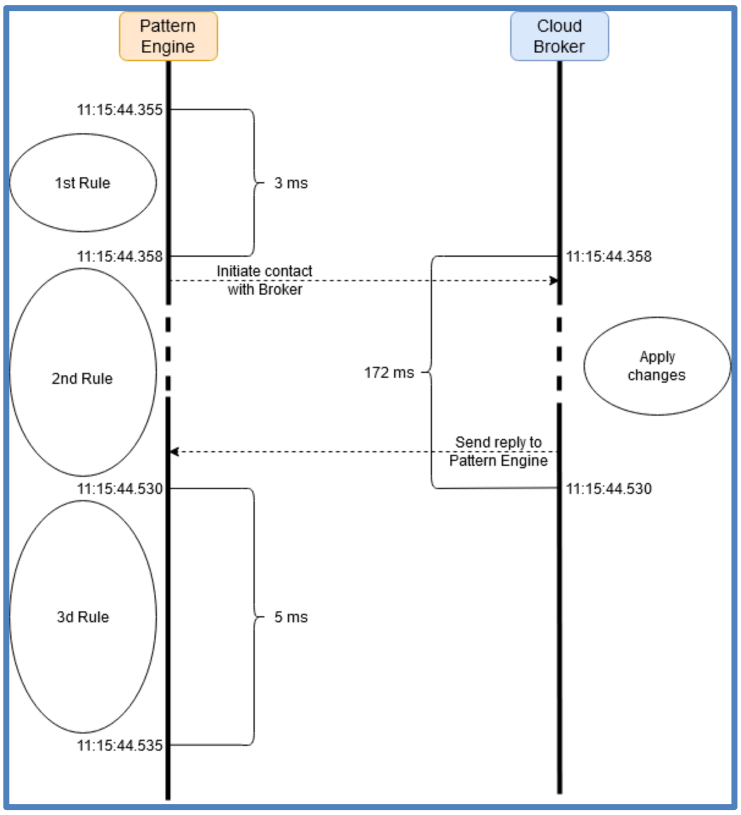


FIGURE 54: MEASUREMENTS OF ADAPTATION MECHANISM

8.9 KPI-4.6: Development of new security Mechanisms/Controls

In the context of UC1, sensors are communicating with an MQTT Broker. The security of this communication and particularly the encryption of the traffic, is controlled by an appropriate pattern rule preinstalled in the Backend Pattern Engine (Figure 55). This pattern rule allows the enforcement of encryption on the MQTT Broker with the help of an application developed on the Broker side.

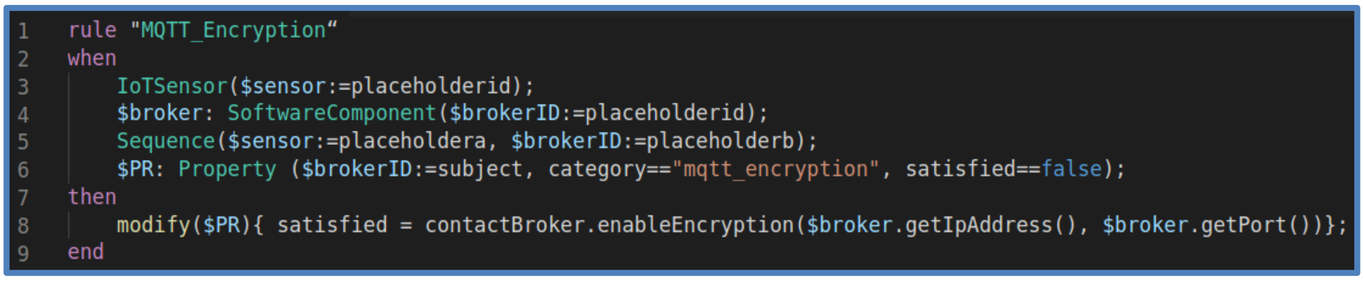
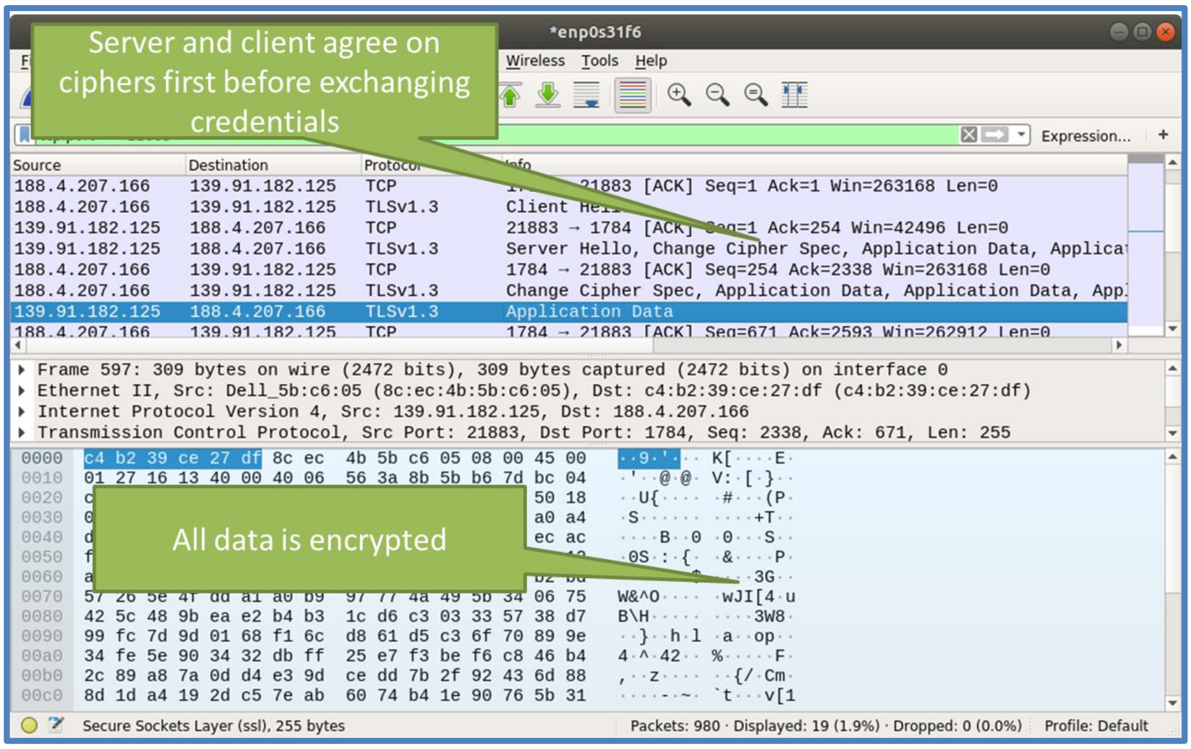

```
rule "MQTT_Encryption“
when
    IoTSensor($sensor:=placeholderid);
    $broker: SoftwareComponent($brokerID:=placeholderid);
    Sequence($sensor:=placeholdera, $brokerID:=placeholderb);
    $PR: Property ($brokerID:=subject, category=="mqtt_encryption", satisfied==false);
then
    modify($PR){ satisfied = contactBroker.enableEncryption($broker.getIpAddress(), $broker.getPort())};
end
```

FIGURE 55: PATTERN BASED ENCRYPTION ENFORCEMENT

In the configuration of the MQTT Broker, all the related information for incoming connections such as port, protocol certificates etc., is grouped and the terminology used to describe that group is “listener”. The “enableEncryption” alters the configuration of the Broker in such a manner that it will include certificates for the available listener for incoming connections thus enforcing TLS encryption as shown in packet capture of Figure 56.


Server and client agree on ciphers first before exchanging credentials
All data is encrypted
*enp0s31f6
Wireless Tools Help
Expression...

```
Source          Destination     Protocol  Info
188.4.207.166   139.91.182.125  TCP       1784 → 21883 [ACK] Seq=1 Ack=1 Win=263168 Len=0
188.4.207.166   139.91.182.125  TLSv1.3   Client Hello
139.91.182.125  188.4.207.166   TCP       21883 → 1784 [ACK] Seq=1 Ack=254 Win=42496 Len=0
139.91.182.125  188.4.207.166   TLSv1.3   Server Hello, Change Cipher Spec, Application Data, Applicat
188.4.207.166   139.91.182.125  TCP       1784 → 21883 [ACK] Seq=254 Ack=2338 Win=263168 Len=0
188.4.207.166   139.91.182.125  TLSv1.3   Change Cipher Spec, Application Data, Application Data, Appl
139.91.182.125  188.4.207.166   TLSv1.3   Application Data
188.4.207.166   139.91.182.125  TCP       1784 → 21883 [ACK] Seq=671 Ack=2593 Win=262912 Len=0

▸ Frame 597: 309 bytes on wire (2472 bits), 309 bytes captured (2472 bits) on interface 0
▸ Ethernet II, Src: Dell_5b:c6:05 (8c:ec:4b:5b:c6:05), Dst: c4:b2:39:ce:27:df (c4:b2:39:ce:27:df)
▸ Internet Protocol Version 4, Src: 139.91.182.125, Dst: 188.4.207.166
▸ Transmission Control Protocol, Src Port: 21883, Dst Port: 1784, Seq: 2338, Ack: 671, Len: 255

0000  c4 b2 39 ce 27 df 8c ec  4b 5b c6 05 08 00 45 00   ··9·'···K[····E·
0010  01 27 16 13 40 00 40 06  56 3a 8b 5b b6 7d bc 04   ·'··@·@·V:·[·}··
0020  c                                            50 18   ··U{·····#···(P·
0030  0                                            a0 a4   ·S·········+T··
0040  d                                            ec ac   ····B··0·0···S··
0050  f                                                    ·0S·:·{··&····P·
0060  a                                                    ··········3G··
0070  57 26 5e 4f dd a1 a0 b9  97 77 4a 49 5b 34 06 75   W&^O·····wJI[4·u
0080  42 5c 48 9b ea e2 b4 b3  1c d6 c3 03 33 57 38 d7   B\H·········3W8·
0090  99 fc 7d 9d 01 68 f1 6c  d8 61 d5 c3 6f 70 89 9e   ··}··h·l·a··op··
00a0  34 fe 5e 90 34 32 db ff  25 e7 f3 be f6 c8 46 b4   4·^·42··%·····F·
00b0  2c 89 a8 7a 0d d4 e3 9d  ce dd 7b 2f 92 43 6d 88   ,··z······{/·Cm·
00c0  8d 1d a4 19 2d c5 7e ab  60 74 b4 1e 90 76 5b 31   ····-·~·`t···v[1

Secure Sockets Layer (ssl), 255 bytes     Packets: 980 · Displayed: 19 (1.9%) · Dropped: 0 (0.0%)     Profile: Default
```

FIGURE 56: PACKET CAPTURE AFTER ENCRYPTION IS ENABLED

The interaction between the pattern Engine and the MQTT Broker occurs during runtime and the encryption is enabled on the fly. The immediate effect of the above procedure is to break the communication with the sensors until they are configured to use encryption from their side too. This way, the security of the system is hardened, allowing only secure communication between the sensors and the MQTT Broker.

8.10 KPI-6.1: Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%

In order to address this KPI we briefly describe how automation systems are engineered nowadays. We then compare this procedure with our approach, which significantly reduces the effort.
The target of 80% is very much an exact number, which cannot be proven in general. Various automation systems and applications differ significantly in their complexity. To reduce manual intervention is also a qualitative measure (not only quantitative one as stated in this KPI). Nevertheless, in the following we attempt to provide justification for our approach, which eases the engineering process.

The state of the art in engineering of automation systems is divided into five phases: 1. Design phase; 2. Development phase; 3. Engineering phase; 4. Commissioning phase, and 5. Operation phase.
An engineer starts the Design phase by defining an automation function, which needs to be implemented on a field device. The function is defined with a model in accordance with the model-driven design. The engineer runs the code generator to generate the skeleton of the function and implements the function in the Development phase. The engineer will then typically engineer an automation system. The system contains the field device with a newly developed function. Usually, it contains other field devices too. The devices need to be networked with defined interactions between them (including protocols, IP addresses, ports etc.). The semantics of data exchanged between devices need to be defined (data formats, data types, units, quantities etc.). Finally, the engineer deploys and configures the system on an automation station in the Commissioning phase and starts the system with a certain functionality (application).

If a new application, with a new device, needs to be realized (as it is the case in Use Case 1) then the automation system has to be re-engineered. The new device is required to communicate over the same protocol used by existing sensors. This is in contradiction to the requirements of our use case. Our assumption is that a wind turbine has a lifecycle of 20 to 30 years. In this long period of time, new applications with new devices and with different communication capabilities are bound to emerge, especially considering the fast-paced landscape of the IIoT era. Thus, we need to make interoperable brownfield devices with new devices.

With our approach the IoT Gateway integrates both brownfield and greenfield devices and enables a new application. In tradition automation system, for devices with diverse communication capabilities or different semantics of data, this is not possible at all. In contrast, the SEMIoTICS IIoT Gateway can do this, as it features pre-created protocol bindings for devices with diverse communication capabilities and provides a unified Thing Description as a common semantics for devices and their data. Moreover, it is integrated with a Software Defined Network to ensure networking capabilities of field devices tailored to a given application. It scans the network consisting of diverse field devices, and in a few minutes, it can bootstrap them. Furthermore, it automatically generates programmable access to these devices. An engineer may then implement an application knowing only one API and one semantic data model for all diverse field devices, their capabilities, and the capabilities of the underlying network.

In our evaluation, an engineer was able to bootstrap few field devices and develop an application in less than 15 minutes. In contrast to this, the traditional engineering process as described above requires a day or longer. This always depends on the number of field devices and the size of an application (automation system). Our evaluation was constrained to the class of applications as developed in this project.

8.11 KPI-6.3: Delivery of 3 prototypes of IIoT/IoT applications

In addition to Use Cases 2 and 3, Use Case 1 presents a prototype of an IoT platform implementing the SEMIoTICS architecture. Therefore, the 3 use case demonstrators delivered by the project satisfy this KPI.

Regarding UC1 in specific, Sub Use Case 1 and Sub Use Case 2 use infrared camera and microphone hardware sensors, in addition to an AI-based imagery/audio classifier interlinked via a recipe and deployed on multiple devices networked via SDN. Based on the result of the inference of the imagery and audio classifiers, a running system is triggered with an adaptation request, i.e., to stop turbine rotation, in real-time. In addition, alarms are raised using remote Cloud platforms as enablers, in order to notify the system owner of ongoing malfunctions and events. Sub Use Case 3 deploys an inclinometer mounted on the model wind turbine to report the sensed measurements over the SDN to a Cloud platform (Siemens MindSphere) where the data is recorded and available for analytics in the Predictive Maintenance and Condition Monitoring applications.

8.12 KPI-7.1 Provision the SEMIoTICS building blocks

All components deployed in UC1 fulfil by definition the TRL level 4, due to their successful deployment in the lab environment.

Some of the tested blocks additionally are validated for compatibility with the technological building blocks at TRL 5. Specifically, the IIoT Gateway components used in Use Cases 1, 2, and 3, among other functionalities, exposes an interface to interact with a PLC brownfield device (i.e., the SIMATIC S7 PLC). This turbine controller emulates the functionality of the turbine controller hosted in the operational Wind Park.

Details of the realization of the interface for brownfield interactions are provided in Section 6.

9 CONCLUSION

9.1 Observed obstacles

Main obstacles encountered during the demonstrator development were related to the ongoing COVID-19 pandemic, imposing limitations in access to the integrated environment.

In particular, physically preparing the testbed (e.g., the demonstration case shown in Figure 3) and the execution of tests where physical input was required (e.g., putting grease on the turbine model and subsequently collecting sufficient amount of images to train the auto-encoder with good accuracy), imposed significantly delays in the development of field layer applications.

Additionally, remote access had to be established to the system due to limited physical access and connection to the testbed. However, connecting to each device in the distributed setup over provided SDN infrastructure proved unstable, due to dependency of the SDN's correct functioning on the SDN controller. Thus, whenever the SDN controller was impacted, e.g., due to a planned code upgrade or test execution, the remote access would accordingly be affected, or even made unavailable, which made the progress of SSC implementation harder. In order to circumvent this issue, we deployed an out-of-band network to each device in the system, configured it with a separate subnet and made it available for access to all contributors of the use case.

The limitation of not being able to execute technical workshops further delayed implementation and integration achievements, whenever dependency on hardware or isolated testbed access was necessary. The need for social distancing resulted in the adoption a token-based approach for persons requiring accessing the testbed. Common integration rounds were, thus, unviable. Due to testbed being hosted at SAG premises, SAG had to act as the sole point of contact and integrator in order to deploy the components developed by BS (SEMIoTICS GUI - GUIHub) and FORTH (Pattern Engines, Pattern Orchestrator).

9.2 Concluding Remarks

The main goal of UC1 was to demonstrate the seamless coexistence of a highly integrated control system and an agile and SPDI-aware IIoT ecosystem based on the SEMIoTICS framework, allowing service providers to deploy new value-added services faster and provide effective access to data, from both existing control system and the newly deployed edge sensor networks. The demonstrator of UC1, therefore, addresses some of the common challenges in extending the capabilities of an existing control system in a brownfield environment in the Wind Energy domain while highlighting the general benefits of the SEMIoTICS platform.

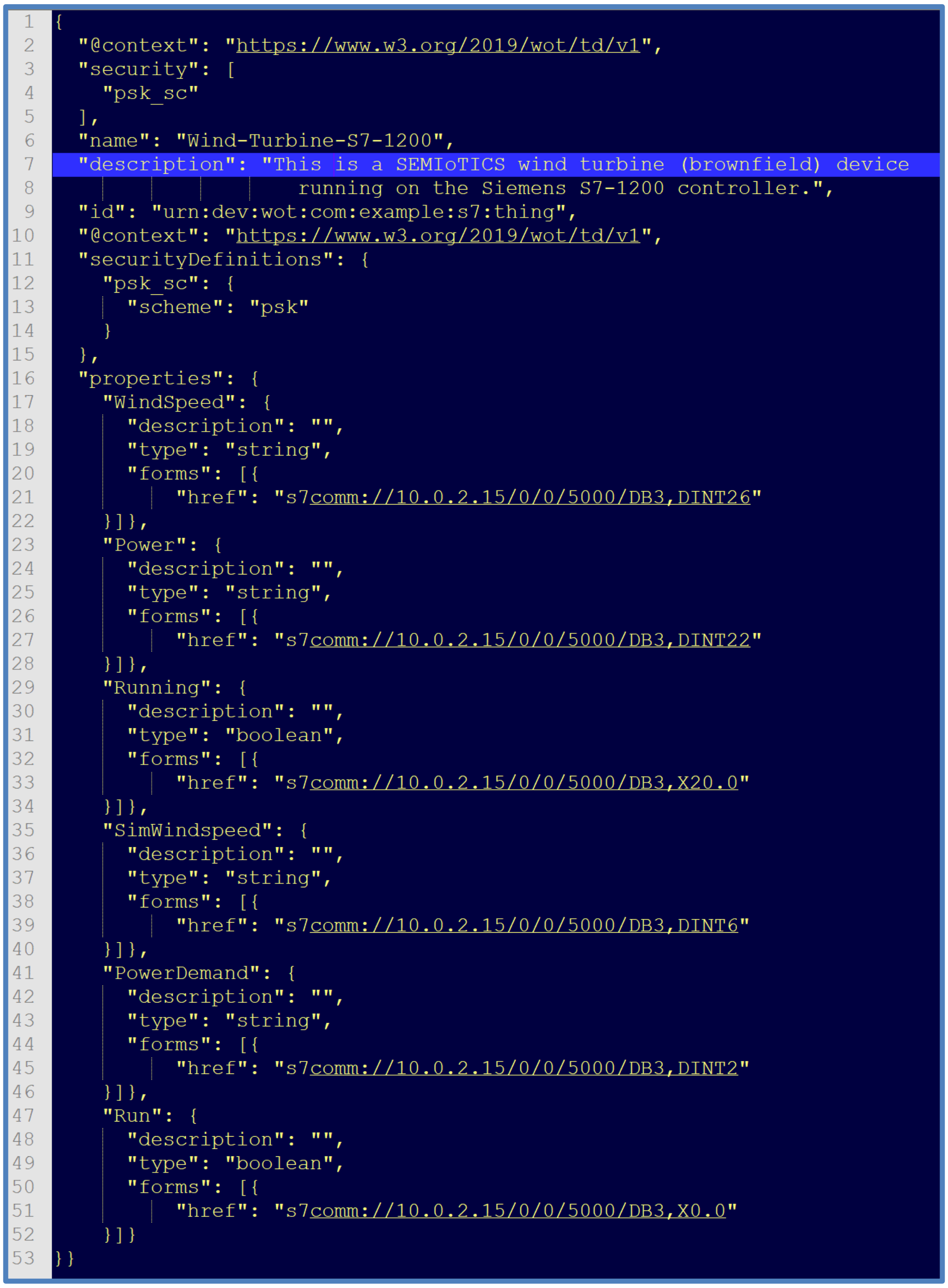
Deliverable D5.4 provided the initial insights on the development of UC1, while D5.9 presented herein the final status of development of SEMIoTICS' UC1 and the four associated Sub Use Cases. Compared to D5.4, D5.9 extended the description of D5.4 with the final status of implementation and integration of the UC1 components and elements of the SEMIoTICS platform and has presented the methodology and measurement points (where applicable) used to quantify the reaching of KPIs in UC1, as defined in the D5.1

Therefore, this deliverable presented the final achievements of the consortium regarding the integration of physical equipment comprising industrial programmable logic controllers, actuation, the greenfield and brownfield sensory and actuation devices, private and public clouds environments and hosted applications, the orchestration mechanisms introduced in SEMIoTICS, as well as the resources orchestrated, and applications dynamically instantiated and interconnected by our framework.

10 APPENDIX

10.1 Thing Description for Brownfield Device

This is an example of a Thing Description for SEMIoTICS wind turbine (brownfield device).

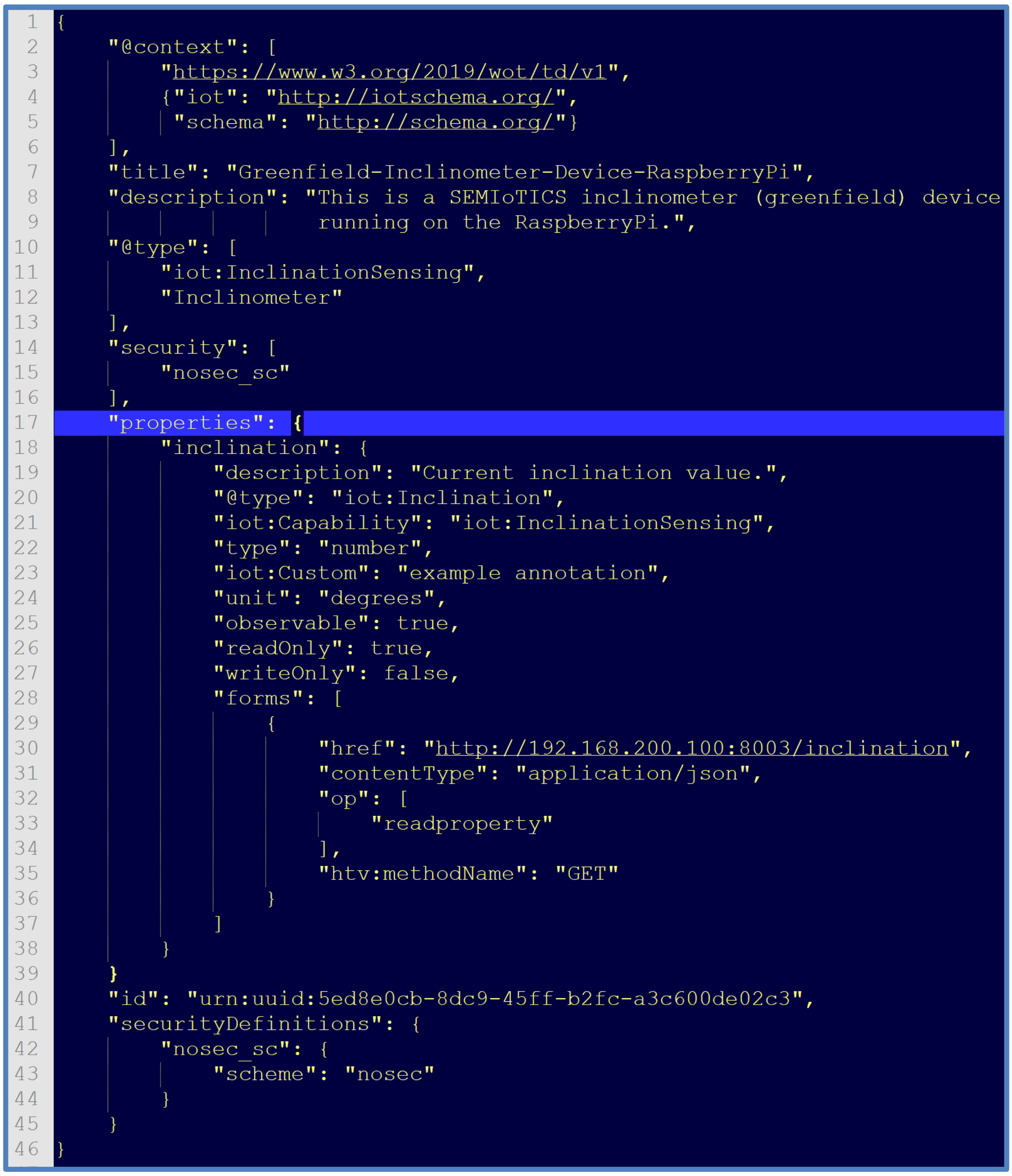


```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "security": [
    "psk_sc"
  ],
  "name": "Wind-Turbine-S7-1200",
  "description": "This is a SEMIoTICS wind turbine (brownfield) device
                  running on the Siemens S7-1200 controller.",
  "id": "urn:dev:wot:com:example:s7:thing",
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "securityDefinitions": {
    "psk_sc": {
      "scheme": "psk"
    }
  },
  "properties": {
    "WindSpeed": {
      "description": "",
      "type": "string",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,DINT26"
    }]},
    "Power": {
      "description": "",
      "type": "string",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,DINT22"
    }]},
    "Running": {
      "description": "",
      "type": "boolean",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,X20.0"
    }]},
    "SimWindspeed": {
      "description": "",
      "type": "string",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,DINT6"
    }]},
    "PowerDemand": {
      "description": "",
      "type": "string",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,DINT2"
    }]},
    "Run": {
      "description": "",
      "type": "boolean",
      "forms": [{
          "href": "s7comm://10.0.2.15/0/0/5000/DB3,X0.0"
    }]}
}}
```

FIGURE 57: THING DESCRIPTION FOR WIND TURBINE

10.2 Thing Description for Greenfield Device

This is an example of a Thing Description for SEMIoTICS inclinometer (greenfield device).

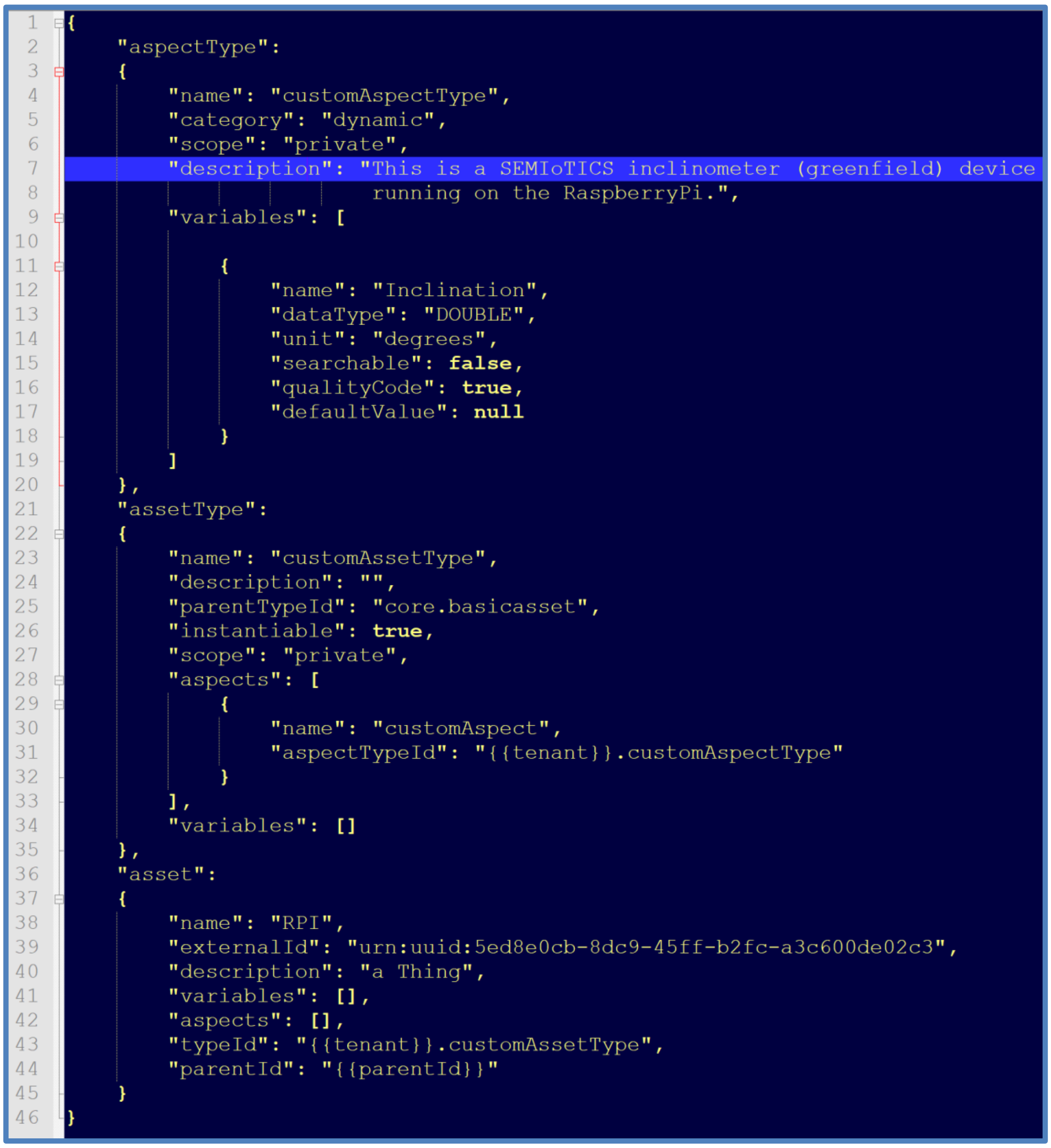


```json
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        {"iot": "http://iotschema.org/",
         "schema": "http://schema.org/"}
    ],
    "title": "Greenfield-Inclinometer-Device-RaspberryPi",
    "description": "This is a SEMIoTICS inclinometer (greenfield) device running on the RaspberryPi.",
    "@type": [
        "iot:InclinationSensing",
        "Inclinometer"
    ],
    "security": [
        "nosec_sc"
    ],
    "properties": {
        "inclination": {
            "description": "Current inclination value.",
            "@type": "iot:Inclination",
            "iot:Capability": "iot:InclinationSensing",
            "type": "number",
            "iot:Custom": "example annotation",
            "unit": "degrees",
            "observable": true,
            "readOnly": true,
            "writeOnly": false,
            "forms": [
                {
                    "href": "http://192.168.200.100:8003/inclination",
                    "contentType": "application/json",
                    "op": [
                        "readproperty"
                    ],
                    "htv:methodName": "GET"
                }
            ]
        }
    }
    "id": "urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3",
    "securityDefinitions": {
        "nosec_sc": {
            "scheme": "nosec"
        }
    }
}
```

FIGURE 58: THING DESCRIPTION FOR INCLINOMETER

10.3 MindSphere Asset Model

This is an example of MindSphere Asset model for SEMIoTICS inclinometer.



```
{
    "aspectType":
    {
        "name": "customAspectType",
        "category": "dynamic",
        "scope": "private",
        "description": "This is a SEMIoTICS inclinometer (greenfield) device
                running on the RaspberryPi.",
        "variables": [

            {
                "name": "Inclination",
                "dataType": "DOUBLE",
                "unit": "degrees",
                "searchable": false,
                "qualityCode": true,
                "defaultValue": null
            }
        ]
    },
    "assetType":
    {
        "name": "customAssetType",
        "description": "",
        "parentTypeId": "core.basicasset",
        "instantiable": true,
        "scope": "private",
        "aspects": [
            {
                "name": "customAspect",
                "aspectTypeId": "{{tenant}}.customAspectType"
            }
        ],
        "variables": []
    },
    "asset":
    {
        "name": "RPI",
        "externalId": "urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3",
        "description": "a Thing",
        "variables": [],
        "aspects": [],
        "typeId": "{{tenant}}.customAssetType",
        "parentId": "{{parentId}}"
    }
}
```

FIGURE 59: MINDSPHERE ASSET MODEL FOR SEMIOTICS INCLINOMETER

10.4 Mapping Rule of GW Semantic Mediator

This is an example of a declarative mapping in GW Semantic Mediator. It provides an automated mapping of a Thing Description (e.g., SEMIoTICS inclinometer shown in Figure 58) into a MindSphere Asset model (see Figure 59). Figure 60 depicts the complete example mapping (note that all three parts below belong to the same mapping rule).

```
PREFIX gr: <http://example.com/graph/>
PREFIX ms: <https://mindsphere.io/#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/>
PREFIX td: <https://www.w3.org/2019/wot/td#>
PREFIX transit: <http://temporary.transition.org/>
PREFIX this: <http://this.org/>
PREFIX pdc: <http://purl.org/dc/terms/>

INSERT {
    transit:externalId this:correspondTo ms:externalId.
    graph gr:f_example {[] transit:externalId ?object.}}
WHERE {
    graph gr:f_example {
        ?a pdc:title ?b}
      filter (
            not exists
      {graph gr:f_example {   [] transit:externalId ?a.     }}
    )
    bind( if(isIRI(?a),str(?a),?a) as ?object )
};
```

```
INSERT { GRAPH gr:t_model{
   ?bn  ms:variables    [
                            ms:name ?name;
                            ms:dataType ?type;
                            ms:unit ?unit;
                            ms:qualityCode true;
                            ms:searchable false
                        ]}
} WHERE {
    graph gr:f_example{
        ?ident td:hasPropertyAffordance [
            td:name ?name;
            rdf:type ?oldtype;
            schema:unitCode ?unit
      ]
}
    graph gr:t_model{
        [ms:aspectType ?bn].
    }
    filter (
        not exists
        {
            graph gr:t_model {
                  [     ms:variables [
                        ms:name ?name;
                        ms:dataType ?type;
                        ms:unit ?unit;
                        ms:qualityCode true;
                        ms:searchable false
                        ]
                  ]
            }
        }
      )
    bind(if(STR(?oldtype)="https://www.w3.org/2019/wot/json-
schema#NumberSchema","DOUBLE",?oldtype) as ?type )
};
```

```
WITH gr:t_model
DELETE {
    ?bn ms:variables ?s.
    ?s ?p ?o
} WHERE {
      ?s ms:dataType "{{dataType}}".
      ?bn ms:variables ?s.
      ?s ?p ?o
} ;
DELETE {
    graph gr:t_model{?bn ms:name ?oldname.}
}
INSERT {
    graph gr:t_model{?bn ms:name ?newname}
}
WHERE {
    graph gr:t_model{?bn ms:name ?oldname.}
    [ms:asset ?bn].
    graph gr:f_example{ ?s <http://purl.org/dc/terms/title> ?newname.}
};
DELETE {
    graph gr:t_model{?bn ms:description ?olddescription.}
}
INSERT {
    graph gr:t_model{?bn ms:description ?newdescription;}

} WHERE {
    graph gr:t_model{  ?bn ms:description ?olddescription}
    graph gr:f_example{
      ?s <http://purl.org/dc/terms/description> ?newdescription;
      <http://purl.org/dc/terms/title> ?title               }
      [ms:asset ?bn].
}
```

FIGURE 60: AN EXAMPLE OF A DECLARATIVE MAPPING IN GW SEMANTIC MEDIATOR