Ref. Ares(2020)3433041 - 30/06/2020

SEMIoTICS

Deliverable D4.13
Implementation of Backend API (Final Cycle)

Deliverable release date	30/06/2020
Authors	1. Arne Broering (SAG) 2. Eftychia Lakka, Emmanouil Michalodimitrakis, Tsirantonakis Georgios, Georgopoulos Konstantinos (FORTH) 3. Konstantinos Fysarakis, Iasonas Somarakis, Michail Smyrlis (STS) 4. Bartłomiej Lipa, Michał Rubaj, Urszula Stawicka (BS) 5. Felix Klement, Korbinian Spielvogel, Henrich C. Pöhls (UP)
Responsible person	Bartłomiej Lipa (BS)
Reviewed by	Mirko Falchetto (ST-I), Konstantinos Fysarakis (STS), Felix Klement (UP), Eftychia Lakka, Nikolaos Petroulakis, Manolis Michalodimitrakis (FORTH)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Urlich Hansen)
Status of the Document	Final version
Version	1.0
Dissemination level	Confidential

Table of Contents

TABLE 1 ACRONYMS TABLE

Acronym	Definition
AEP	Authentication Enforcement Point
API	Application Programming Interface
BO	Backend Orchestrator
CD	Continuous Development
CI	Continuous Integration
CPU	Central Processing Unit
CRUD	Create, Remove, Update, Delete
DVCS	Distributed Version Control System
EMF	Eclipse Modelling Framework
GUI	Graphical User Interface
GW	Gateway
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IIoT	Industrial Internet of Things
IoT	Internet of Things
JSON	JavaScript Object Notation
JSON-LD	JSON for Linking Data
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
PaaS	Platform as a Service
PEP	Policy Enforcement Point
PoC	Proof of Concept
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software-Defined Networking
SEMIoTICS	Smart End-to-end Massive IoT Interoperability, Connectivity and Security
SPDI	Security, Privacy, Dependability and Interoperability
SW	Software
TCP	Transmission Control Protocol
TD	Thing Description
TLS	Transport Layer Security protocol
TTL	Time To Live
UC	Use Case
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
vSwitch	Virtual Switch
W3C	World Wide Web Consortium
WoT	Web of Things
WP	Work Package

1 INTRODUCTION

SEMIoTICS aims to deliver an open source, proof-of-concept implementation of the SEMIoTICS framework, integrating the core interoperability, monitoring, intelligence, adaptation, and networking capabilities. In this context, the implementation of the backend API of SEMIoTICS will cover not only the implementation of the necessary algorithms, techniques, and components but also deliver an open API set giving access to them.

Said backend API will provide communication across the layers and communication with external systems and partners. Any kind of connection within the IoT platform will be monitored in order to ensure Security, Privacy, Dependability, and Interoperability (SPDI) requirements relevant for each component. Delivery of 3 prototypes (use cases) of IoT applications will demonstrate the business and technological capabilities of the SEMIoTICS framework, in the domains of Wind Energy, Healthcare and Smart Sensing.

Looking from an implementation perspective, the first implementation cycle (Cycle 1 due M17), the second implementation cycle (Cycle 2 due M23) and the final implementation cycle (Cycle 3 due M30) combined together provide the implementation of algorithms, techniques and components in WP4 (Tasks 4.1 - 4.5) and deliver set of dedicated APIs giving access to them. As it has been stated in the project description of action, this API provides IoT components communication across layers and integration with external systems and partners.

Based on the above, Deliverable 4.13 “Implementation of SEMIoTICS Backend API (Cycle 3)”, being the final output of T4.6 (Implementation of SEMIoTICS backend API), provides the status of the final implementation cycle, describes the implementation approach and establishes which backend architectural components (see D2.5) are developed in which SEMIoTICS development cycle. Within this deliverable, the implementation status of the final algorithms, techniques, components. (as specified in T4.1 to T4.5 and the respective deliverables) and API for accessing them are described.

In more detail, this document deliverable D4.13 is structured as follows:

- Section 2 describes the SEMIoTICS implementation approach.
- Section 3 establishes which backend architectural components (more details available in Deliverable 2.5 “SEMIoTICS high level architecture (final)”) are developed in which SEMIoTICS development cycle.
- Section 4 covers the development status of Cycle 3 and describes the development of each of the components related to cycle 3 in dedicated subsections.
- Finally, Section 5 validates and Section 6 concludes the work done within this cycle.

1.1 PERT chart of SEMIoTICS

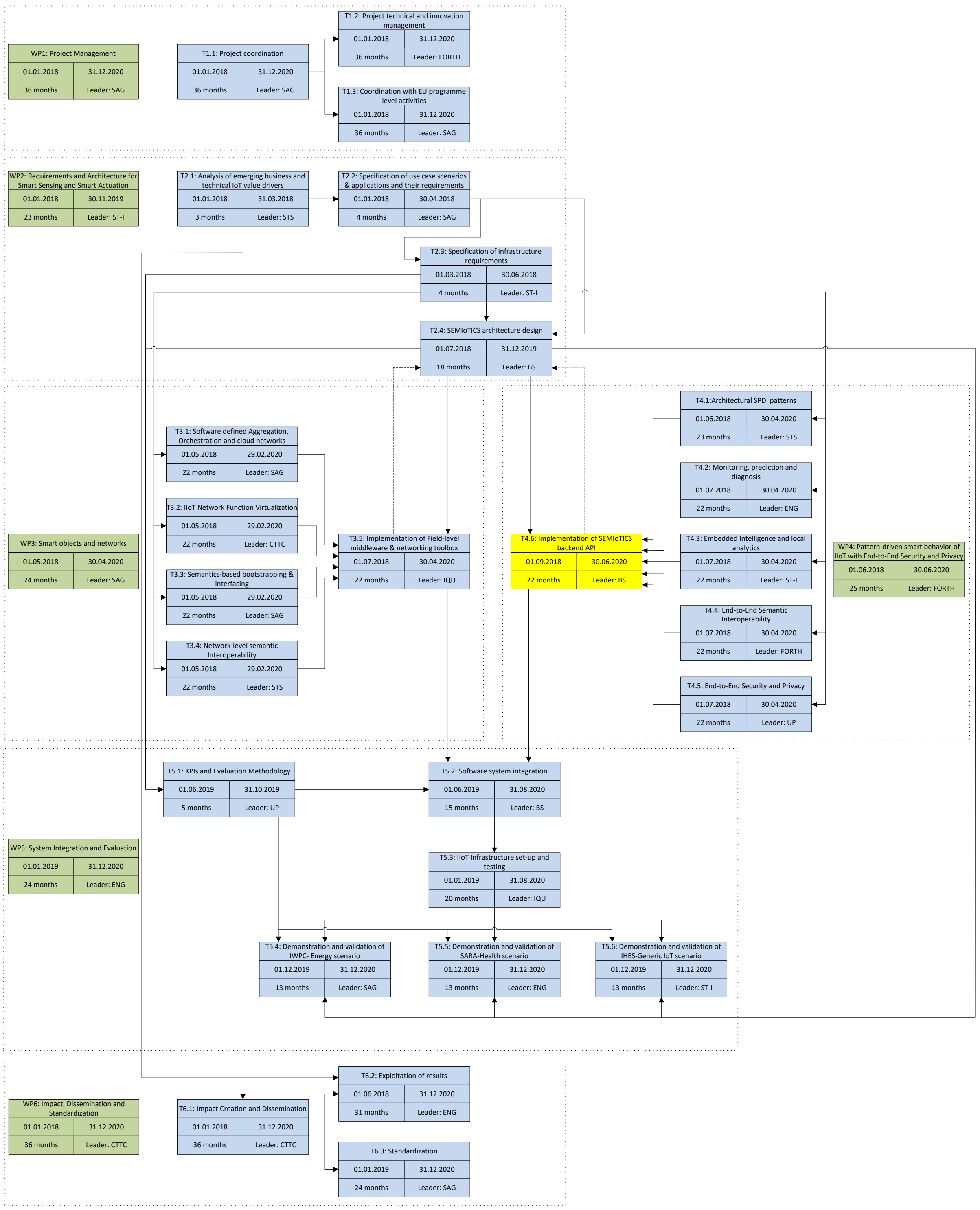

WP1: Project Management
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.1: Project coordination
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.2: Project technical and innovation management
01.01.2018
31.12.2020
36 months
Leader: FORTH
T1.3: Coordination with EU programme level activities
01.01.2018
31.12.2020
36 months
Leader: SAG
WP2: Requirements and Architecture for Smart Sensing and Smart Actuation
01.01.2018
30.11.2019
23 months
Leader: ST-I
T2.1: Analysis of emerging business and technical IoT value drivers
01.01.2018
31.03.2018
3 months
Leader: STS
T2.2: Specification of use case scenarios & applications and their requirements
01.01.2018
30.04.2018
4 months
Leader: SAG
T2.3: Specification of infrastructure requirements
01.03.2018
30.06.2018
4 months
Leader: ST-I
T2.4: SEMIoTICS architecture design
01.07.2018
31.12.2019
18 months
Leader: BS
T3.1: Software defined Aggregation, Orchestration and cloud networks
01.05.2018
29.02.2020
22 months
Leader: SAG
T3.2: IIoT Network Function Virtualization
01.05.2018
29.02.2020
22 months
Leader: CTTC
T3.3: Semantics-based bootstrapping & interfacing
01.05.2018
29.02.2020
22 months
Leader: SAG
T3.4: Network-level semantic Interoperability
01.05.2018
29.02.2020
22 months
Leader: STS
WP3: Smart objects and networks
01.05.2018
30.04.2020
24 months
Leader: SAG
T3.5: Implementation of Field-level middleware & networking toolbox
01.07.2018
30.04.2020
22 months
Leader: IQU
T4.6: Implementation of SEMIoTICS backend API
01.09.2018
30.06.2020
22 months
Leader: BS
T4.1:Architectural SPDI patterns
01.06.2018
30.04.2020
23 months
Leader: STS
T4.2: Monitoring, prediction and diagnosis
01.07.2018
30.04.2020
22 months
Leader: ENG
T4.3: Embedded Intelligence and local analytics
01.07.2018
30.04.2020
22 months
Leader: ST-I
T4.4: End-to-End Semantic Interoperability
01.07.2018
30.04.2020
22 months
Leader: FORTH
T4.5: End-to-End Security and Privacy
01.07.2018
30.04.2020
22 months
Leader: UP
WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy
01.06.2018
30.06.2020
25 months
Leader: FORTH
T5.1: KPIs and Evaluation Methodology
01.06.2019
31.10.2019
5 months
Leader: UP
T5.2: Software system integration
01.06.2019
31.08.2020
15 months
Leader: BS
WP5: System Integration and Evaluation
01.01.2019
31.12.2020
24 months
Leader: ENG
T5.3: IIoT Infrastructure set-up and testing
01.01.2019
31.08.2020
20 months
Leader: IQU
T5.4: Demonstration and validation of IWPC- Energy scenario
01.12.2019
31.12.2020
13 months
Leader: SAG
T5.5: Demonstration and validation of SARA-Health scenario
01.12.2019
31.12.2020
13 months
Leader: ENG
T5.6: Demonstration and validation of IHES-Generic IoT scenario
01.12.2019
31.12.2020
13 months
Leader: ST-I
T6.2: Exploitation of results
01.06.2018
31.12.2020
31 months
Leader: ENG
WP6: Impact, Dissemination and Standardization
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.1: Impact Creation and Dissemination
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.3: Standardization
01.01.2019
31.12.2020
24 months
Leader: SAG

FIGURE 1 PERT CHART

2 IMPLEMENTATION APPROACH

In SEMIoTICS, Task 4.6 is the main implementation task of WP4, which will deliver the SEMIoTICS components developed in WP4 in incremental release cycles. In the following sections, the software development and release processes will be detailed. Implementation has been divided into 3 development cycles as per Definition of Action. This has been consciously chosen due to the fact that entire project plan has been aligned with such an approach. Moreover, in this section there is detailed description of the development and release cycles based on Agile and Continuous Integration/Development (CI/CD) best practices which have been proven to be very efficient approach in the IT domain.

2.1 SEMIoTICS Development and Release Cycles

In the context of Task 2.4, we have designed the SEMIoTICS architecture and defined the architectural components of each layer. Each architectural component is associated with a respective functional module (i.e. component) with an owner assigned. These components are implemented with an iterative process, which follows the concept of CI. Such an iterative development process is performed in cycles, with each cycle ending with a new software release. Each release cycle consists of the following phases, also illustrated in Figure 2, and lasts approximately 4 months:

1. **Feature planning**: The consortium agrees on the features that will be implemented in the next release. This might occur during a feature planning meeting, or during the regular project meetings and calls. It defines all required mechanisms and interfaces in a high-level specification document, which also includes the test cases which will be adopted during system verification. This phase requires approximately 1 month.
2. **Development**: With the requirements document at hand, all required features are implemented by the responsible developers coordinated by component owners. Each developer is responsible for ensuring that the proposed features are properly implemented in the associated architectural component, as defined in Task 2.4, additionally ensuring that all related functionalities including legacy functionalities of the component are preserved. Furthermore, appropriate testing will ensure that the developed components and feature sets perform as specified. Development requires 2 months.
3. **Integration**: After completion of the development phase, changes are integrated into the main SEMIoTICS codebase. Automated non-regression and sanity tests are performed to rule-out regressions. This task requires 1-2 weeks.
4. **System testing**: The testing team deploys the new software release to the testbed and performs all the required system tests to validate that it runs as specified, further, this is essential to ensure that and new modules and features correctly interoperate with the rest of the system. In case of issues, they report back to the responsible developers and depending on the required effort, further, development might occur to fix the issue or move the issues for resolution in upcoming releases. This phase requires 2-3 weeks.
5. **System release**: Eventually, the developer generates all the release artifacts and documents and tags the current version of the software. In addition, a system release review meeting takes place to identify and discuss problems encountered during this release cycle.

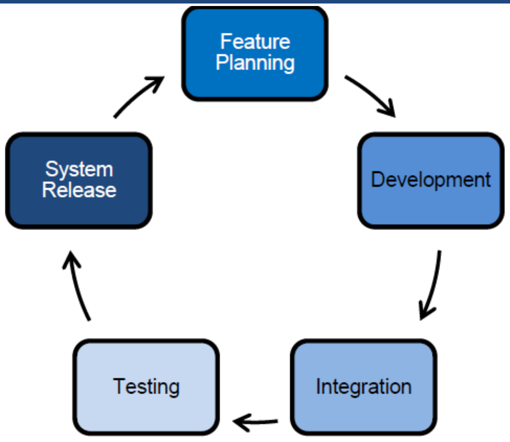


FIGURE 2 SEMIOTICS RELEASE CYCLE

Tentatively, the consortium adopted the following release schedule.

- M15 marked the start of the development process.
- On M17 (Cycle 1), the first software release was delivered, including the basic functionality of the SEMIoTICS backend implementation.
- On M23 (Cycle 2), the second software release was delivered, incorporating the pattern-driven smart behavior.
- On M30 (Cycle 3), the third release delivers the SEMIoTICS end-to-end architecture implementation.

2.2 SEMIoTICS development workflow

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

2.2.1 SEMIOTICS GIT BRANCHES

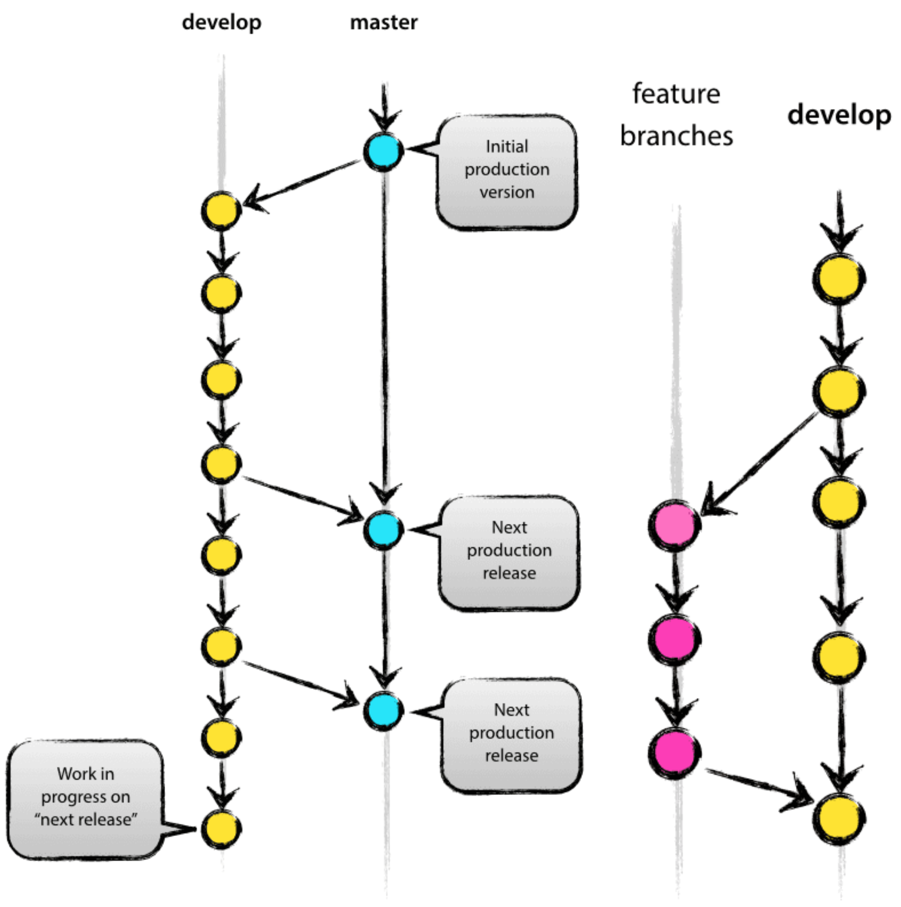


FIGURE 3 SEMIOTICS GIT REPOSITORY BRANCHES

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

In addition to the main branches (i.e., *master* and *develop*), *feature* branches may be used to develop new features for the upcoming or a future release. *Feature* branches generally exist as long as a new feature is in development and will eventually be merged back into the *develop* branch, to ultimately add the new feature to an upcoming release, or even discarded in case of an experiment that led to a dead-end. *Feature* branches are also created in the origin repo, so multiple developers can push to the same *feature* branch. Multiple *feature* branches may exist at a time.

2.2.2 CONTINUOUS INTEGRATION PIPELINE

A CI/CD pipeline is also part of GitLab features, in the form of a web application with an API that stores its state in a database. It manages the project builds and provides a Graphical User Interface (GUI) which gives an easy to understand overview of the project development process. Most importantly, the CI pipeline is closely integrated with the core features of GitLab. The GitLab CI pipeline is part of the SEMIoTICS testing framework and includes all required unit tests and integration tests. Tests can be authored by the respective developers or a separate testing team. Only if tests pass, then a new code is committed to the source code repository. Furthermore, the system performs nightly builds and in case of build failure notifies the responsible developers to fix the issue. The SEMIoTICS Continuous Integration processes include the following, which may be accomplished via the GitLab system, or additional tools:

- A ticketing system to assign tasks and feature requests to partners
- A task planning system to assign features to future releases
- Team collaboration tools (e.g., Messaging, File sharing, etc.)

It should be noted that access to the GitLab project is granted only for Consortium Members as per the Consortium Agreement.

3 CYCLE PLAN

The development of the WP4 components has been planned according to development cycles – from 1 to 3 (final) – as defined above. The plan of the cycles is related to the outputs of the different Tasks and the respective components as depicted in the SEMIoTICS Architectural Framework (FIGURE 4). More specifically, Task 4.6 provides the implementation of components defined within WP4 as well as the development of the backend API. Moreover, partial integration of the respective components that are also related to the outputs of the tasks as depicted in Figure 4 below is an important part of efforts within T4.6 however the main effort on that is planned within WP5.

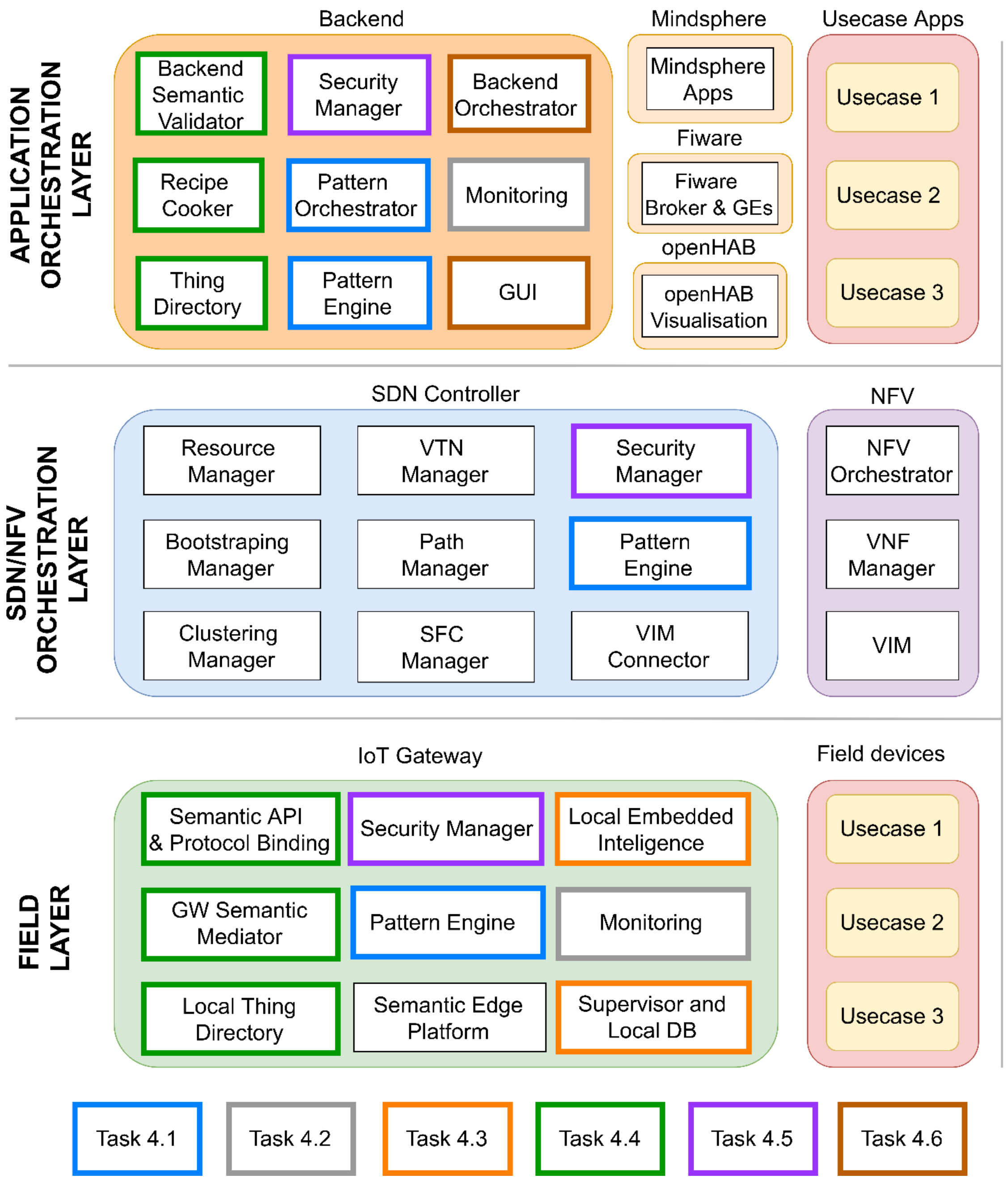


FIGURE 4 SEMIOTICS ARCHITECTURAL FRAMEWORK

Various components from SDN/NFV orchestration layer and field layer are mostly implemented in WP3, thus Table 2 only shows the cycle-assignment of components implemented within WP4 to development cycles. Each component is developed in at least two cycles.
It should be reiterated that this document is part of a sequence (D4.6, D4.7, and D4.13), with the current deliverable (D4.13) covering the final cycle. More details about the individual components can be found in Section 4.

TABLE 2 ASSIGNMENT OF COMPONENTS TO CYCLES

Component	Owner	Cycle 1	Cycle 2	Cycle 3
Backend Orchestrator	BS	Part 1	Part 2	Part 3
Pattern Orchestrator	STS	Part 1	Part 2	Part 3
Pattern Engine	STS	Part 1	Part 2	Part 3
Monitoring	ENG	-	Part 1	Part 2
Backend Semantic Validator	FORTH	Part 1	Part 2	Part 3
GUI	BS	Part 1	Part 2	Part 3
Backend Security Manager	UP	-	Part 1	Part 2
Recipe Cooker	SAG	Part 1	Part 2	Part 3
Thing Directory	SAG	Part 1	-	Part 2
Local Embedded Intelligence	ST	-	Part 1	Part 2

Every cycle plan is monitored with the use of the GitLab tool, while the feature backlog definition identified at the early stage of the project, is provided within Section 4. As per the Agile methodology, the backlog is constantly updated throughout the project.

4 FINAL CYCLE COMPONENTS

As mentioned above, an implementation of the SEMIOTICS framework solution imposes not only the implementation of the components but also designing suitable interactions between them. Not only the definition of components APIs is required, but also defining which components will be a consumer of which component API.
The landscape definition of the component interactions with API definitions have been initiated in Cycle 1, continued in an early stage of Cycle 2 and finalized in Cycle 3 as crucial for further development of the specific components.

4.1 Graphical User Interface (GUI)

As described in D4.6 and D.4.7, GUI is a component responsible for giving meaningful insights into the platform and centralized visualization of the whole framework as well as is a layer of presentation for specific use cases. During cycle 1 and cycle 2, there has been extensive analysis run, which outcome is designed. The following approaches have been taken and implemented:

- GUI that communicates through the API with an external application.
- GUI that loads the view itself from the external application.
- GUI that is dedicated to the given backend application.

Several views have been developed within cycle 3 and few of them have been updated. Also, the architecture of GUI has been enriched with several new components.

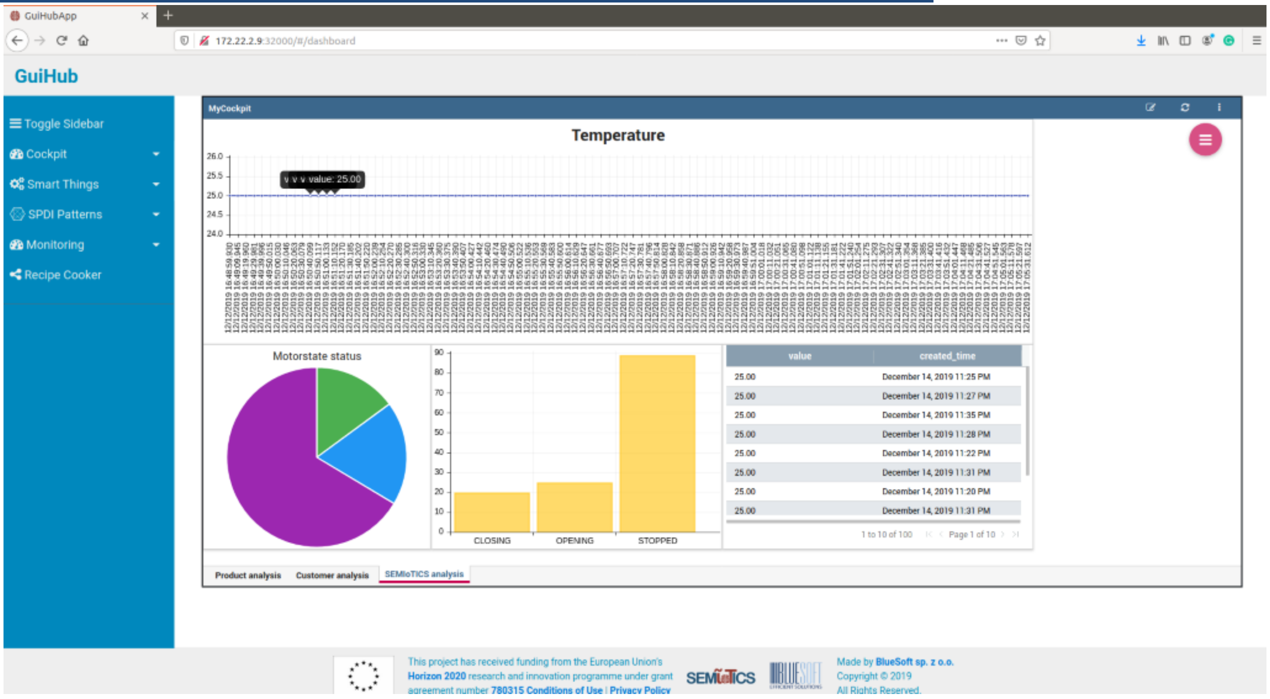
TABLE 3 GUI BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Initialize GUI application	Create a SpringBoot & Angular application	Cycle 1	Delivered
Create a view to perform basic actions on Things	Create all necessary endpoints for GUI. Create a graphical user interface. The interface should allow to register a thing description, delete a thing description, display all registered things, display things' details.	Cycle 1	Delivered
Provide support for multiple environments	Create maven profiles to facilitate the process of the application deployment	Cycle 1	Delivered
Prepare GUI for deployment on Backend Orchestrator	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes	Cycle 1	Delivered
Create a database for GUI	Create a database that should store information about registered things, their details including all the properties and actions and all the data gathered from them. Create entities, services, repositories. Add a database connection handler. Change the already existing implementation of methods so they can use database	Cycle 2	Delivered
Add dashboard functionality	Create PoC that allows a user to perform basic CRUD operations on dashboards and widgets.	Cycle 2	Delivered

Create a simulator of a thing.	Create a mock-up application which imitates the behavior of the real IoT device	Cycle 2	Delivered
Add a mechanism to gather historic data from IoT devices	Create a mechanism to collect data from IoT devices and save them in the database	Cycle 2	Delivered
Create a view that displays SPDI Patterns	Create a service that allows getting the SPDI Patterns from Pattern Orchestrator and to prepare them to be displayed in GUI. Create an interface that displays SPDI patterns from all of the SEMIoTICS's architecture layers and their details	Cycle 2	Delivered
Create a view that displays SPDI Recipes	Create a service that allows getting the SPDI Recipes from Pattern Orchestrator and to prepare them to be displayed in GUI. Create a GUI that displays SPDI recipes in the form of graphs.	Cycle 2	Delivered
Create a view to interact with Things.	Create a graphical user interface and a service that mediates between GUI and IoT Devices and allows to: get real-time properties values of sensors, perform an action on actuators,	Cycle 2	Delivered
Implement a fully functional user dashboard with widgets	Implement all the essential functions and views	Cycle 3	Delivered
Add routing to other SEMIoTICS' components	Create a bar that allows navigating through other SEMIoTICS' components	Cycle 3	Delivered

4.1.1 DEVELOPMENT STATUS

The main two tasks planned for development in Cycle 3 were the implementation of widgets and the placement of other components' URLs in the navbar. Widgets has been provided by integrating GUI with a FIWARE KNOWAGE component. The scope of the task delivered all essential functionalities for widget management in GUI. The pictures below depict the view of user's widgets and the visualisation of the data gathered from IoT sensors. The navbar expect for GUI's subsites contains redirection to other SEMIoTICS components like e.g. Recipe Cooker.



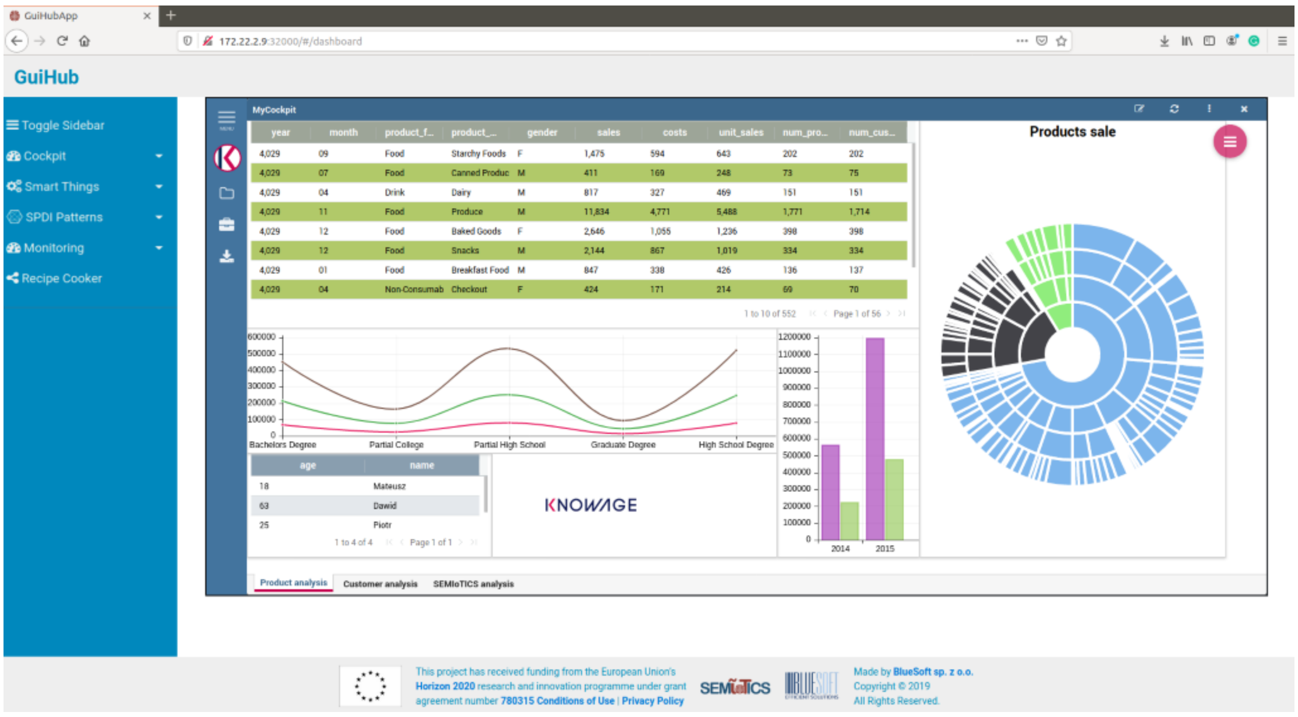
value	created_time
25.00	December 14, 2019 11:25 PM
25.00	December 14, 2019 11:27 PM
25.00	December 14, 2019 11:35 PM
25.00	December 14, 2019 11:28 PM
25.00	December 14, 2019 11:22 PM
25.00	December 14, 2019 11:31 PM
25.00	December 14, 2019 11:20 PM
25.00	December 14, 2019 11:31 PM

1 to 10 of 100 Page 1 of 10

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o. Copyright © 2019 All Rights Reserved.

FIGURE 5 DASHBOARD



year	month	product_f...	product_...	gender	sales	costs	unit_sales	num_pro...	num_cus...
4,029	09	Food	Starchy Foods	F	1,475	594	643	202	202
4,029	07	Food	Canned Produc	M	411	169	248	73	75
4,029	04	Drink	Dairy	M	817	327	469	151	151
4,029	11	Food	Produce	M	11,834	4,771	5,488	1,771	1,714
4,029	12	Food	Baked Goods	F	2,646	1,055	1,236	398	398
4,029	12	Food	Snacks	M	2,144	867	1,019	334	334
4,029	01	Food	Breakfast Food	M	847	338	426	136	137
4,029	04	Non-Consumab	Checkout	F	424	171	214	69	70

1 to 10 of 552 Page 1 of 56

age	name
18	Mateusz
63	Dawid
25	Piotr

1 to 4 of 4 Page 1 of 1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

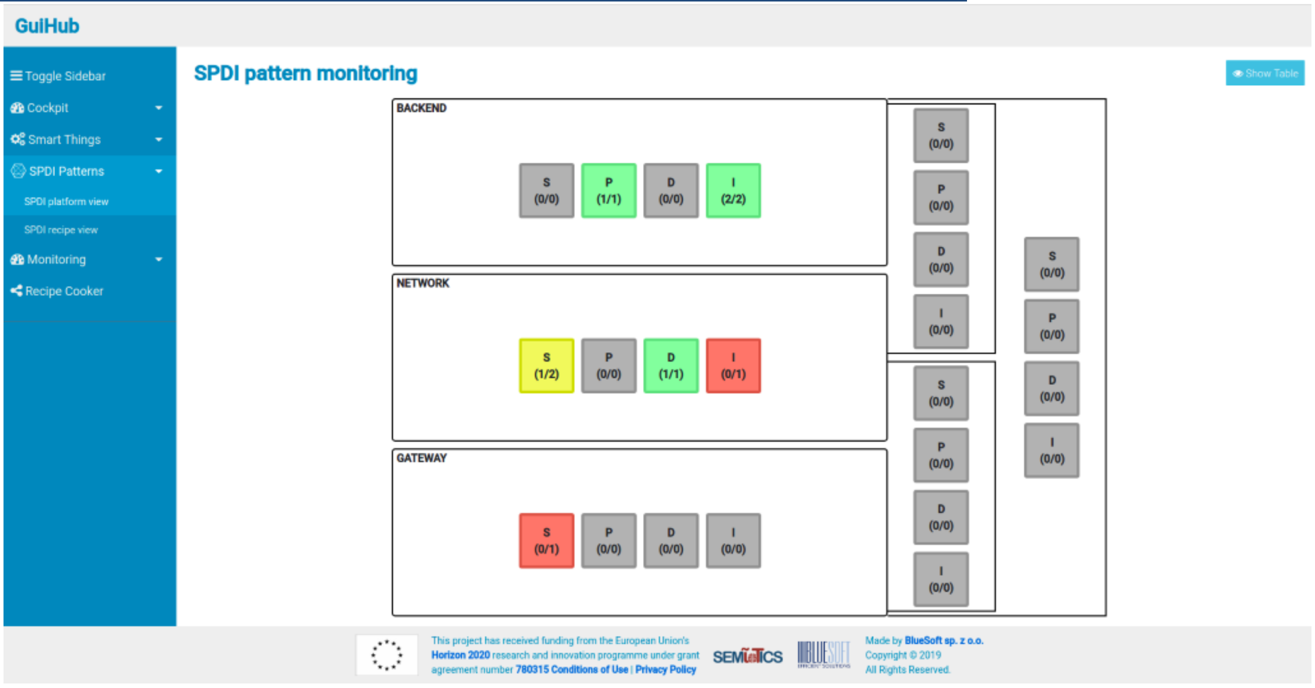
Made by BlueSoft sp. z o.o. Copyright © 2019 All Rights Reserved.

FIGURE 6 GUI

During the cycle, some of the project requirements changed and after having acquainted with the opinion of other components owners it was decided that a few functionalities which have already been delivered in GUI should be an amendment and a few new functionalities should be implemented and added to GUI. Table 4 shows the tasks which were additionally created during the third cycle.

TABLE 4 LIST OF ADDED AND AMENDMENT FUNCTIONALITIES

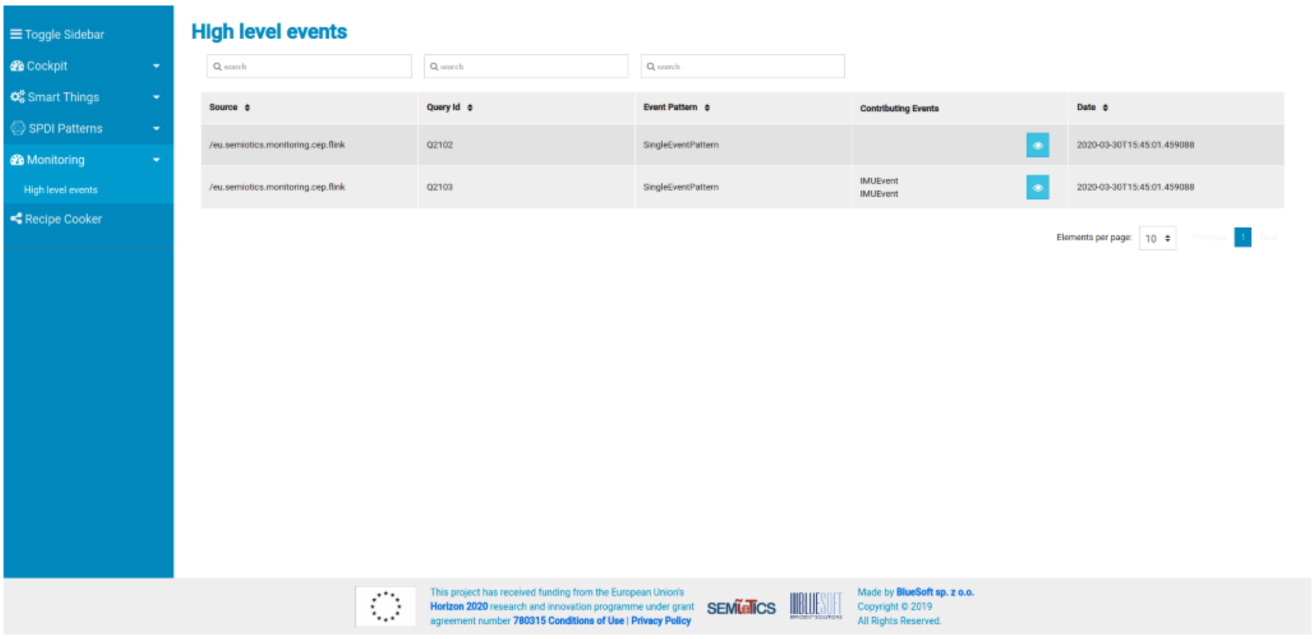
Feature/task scope	**Short description**	**New functionality / Amendment of already existing**	**Status**
Displaying Things per Thing Directory	A user has an availability to specify which Thing Directory he wants to display Things from	Amendment of already existing	Delivered
Displaying Monitoring High Level Events in GUI	New view with the visualisation of high-level events.	New functionality	Delivered
Refreshing SPDI Patterns	SPDI Pattern should be constantly refreshed and the user should be notified whenever the status of pattern changes	New functionality	Delivered
Implementation of oath 2.0	The access to GUI should be given only to users authorized in Security Manager	New functionality	Delivered
Improvement of graphical user interface	Refactor of visualisation	Amendment of already existing	Delivered



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 7 NEW SPDI PATTERN VIEW



High level events

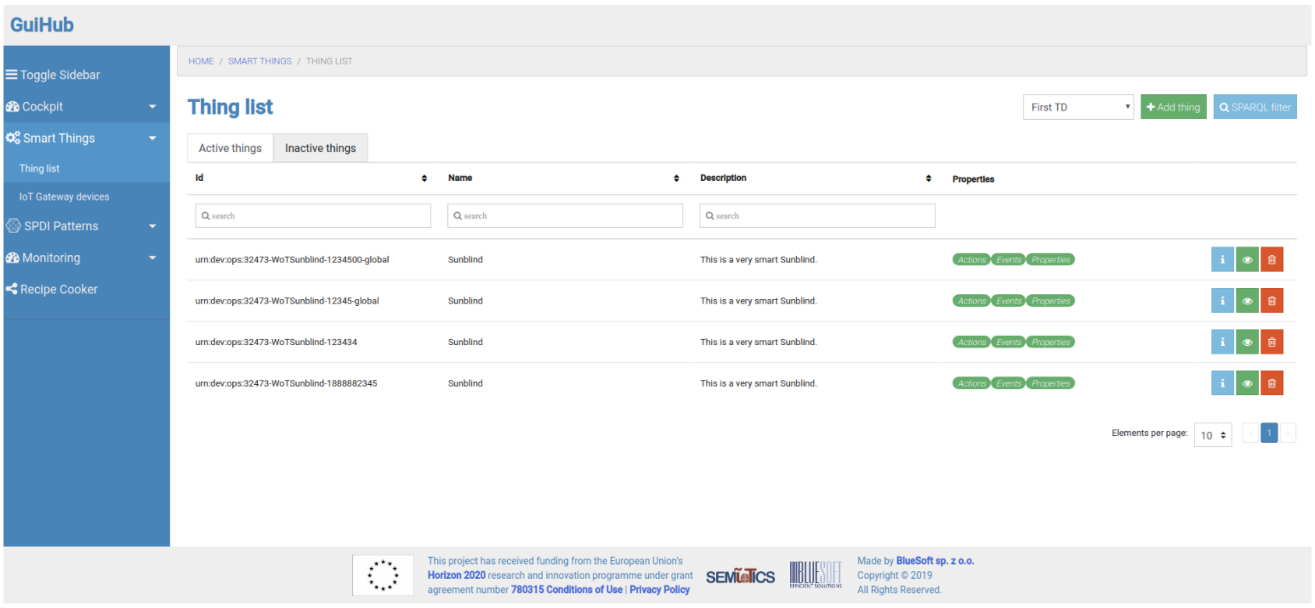
Source	Query Id	Event Pattern	Contributing Events	Date
/eu.semiotics.monitoring.cep.flink	Q2102	SingleEventPattern		2020-03-30T15:45:01.459088
/eu.semiotics.monitoring.cep.flink	Q2103	SingleEventPattern	IMUEvent IMUEvent	2020-03-30T15:45:01.459088

Elements per page: 10

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 8 NEW MONITORING EVENT VIEW



GuiHub

Toggle Sidebar · Cockpit · Smart Things · Thing list · IoT Gateway devices · SPDI Patterns · Monitoring · Recipe Cooker

HOME / SMART THINGS / THING LIST

Thing list

First TD · Add thing · SPARQL filter

Active things · Inactive things

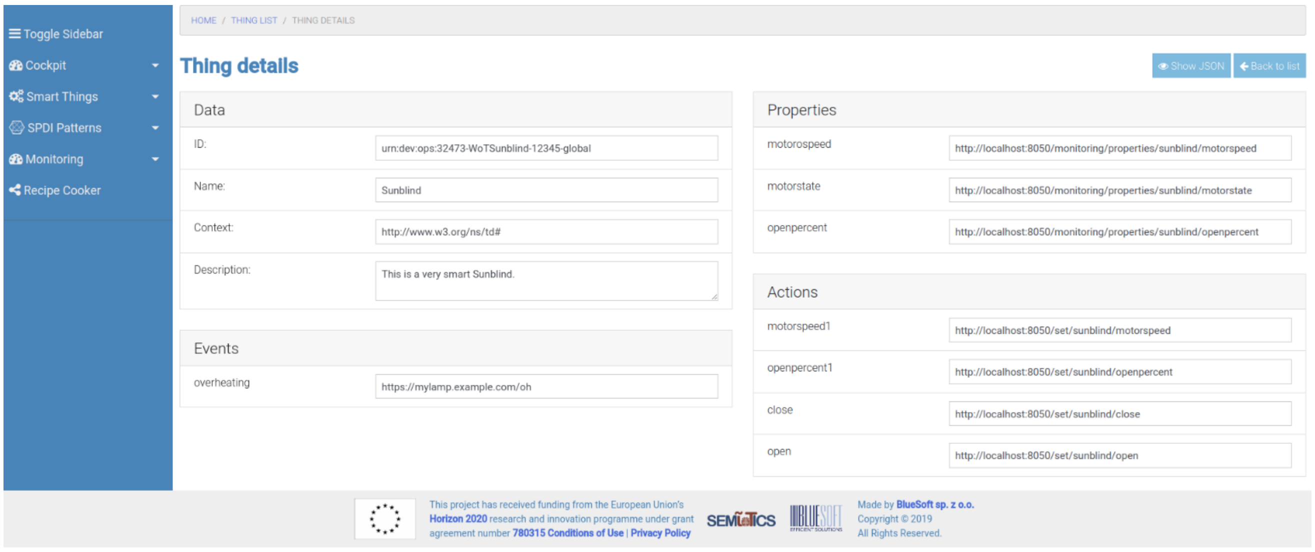
Id	Name	Description	Properties
urn:dev:ops:32473-WoTSunblind-1234500-global	Sunblind	This is a very smart Sunblind.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-12345-global	Sunblind	This is a very smart Sunblind.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-123434	Sunblind	This is a very smart Sunblind.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-1888882345	Sunblind	This is a very smart Sunblind.	Actions Events Properties

Elements per page: 10 · 1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o. Copyright © 2019 All Rights Reserved.

FIGURE 9 UPDATED THING LIST VIEW



Toggle Sidebar · Cockpit · Smart Things · SPDI Patterns · Monitoring · Recipe Cooker

HOME / THING LIST / THING DETAILS

Thing details

Show JSON · Back to list

Data

ID:	urn:dev:ops:32473-WoTSunblind-12345-global
Name:	Sunblind
Context:	http://www.w3.org/ns/td#
Description:	This is a very smart Sunblind.

Events

overheating	https://mylamp.example.com/oh

Properties

motorospeed	http://localhost:8050/monitoring/properties/sunblind/motorspeed
motorstate	http://localhost:8050/monitoring/properties/sunblind/motorstate
openpercent	http://localhost:8050/monitoring/properties/sunblind/openpercent

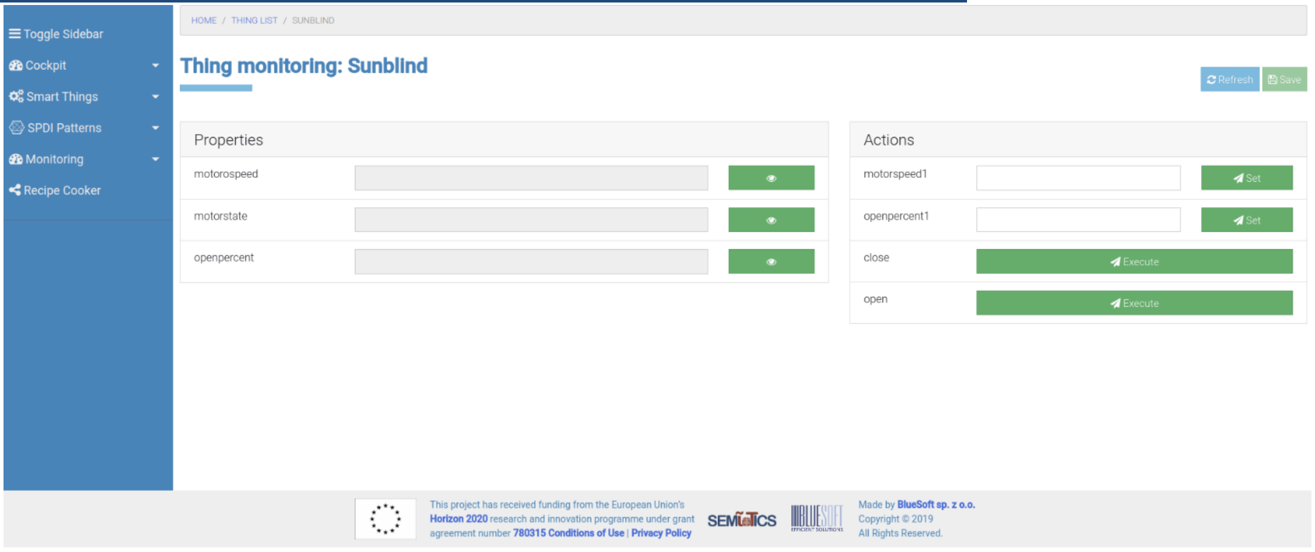
Actions

motorspeed1	http://localhost:8050/set/sunblind/motorspeed
openpercent1	http://localhost:8050/set/sunblind/openpercent
close	http://localhost:8050/set/sunblind/close
open	http://localhost:8050/set/sunblind/open

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o. Copyright © 2019 All Rights Reserved.

FIGURE 10 UPDATED THING DETAILS VIEW



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 11 UPDATED THING PROPERTIES AND ACTIONS VIEW

4.1.2 COMPONENT API INTERACTIONS DESCRIPTION

GUI is a module that overlays some components of the SEMIoTICS projects. Its main purpose is to support the visualization of individual components and the presentation of collected data in one IoT platform. According to the project assumptions, GUI integrates with Thing Directory, Local Thing Directories, IoT Gateway, Pattern Orchestrator, Monitoring, FIWARE Knowage, Security Manager and Recipe Cooker. Due to that fact, the description of each API created for integration with these components is provided in the table below. The technical aspect of each API is presented in the screenshots with the Swagger documentation.

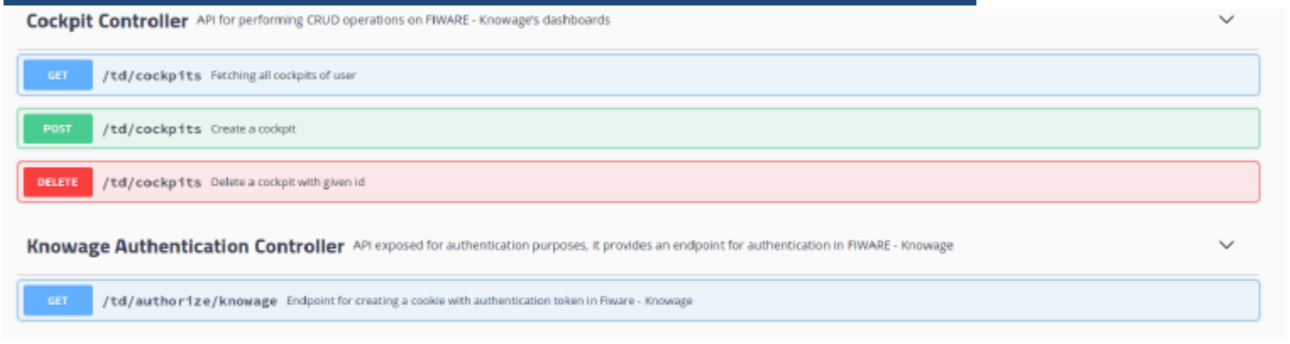
TABLE 5 GUI API'S

API	Status	API access type	Additional comments
GET td/authorize/knowage	Deployed	Internal	Access only through GUI component. Special sessionKey required.
GET td/cockpits	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
POST td/cockpits	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
DELETE td/cockpits	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/spdi/getGraphData	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/spdi/getRecipeList	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/spdi/getSpdiMonitoringData	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/spdi/getSpdiTableData	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/thingMonitoring getMonitoredValues	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.

POST td/thingMonitoring saveProperties	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
POST td/thingMonitoring/executeAction	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/{directoryId}/things	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
POST td/{directoryId}/things	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
DELETE td/{directoryId}/things	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/{directoryId}/things /{thingId}	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/filterAllThings	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/getDeletedThing	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/iot-gateway/devices	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/iot-gateway/ip-addresses	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
POST td/iot-gateway/register	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/monitoring/contributing-events	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
GET td/monitoring/high-level-events	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
POST td/monitoring/high-level-events	Deployed	Internal	Access only through GUI after login. API is inaccessible outside Kubernetes network.
Login	In progress	External	

4.1.2.1 APIS FOR FIWARE KNOWAGE INTEGRATION

Integration GUI with Knowage was created to visualize collected data from sensors and devices on highly extensive and efficient dashboards. To use all main functionalities provided by FIWARE Knowage, the above-mentioned APIs were developed. User can list all available cockpits (HTTP GET 'td/cockpits'), create or edit dashboards (HTTP POST 'td/cockpits') and delete existing cockpits(HTTP DELETE 'td/cockpits'). To enable communication with Knowage, GUI authorizes in Knowage and receives a special token.


Cockpit Controller API for performing CRUD operations on FIWARE - Knowage's dashboards

GET /td/cockpits Fetching all cockpits of user

POST /td/cockpits Create a cockpit

DELETE /td/cockpits Delete a cockpit with given id

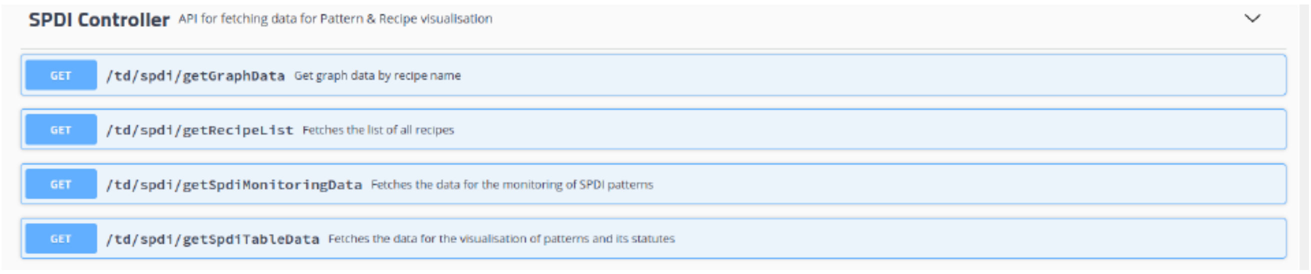
Knowage Authentication Controller API exposed for authentication purposes, it provides an endpoint for authentication in FIWARE - Knowage

GET /td/authorize/knowage Endpoint for creating a cookie with authentication token in Fiware - Knowage

FIGURE 12 APIS FOR INTEGRATION WITH KNOWAGE

4.1.2.2 APIS FOR PATTERN ORCHESTRATOR INTEGRATION

This integration aims to support Pattern Orchestrator in monitoring the current state of SPDI patterns from all recipes and location SPDI patterns in an individual layer e.g. backend, network, gateway. Additionally, GUI can present existing recipes in the interactive graph form as a combination of nodes, links, and layers. APIs developed in the GUI allows user to view SPDI patterns in real-time in two forms, in view with tiles (HTTP GET 'td/spdi/getSpdiMonitoringData') or in table view (HTTP GET 'td/spdi/getSpdiTableData') . User can also see all existing recipes (HTTP GET 'td/spdi/getRecipeList') or watch recipe in graph form (HTTP GET 'td/spdi/getGraphData').


SPDI Controller API for fetching data for Pattern & Recipe visualisation

GET /td/spdi/getGraphData Get graph data by recipe name

GET /td/spdi/getRecipeList Fetches the list of all recipes

GET /td/spdi/getSpdiMonitoringData Fetches the data for the monitoring of SPDI patterns

GET /td/spdi/getSpdiTableData Fetches the data for the visualisation of patterns and its statutes

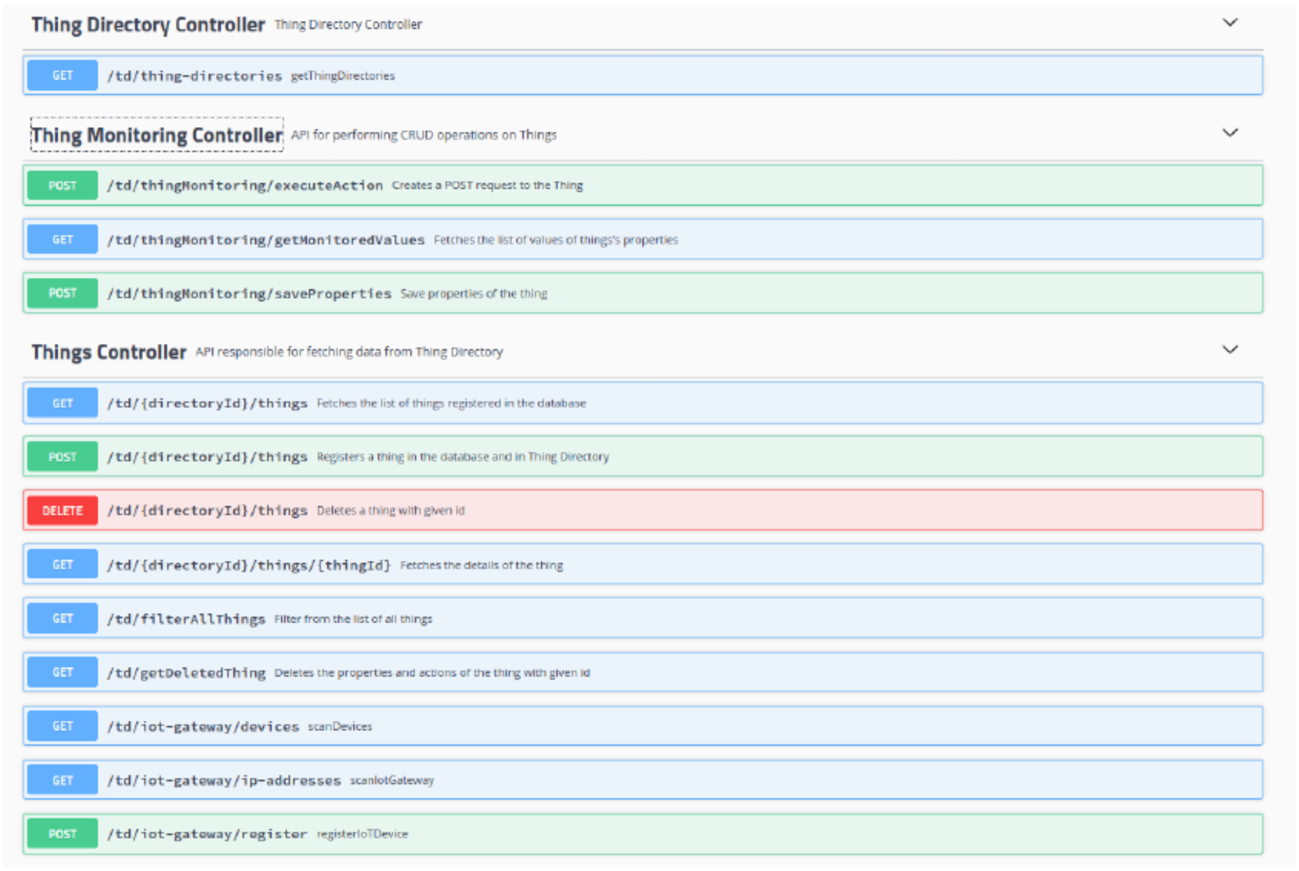
FIGURE 13 APIS FOR INTEGRATION WITH PATTERN ORCHESTRATOR

4.1.2.3 APIS FOR THING DIRECTORY AND IOT GATEWAY INTEGRATION

Integration with Thing Directory, Local Thing Directories, and IoT Gateway provides the largest number of endpoints. This kind of integration was created to visualize devices registered in Global Thing Directory or Local Thing Directories, register, and delete new ones. Additionality allows users to interact with devices and watch their properties. The development of APIs for IoT Gateway allows finding new devices and register them directly through this component. User can use one of the existing endpoints:

- HTTP GET 'td/thing-directories' to list all available Local Thing Directories and one Global Thing Directory,
- HTTP POST 'td/thingMonitoring/executeAction' to run one of selected action from device,
- HTTP GET 'td/thingMonitoring/getMonitoredValues' to get current values of thing's properties,
- HTTP POST *td* thingMonitoring/saveProperties' to start collecting data for selected property of
- device,
- HTTP GET 'td/{directoryId}/things' to get all thing for selected Thing Directory,
- HTTP POST 'td/{directoryId}/things' to register a new thing in selected Thing Directory,
- HTTP DELETE 'td/{directoryId}/things' to delete thing for selected Thing Directory,
- HTTP GET 'td/{directoryId}/things/{thingId}' to get details of selected thing,

- HTTP GET 'td/filterAllThings' to filter Thing Directory using SPARQL syntax query,
- HTTP GET 'td/getDeletedThing' to get details of deleted thing for given ip,
- HTTP GET 'td/iot-gateway/devices' to scan IoT Gateway in the range of ip addresses to find new
    - devices,
- HTTP GET 'td/iot-gateway/ ip-addresses' to scan IoT Gateway in the range of ip addresses to find ip
    - addresses of devices,
- HTTP POST 'td/iot-gateway/register' to register a new device in Global Thing Directory using IoT
    - Gateway.


Thing Directory Controller Thing Directory Controller

GET /td/thing-directories getThingDirectories

Thing Monitoring Controller API for performing CRUD operations on Things

POST /td/thingMonitoring/executeAction Creates a POST request to the Thing

GET /td/thingMonitoring/getMonitoredValues Fetches the list of values of things's properties

POST /td/thingMonitoring/saveProperties Save properties of the thing

Things Controller API responsible for fetching data from Thing Directory

GET /td/{directoryId}/things Fetches the list of things registered in the database

POST /td/{directoryId}/things Registers a thing in the database and in Thing Directory

DELETE /td/{directoryId}/things Deletes a thing with given id

GET /td/{directoryId}/things/{thingId} Fetches the details of the thing

GET /td/filterAllThings Filter from the list of all things

GET /td/getDeletedThing Deletes the properties and actions of the thing with given id

GET /td/iot-gateway/devices scanDevices

GET /td/iot-gateway/ip-addresses scanIotGateway

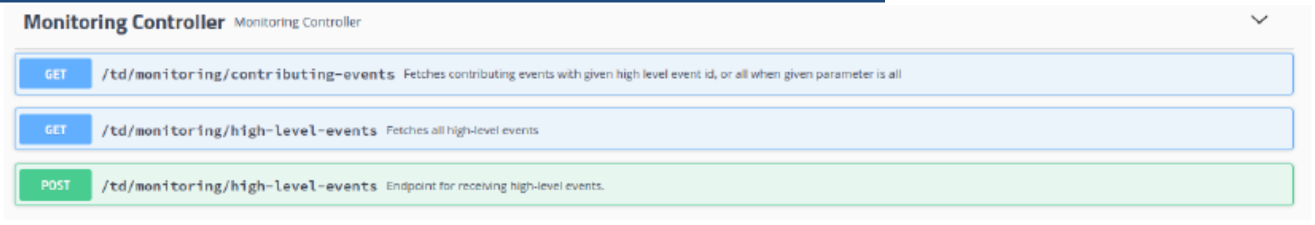
POST /td/iot-gateway/register registerIoTDevice

FIGURE 14 APIS FOR INTEGRATION WITH TDS AND IOT GATEWAY

4.1.2.4 APIS FOR MONITORING INTEGRATION

Integration between GUI and Monitoring component was created to provide visualization for predictive monitoring of events that might occur in the whole SEMIoTICS platform. It allows users to view high-level-events and contributing-events, and preparing queries with the definition of high-level-event. User can use one of the existing endpoints:

- HTTP GET 'td/monitoring/contributing-events' to get all contributing events for selected high-level-event,
- HTTP GET 'td/monitoring/high-level-events' to get all high-level-events that had occurred ,
- HTTP POST 'td/monitoring/high-level-events' to register query with definition of high-level-event.



Monitoring Controller Monitoring Controller

GET /td/monitoring/contributing-events Fetches contributing events with given high level event id, or all when given parameter is all

GET /td/monitoring/high-level-events Fetches all high-level events

POST /td/monitoring/high-level-events Endpoint for receiving high-level events.

FIGURE 15 APIS FOR INTEGRATION WITH MONITORING

4.1.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

After all 3 cycles, GUI is a fully working component integrated with other SEMIoTICS components as presented in Deliverable 5.2. A detailed description of development progress can be found in D4.6 for cycle 1 and in D4.7 for cycle 2. According to the initial assumption, GUI should meet 3 basic requirements:

- communication through the API with an external application,
- loading the view itself from the external application,
- redirecting to the given backend application.

All of the abovementioned requirements were implemented in the next steps. In cycle 2 communication was established through the API with external applications that include:

- communication with Thing Directory to show and interact with all registered devices,
- communication with Pattern Orchestrator to visualize real-time SPDI patterns.

Communication with Monitoring API to visualize high-level-events was implemented in cycle 3. In this cycle it was also added loading the view from an external application which was Knowage- one of the FIWARE Generic Enabler for complex data visualization. Additionally, in GUI was implemented redirection to Recipe Cooker that is one of SEMIoTICS backend applications. The redirection to SDN/NFV development view, that is a dedicated GUI for SPDI patterns is still in progress. The full sidebar with navigation to all SEMIoTICS views and APIs are presented below.

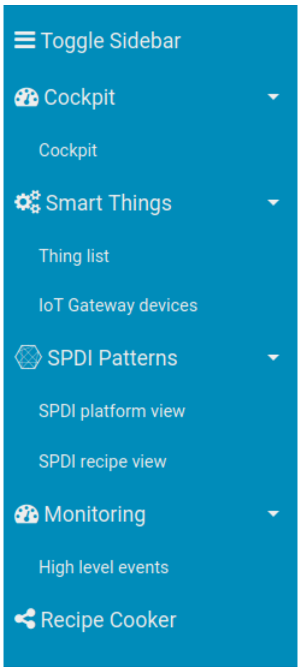
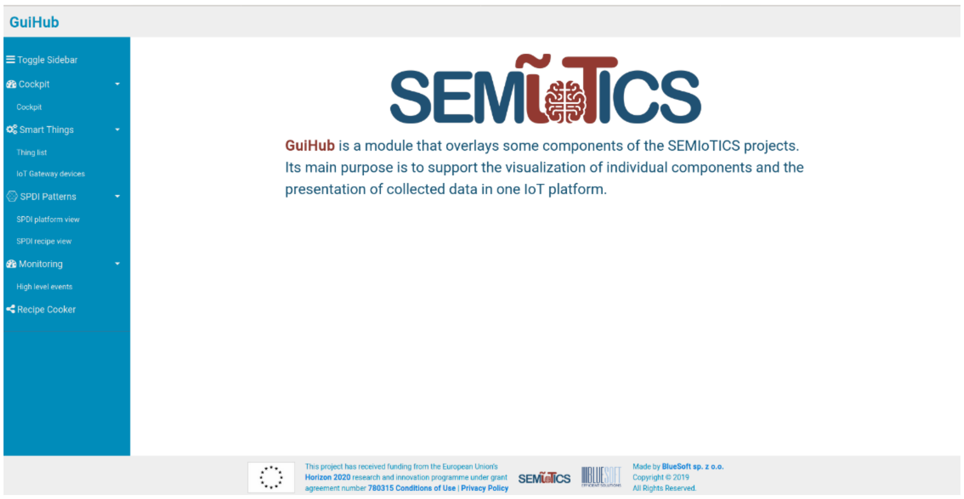


FIGURE 16 FULL SIDEBAR WITH NAVIGATION TO ALL COMPONENTS

During all 3 cycles, the basic assumptions were expanded and additional functionalities were added to adjust to the project requirements. All changes were easily applicable thanks to generic component architecture. The most significant improvements that were developed:

- support for all Local Thing Directories, not only for one Global Thing Directory,
- scanning and registering new devices through IoT Gateway,
- presenting recipe combined with SPDI patterns as an interactive graph.

The Picture below depicts GUI in the final version with the full navigation to external components.



GuiHub is a module that overlays some components of the SEMIoTICS projects. Its main purpose is to support the visualization of individual components and the presentation of collected data in one IoT platform.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 17 FINAL GUI VIEW

The most significant aspect during development and integration components was to provide safe and secure communication. To achieve this aim a special component AEP and Policy Enforcer Point (PEP) were developed. Each time the GUI component sends a request to another component it is signed by AEP and a special application token is added. Before the request is received by an external component, PEP gets an application token and validates it in Security Manager. If GUI has permission to communicate with the application, PEP sends a request to this app and returns a response to GUI as presented below.

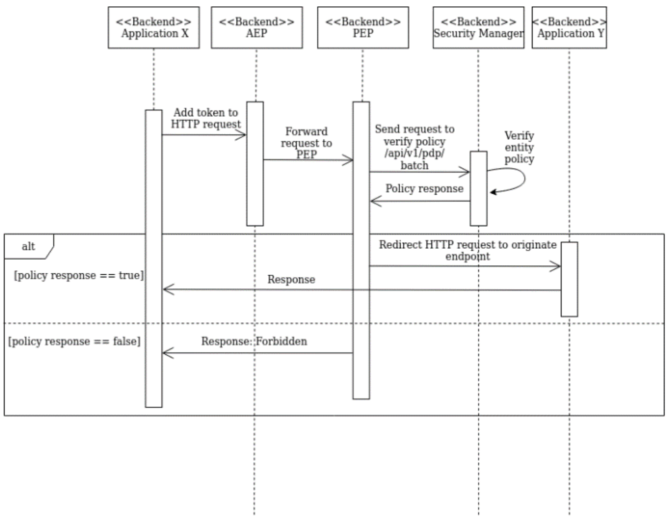

<<Backend>> Application X
<<Backend>> AEP
<<Backend>> PEP
<<Backend>> Security Manager
<<Backend>> Application Y
Add token to HTTP request
Forward request to PEP
Send request to verify policy /api/v1/pdp/batch
Verify entity policy
Policy response
alt
[policy response == true]
Redirect HTTP request to originate endpoint
Response
[policy response == false]
Response: Forbidden

FIGURE 18 DIAGRAM WITH SECURED COMMUNICATION PROVIDED BY AEP AND PEP

Using AEP and PEP components in SEMIoTICS architecture protects against access to data of unauthorized users or applications.

4.1.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO GUI

GUI component should meet Semiotics requirements and KPIs as described in D5.1. According to these requirements customized solutions were implemented during the development process. A description of KPIs with the detailed substantiation is presented in the table below.

KPI id	Description	Status of development	Substantiation
R.P.1	The collection of raw data MUST be minimized.	Done	GUI application uses and storages only necessary data collected from all devices to the SEMIoTICS platform. GUI contains internal components ThingWorker and ThingOrchestrator dedicated to collecting data. Thing Orchestrator is a component responsible for creating and distribute jobs between Thing Workers and also for deleting assigned jobs to Thing Workers. Job is created when a user of the GUI starts collecting measurements. Thing Worker is responsible for collecting data for specific properties of devices with a set by user frequency. Thus, not all properties of devices are collected but only selected. Thing Worker stops saving measurements form devices when the job is deleted or when devices are off. In GUI there is no redundant and unnecessary data.
R.P.2	The data volume that is collected or requested by an IoT application MUST be minimized (e.g. minimize sampling rate, amount of data, recording duration, different parameters).	Done	GUI application uses and storages only necessary data for communication with related components of the SEMIoTICS platform. Data that is used in the GUI application but stored in databases of individual components is only requested and immediately visualized. Examples of such interactions are communication with the Thing Directory and Pattern Orchestrator components, where received data is mapped and transformed to show it in a form that the user can understand. Therefore, there are no tables repetitions and no data redundancy. GUI also sends requests with different parameters for individual requests to reduce the amount of data transferred. For data that is used only in GuiHub dedicated PostgreSQL database was created. A relational data model is the most suitable solution and ensures that the database is lightweight because it contains only a few tables. The tables in this database are tailored to the needs of the application and data models used. GUI database collects and stores only selected measurements from devices registered to the SEMIoTICS platform.
R.P.3	Storage of data MUST be minimized.	Done	GUI database stores mostly thing's metadata such as properties, actions, the

			data essential for the worker orchestrator, and finally the values gathered from sensors. This is the minimum data that is required to accomplish the given tasks.
R.P.4.	A short data retention period MUST be enforced, and maintaining data for longer than necessary avoided.	Done	The user can specify the retention period of the measurement gathered by IoT devices. Data from IoT devices is collected by default by 1 month and can be customized by user to 3 months. When a data collection expires, the measurements are not saved in the database.
R.P.9	Repeated querying for specific data by applications, services, or users that are not intent to act in this manner SHALL be blocked.	In progress	In this stage of development of SEMIoTICS, we do not provide this functionality. We have not data to determine the minimal intervals of repeated queries for specific data. This is an area that can be improved after collecting the requirements from users. Currently, database performance is the only limitation in the frequency of data access.

4.2 Backend Orchestrator

Backend orchestrator is a component responsible for integrating all backend services and exposing APIs. Kubernetes (https://kubernetes.io) has been chosen as a component responsible for orchestration of the SEMIoTICS backend. Kubernetes is an open source project that enables declarative framework orchestration that has become a standard and is available to install on most of the platforms. Additionally, this technology is in line with proposed micro services architecture (described widely in D2.5).
Most important Kubernetes features are[1]:

- Kubernetes provides a container-centric management environment,
- Kubernetes orchestrates computing, networking, and storage infrastructure on behalf of user workloads,
- Kubernetes provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.

The development of the Backend Orchestrator component has been continued within Cycle 3.

Within Table 6 updated backlog of the tasks planned for the component is visible with the given status of the implementation. Further sections provide more details of the implementation.

The technologies which have been chosen to be used for backend orchestration are the following:

- Kubernetes – technology used as backend orchestrator to orchestrate backend applications
- Ansible 2.5 – technology used as a tool to automatize installation of Kubernetes and its dependencies
- Docker – technology used for containerization of applications written in different languages

TABLE 6 BACKEND ORCHESTRATOR BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Comparison and choosing the technology for Backend Orchestrator	Comparison of OpenStack, Kubernetes, and OpenShift.	Cycle 1	Delivered
Installation of Kubernetes on a cloud server for fast testing the chosen technology	Creating an instance of Kubernetes Cluster on AWS. Testing process to determine the size of resources for the physical cluster.	Cycle 1	Delivered
Creating a docker images repository	Creating a repository for Docker images on the GitLab.	Cycle 1	Delivered
First installation of Backend Orchestrator on BLS cluster	Creating the ansible script for installing required tools on a cluster. Testing one node Kubernetes architecture	Cycle 1	Delivered
Changing the internal architecture of Backend Orchestrator	Creating at least two nodes. There have to be a master node and a slave.	Cycle 1	Delivered
Implement a proxy mechanism in PEP	Implement a proxy mechanism to intercept HTTP traffic going to the main application and authorize the request in Security Manager	Cycle 1&2	Delivered
Add a proxy application to authenticate requests	Add mitmproxy application as an Authentication Enforcement Point which adds the client's token to an HTTP request	Cycle 1&2	Delivered
Preparation of PEP for deployment on Backend Orchestrator	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes	Cycle 1&2	Delivered

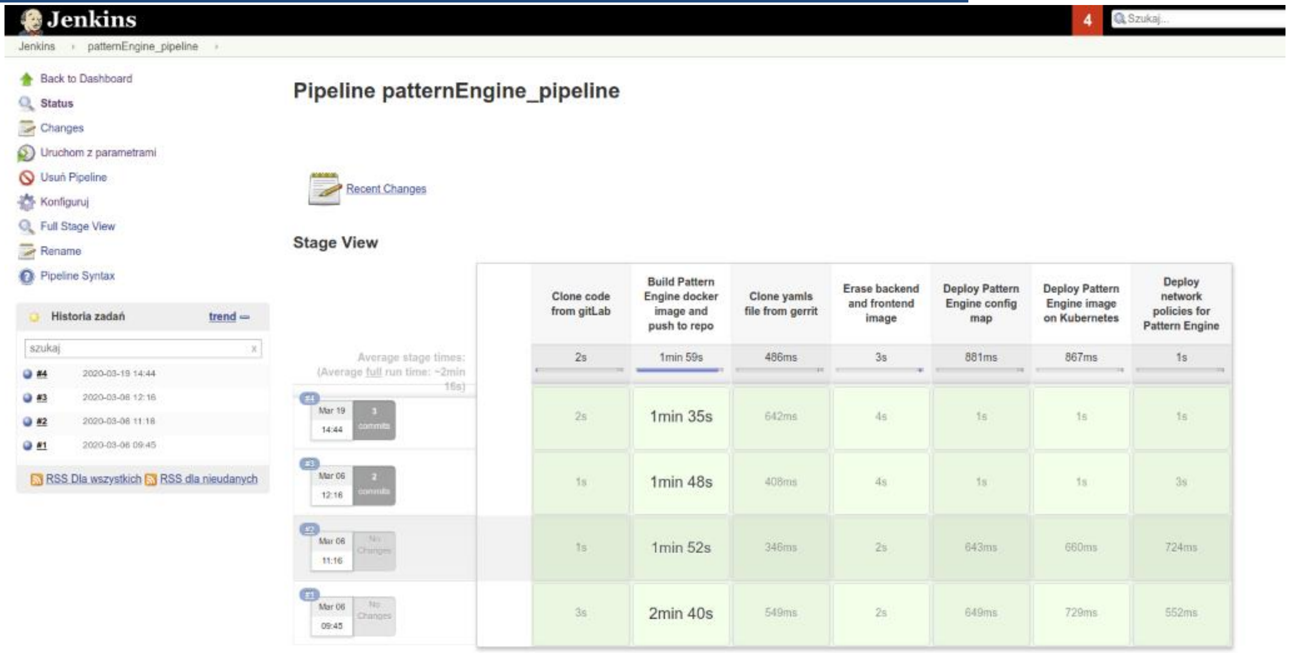
[1] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Set of access rules for consortium partners to Backend Orchestrator	Configure the namespaces and roles on Kubernetes. Creating the script that assigns permission per roles and namespaces.	Cycle 2&3	Delivered
Develop the scheme of deploying the component	Creating the script of deployment, service and config map for example component.	Cycle 2	Delivered
Deploying GUI and Thing Simulator on Kubernetes cluster	Developing the structure of deployment files for GUI and Thing Simulator	Cycle 2	Delivered
Automate the repeatable process of deploying the components.	Installation Jenkins on the machine. Creating the access rules for Jenkins to the BO.	Cycle 2	Delivered
Enabling communication between components deployed on Backend Orchestrator and to external applications	Develop the scripts that allow to expos the component to externals networks	Cycle 2	Delivered
Performing the test of communication between components	Testing internal and external communication between deployed components	Cycle 2	Delivered
Develop the way of storage and updating the credentials for externals applications	Set of rules about storage and user credentials for GitLabRepository.	Cycle 2	Delivered
Manual deployment of Thing Directory	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes. Deployment component	Cycle 2	Delivered
Creating and configuration of the tool for the administrator of Backend Orchestrator	Configure a dashboard that shows the state of the cluster. Creating the notification when the dangerous state of a cluster.	Cycle 2	Delivered
Creating deployment files for AOL components	Create deployment .yml files for GUI, Security Manager, Think Directory, Think Simulator and Think Worker, which allows deployment on Kubernetes.	Cycle 2	Delivered
Create automatized jobs for deploying components.	Create CI/CD pipeline that allows deploying the following components GUI, Security Manager, Think Directory, Think Simulator and Think Worker on Kubernetes after manual initialization.	Cycle 2	Delivered
Creating deployment files for AOL components	Create deployment .yml files for Pattern Engine, Recipe Cooker, Backend Semantic Validator, Pattern Orchestrator which allows deployment on Kubernetes.	Cycle 3	Delivered
Create automatized jobs for deploying components.	Create CI/CD pipeline that allows deploying the following components: Pattern Engine, Recipe Cooker, Backend Semantic Validator, Pattern Orchestrator on Kubernetes after manual initialization.	Cycle 3	Delivered

4.2.1 DEVELOPMENT STATUS

The development outcome of cycle 3 were deployment files for all the component that have not been deployed during the previous cycles e.g. Pattern Engine, Backend Semantic Validator, Recipe Cooker, Pattern Orchestrator. Each of those deployment has dedicated pipeline in Jenkins to facilitate the process of deployment. Every pipeline fetches the code of an application, builds the docker image and eventually, deploys it on the Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: backend-semantic-validator
 namespace: semiotics
 labels:
   app: bsv
spec:
 replicas: 1
 selector:
   matchLabels:
     app: bsv
 template:
   metadata:
     labels:
       app: bsv
   spec:
     containers:
       - name: bsv
         image: registry.gitlab.com/semiotics/backend/semantic-mediator:latest          resources:
           requests:
             memory: "230Mi"
             cpu: "100m"
           limits:
             memory: "460Mi"
             cpu: "200m"
         imagePullPolicy: Always
         ports:
           - containerPort: 8086
     imagePullSecrets:
       - name: blue-k8s
---
kind: Service
apiVersion: v1
metadata:
 name: bsv-svc
 namespace: semiotics
spec:
 ports:
   - nodePort: 31006
     port: 8086
     targetPort: 8086
 selector:
   app: bsv
 sessionAffinity: None
```



#4	2020-03-19 14:44
#3	2020-03-08 12:16
#2	2020-03-08 11:18
#1	2020-03-06 09:45

	Clone code from gitLab	Build Pattern Engine docker image and push to repo	Clone yamls file from gerrit	Erase backend and frontend image	Deploy Pattern Engine config map	Deploy Pattern Engine image on Kubernetes	Deploy network policies for Pattern Engine
Average stage times: (Average full run time: ~2min 16s)	2s	1min 59s	486ms	3s	881ms	867ms	1s
#4 Mar 19 14:44 3 commits	2s	1min 35s	642ms	4s	1s	1s	1s
#3 Mar 06 12:16 2 commits	1s	1min 48s	408ms	4s	1s	1s	3s
#2 Mar 06 11:16 No Changes	1s	1min 52s	346ms	2s	643ms	660ms	724ms
#1 Mar 06 09:45 No Changes	3s	2min 40s	549ms	2s	649ms	729ms	552ms

FIGURE 19 AN EXAMPLE OF JENKINS PIPELINE

4.2.2 COMPONENT API INTERACTIONS DESCRIPTION

Backend Orchestrator is based entirely on Kubernetes. To successfully accomplish the functionalities of the component, already existing APIs have been adjusted and configured in Kubernetes accordingly to the project's needs and use cases. Because of aforementioned reasons, none of APIs have been developed throughout any cycle. The table below shows Kubernetes API which have been used for the development of Backend Orchestrator.

TABLE 7 KUBERNETES API DETAILS

API	Status	API access type	Additional comments
K8s HPA	Discarded	Internal	Access only for components of Kubernetes network. The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization
K8s CronJobs	Discarded	Internal	Currently no use case is foreseen for this service. If any need will be identified, it can be set up.
K8s Ingress	Deployed	Internal	Access only for components of Kubernetes network. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster
K8s PVCs	Deployed	Internal	Access only for components of Kubernetes network. a request for storage by a user.
K8s PVs	Deployed	Internal	Access only for components of Kubernetes network. a piece of storage in the cluster. used to store data in a way that it persists beyond the lifetime of a pod
K8s Secrets	Deployed	Internal	Access only for components of Kubernetes network.

			an object that contains a small amount of sensitive data such as a password, a token, or a key.
K8s Jobs	Deployed	Internal	Access only for components of Kubernetes network. a **Job** is a controller object that represents a finite task.
K8s Deployments	Deployed	Internal	Access only for components of Kubernetes network. An object for management a set of identical pods.
K8s Services	Deployed	Internal	Access only for components of Kubernetes network. defines a logical set of Pods and a policy by which to access them
K8s Pods	Deployed	Internal	Access only for components of Kubernetes network. is a group of one or more containers , with shared storage/network, and a specification for how to run the containers.
K8s Variables	Deployed	Internal	Access only for components of Kubernetes network. An Object which stores a non-confidential data as key-value pairs.

4.2.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

During development cycles, the best tool for orchestration has been chosen, tested, and used in the working environment. During the first cycle, three possible orchestrators have been investigated: OpenStack, OpenShift, and Kubernetes. The best option for the SEMIoTICS proved to be Kubernetes. Its capabilities most suit microservices architecture of the SEMIoTICS platform. Tests have been performed on bare metal as well on the cloud provider cluster. There was initial implementation performed and administration rules developed. The architecture of a working cluster has been established and configured. The repository for Docker images has been established and tested with two components.

In the second cycle, the priority was the automatization of component deployment. Jenkins has been chosen to create an automatic process of deployment which consists of a compilation of pushed code to GitLab repository, creation of Docker image, and deployment on a Kubernetes cluster. In the testing process, the deployment files for each component have been written and test used. Security policies have been also implemented, the roles on Kubernetes have been assigned to each user based on best practices used in CI/CD solutions. Role configuration used in Backend Orchestrator ensures that each user has limited access to a cluster which might sometimes bring difficulties but prevents many actions that may disturb the proper operation of orchestrator. The connectivity aspects have been tested and implemented with appropriate policies and workflows in internal networking. Configuration of externals connectivity has been done and tested. The secure process of storage and usage of technical user credentials has been created and used for connectivity with external applications. Tools for administrating the cluster have been configured and tested. Administration tools allow to remotely connect to the Backend Orchestrator, monitor its state, and make necessary adjustments. Some of the components get their automated deployment pipelines.

The third cycle has been used for further testing of functionalities developed in previous cycles. More pipelines have been created and monitor their functionality. The exposed API of Kubernetes has been tested and incorporated in the monitoring process. In this period, we have intensively use Backend Orchestrator for deployment and hosting SEMIoTICS platform components.

4.2.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO BACKEND ORCHESTRATOR

Requirements for the Backend Orchestrator have been formulated in Deliverable D2.3 and core functionalities have been formulated in section 3.1.1 of deliverable D2.4.

Details of Backend Orchestrator implementation are described in section 4.2 of deliverables D4.6, D4.7, and D4.13.

The table below shows the requirements of the Backend Orchestrator, a short description of fulfillin g it and status of delivery.

Requirement	Status	Description of implementation
Secure communication among the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules)	Delivered	Backend Orchestrator has a built-in network policy object. It is a specification of how groups of pods are allowed to communicate with each other and other network endpoints. By default, all pods in a Kubernetes cluster can communicate freely with each other without any issues therefore by applying network restrictions we can isolate the services running in pods from each other. Network policies have been applied to all of the backend components to prevent any security leaks and to maintain the traffic that is required by architecture of the SEMIoTICS.
End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	Delivered	Every application managed by Backend Orchestrator has its instance of service object. A service exposes a running application in a pod as a network service and because of that, the APIs of applications are exposed to any external HTTP call. As long as IoT devices have built-in mechanism to create HTTP requests, the end-to-end connectivity is provided.
Scalable infrastructure due to the fast-paced growth of IoT devices	Delivered	Backend Orchestrator(BO) in SEMIoTICS project is a Kubernetes instance that provides easy scalability and high availability. The unit of deployment in BO is a "pod" which contains one application or one SEMIoTICS component. Kubernetes gives a wide range of options for managing and scaling pods setting simple parameters like "ReplicaSet" in the deployment file. When one application of the SEMIoTICS platform is overloaded, BO can create the next instance of an application to balance the load between them. A similar situation occurs when one of the components will be inactive or breaks down, then BO deletes the wrong application and creates a new one. The fast-paced growth of IoT devices connected to SEMIoTICS platform does not affect the BO operation because the number of applications at any time is adapted to the number of devices using them.

4.3 Pattern Orchestrator

The Pattern Orchestrator is a module responsible for automated configuration, coordination, and management of different patterns and their deployment.

In further detail, the Pattern Orchestrator is able to:

1. Receive instantiated recipes from Recipe Cooker via defined API
2. Extract SPDI & QoS properties/requirements from instantiated recipes and convert to patterns
3. Convert patterns to Drools
4. Classify and distribute patterns (as Drools) to the different pattern engines in three layers (Backend, Network, Field)

Cycle 3 includes:

- Extension of the Pattern Orchestrator - Pattern Engines interfacing with more REST services
- Integration with SEMIoTICS GUI
- New classes for the instantiation of Drools facts

TABLE 8 PATTERN ORCHESTRATOR BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
API definition between Recipe Cooker and Pattern Orchestrator	Recipe Cooker needs to submit an instantiated recipe to the Pattern Orchestrator and expects a response that indicates whether the recipe definition is feasible to execute. For that reason, an API needs to be defined.	Cycle 1	Delivered
Transformation of an instantiated recipe to patterns	Pattern Orchestrator must understand the instantiated Recipes it receives, as defined by the Recipe Cooker and transform them into patterns.	Cycle 1	Delivered
Communication with the three Pattern Engines	Interfacing with Pattern Engines on all layers (Backend, Network, and Field) needs to be implemented and tested.	Cycle 1	Delivered
Store patterns (as Drools) in the backend pattern repository	The patterns created by Pattern Orchestrator need to be communicated to the Backend Pattern Engine for storing in the local repository.	Cycle 2	Delivered
Classify and distribute patterns (as Drools) to the different pattern engines	Pattern Orchestrator must be able to decide for each of the Drools Rules/Facts (patterns), which is the appropriate Pattern Engine to deliver it.	Cycle 2 & 3	Delivered
IoT service orchestration adaptation	In case an SPDI or QoS property is no longer guaranteed, adaptation actions must be taken, changing a number of orchestration components. In that way, the Pattern Engines can guarantee that the SPDI/QoS property in question is henceforward satisfied.	Cycle 3	Delivered

Regarding the distribution of patterns to different pattern engines, the decision mechanism of the Pattern Orchestrator, although it delivered in its current state, updates may be made. This is due to the fact that new patterns (rules, facts) are created constantly and will continue to be created until the end of the project. Therefore, the decision mechanism is constantly updated in order to include the newly created patterns.

4.3.1 DEVELOPMENT STATUS

According to Table 8 the first feature that was added to Pattern Orchestrator during cycle 3 is the classification and distribution of the Recipe components as Drools facts to the three different Pattern Engines.

An instantiated Recipe, which serves as input to the Pattern Orchestrator, constitutes by a number of Recipe components of different types such as:

- Placeholders in the form of Host, IoTSensor, IoTActuator, IoTGateway, SoftwareComponent, SoftwareService, NetworkComponent,
- Orchestrations in the form of Sequences, Merges, Splits, Choices
- Properties of all the above
- Interfaces and Operations of the Placeholders

The classification of the Recipe components is done based on their type.

Placeholders: Each of the Recipe components is recognized, based on an ANTLR parser, and is sent to the appropriate Pattern Engine, to be added as Drools facts in their working memory. The type of each Recipe components determines the layer of the Pattern Engine it is sent to. For example, IoTSensors, IoTActuators and IoTGateways are sent to the Pattern Engine at the Field layer. NetworkComponents, on the other hand, are sent to the Pattern Engine at the network layer. SoftwareComponents and SoftwareServices are sent to the Pattern Engine at the Backend layer.

Orchestrations: The layer decision for the different types of placeholders is pretty straightforward. However, it is not clear for Orchestrations to which layer they belong. In order to define the layer of an Orchestrati on we have to take under consideration the layers of the involved placeholders.

Let's take a Sequence as an example case. A sequence consists of two Placeholders, the output of the first becomes input of the second. If the layer of the first placeholder matches the layer of the second, the layer of their Sequence is set to that very same layer. Nevertheless, if the layers of the two Placeholders does not match, the layer of their Sequences falls to one of the three cross layer cases named, Backend-Network, Backend-Field, Field-Network. What we have just described is depicted as code snippet below.

```
    // Set the Layer of all Sequences based to the layers of their placeholders
    private static void setSequencesLayer() {
        for (Sequence sequence : sequencestosend) {
            String plaId = sequence.getPlaceholdera();
            String plaType = getPlaceholderType(plaId);
            String plaLayer = getPlaceholderLayer(plaId, plaType);

            String plbId = sequence.getPlaceholderb();
            String plbType = getPlaceholderType(plbId);
            String plbLayer = getPlaceholderLayer(plbId, plbType);

            if (plaLayer.equals(plbLayer)) {
                sequence.setLayer(plaLayer);
            }
            else {
                if ((plaLayer.toLowerCase().equals("backend") && plbLayer.toLowerCase().equals("network"
)) || (plbLayer.toLowerCase().equals("backend") && plaLayer.toLowerCase().equals("network"))) {
                    sequence.setLayer("CROSSLAYERBN");
                }
                if ((plaLayer.toLowerCase().equals("backend") && plbLayer.toLowerCase().equals("gateway"
)) || (plbLayer.toLowerCase().equals("backend") && plaLayer.toLowerCase().equals("gateway"))) {
                    sequence.setLayer("CROSSLAYERGB");
                }
                if ((plaLayer.toLowerCase().equals("network") && plbLayer.toLowerCase().equals("gateway"
)) || (plbLayer.toLowerCase().equals("network") && plaLayer.toLowerCase().equals("gateway"))) {
                    sequence.setLayer("CROSSLAYERNG");
                }
            }

        }
    }
```

The setSequencesLayer() method run through all the sequences of a Recipe and, first of all, gets the layers of the two involved Placeholders in the plaLayer and plbLayer variables. Then if these variables are equal, the layer of the current Sequence is set to the same layer (if statement). If they are not, the Sequence layer is set to one of the three cross layer cases (else statement).

Properties: Properties describe a characteristic of a Recipe component. As a result, each Property has a subject that it is referred to. This subject defines the layer of the Property itself. As we can see in the code snippet below, the definePropertyLayer() method takes as parameters a subject and a layer and runs through all the Recipe properties. The parameter subject is actually a Recipe component and the parameter layer is its layer. If the property iteration spots a Property with subject the said Recipe component, the Property layer is set to the Recipe component's layer.

```
    // Set the Layer of all Properties that are subject to a specific placeholder based on the layer of
the placeholder
    private static void definePropertyLayer(String subject, String layer) {
        for (Property property: propertiestosend) {
            if (property.getSubject().equals(subject)) {
                property.setLayer(layer);
            }
        }
    }
```

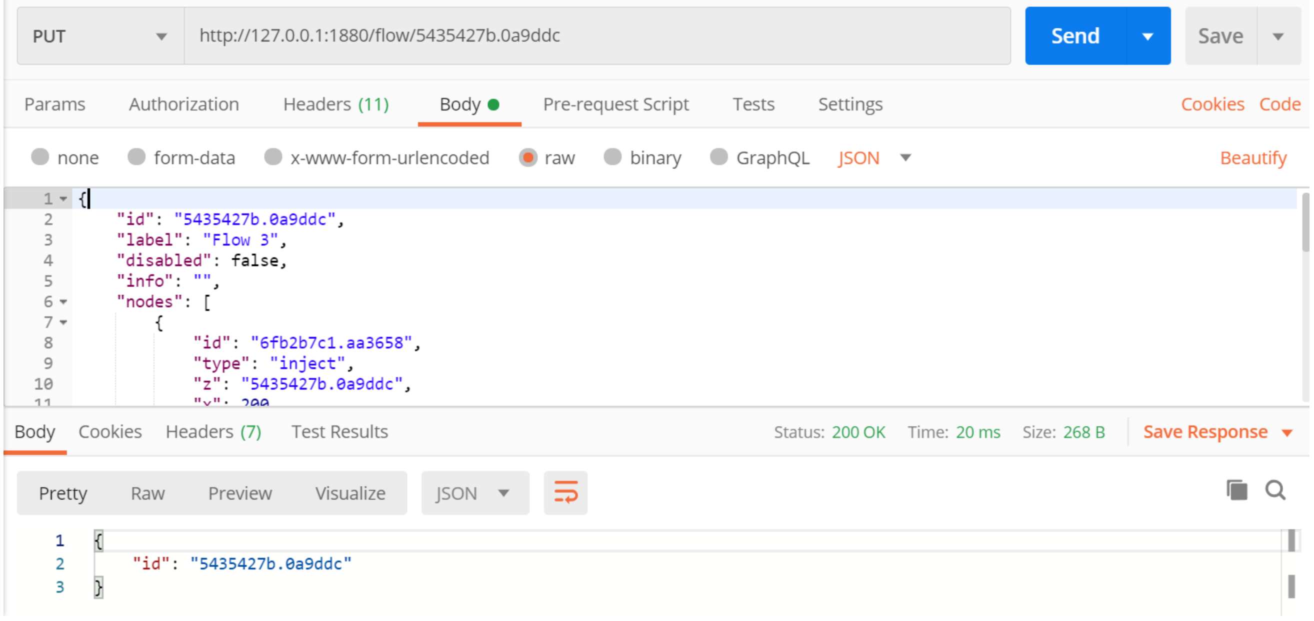
Interfaces and Operations: The same way we define the layer of a property, an interface or operation layer is defined. Interfaces and Operations have also subjects that are referred to Recipe components. The methods defineInterfaceLayer() and defineOperationLayer() are equivalent to the definePropertyLayer() we described above.

```
    // Set the Layer of all Interfaces that are subject to a specific placeholder based on the layer of
the placeholder
    private static void defineInterfaceLayer(String subject, String layer) {
        for (Interface interface: interfacestosend) {
            if (interface.getSubject().equals(subject)) {
                interface.setLayer(layer);
            }
        }
    }

    // Set the Layer of all Operations that are subject to a specific placeholder based on the layer of
the placeholder
    private static void defineOperationLayer(String subject, String layer) {
        for (Operation operation: operationstosend) {
            if (operation.getSubject().equals(subject)) {
                operation.setLayer(layer);
            }
        }
    }
```

Another feature that was added to Pattern Orchestrator during cycle 3 is its ability to communicate with the Recipe Cooker to send the potentially altered flow in case adaptation actions are needed. Adaptation actions may take place in case an SPDI or QoS Property, referred to the whole Recipe or to a part of it, does not hold. The said adaptation action ends up to a new, updated Recipe, that is communicated back to the Recipe Cooker in order to be deployed again. The updated version of the Recipe may have additional components or substituted components.

Recipe Cooker is based on Node-Red. The latter exposes an API that allows the update of a given flow. The flow is represented as a tab within the Node-Red editor and all its nodes are stopped before the new flow configuration is started. A PUT request is required at the following URL: http://“nodeRedIP”:“nodeRedPort”/flow/“RecipeID”. In the body of the request, the updated flow is inserted in JSON format. The expected response is the ID of the updated flow. All the above are depicted in Figure 20 , where an update-flow request is shown.



http://127.0.0.1:1880/flow/5435427b.0a9ddc

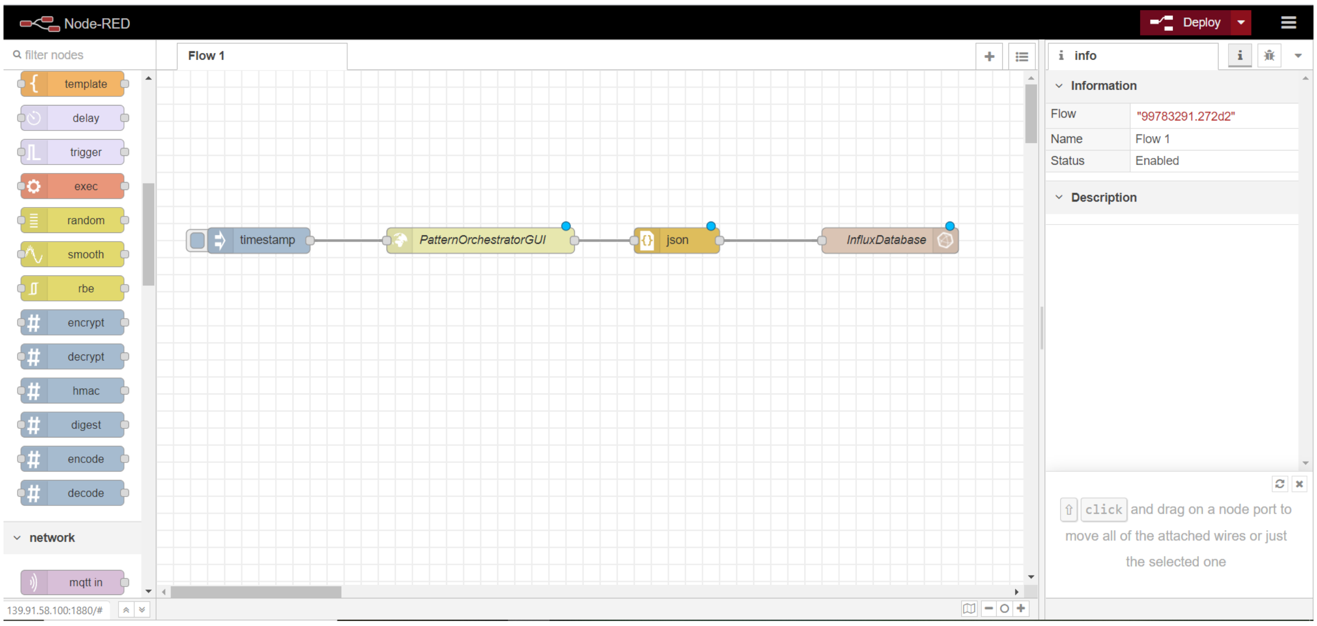
```
{
    "id": "5435427b.0a9ddc",
    "label": "Flow 3",
    "disabled": false,
    "info": "",
    "nodes": [
        {
            "id": "6fb2b7c1.aa3658",
            "type": "inject",
            "z": "5435427b.0a9ddc",
            "x": 200
```

Status: 200 OK Time: 20 ms Size: 268 B

```
{
    "id": "5435427b.0a9ddc"
}
```

FIGURE 20 UPDATEFLOW REQUEST TO NODE-RED

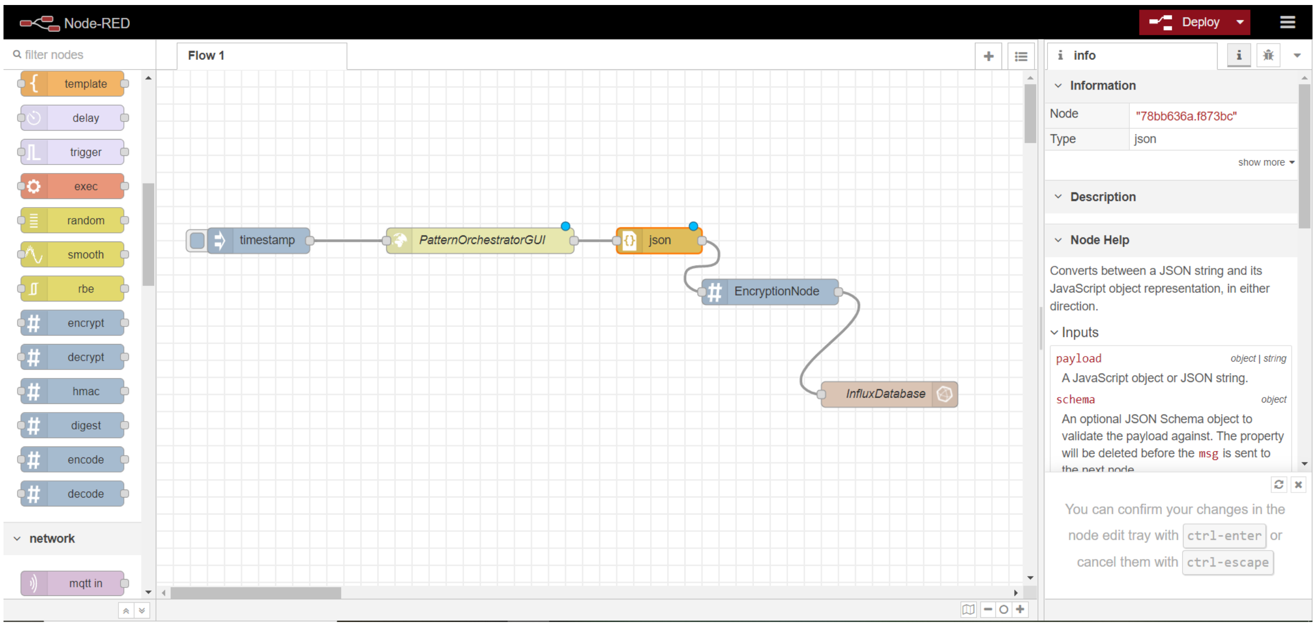
Let's assume that the flow depicted in FIGURE 21 is sent to Pattern Orchestrator in order the Property *Encryption* to be verified against the sequence of the two nodes *PatternOrchestratorGUI* and *InfluxDatabase.* Since no encryption takes place between these two nodes, the verification Drools rules that will be triggered in the corresponding Pattern Engine will respond that the *Encryption* Property is not satisfied. In that case, the Pattern Engine will send a request to the *modifyRecipe* API of the Pattern Orchestrator with the SPDI/QoS Property that is not satisfied. In that way, all the needed information such as the Recipe ID and the subject of the Property become available.



Flow	"99783291.272d2"
Name	Flow 1
Status	Enabled

FIGURE 21 ORIGINAL FLOW WITH UNENCRYPTED COMMUNICATION

As soon as, the Pattern Orchestrator receives the said request, sends its own request to Node-Red asking for the flow in JSON format. After that, an extra node is added in the flow, changing its JSON representation. The new, changed JSON file is sent to Node-Red using the update-flow API. In order to satisfy the Encryption Property , an *EncryptionNode* is added which is able to encrypt the transferred message using one of the most well-known encryption algorithms and a provided secret key.



Node	"78bb636a.f873bc"
Type	json

Converts between a JSON string and its JavaScript object representation, in either direction.

payload object | string
A JavaScript object or JSON string.
schema object
An optional JSON Schema object to validate the payload against. The property will be deleted before the msg is sent to the next node.

You can confirm your changes in the node edit tray with ctrl-enter or cancel them with ctrl-escape

FIGURE 22 UPDATED FLOW WITH ENCRYPTED COMMUNICATION

The code snippet below depicts what is added by Pattern Orchestrator to the JSON representation of the flow. These lines describe the *EncryptionNode*. As we can see, its id, name and type are defined. The z attribute represents the flow to which this new node is added. Moreover, we see that the encryption algorithm and the secret key are defined. Finally, the wires attribute includes the ids of the nodes that use the output of the node in question as their input. In our case, it includes the id of the InfluxDatabase node.

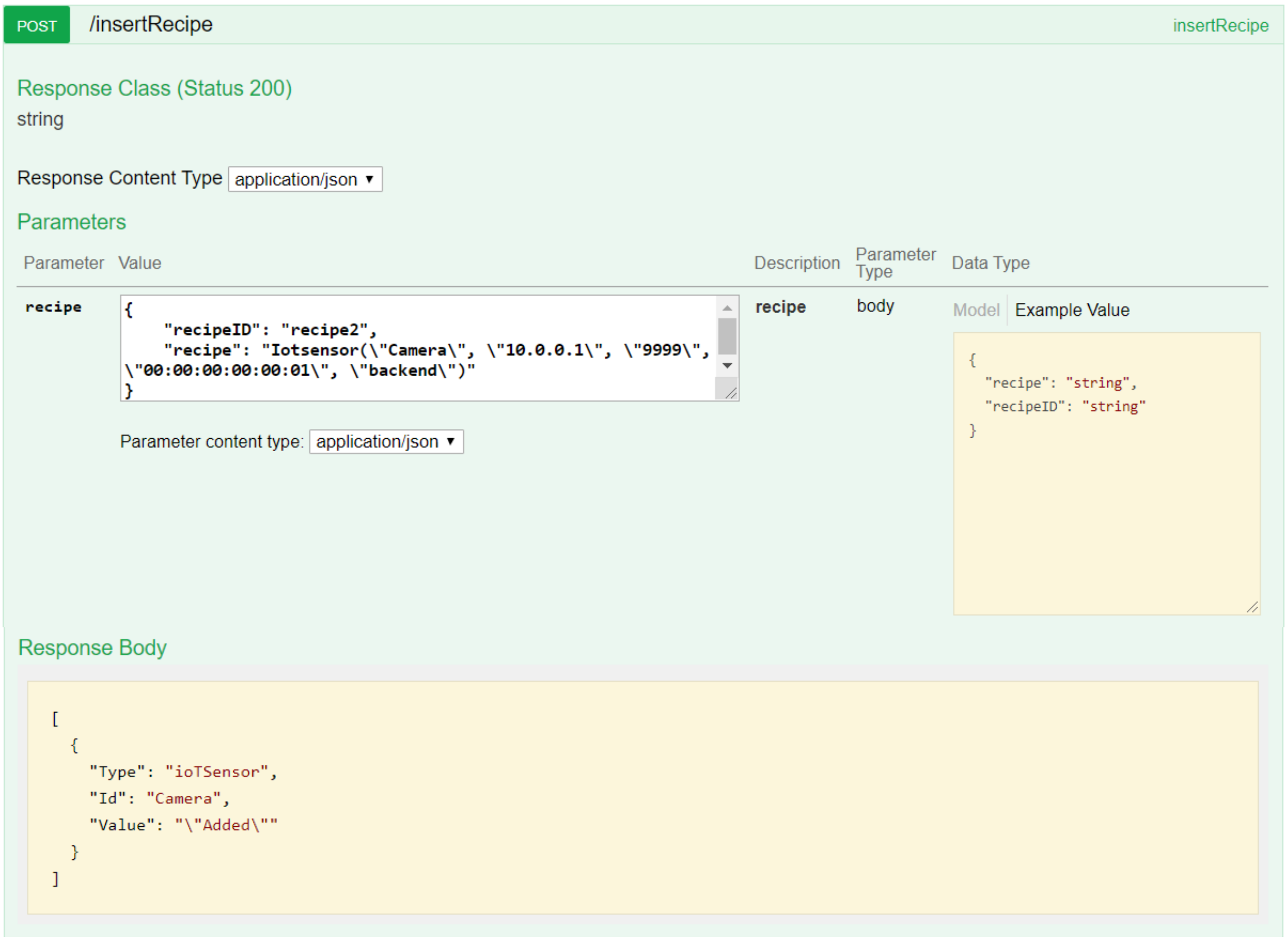
```
{
        "id": "6757fce7.108a94",
        "type": "encrypt",
        "z": "99783291.272d2",
        "name": "EncryptionNode",
        "algorithm": "AES",
        "key": "semiotics",
        "x": 720,
        "y": 260,
        "wires": [
            [
                "eae7b2ac.b98fc"
            ]
        ]
        }
```

4.3.2 COMPONENT API INTERACTIONS DESCRIPTION

The developed APIs within Pattern Orchestrator can be seen in Table 9 below. We present each of them using Swagger API documentation. We show what are the parameters of a request to these APIs, and how the response body should look like.

TABLE 9: LIST OF API DEVELOPED WITHIN PATTERN ORCHESTRATOR

API	Status	API access type	Additional comments
gui	Deployed	internal	Used by the GUI hub of SEMIoTICS
insertRecipe	Deployed	internal	Used by the Recipe Cooker
removeRecipe	Deployed	internal	Used by the Recipe Cooker
modifyRecipe	Deployed	internal	Used by the Pattern Engines


POST /insertRecipe insertRecipe

Response Class (Status 200)
string

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
recipe	see below	recipe	body	Model / Example Value

```
{
    "recipeID": "recipe2",
    "recipe": "Iotsensor(\"Camera\", \"10.0.0.1\", \"9999\",
\"00:00:00:00:00:01\", \"backend\")"
}
```

Parameter content type: application/json

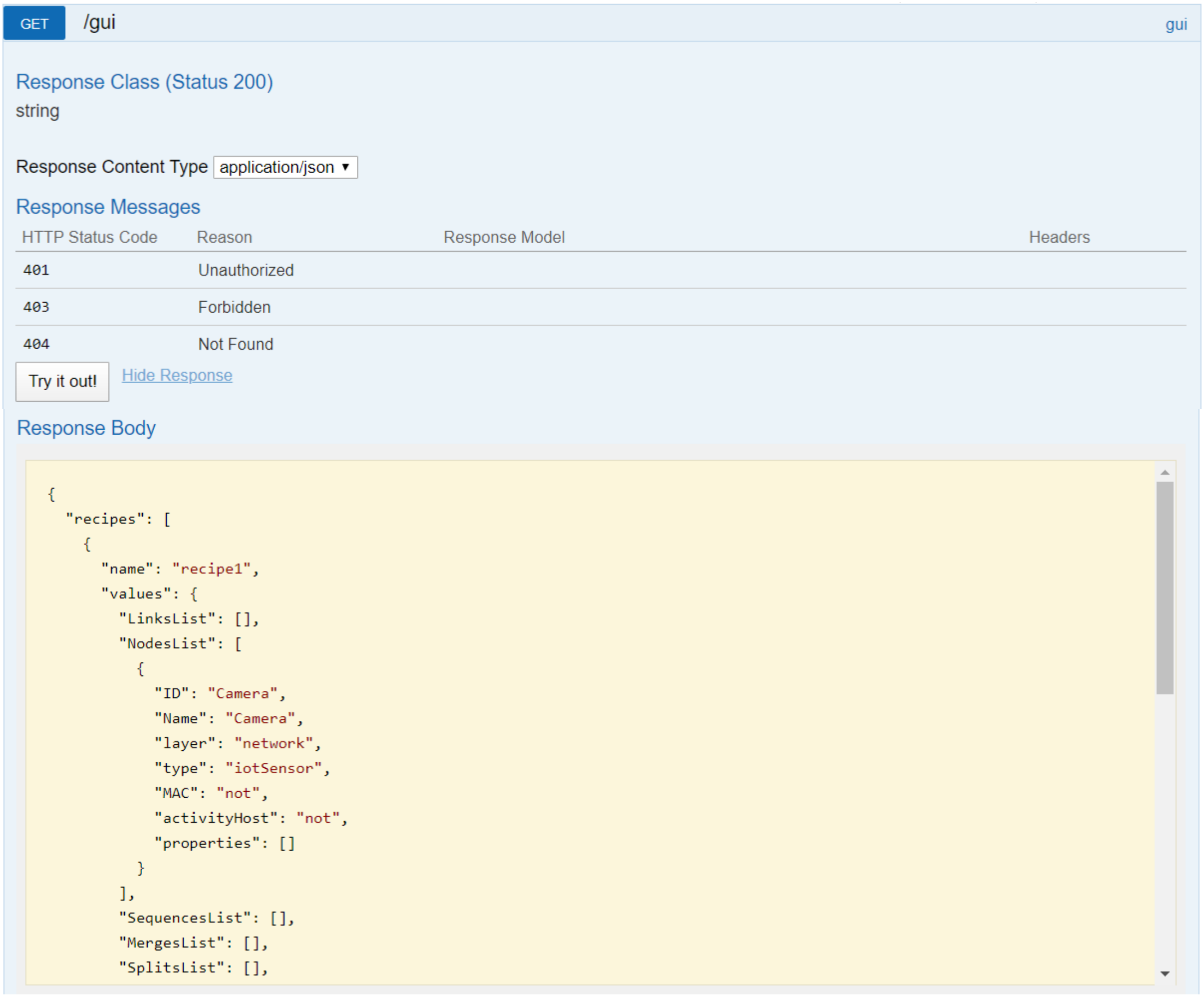
Example Value:

```
{
  "recipe": "string",
  "recipeID": "string"
}
```

Response Body

```
[
  {
    "Type": "ioTSensor",
    "Id": "Camera",
    "Value": "\"Added\""
  }
]
```

FIGURE 23 INSERTRECIPE API USING SWAGGER



GET /gui gui

Response Class (Status 200)
string

Response Content Type application/json

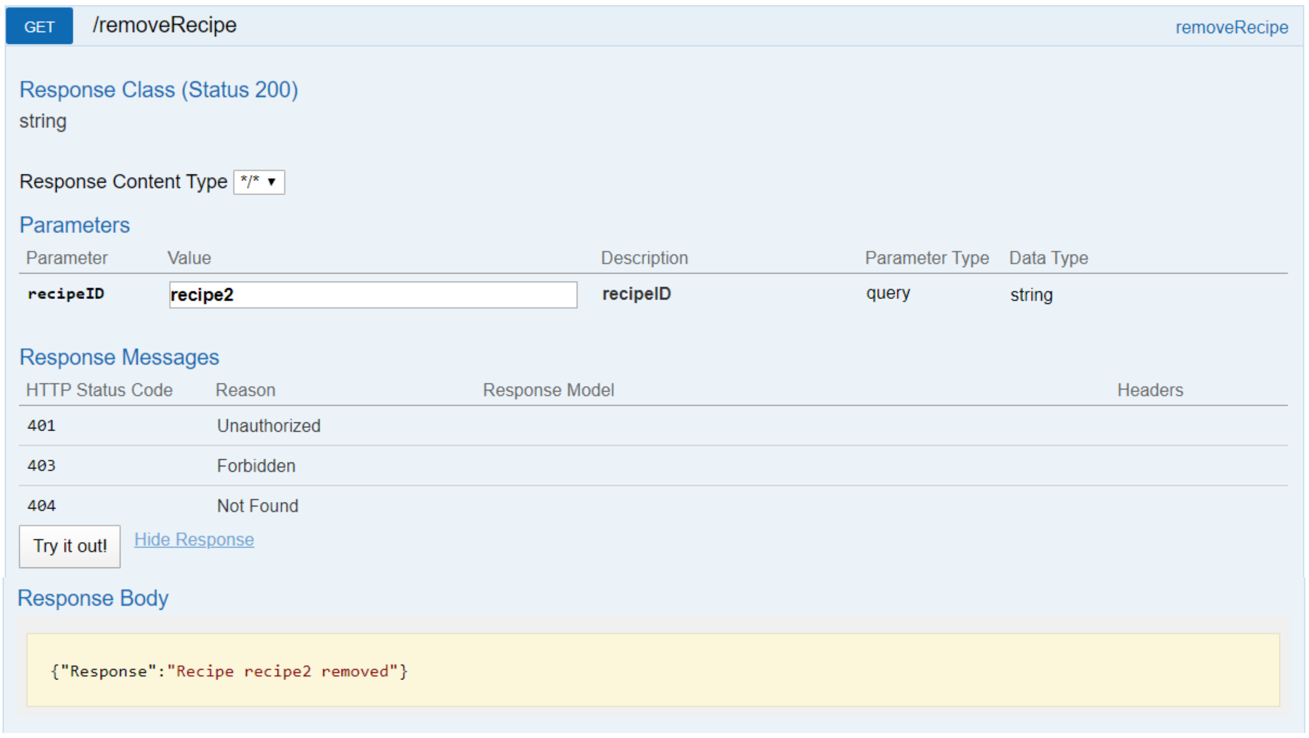
Response Messages

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

Response Body

```
{
  "recipes": [
    {
      "name": "recipe1",
      "values": {
        "LinksList": [],
        "NodesList": [
          {
            "ID": "Camera",
            "Name": "Camera",
            "layer": "network",
            "type": "iotSensor",
            "MAC": "not",
            "activityHost": "not",
            "properties": []
          }
        ],
        "SequencesList": [],
        "MergesList": [],
        "SplitsList": [],
```

FIGURE 24 GUI API USING SWAGGER


GET /removeRecipe removeRecipe

Response Class (Status 200)
string

Response Content Type */*

Parameters

Parameter	Value	Description	Parameter Type	Data Type
recipeID	recipe2	recipeID	query	string

Response Messages

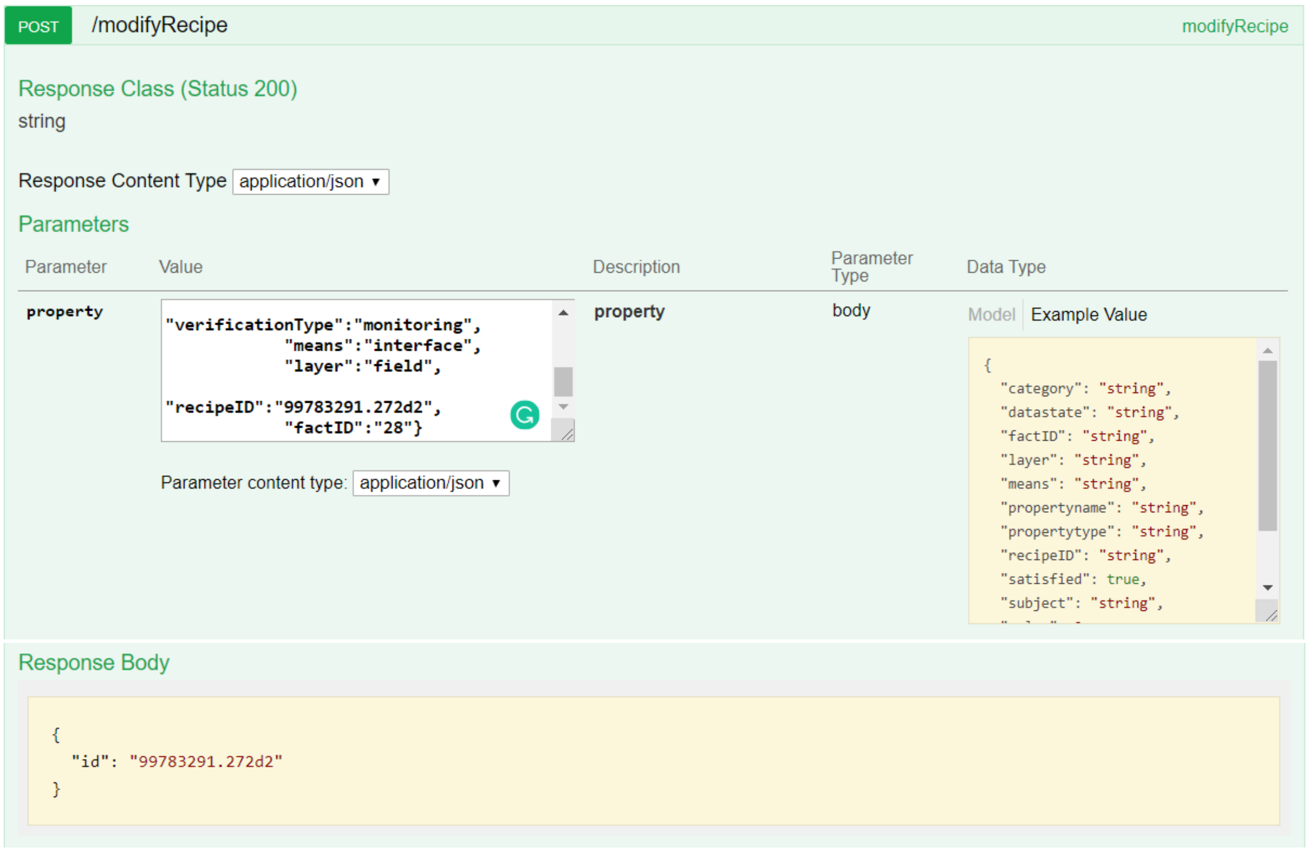
HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

Try it out! Hide Response

Response Body

```
{"Response":"Recipe recipe2 removed"}
```

FIGURE 25 REMOVERECIPE API USING SWAGGE


POST /modifyRecipe modifyRecipe

Response Class (Status 200)
string

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
property	"verificationType":"monitoring", "means":"interface", "layer":"field", "recipeID":"99783291.272d2", "factID":"28"}	property	body	Model Example Value

```
"verificationType":"monitoring",
            "means":"interface",
            "layer":"field",

"recipeID":"99783291.272d2",
            "factID":"28"}
```

Parameter content type: application/json

```
{
  "category": "string",
  "datastate": "string",
  "factID": "string",
  "layer": "string",
  "means": "string",
  "propertyname": "string",
  "propertytype": "string",
  "recipeID": "string",
  "satisfied": true,
  "subject": "string",
```

Response Body

```
{
  "id": "99783291.272d2"
}
```

FIGURE 26 MODIFYRECIPE API USING SWAGGER

4.3.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

During cycle 1 the system model was defined using the Eclipse Modeling framework (EMF). A UML diagram was created using the EMF graphical editor and based on that, an EBNF grammar was created. The expected input of the Pattern Orchestrator is a flow expressed with the system model. Moreover, using the Eclipse ANTLR4 plugin, the said grammar produced a parser and a listener allowing for syntactic verification of flows expressed with the system model. A more detailed description can be found in D4.6.

During cycle 2, Rest web services were built for Pattern Orchestrator using the Spring Framework. In that way, other SEMIoTICS components, such as Recipe Cooker and SEMIoTICS GUI, are able to make REST requests to the Pattern Orchestrator API using REST clients. Recipe Cooker uses the InsertRecipe API to communicate an instantiated Recipe to the Pattern Orchestrator, while GUI uses the patternStatus API to get information of the Recipe patterns in order to visualize their status. The description of those APIs can be found in D4.7. A REST client was also created in order the Pattern Orchestrator to be able to send requests to the REST APIs of the three Pattern Engines in the backend, network and field layers.

During cycle 3, additional code to the Pattern Orchestrator, allowed the later to be able to classify the different Recipe components and to decide, based on their classification, to which Pattern Engine to communicate them. The description on how this is done can be found on section 4.3.1 above. Moreover, one additional REST API was created for the communication of the information needed for an adaptation action to take place. Pattern Engines send requests to this new API. Respectively, new REST client was created for the communication between the Pattern Orchestrator and the Recipe Cooker. This communication is depicted in details in section 4.3.1.

4.3.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO PATTERN ORCHESTRATOR

SEMIoTICS Requirement		Pattern language considerations	Reference
Req. ID	**Description**		
R.BC.18	The backend layer must feature SPDI pattern reasoning embedded intelligence capabilities	This is a core set of requirements for the SPDI capabilities that must be covered within the pattern-driven approach developed within T4.1. Individual Pattern reasoning components should be developed and deployed at all layers, while the backend should feature global reasoning capabilities. All reasoning engines should aggregate (through interfacing with monitoring) relevant information needed for said reasoning.	The system model and associated pattern language developed are tailored to the multi-layer approach of SEMIoTICS, also anticipating intra- and cross- layer reasoning. Furthermore, Pattern reasoning components (referred to as Pattern Engines) are embedded at all layers; see subsection 3.7.2.2 of D4.8. The real-time reasoning will be achieved in conjunction with
R.BC.19	The backend layer should feature pattern-driven cross-layer orchestration capabilities		
R.BC.20	The backend layer must aggregate intra-layer as well as inter-layer SPDI status information to enable local and global intelligence reasoning and adaptation		
R.NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities		
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation		
R.FD.14	The field layer must feature SPDI pattern reasoning local embedded intelligence capabilities		

R.FD.15	The field layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation		the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning.
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	While an indirect set of requirements, the various cross platform and cross layer interactions (including E2E between field and backend) with heterogeneous components will need to be supported and their SPDI properties monitored accordingly.	As can be seen in subsections 3.2 (Language Model) and 3.3 (Language Constructs) of D4.8, instances of Java class *Link* allow Pattern Engines to monitor and verify connectivity among IoT service orchestration components. This also encompasses the pattern-driven interoperability mechanisms developed in the context T3.4 (and which are further described in D3.4), which leverage the language and pattern definitions. Through the above and the integration of pattern-based capabilities at the network level (SDN pattern engine), connectivity and QoS parameters can also be monitored.
R.UC1.1	Automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders		
R.UC2.3	The SEMIoTICS platform SHOULD guarantee proper connectivity between the various components of the SARA distributed application. The SARA solution is a distributed application not only because it uses different cloud services (e.g. AREAS Cloud services, AI services) from different remote computational nodes, but also because the SARA application logic itself is distributed across various edge nodes (SARA Hubs).		
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication)	Other than the aspects of availability and dependability (and	As can be seen in subsections 3.3 (Language Model)

R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration needed by SPDI pattern	associated concepts; e.g. fault tolerance) that are already integral in the SPDI properties, other QoS-related parameters (e.g. latency) can also be accommodated by the pattern language adopted. Moreover, the pattern language must be able to leverage appropriate monitors and interface with the necessary mechanisms to act as an enabler for configuring the network and triggering network updates / reconfigurations, as needed (e.g. for fault tolerance or QoS).	and 3.4 (Language Constructs) of D4.8, Java class *Property* owns an attribute *Category,* allowing Pattern Engines to monitor QoS properties of the components of an IoT service orchestration. Moreover, the properties associated with the *Link* class directly affect the requirements relayed to the network layer (with the associated properties reasoned by the Pattern Engine embedded at the SDN controller; see subsection 3.7.2.2 of D4.8).
R.GP.7	SDN controller giving feedback for a future generation of SPDI patterns to avoid using the same pattern in case of failure		
R.UC1.5	Fail-over and highly available network management SHALL be performed in the face of either controller or data-plane failures.		
R.UC1.3	There MUST be enabled the definition of network QoS on application-level and automated translation into SDN controller configurations.		
R.UC1.4	Network resource isolation MUST be performed for guaranteed Service properties – i.e. reliability, delay and bandwidth constraints.		
R.UC2.15	The SEMIoTICS platform SHOULD provide low latency connectivity between the SARA hubs and cloud services (i.e. AREAS cloud services and AI services) to allow offloading of near real-time computation intensive tasks to the cloud. Therefore, SARA hubs need to send with minimal delay: • raw range data (e.g. from Lidar sensors) to identify proximal objects/objects, • real-time audio stream for speech analysis, and real-time raw video stream (object/people recognition, gesture recognition, posture analysis).		
R.GSP.1	The Intrusion Detection System (IDS) MUST capture and process suspicious traffic.	Concerns regarding any sensitive data that is generated, processed, stored and exchanged at all layers must be considered, enforcing and monitoring the corresponding security mechanisms, especially when different trust domains are involved. Proper authentication and authorisation services are a necessity when trying to safeguard the security and privacy of data and services. These aspects	Security-related properties (such as Confidentiality) are at the core of the properties covered in the SEMIoTICS system model (subsection 3.3 of D4.8) and associated language (subsection 3.4 of D4.8). Moreover, a first version of security-related pattern rules can be seen in
R.NL.11	Secure communication with the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules), as well as the communication between VIM, SDN Controller, and MANO, with data paths acting as computing nodes for VNF spinoff.		
R.S.7	The negotiation interface of the SDN Controller SHALL be secure against network-based attacks		

R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.	must be defined in the pattern language, monitored and enforced, considering the different types of devices (e.g. sensors, network controllers, backend servers), actors (e.g. humans, machines/applications) and interaction types (e.g. maintenance or medical staff, simple users). These, along with cryptographic mechanisms, will need to be used to establish trust within and across domains. Moreover, privacy considerations will have to be included (e.g. protection of private data at rest and in transit, data anonymization and minimisation, data retention; see section 2.2.1 above). In addition to the above, patterns can also be leveraged to monitor and enforce the presence of security mechanisms in different IoT orchestrations.	subsection 4.1 of D4.8, while a first set of Privacy Patterns can be seen in subsection 4.1.5 of D4.8. Moreover, using the pattern language, different verification types can be declared for each of the properties (see subsection 3.3 of D4.8); this can be exploited to define interfaces with the various security mechanisms which will allow the verification of the different SPDI properties associated with them (e.g., monitoring encryption mechanisms that provide the property of Confidentiality). This will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning on relevant security and privacy - related aspects, such as secure deletion of unnecessary data, limitation of
R.S.6	Sensors SHALL be able to encrypt the data they generate, i.e. their CPU and memory SHALL be sufficient to perform these cryptographic operations.		
R.S.2	Authentication and authorisation of the stakeholders MUST be enforced by the Network controller, e.g. through access and role-based lists for different levels of function granularities (overlay, customized access to service, QoS manipulation, etc.)		

			sampling via a variant of the mechanisms used to ensure QoS parameters, etc.
R.S.3	Sensors SHALL be identifiable (e.g. by a TPM module/smartcard) and authenticated by the gateway.	These Security and Privacy requirements are indirectly related to the pattern approach presented herein. Nevertheless, the SEMIoTICS patterns need to be able to accommodate all these requirements, monitoring the status of the corresponding components implementing these security and privacy requirements, and triggering adaptations if needed.	All key security and privacy properties are covered within the SEMIoTICS patterns (see Section 4 of D4.8). Furthermore, the language expressiveness allows the definition of the appropriate conditions (facts) to be verified in order to provide real-time verification of the properties sketched by these requirements (see subsections 3.4 and 3.9 of D4.8).
R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.		
R.S.5	Before sensitive data is being transmitted, the respective components SHALL be authenticated as defined by requirements R.S.3 and R.S.4		
R.S.17	There MUST be an interface between the network controller and the network administrators for the designation of the applications' permissions.		
R.S.18	All network functions SHALL be mapped to application permissions		
R.GSP.4	Platforms, e.g. cloud platform and sensor, SHALL be trusted.		
R.GSP.9	The SARA system SHALL provide robust mechanisms to protect Patient-related data.		
R.GSP.10	The SARA system MUST fully comply with all relevant Italian laws governing the privacy, security and storage of sensitive Patient health-related data.		
R.P.1	The collection of raw data MUST be minimized.	Coverage of privacy requirements within the SEMIoTICS patterns is needed.	As documented in subsection 4.2 of D4.8, the SEMIoTICS patterns (and by extension the pattern-driven reasoning capabilities of SEMIoTICS at all layers) include all key privacy properties.
R.P.3	Storage of data MUST be minimized.		
R.P.4	A short data retention period MUST be enforced and maintaining data for longer than necessary avoided.		
R.P.6	Data MUST be anonymized wherever possible by removing the personally identifiable information in order to decrease the risk of unintended disclosure.		
R.P.8	Data MUST be stored in encrypted form.		
R.P.9	Repeated querying for specific data by applications, services, or users that are not intent to act in this manner SHALL be blocked.		

R.UC1.6	Decisions made by unreliable, i.e. faulty or malicious SDN controllers, SHALL be identified and excluded.	Events received from monitoring critical aspects of the systems' and subsystems' operation, as highlighted by the pattern language, will need to be aggregated and evaluated by the pattern engine. These will need to encompass SPDI and other parameters (e.g. QoS related), as well as anomalies, indicators of malicious actions, malfunction, resource depletion, failures etc., across the different layers and (physical & logical) components of the SEMIoTICS deployment. Pattern-driven interoperability mechanisms will ensure that these connections can be established, further explored in D3.4. In cases of privacy-sensitive monitoring data (e.g. location of the device), the necessary privacy provisions will need to be enforced.	As can be seen in section 3.2 (Language Model) and 3.3 (Language Constructs), the pattern language that has been created can declare Properties whose verification type is *Monitoring*. That allows for capturing the monitoring critical aspects and enabling the reasoning on parameters related to properties such as reliability. As above, the necessary inputs will be aggregated from the monitoring framework of SEMIoTICS (T4.2/D4.2).
R.GSP.7	The cloud platform SHALL be able to monitor the execution of an app, in particular its interactions with other apps, the network interface, and APIs.		
R.UC3.7	MCU IoT Sensing unit shall be able to send change detection and signal local changes / anomalies to IoT Sensing gateway.	This set of requirements indirectly affects the development of the	Availability and Dependability patterns

R.UC3.16	Each registered sensing unit should send to the sensing gateway a keep alive signal on a specified period (e.g. few seconds) to notify the gateway it is correctly working. The sensing gateway should detect by this mean any non-working sensing unit and reconfigure the system accordingly.	SEMIoTICS pattern solution. The Availability and Dependability aspects integrated into the pattern approach need to support these UC requirements.	developed within SEMIoTICS are able to accommodate the monitoring defined in these requirements (see subsections 4.1.3 and 4.3 of D4.8). These features will be further explored and demonstrated in the context of the UC3 scenarios, as detailed in subsection 7.3.
R.UC3.18	Sensing units may be equipped with dedicated FW to detect relevant sensors malfunctioning and report that to the gateway		
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised	Logging is an integral part of security, enabling auditing functions and providing accountability. Moreover, regulatory drivers also necessitate it (e.g. transparency through logging is essential under GDPR). This must be considered in the definition of the pattern language, the associated engine and its monitors, enabling the provision of reliable and trustworthy logging mechanisms both for the various actors as well as the events and reasoning of the pattern engine itself.	All pattern engine components (see subsection 3.7.2.2 of D4.8) feature integrated logging mechanisms that allow for auditing on all pattern-driven reasoning and adaptation actions triggered. In other parts of the SEMIoTICS framework and protected infrastructure, the deployment and monitoring of the proper operation of the logging functions can be introduced as with any other mechanism (see subgroups of requirements above).

4.4 Pattern Engine (backend)

As described in D4.6 and D4.7, the Backend Pattern Engine is a module featuring an underlying semantic reasoner processing Drools rules and facts. It also supports the capability to insert, modify, execute and retract patterns at design time (via Pattern Orchestrator) or at runtime in the SEMIoTICS backend. Using Drools rule engine, along with monitoring capabilities present at the backend layer, the Pattern Engine is able to reason on the SPDI and QoS properties of aspects pertaining to the operation of the SEMIoTICS backend.

During runtime, the Backend Pattern Engine Module is able to receive fact updates from the Pattern engines of lower layers (Network & Field), in order to have an up-to-date view of the SPDI state of all the layers and the corresponding components.

Cycle 3 development includes:

- Refinement of classes implemented in Cycle 2 for the instantiation of Drools facts.
- Add encryption to the endpoints.
- Update of Backend Pattern Engine status based on information from the SDN/NFV layer and Field layer
- Adaptation to maintain desired properties.

Please refer to Table 10 for more details.

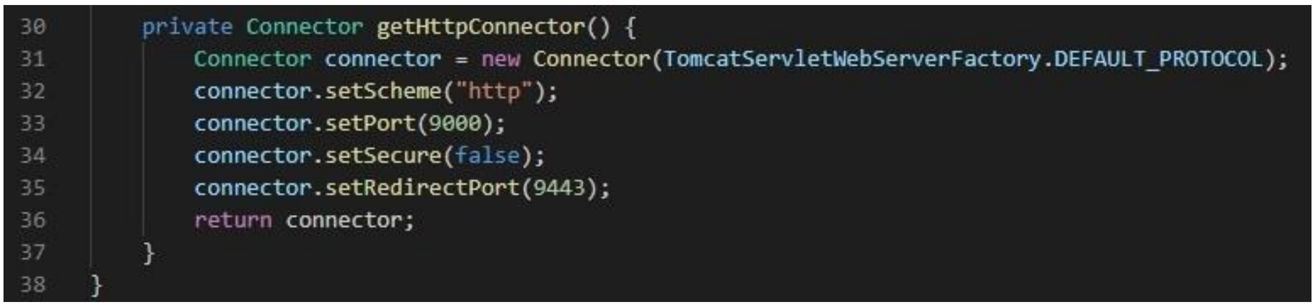
TABLE 10 PATTERN ENGINE BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
API Definition	Pattern Engines in all layers need a common API for the interactions between them, therefore the first step is to define the API.	Cycle 1	Delivered
Drools pattern rules instantiation	Patterns in the form of Drools Rules must be created and instantiated inside the Drools Engine of the Backend Pattern Engine.	Cycle 1	Delivered
Drools pattern rules storage in a standalone repository	A standalone repository is needed for the Drools pattern rules in order to maintain them in the case of restarting the engine.	Cycle 1	Delivered
Communication of network and field updates to Backend Pattern Engine	The Backend Pattern Engine must have a global view of the SPDI properties, therefore, Pattern Engines in the field and network layer must propagate their updates to Backend Pattern Engine	Cycle 2	Delivered
Successful testing of flow from Recipe Cooker	The Recipe Cooker is the point of start for an IoT service orchestration to be deployed with SPDI properties assigned to it. The IoT service orchestration must be communicated to the relevant Pattern Engines through the Pattern Orchestrator (please see the comment below).	Cycle 2 & 3	Delivered
Refinement of classes from Cycle 2	The classes used for the instantiation of Drools facts, needed to be adapted to fit the needs of Use Cases.	Cycle 3	Delivered
Add encryption to the REST endpoints	In order to increase the level of security all the REST endpoints are encrypted.	Cycle 3	Delivered
Update of Backend Pattern Engine status based on information from the SDN/NFV layer and Field layer	Update of Backend Pattern Engine on status based on instantiated paths with different properties, and adaptation of network to maintain desired properties and used SFC chains.	Cycle 3	Delivered
Adaptation to maintain desired properties	When the desired property is no longer satisfied, the Backend Pattern Engine must take adaptation actions accordingly.	Cycle 3	Delivered

4.4.1 DEVELOPMENT STATUS

Regarding the Backend Pattern Engine the environment that was adapted during Cycle 2 after replacing gRPC and Protocol buffers with the corresponding REST approach, was capable to fulfill all the needs with other SEMIoTICS components and therefore there was no need to make additional changes on that front.

In addition, the REST web services that were built on Cycle 2 are now secured with encryption with the use of SSL. The necessary certificates are also included in the other Pattern Engines in order to communicate with the Pattern Engine at the Backend. The Spring Framework was also configured to redirect all http traffic to https as depicted in Figure 27.



```
    private Connector getHttpConnector() {
        Connector connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
        connector.setScheme("http");
        connector.setPort(9000);
        connector.setSecure(false);
        connector.setRedirectPort(9443);
        return connector;
    }
}
```

FIGURE 27 REDIRECTION OF HTTP TRAFFIC TO HTTPS

In order for the Pattern Engine at the Backend to have an up to date information about all the components involved in all layers, two steps were taken. The first step was in cooperation with the Pattern Orchestrator and the second was in cooperation with the Pattern Engines at the other layers. Pattern Orchestrator decides which facts should arrive to which Pattern Engine. The Pattern Engine at the Backend, in addition to the facts that are specific for the backend layer, will also receive all the facts from the Pattern Orchestrator that are sent to the other layers. This process is accomplished with the use of the addFact API.

Due to the fact that the Pattern Engine at the Backend is not in position to verify on its own, the SPDI/QoS properties that exist in the other layers, it needs this information from the other Pattern Engines. The Pattern Engines at the other layers, after triggering their local rules and having reasoned with all their available facts, conclude to an updated status whether the said properties are satisfied or not. Afterwards, they transmit these properties to the Pattern Engine at the Backend using the factUpdate API. Some refinement was necessary to the classes introduced during cycle 2 but the core of them remained the same.

When a desired property is no longer satisfied, the Backend Pattern Engine is able to take adaptation actions accordingly. These actions are dictated accordingly from the Pattern Rules. We consider the scenario where a database component exists in the recipe, along with a requirement for encrypted storage. A Pattern Rule would identify that the database doesn't support encryption and initiate a chain of actions that will result in the modification of the initial recipe. The new recipe will have an encryption node ahead of the database component thus satisfying the need for encrypted storage.

The following rule is the implementation of the above scenario:

```
rule "Encryption Adaptation"
when
    SoftwareComponent($pid:=placeholderid);
    $pr:Property($pid:=subject, category=="storageencryption", satisfied==false);
then
    System.out.println("Contacting Pattern Orchestrator ...");
    Application.contactPO.flowUpdate($pr);
end
```

This rule is triggered whenever a software component such as a database exists along with a requirement for encrypted storage. The "then" part of the rule will contact Pattern Orchestrator, providing him will all the necessary information for updating the recipe.

The following snippet of code is the method that will contact Pattern Orchestrator in the above scenario.

```
public String flowUpdate(Property pr){
    String jsonString ="";
    Boolean connectivity=false;

    //Convert Java Plain Object into JSON
    Gson gsonBuilder = new GsonBuilder().create();
    String jsonFact = gsonBuilder.toJson(pr);

    try {
        URL url = new URL (baseURL + "/modifyRecipe");
        HttpURLConnection con = (HttpURLConnection)url.openConnection();
        con.setRequestMethod("POST");
        con.setRequestProperty("Content-Type", "application/json");
        con.setRequestProperty("Accept", "application/json");
        con.setDoOutput(true);
        con.setConnectTimeout(5000); //set timeout to 5 seconds
        String body = jsonFact;
        try(OutputStream os = con.getOutputStream()) {
            byte[] input = body.getBytes("utf-8");
            os.write(input, 0, input.length);
        }
        try(BufferedReader br = new BufferedReader(
                new InputStreamReader(con.getInputStream(), "utf-8"))) {
            StringBuilder response = new StringBuilder();
            String responseLine = null;
            while ((responseLine = br.readLine()) != null) {
                response.append(responseLine.trim());
            }
            jsonString = response.toString();
            connectivity=true;

        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch(NoRouteToHostException e){
        System.out.println("no connection to Pattern Orchestrator at "+poIP+":"+poPort);
    }catch (ConnectException e){
        System.out.println(e.getMessage() + ". Make sure that Pattern Orchestrator is runni
ng at "+poIP+":"+poPort);
    }
    catch (SocketTimeoutException e){
        System.out.println(e.getMessage() + ". Make sure that Pattern Orchestrator is runni
ng at "+poIP+":"+poPort);
```

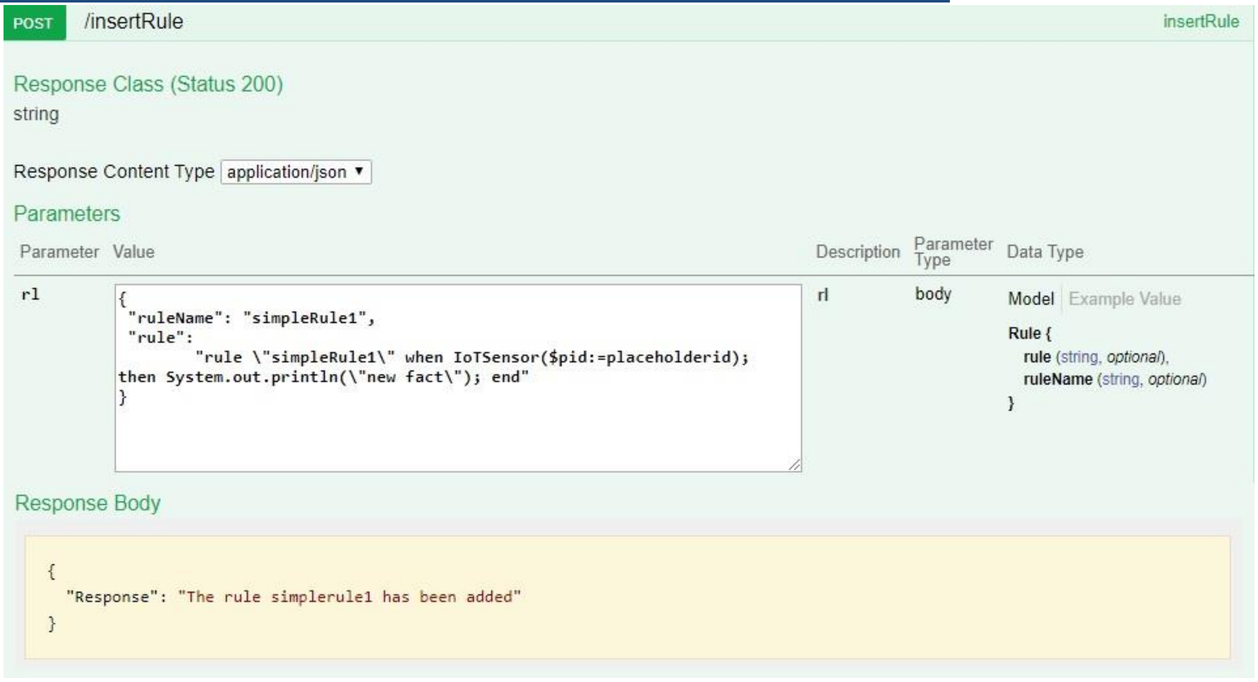
```
    }
    String jsonFromString="";
    if(connectivity) {
        jsonFromString = jsonString;
    }
    return jsonFromString;
}
```

4.4.2 COMPONENT API INTERACTIONS DESCRIPTION

The following table includes the main set of APIs that were developed. The APIs are either internal to the same layer or external i.e. cross-layer.

API	Status	API access type	Additional comments
insertRule	Deployed	internal	Access only for Pattern Orchestrator
removeRule	Deployed	internal	Access only for Pattern Orchestrator
getRule	Deployed	internal	Access only for Pattern Orchestrator
addFact	Deployed	external	Access for Pattern Orchestrator and cross-layer access from other Pattern Engines
factUpdate	Deployed	external	Access for Pattern Orchestrator and cross-layer access from other Pattern Engines
factStatus	Deployed	external	Access for Pattern Orchestrator and cross-layer access from other Pattern Engines
factRemove	Deployed	external	Access for Pattern Orchestrator and cross-layer access from other Pattern Engines

In the following figures all the APIs are presented with a sample input along with the corresponding response using SWAGGER.



POST /insertRule insertRule

Response Class (Status 200)
string

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type

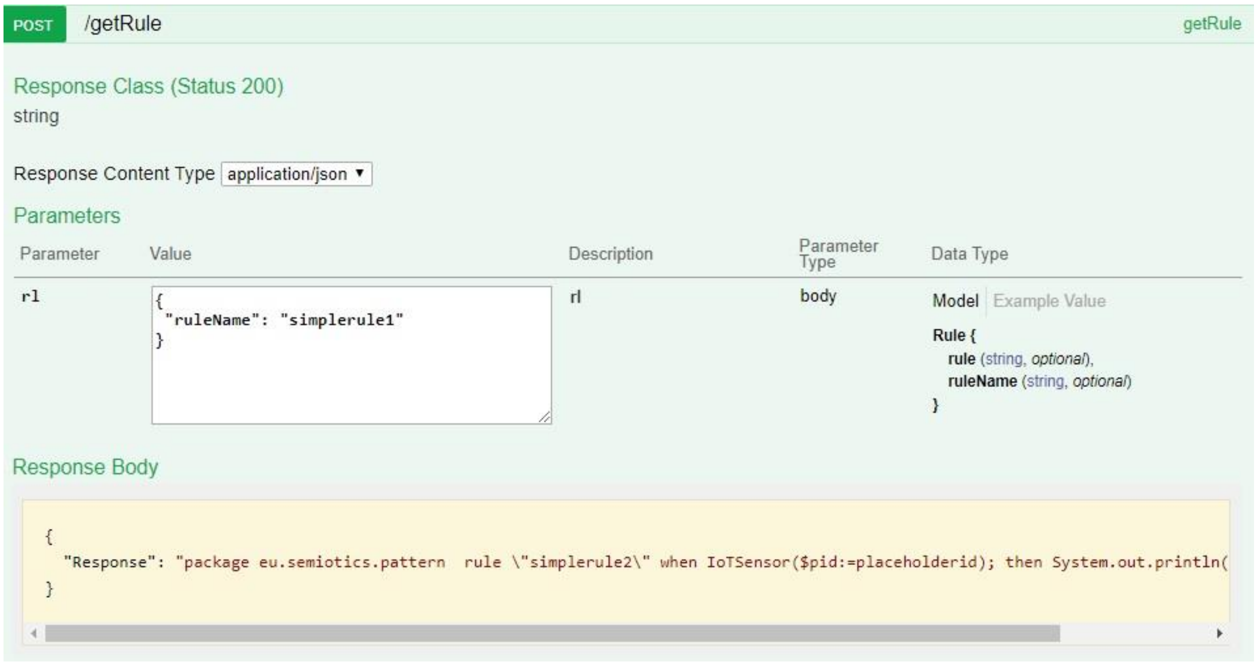
rl	{ "ruleName": "simpleRule1", "rule": "rule \"simpleRule1\" when IoTSensor($pid:=placeholderid); then System.out.println(\"new fact\"); end" }	rl	body	Model / Example Value
				Rule {
				rule (string, optional),
				ruleName (string, optional)
				}

```
{
 "ruleName": "simpleRule1",
 "rule":
        "rule \"simpleRule1\" when IoTSensor($pid:=placeholderid);
then System.out.println(\"new fact\"); end"
}
```

Response Body

```
{
  "Response": "The rule simplerule1 has been added"
}
```

FIGURE 28 INSERTRULE API



POST /getRule getRule

Response Class (Status 200)
string

Response Content Type application/json

Parameters

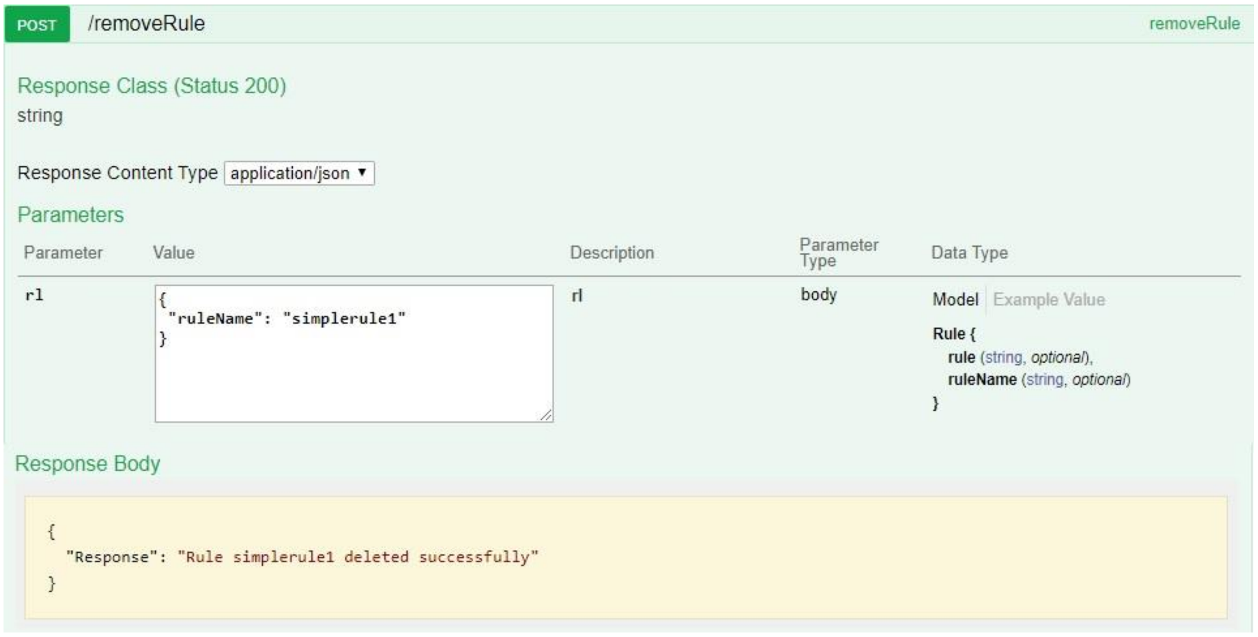
Parameter	Value	Description	Parameter Type	Data Type

rl	{ "ruleName": "simplerule1" }	rl	body	Model / Example Value
				Rule {
				rule (string, optional),
				ruleName (string, optional)
				}

Response Body

```
{
  "Response": "package eu.semiotics.pattern  rule \"simplerule2\" when IoTSensor($pid:=placeholderid); then System.out.println(
}
```

FIGURE 29 GETRULE API



POST /removeRule removeRule

Response Class (Status 200)
string

Response Content Type application/json ▾

Parameters

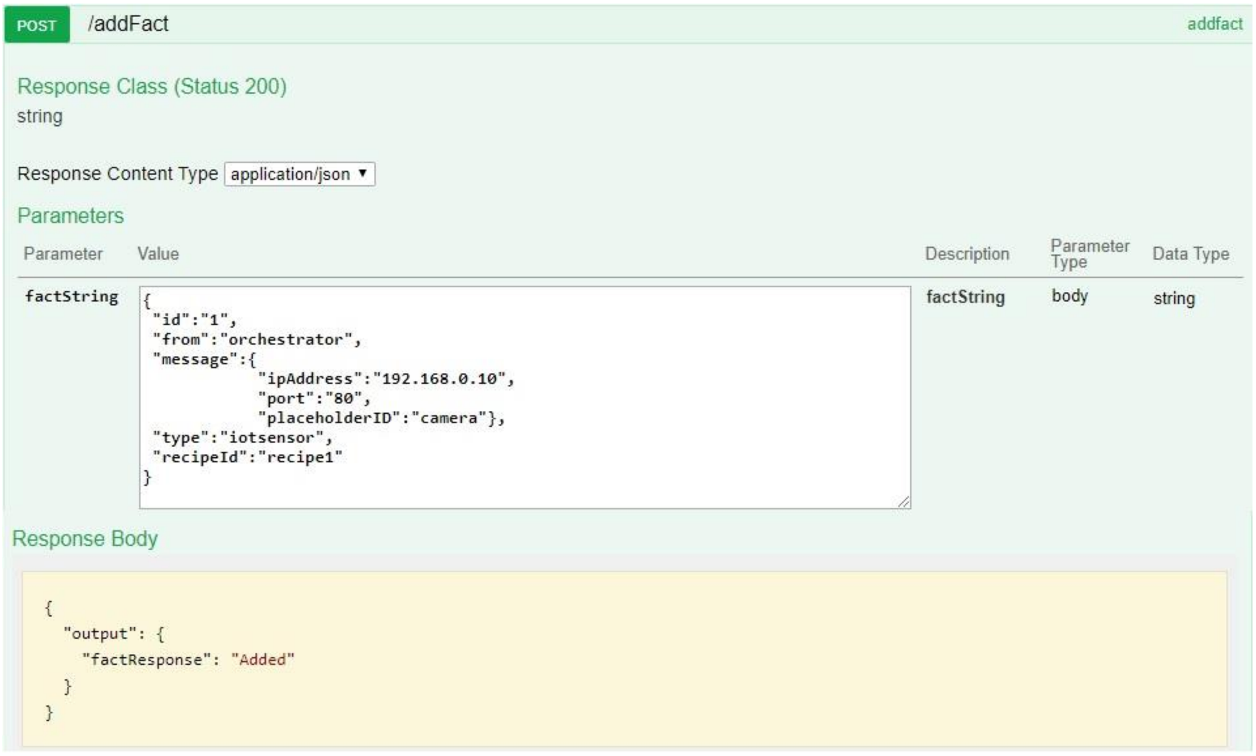
Parameter	Value	Description	Parameter Type	Data Type

rl	`{ "ruleName": "simplerule1" }`	rl	body	Model | Example Value
				Rule {
				rule (string, *optional*),
				ruleName (string, *optional*)
				}

Response Body

```
{
  "Response": "Rule simplerule1 deleted successfully"
}
```

FIGURE 30 REMOVERULE API



POST /addFact addfact

Response Class (Status 200)
string

Response Content Type application/json ▾

Parameters

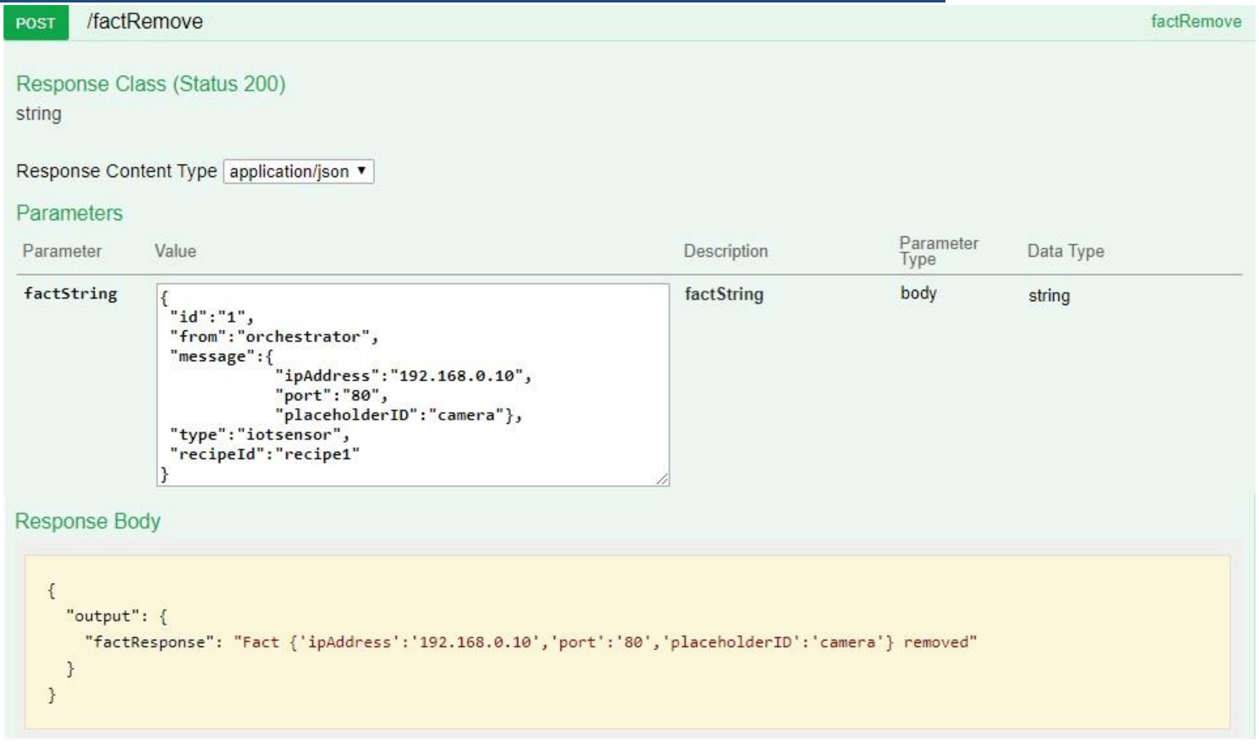
Parameter	Value	Description	Parameter Type	Data Type
factString	see below	factString	body	string

```
{
 "id":"1",
 "from":"orchestrator",
 "message":{
             "ipAddress":"192.168.0.10",
             "port":"80",
             "placeholderID":"camera"},
 "type":"iotsensor",
 "recipeId":"recipe1"
}
```

Response Body

```
{
  "output": {
    "factResponse": "Added"
  }
}
```

FIGURE 31 ADDFACT API



POST /factRemove factRemove

Response Class (Status 200)
string

Response Content Type application/json

Parameters

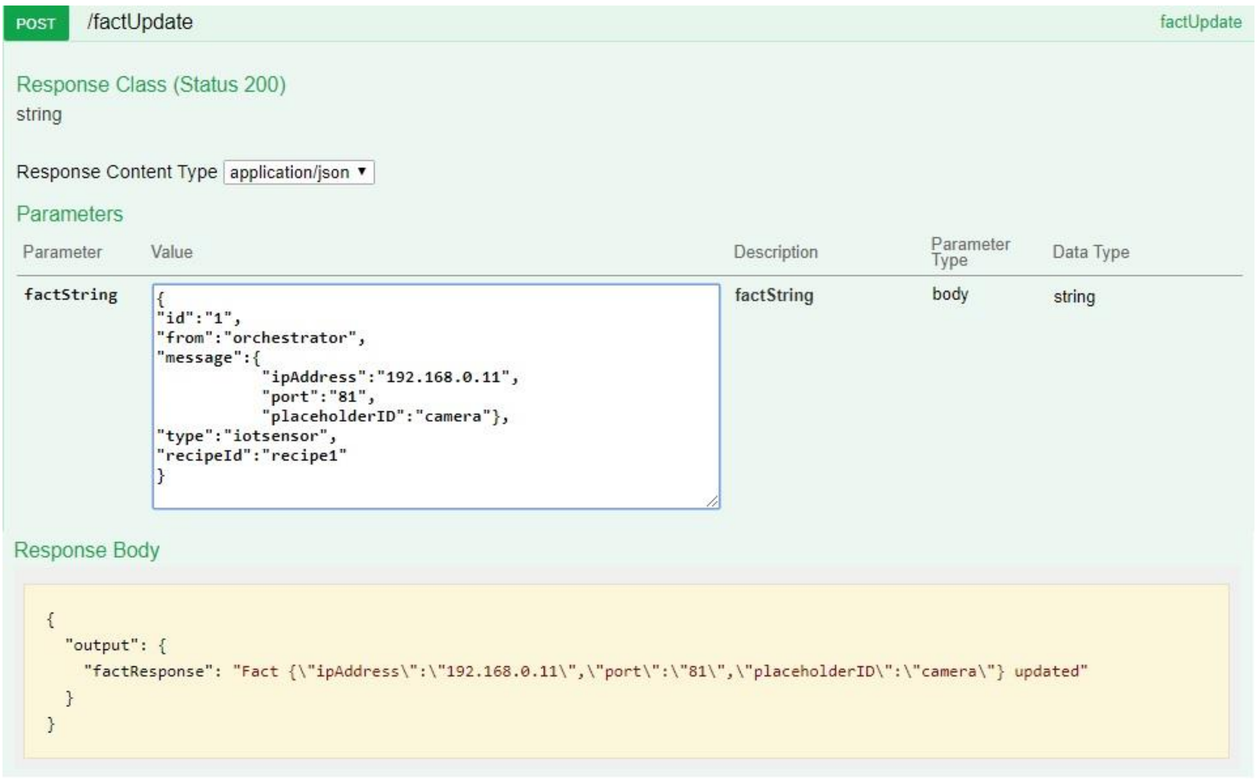
Parameter	Value	Description	Parameter Type	Data Type
factString	(see below)	factString	body	string

```
{
 "id":"1",
 "from":"orchestrator",
 "message":{
           "ipAddress":"192.168.0.10",
           "port":"80",
           "placeholderID":"camera"},
 "type":"iotsensor",
 "recipeId":"recipe1"
}
```

Response Body

```
{
  "output": {
    "factResponse": "Fact {'ipAddress':'192.168.0.10','port':'80','placeholderID':'camera'} removed"
  }
}
```

FIGURE 32 FACTREMOVE API



POST /factUpdate factUpdate

Response Class (Status 200)
string

Response Content Type application/json

Parameters

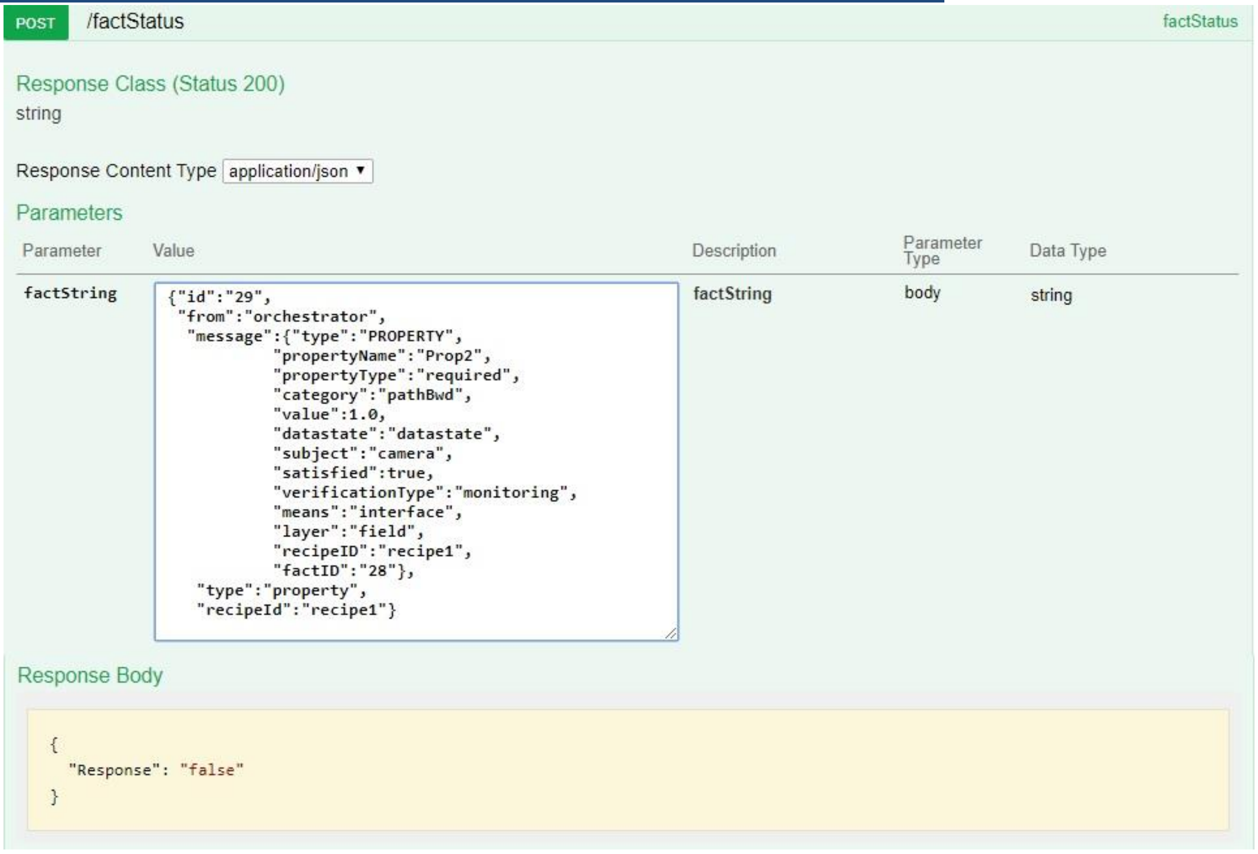
Parameter	Value	Description	Parameter Type	Data Type
factString	(see below)	factString	body	string

```
{
"id":"1",
"from":"orchestrator",
"message":{
          "ipAddress":"192.168.0.11",
          "port":"81",
          "placeholderID":"camera"},
"type":"iotsensor",
"recipeId":"recipe1"
}
```

Response Body

```
{
  "output": {
    "factResponse": "Fact {\"ipAddress\":\"192.168.0.11\",\"port\":\"81\",\"placeholderID\":\"camera\"} updated"
  }
}
```

FIGURE 33 FACTUPDATE API


POST
/factStatus
factStatus
Response Class (Status 200)
Response Content Type application/json
Parameters
Response Body

string

Parameter	Value	Description	Parameter Type	Data Type
factString	see below	factString	body	string

```
{"id":"29",
 "from":"orchestrator",
  "message":{"type":"PROPERTY",
            "propertyName":"Prop2",
            "propertyType":"required",
            "category":"pathBwd",
            "value":1.0,
            "datastate":"datastate",
            "subject":"camera",
            "satisfied":true,
            "verificationType":"monitoring",
            "means":"interface",
            "layer":"field",
            "recipeID":"recipe1",
            "factID":"28"},
   "type":"property",
   "recipeId":"recipe1"}
```

```
{
  "Response": "false"
}
```

FIGURE 34 FACTSTATUS

4.4.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

During Cycle 1 the environment was based on Apache Maven 3.6.1, JBoss Drools[2] 7.15, and gRPC[3] with Protocol Buffers[4] Version 3. Based on that, we had successfully created a gRPC server loading the Pattern Engine with a basic set of Drools rules. Using a test client, we were able to successfully make gRPC calls to the server to request verification of specific pattern rule. API definition, pattern rules instantiation and a standalone repository for pattern rules were implemented.

During Cycle 2 gRPC and Protocol Buffers were replaced with a corresponding REST approach for compatibility purposes with other SEMIoTICS components. The Spring Framework was adopted to build REST web services. Using REST clients, other SEMIoTICS components are able to successfully make REST requests to the Backend Pattern Engine API. Communication of network and field updates to Backend Pattern Engine were implemented as well as testing of flow from Recipe Cooker.

During Cycle 3 the corresponding REST approach, was capable to fulfill all the needs with other SEMIoTICS components and therefore there was no need to make additional changes on that front. Classes that were introduced during Cycle 2 were refined, encryption was added to the REST endpoints and adaptation to maintain desired properties were implemented with pattern rules.

[2] https://docs.jboss.org/drools/release/7.15.0.Final/drools-docs/html_single/index.html
[3] https://grpc.io/
[4] https://developers.google.com/protocol-buffers/docs/proto3

4.4.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO PATTERN ENGINE (BACKEND)

SEMIoTICS Requirement		Pattern language considerations	Reference
Req. ID	Description		
R.BC.18	The backend layer must feature SPDI pattern reasoning embedded intelligence capabilities	The system model and associated pattern language developed are tailored to the multi-layer approach of SEMIoTICS, also anticipating intra- and cross-layer reasoning. Furthermore, Pattern reasoning components (referred to as Pattern Engines) are embedded at all layers; see subsection 3.7.2.2 of D4.8.	The system model and associated pattern language developed are tailored to the multi-layer approach of SEMIoTICS, also anticipating intra- and cross- layer reasoning. Furthermore, Pattern reasoning components (referred to as Pattern Engines) are embedded at all layers; see subsection 3.7.2.2 of D4.8. The real-time reasoning will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning.
R.BC.20	The backend layer must aggregate intra-layer as well as inter-layer SPDI status information to enable local and global intelligence reasoning and adaptation		
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	The real-time reasoning will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning.	As can be seen in subsections 3.2 (Language Model) and 3.3 (Language Constructs) of D4.8, instances of Java class *Link* allow Pattern Engines to monitor and verify connectivity among IoT service orchestration

R.UC2.3	The SEMIoTICS platform SHOULD guarantee proper connectivity between the various components of the SARA distributed application. The SARA solution is a distributed application not only because it uses different cloud services (e.g. AREAS Cloud services, AI services) from different remote computational nodes, but also because the SARA application logic itself is distributed across various edge nodes (SARA Hubs).		components. This also encompasses the pattern-driven interoperability mechanisms developed in the context T3.4 (and which are further described in D3.4), which leverage the language and pattern definitions. Through the above and the integration of pattern-based capabilities at the network level (SDN pattern engine), connectivity and QoS parameters can also be monitored.
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication)	As can be seen in subsections 3.2 (Language Model) and 3.3 (Language Constructs) of D4.8, instances of Java class Link allow Pattern Engines to monitor and verify connectivity among IoT service orchestration components. This also encompasses the pattern-driven interoperability mechanisms developed in the context T3.4 (and which are further described in D3.4), which leverage the language and pattern definitions. Through the above and the integration of pattern-based capabilities at the network level (SDN pattern engine), connectivity and QoS parameters can also be monitored.	As can be seen in subsections 3.3 (Language Model) and 4.5 (Language Constructs) of D4.8, Java class *Property* owns an attribute *Category,* allowing Pattern Engines to monitor QoS properties of the components of an IoT service orchestration. Moreover, the properties associated with the *Link* class directly affect the requirements relayed to the network layer (with the associated properties reasoned by the Pattern Engine embedded at the SDN controller; see subsection 3.7.2.2 of D4.8).
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration needed by SPDI pattern		
R.UC2.15	The SEMIoTICS platform SHOULD provide low latency connectivity between the SARA hubs and cloud services (i.e. AREAS cloud services and AI services) to allow offloading of near real-time computation intensive tasks to the cloud. Therefore, SARA hubs need to send with minimal delay: • raw range data (e.g. from Lidar sensors) to identify proximal objects/objects, • real-time audio stream for speech analysis, and real-time raw video stream (object/people recognition, gesture recognition, posture analysis).		

R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.	Concerns regarding any sensitive data that is generated, processed, stored and exchanged at all layers must be considered, enforcing and monitoring the corresponding security mechanisms, especially when different trust domains are involved. Proper authentication and authorisation services are a necessity when trying to safeguard the security and privacy of data and services. These aspects must be defined in the pattern language, monitored and enforced, considering the different types of devices (e.g. sensors, network controllers, backend servers), actors (e.g. humans, machines/applications) and interaction types (e.g. maintenance or medical staff, simple users). These, along with cryptographic mechanisms, will need to be used to establish trust within and across domains. Moreover, privacy considerations will have to be included (e.g. protection of private data at rest and in transit, data anonymization and minimisation, data retention; see section 2.2.1 above). In addition to the above, patterns can also be leveraged to monitor and enforce the presence of security mechanisms in different IoT orchestrations.	Security-related properties (such as Confidentiality) are at the core of the properties covered in the SEMIoTICS system model (subsection 3.3 of D4.8) and associated language (subsection 3.4 of D4.8). Moreover, a first version of security-related pattern rules can be seen in subsection 4.1 of D4.8, while a first set of Privacy Patterns can be seen in subsection 4.1.5 of D4.8. Moreover, using the pattern language, different verification types can be declared for each of the properties (see subsection 3.3 of D4.8); this can be exploited to define interfaces with the various security mechanisms which will allow the verification of the different SPDI properties associated with them (e.g., monitoring encryption mechanisms that provide the property of Confidentiality). This will be achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.2), which can be used for providing Pattern Rules with the appropriate input for reasoning on relevant security and privacy -related aspects, such as secure deletion of unnecessary data, limitation of sampling via a variant of the mechanisms used to ensure QoS parameters, etc.

R.S.5	Before sensitive data is being transmitted, the respective components SHALL be authenticated as defined by requirements R.S.3 and R.S.4	As can be seen in subsections 3.3 (Language Model) and 3.4 (Language Constructs) of D4.8, Java class Property owns an attribute Category, allowing Pattern Engines to monitor QoS properties of the components of an IoT service orchestration. Moreover, the properties associated with the Link class directly affect the requirements relayed to the network layer (with the associated properties reasoned by the Pattern Engine embedded at the SDN controller; see subsection 3.7.2.2 of D4.8).	All key security and privacy properties are covered within the SEMIoTICS patterns (see Section 4 of D4.8). Furthermore, the language expressiveness allows the definition of the appropriate conditions (facts) to be verified in order to provide real-time verification of the properties sketched by these requirements (see subsections 3.4 and 3.9 of D4.8).
R.P.1	The collection of raw data MUST be minimized.	Security-related properties (such as Confidentiality) are at the core of the properties covered in the SEMIoTICS system model (subsection 3.3 of D4.8) and associated language (subsection 3.4 of D4.8). Moreover, a first version of security-related pattern rules can be seen in subsection 4.1 of D4.8, while a first set of Privacy Patterns can be seen in subsection 4.1.5 of D4.8. Moreover, using the pattern language, different verification types can be declared for each of the properties (see subsection 3.3 of D4.8); this can be exploited to define interfaces with the various security mechanisms which will allow the verification of the different SPDI properties associated with them (e.g., monitoring encryption mechanisms that provide the property of Confidentiality).	As documented in subsection 4.2 of D4.8, the SEMIoTICS patterns (and by extension the pattern-driven reasoning capabilities of SEMIoTICS at all layers) include all key privacy properties.
R.P.3	Storage of data MUST be minimized.		
R.P.6	Data MUST be anonymized wherever possible by removing the personally identifiable information in order to decrease the risk of unintended disclosure.		
R.P.8	Data MUST be stored in encrypted form.		
R.P.9	Repeated querying for specific data by applications, services, or users that are not intent to act in this manner SHALL be blocked.		

		This is achieved in conjunction with the monitoring framework (developed in the context of T4.2, and documented in D4.9), which can be used for providing Pattern Rules with the appropriate input for reasoning on relevant security and privacy -related aspects, such as secure deletion of unnecessary data, limitation of sampling via a variant of the mechanisms used to ensure QoS parameters, etc.	
R.GSP.7	The cloud platform SHALL be able to monitor the execution of an app, in particular its interactions with other apps, the network interface, and APIs.	All key security and privacy properties are covered within the SEMIoTICS patterns (see Section 4 of D4.8). Furthermore, the language expressiveness allows the definition of the appropriate conditions (facts) to be verified in order to provide real-time verification of the properties sketched by these requirements (see subsections 3.4 and 3.9 of D4.8).	As can be seen in section 3.2 (Language Model) and 3.3 (Language Constructs) of D4.8, the pattern language that has been created can declare Properties whose verification type is *Monitoring*. That allows for capturing the monitoring critical aspects and enabling the reasoning on parameters related to properties such as reliability. As above, the necessary inputs will be aggregated from the monitoring framework of SEMIoTICS (T4.2/D4.2).

			Availability and Dependability patterns developed within SEMIoTICS are able to accommodate the monitoring defined in these requirements (see subsections 4.1.3 and 4.3 of D4.8). These features will be further explored and demonstrated in the context of the UC3 scenarios, as detailed in subsection 7.3.
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised	As documented in subsection 4.2 of D4.8, the SEMIoTICS patterns (and by extension the pattern-driven reasoning capabilities of SEMIoTICS at all layers) include all key privacy properties.	All pattern engine components (see subsection 3.7.2.2 of D4.8) feature integrated logging mechanisms that allow for auditing on all pattern-driven reasoning and adaptation actions triggered. In other parts of the SEMIoTICS framework and protected infrastructure, the deployment and monitoring of the proper operation of the logging functions can be introduced as with any other mechanism (see subgroups of requirements above).

4.5 Backend Semantic Validator

The aim of the Backend Semantic Validator (BSV) component is to tackle the semantic interoperability issues that arise in the SEMIoTICS framework, at the application orchestration layer. The Backend Semantic Validator provides:

- validation mechanisms to ensure semantic interoperability,
- connection with Recipe Cooker to resolve the semantic conflicts using the Adaptor Nodes,
- connection with external IoT platforms to enable interoperability between these targeted external IoT enabling platforms and SEMIoTICS and
- adaptability taking to account the interoperability of devices that are used in SEMIoTICS, interacting with the Pattern Engine (Backend layer).

TABLE 11 BSV BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
First installation of a server using gRPC and protocol buffers	In order to receive a request from an IoT application, a service is required from the BSV side. For this reason, a server is implemented with the appropriate endpoints, using gRPC framework and protocol buffers, for the aforementioned communication.	Cycle 1	Delivered
Establish communication between Recipe Cooker and BSV	Recipe Cooker is the primary tool for designing the flow that involves Things as well as other components. In order to be able to guarantee the semantic interoperability between the Things, the Recipe Cooker needs to be able to communicate with BSV. The output of the Recipe cooker is in JSON format that BSV parses.	Cycle 2	Delivered
Re-implement BSV's endpoints using RESTFul services instead of gRPC	Due to compatibility issues with Recipe Cooker, the need to abandon gRPC implementation and replace it with RESTFul Services.	Cycle 2	Delivered
Resolve semantic conflicts using the Adaptor Nodes	Upon receiving a recipe from Recipe Cooker, the BSV checks the semantic validity of the involved Things and responds accordingly to Recipe Cooker. When two Things are not semantically interoperable, the BSV creates an Adaptor Node, which resolves the semantic conflicts between them.	Cycle 2 & 3	Delivered
Communication with the Semantic API & Protocol Binding component	When the request of an IoT application results in the involvement of brownfield systems, it is necessary to forward the request to the Semantic API & Protocol Binding component, which is responsible to trigger the GW Semantic Mediator in the filed layer. Therefore, communication between BSV and Semantic API & Protocol Binding needs to be implemented.	Cycle 3	This feature was not implemented, because it was out of the requirements of the final implementation of the SEMIoTICS UCs scenarios (see D4.11)
Interact with other European platforms (e.g. FIWARE).	The request from an IoT application includes Thing Description in JSON-LD format, which may reference other European schemas different from iot.schema (e.g. schema.lab.fiware.org). Therefore, an example of interaction with at least one European platform should be implemented.	Cycle 3	Delivered
Interact with Pattern related modules	One of the features promised in Pattern Engine regards interoperability properties. The semantic interoperability, in particular, implies the interaction between BSV and Pattern Engine.	Cycle 3	Delivered

4.5.1 DEVELOPMENT STATUS

The phase of the BSV implementation that involves the decision of the interoperability between the things and harmonization of the semantic model capabilities of them with the registration of extra Adaptor Nodes in the Recipe Cooker was implemented during the Cycle 2 (see D4.7- subsection 4.5.2). However, the procedure of the Adaptor Node development, with specific functionality, are modified in Cycle 3. The reason for this further implementation came from the fact that the initial implementation required the re-run of the Recipe Cooker component in order to the new node modules be installed at runtime. Hence, the HTTP API methods from node-RED are used and Function node for the corresponding Adaptor Node are introduced in the flow recipe to resolve the interoperability issues.
Specifically, to enable the interoperability between the flow's Things, a number of different step phases are required, following the corresponding sequence diagram in Figure 36. In fact, the Recipe Cooker component which is responsible for cooking (creating) recipes reflecting user requirements, sends the recipe in Pattern Orchestrator component (using POST method request), which is in charge of the automated configuration, coordination, and management of different patterns and their deployment to express requirements of the flows to guarantee interoperability based on architectural patterns.
The second phase includes the insertion of the interoperability requirement as a POST from the Orchestrator to the Pattern Engine to enforce the respective pattern rules (see Figure 35). The pattern is expressed in a machine-processable Drool rule format of the said semantic interoperability for any inserted flow. The *when* part identify the requested placeholders placed in sequence, required to satisfy the semantic interoperability property. If the conditions are met, the rule in *then* can guarantee that the requested property is satisfied.

```
 rule "Sequence Semantic Interoperability Verification"
 when
        Placeholder($pA:=placeholderid)
        Property ($pA:=subject, category=="semantic", $prvaluein1:=input_value,
          $prvalueout1:=output_value, satisfied==true)
        Placeholder($pB:=placeholderid)
        Property ($pB:=subject, category=="semantic", $prvaluein2:=input_value,
          $prvalueout2:=out_value, satisfied==true)
        Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
          $PR: Property ($sId:=subject, category=="semantic", $prvalueout1==$prvaluein2,
        satisfied==false)
 then
        modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

FIGURE 35 SEMANTIC INTEROPERABILITY VERIFICATION DROOL RULE

Therefore, this rule used by Pattern Engine to trigger the BSV, which resolves semantic interoperability issues, between any link of Things in the flow recipe. Particularly, the BSV receives a request with the flow id from and the Things' id for each link. Based on this information, the component begins the procedure to tackle the semantic interoperability issues between these two things from the said flow. For that reason, it sends SPARQL query to Thing Directory to receive the Thing Description of the Things.
In the sequel, the final phase of the interoperability adaptation is the following. It involves the harmonization of the semantic model capabilities with the registration of extra Adaptor Nodes in the Recipe Cooker if required. Namely, there are three possible results. Firstly, the link source and destination are interoperable, so the BSV updates the Pattern Engine with the TRUE response. Secondly, the link source and destination are not interoperable and the BSV can add Adaptor Nodes in order to guarantee the interoperability. In this case, the BSV not only sends the TRUE response to pattern engine, but it also updates the flow using the PUT method[5] of Recipe Cooker API and create the corresponding Functions nodes. Lastly, when the link source and destination are not interoperable and the validator does not have the required information to develop the adaptor nodes, the validator sends FALSE response to Pattern Engine.

[5] https://nodered.org/docs/api/admin/methods/

Furthermore, another role of the BSV is the connection of SEMIoTICS with other external platforms (e.g. FIWARE). This integration is developed during the Cycle 3 and follows the above procedure with the difference that BSV sends request to the Orion Context Broker FIWARE platform to receive the context data Description of FIWARE Sensor that participates in the recipe flow. More details are mentioned in the D4.11 – subsection 6.2.2.

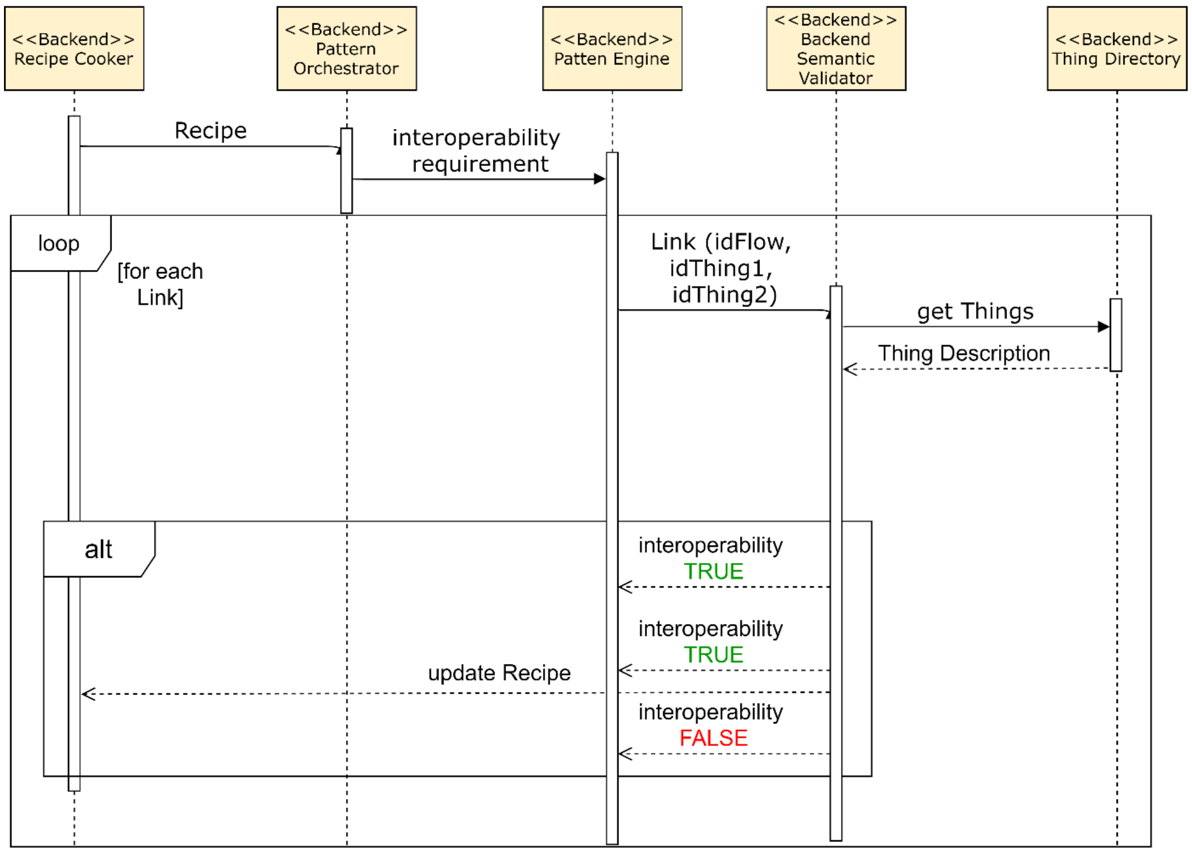

<<Backend>> Recipe Cooker
<<Backend>> Pattern Orchestrator
<<Backend>> Patten Engine
<<Backend>> Backend Semantic Validator
<<Backend>> Thing Directory
Recipe
interoperability requirement
loop
[for each Link]
Link (idFlow, idThing1, idThing2)
get Things
Thing Description
alt
interoperability TRUE
interoperability TRUE
update Recipe
interoperability FALSE

FIGURE 36 SEQUENCE DIAGRAM FOR SEMANTIC INTEROPERABILITY ADAPTATION MECHANISMS

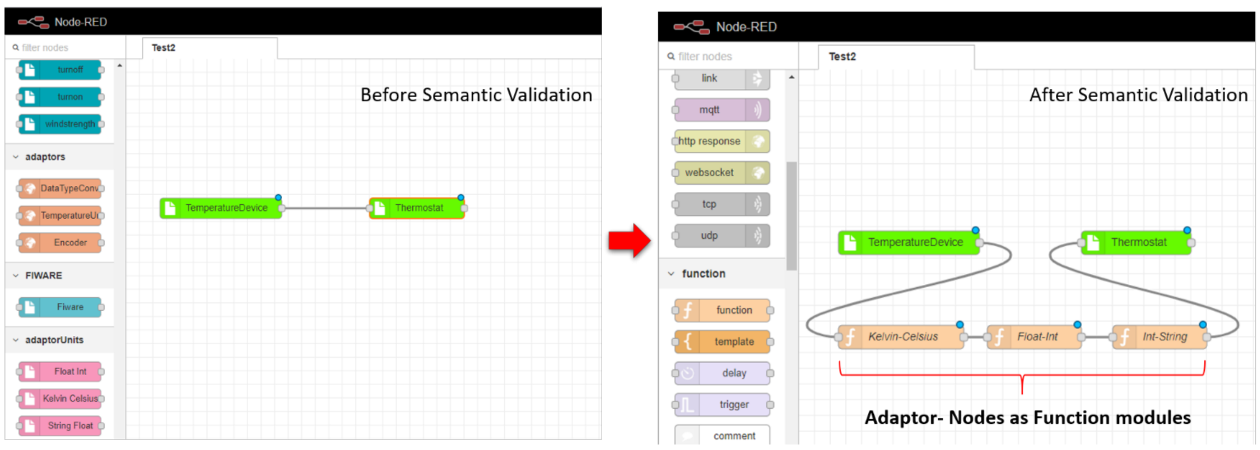


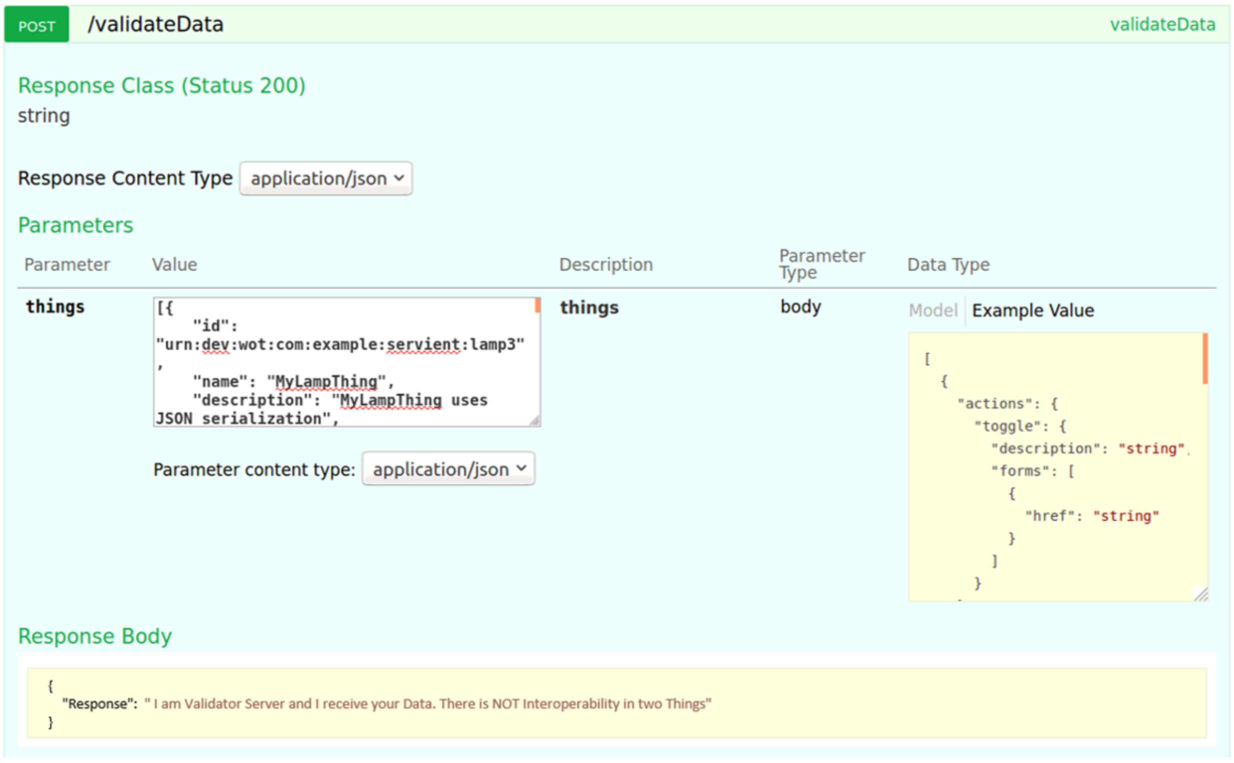
FIGURE 37 SEMANTIC VALIDATION/ADAPTATION MECHANISMS

4.5.2 COMPONENT API INTERACTIONS DESCRIPTION

The following table includes the main set of APIs that were developed. The APIs are either internal to the same layer.

API	Status	API access type	Additional comments
validateData	Deployed	internal	Access for any component (Backend Layer)
validateRecipeFlow	Deployed	internal	Access for Pattern Engine (Backend Layer)

In the following Figure 38 and Figure 39, the APIs are presented with a sample input along with the corresponding response using SWAGGER.


POST /validateData validateData

Response Class (Status 200)
string

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
things	see below	things	body	Model / Example Value

Value:

```
[{
    "id":
"urn:dev:wot:com:example:servient:lamp3"
,
    "name": "MyLampThing",
    "description": "MyLampThing uses
JSON serialization",
```

Parameter content type: application/json

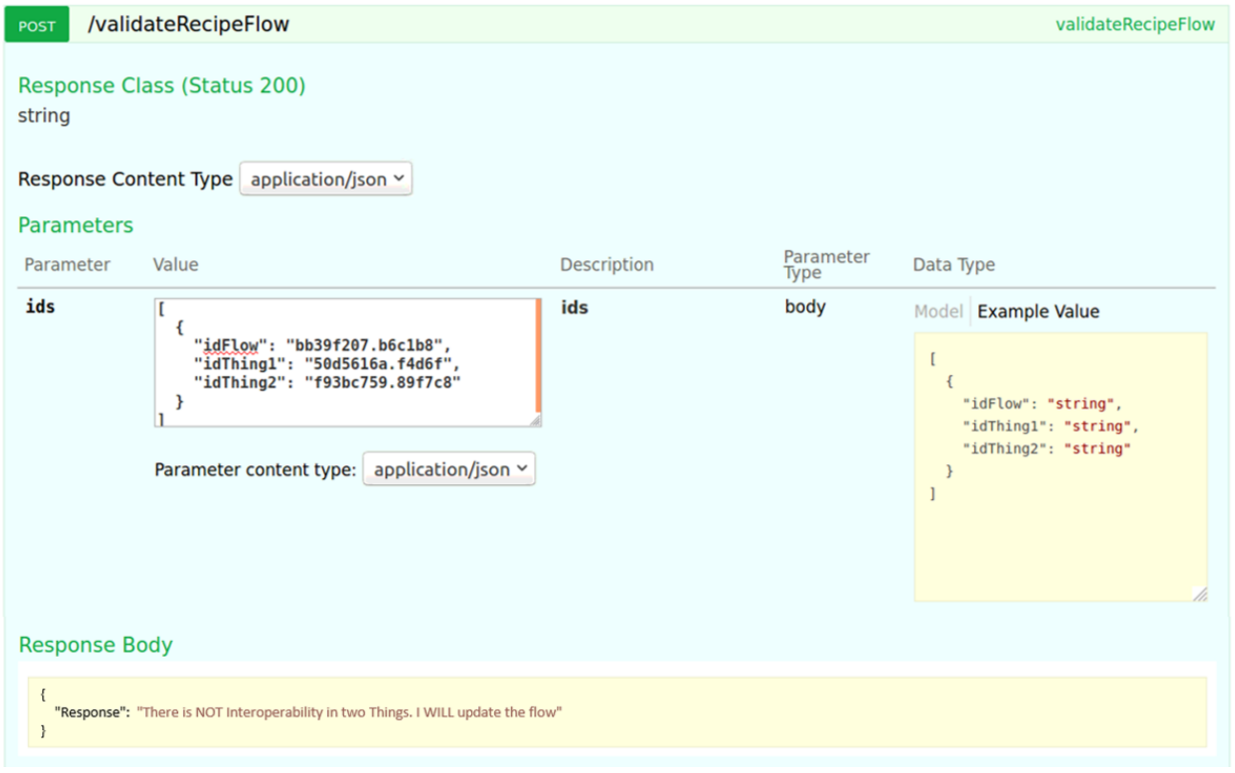
Example Value:

```
[
  {
    "actions": {
      "toggle": {
        "description": "string",
        "forms": [
          {
            "href": "string"
          }
        ]
      }
```

Response Body

```
{
  "Response": " I am Validator Server and I receive your Data. There is NOT Interoperability in two Things"
}
```

FIGURE 38 VALIDATEDATA API


POST /validateRecipeFlow validateRecipeFlow

Response Class (Status 200)
string

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
ids	`[ { "idFlow": "bb39f207.b6c1b8", "idThing1": "50d5616a.f4d6f", "idThing2": "f93bc759.89f7c8" } ]` Parameter content type: application/json	ids	body	Model Example Value `[ { "idFlow": "string", "idThing1": "string", "idThing2": "string" } ]`

Response Body

```
{
  "Response": "There is NOT Interoperability in two Things. I WILL update the flow"
}
```

FIGURE 39 VALIDATE RECIPE FLOW API

4.5.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

During Cycle 1 a server was implemented loading the BSV component, using gRPC[6] framework and protocol buffers, for the verification of semantic interoperability between two Things (i.e. sensor, actuator), which are described with two different TDs. Moreover, the installation and interaction (discovery and registration TDs) with Thing Directory was accomplished (see D4.6).
During Cycle 2 gRPC and Protocol Buffers were replaced with a corresponding REST approach for compatibility purposes with other SEMIoTICS components. The Spring Framework was adopted to build REST web services to provide services for receiving data in a convenient format, creating new data, updating data and deleting data between the interaction of SEMIoTICS architecture components. The first approach for resolving any possible semantic conflicts between the interacting different Things, using or creating the corresponding Adaptor Nodes in Recipe Cooker was implemented (see D4.7).
Lastly, the Cycle 3 includes the modification of the initial Adaptor Nodes implementation for the semantic conflicts of Things in the Recipe Cooker. Additionally, connection with external IoT platforms to enable interoperability between these targeted external IoT enabling platforms and SEMIoTICS was implemented in this cycle. Finally, the adaptation approach, taking to account the interoperability of devices that are used in SEMIoTICS and interacting with the Pattern Engine (Backend layer) is part of the cycle 3 (see D4.11).

4.5.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO BACKEND SEMANTIC VALIDATOR

Requirements (D2.3)	Description	Reference

[6] https://grpc.io/

R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	**D4.11 -** Section 6
R.UC1.8	Semantic and robust bootstrapping/registration of IIoT sensors and actuators with IIoT gateway MUST be supported	**D4.11 -** Subsection 2.2.1
R.UC1.9	Semantic interaction between use-case specific application on IIoT Gateway and legacy turbine control system MUST be supported	**D4.11 -** Section 8, Section 9
R.UC1.12	Standardized semantic models for semantic-based engineering and IIoT applications SHALL be utilized	**D4.11 -** Subsection 4.2, Subsection 4.3
R.UC2.3	The SEMIoTICS platform SHOULD guarantee proper connectivity between the various components of the SARA distributed application. The SARA solution is a distributed application not only because it uses different cloud services (e.g. AREAS Cloud services, AI services) from different remote computational nodes, but also because the SARA application logic itself is distributed across various edge nodes (SARA Hubs).	**D4.11 –** Section 6, Subsection 9.2
R.UC2.6	The SEMIoTICS platform SHOULD allow the SARA solution to retrieve the resources exposed by registered devices via their object model (i.e. a data structure wherein each element represents a resource, or a group of resources, belonging to a device). The SEMIoTICS platform SHOULD support at least the OMA LWM2M object model.	**D4.11 -** Subsection 3.2, Subsection 9.2
R.UC2.11	The SEMIoTICS platform SHOULD allow a SARA component to request a group of devices to take/initiate an action (e.g. turn on/off a light bulb).	**D4.11 –** Subsection 2.2, Subsection 9.2
R.UC3.1	IoT Sensing unit shall be able to embed environmental (e.g. temperature, pressure, humidity, light) and inertial sensors (accelerometer, gyroscope).	**D4.11 -** Subsection 2.2, Subsection 9.3
R.UC3.15	A use case specific serialized message protocol is required to coordinate the gateway and its associated units and exchange data / events / anomalies between them. JSON will be the preferred serialization format adopted.	**D4.11 -** Subsection 5.1.1, Subsection 9.3

4.6 Thing Directory

Thing Directory is a component hosting Thing Descriptions (TDs) of registered things and can be used to browse and discover Things based on their TDs. This is the Thing Directory deployed on the *backend level* (or *network level* depending on demo setup). It interacts with the *Local Thing Directory* that runs on IIoT GW.

Table 12 presents the identified backlog scope and assignment to development cycles planned including the implementation status.

TABLE 12 THING DIRECTORY BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Implementation	Based on an existing, open-source implementation of the Thing Directory, we provided a solution, which is compliant	Cycle 1	Delivered

	with the W3C Thing Description. We packaged the Thing Directory as a Docker container for easy deployment.		
Deployment: Cloud-level	We deployed the Thing Directory in an AWS Cloud environment, accessible for all project partners	Cycle 1	Delivered
Implementation: Up-to-date TDs	A mechanism is being implemented that uploads the latest version of TDs from field devices to the Thing Directory so that components such as the Recipe Cooker will always have the up-to-date view on the field level.	Cycle 3	In progress

4.6.1 DEVELOPMENT STATUS

The works on the mechanism for updating the TDs to their current version is being implemented from the field layer side. The status of the developments is in progress.

4.6.2 COMPONENT API INTERACTIONS DESCRIPTION

API	Status	API access type	Additional comments
registerThing	Done	HTTP	-
deleteThing	Done	HTTP	-
getThings	Done	HTTP	-
getThingsDetails	Done	HTTP	-
searchUsingSPARQL	Done	HTTP	-

4.7 Recipe Cooker

Recipe Cooker component is responsible for cooking (creating) recipes that reflect user requirements on different layers (cloud, edge, network), transforming recipes into understandable rules for each of layer. It uses Thing directory with all necessary models to create these rules.

Table 13 presents the identified backlog scope and assignment to development cycles planned including the implementation status.

TABLE 13 RECIPE COOKER BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Design: merge of recipe + pattern concepts	Introduced the recipe concept, as developed in the BIG IoT project, within the SEMIoTICS architecture. A recipe can be considered as a template for an IoT application. At this point in time, we merged the recipe concept together with the SPDI pattern concept. : By enabling the application-centric definition of recipes and automatically translating them into SPDI patterns and network-specific details, we hide the details of network configuration from the developers and they can fully concentrate	Cycle 1	Delivered

	on the program logic of their IoT application. (Documented in D3.4, Section 2.3 as well as D4.1)		
Design: translation of recipes into facts	After the conceptual merging of the two concepts, we worked out a mechanism for a translation chain from the recipe, over SDN network mode, to patterns and finally facts in the rule engine. We, therefore, enable the semantic description of application-level constraints and their automatic conversion into network configurations. (Documented in D4.4)	Cycle 1	Delivered
Implementation: Recipe Cooker as Node-RED extension	Redefinition of the recipe cooker, as implemented in BIG IoT, by use of the Node-RED visual programming environment. The advantage of Node-RED is that we can build upon a broad ecosystem of nodes for the integration of IoT devices and services. (Documented in Section 4.7.1)	Cycle 1	Delivered
Implementation: Distributed execution of recipes	Extension of the recipe cooker's execution environment for IoT flows to allow their distributed IoT orchestration. The extension enables the deployment of the components of a flow to different devices. Further, the extension allows the definition of application-specific QoS constraints to be auto-translated into patterns for network configuration. Therefore, the so-called 'DirectCom' node was developed, which allows representing the network in the application flows defined via the recipe cooker. For example, an application flow could utilize this 'DirectCom' node to transmit a video stream from a camera to an Edge device that runs an AI pipeline on the video images. In this example, the DirectCom node allows now to define the video frame rate to a minimum of 15 frames per second. This is communicated to the Pattern Orchestrator and then the Pattern Engine for the network to be configured and monitored.	Cycle 2	In finalization
Implementation: Distributed AI	According to the wind turbine use case, a distributed AI approach is implemented with the Recipe Cooker by implementing nodes for the execution of machine learning models that can detect grease leakage in a turbine. Therefore, dedicated nodes for AI inference have been developed to	Cycle 3	Delivered

	implement the AI pipeline and the use case. In a first proof of concept, these nodes realize the functionalities to (1) read images in high frequency from the video stream, (2) convert an image into a tensor, and (3) to classify the tensor according to a defined Neural Network model.		
Implementation: Federated Learning	To support the wind turbine use case on oil leakage detection, existing Neural Network models need to be retrained for the particular imagery expected to be seen inside the turbine – either with leaked grease/oil or without it. This retraining should be done locally at each turbine to avoid sending training data (large imagery data) over the network. However, a central model should aggregate the model updates from the different turbines. Therefore, nodes have to be implemented which allow the retraining, and the federation of the model updates.	Cycle 3	Delivered
Implementation: second proof of concept on audio analytics	In a second proof of concept for the wind turbine use case, we will implement Federated Learning on an unsupervised AI model. This will support the detection of anomalies in the noise generated from the turbine.	Cycle 3	In progress

4.7.1 DEVELOPMENT STATUS

We have implemented components, which realize the inference of distributed AI models as well as their training with a Federated Learning approach. We have realized these components as Node-RED nodes.

We utilized these nodes to build a proof of concept for oil/grease leakage in wind turbines based on video imagery. A second proof of concept for anomaly detection in turbine noises based on audio data is being implemented currently.

4.7.2 COMPONENT API INTERACTIONS DESCRIPTION

API	Status	API access type	Additional comments
user interface	done	visually	The common Node-RED UI is utilized.
recipe access / modification	done	HTTP	Based on flow read/modification offered by Node-RED.

4.8 Security Manager (backend)

The Security Manager in the backend layer is the component that is responsible for ensuring end-to-end security and safety. Its development started in Cycle 2. The Security Manager helps SEMIoTICS to tackle the security and privacy problems that arise from the multi-tenant scenarios in a variety of levels, i.e., from the networking layer to the application layer. Therefore, the SEMIoTICS architectural framework depicted in Table 14 shows several Security Manager components (at the level of the backend and additionally at the network- and field-level) that work together but are controlled by the Security Manager in the backend. The components allow SEMIoTICS to achieve the required functionality in order to:

- provide mechanisms to authenticate users and manage their identities.
- provide mechanisms to manage the identities of other entities, e.g. sensors.
- support use case applications to enforce access to privacy-sensitive information within the application.
- support use case applications to enforce access to privacy-sensitive information when the data is stored in a cloud server, e.g., by using attribute-based encryption and lightweight encryption algorithms.
- provide mechanisms to configure and manage SEMIoTICS end-to-end secure networking capabilities.

All those requirements are covered and managed by one or more of the different software modules of the Security Manager.

TABLE 14 SECURITY MANAGER BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Initialize PEP application	Create a SpringBoot application	Cycle 2	Delivered
Implement a Proxy mechanism in PEP	Implement a Proxy mechanism to intercept HTTP traffic going to the main application and authorize the request in Security Manager	Cycle 2	Delivered
Add a proxy application to authenticate requests	Add mitmproxy application as an Authentication Enforcement Point which adds the client's token to an HTTP request	Cycle 2	Delivered
Prepare PEP for deployment on Backend Orchestrator	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes	Cycle 2	Delivered
Prepare AEP for deployment on Backend Orchestrator	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes	Cycle 2	Delivered
Add a mechanism to configure PEP from a file.	Implement a mechanism that allows configuring mapping between an HTTP request and Security Manager calls	Cycle 3	Delivered
Merge all the existing submodules into one component	Merge all the submodules to one component to simplify the implementation of CI/CD pipeline	Cycle 2	Delivered
Add MongoDB to support Security Manager	Implement MongoDB as a database used by Security Manager to increase the performance of the Security Manager	Cycle 2	Delivered
Implementation of a call to find entities with a visible attribute	Implementation of functionality to find entities with a particular, visible attribute to allow the evaluation of a privacy pattern in Pattern Engine.	Cycle 2	Delivered
Implementation of attribute-based encryption	Implementation of attribute-based encryption and a REST call to generate keys for an entity (based on its attributes or based on its policy)	Cycle 3	Delivered

Integration with Thing Directory	Implementation of calls and methods essential to register new things as soon as they appear available in Security Manager	Cycle 3	In progress
Dockerized Backend Security Manager	Container deployment of the whole Security Manager Architecture	Cycle 3	Delivered
Enable optional cookie support	On top of oauth we can additionally support basic uninitialized sessions.	Cycle 3	Delivered
Security Manager Configuration Service	Node.js based application prototype facilitating the Security Manager API showcases the easy interaction with SecMan functionality, like creating new users; s	Cycle 3	Delivered
Support dynamically added redirect URI's	We enabled a setup in order to support dynamically added redirect URI's provided from the PEP	Cycle 3	Delivered

4.8.1 DEVELOPMENT STATUS

4.8.1.1 SECURITY MANAGER

Within Cycle 3, we have completed the development process for all tasks that were planned there. The deployment of Security Manager's submodules was delivered followed by the extensive testing based on the workflow identified within Use Case 2 on Assisted Living. The integration test with SEMIoTICS Pattern Engine was completed. Moreover, the detailed definition, the implementation and testing on how the Sidecar Proxy subcomponent developed within cycle 1 and interacted with the Security Manager in order to provide the functionality of a Policy Enforcement Point (PEP).

Furthermore, in order to comply with the overall orchestration approach for the backend layer, the Security Manager component in the backend has been finally 100% dockerized. Such an approach allows the easy deployment of the component to the Backend Orchestrator as well as provides the capability of smooth integration with all backend services and exposed APIs.

In order to support an easy way of integrating with the Security Manager, we support on top of oauth an additionally basic uninitialized session logon while initializing a connection. We have devised a configuration file which will allows us to enable or disable this feature.

Another feature to support dynamically redirected URI's was built in. With this, we are able to maintain forcefully given redirect URI's from the policy enforcement point. To empower this setting, it is required to set the redirectURI parameter in the core config file with an asterisk. A complete dynamically enabled redirect configuration looks like the following:

```
{
"id": "AuthEnableRedirect",
"name": "AuthEnableRedirect",
"clientSecret": "Babababanana",
"redirectURI": "*"
},
```

We also implemented a fully working node.js based application prototype, which facilitates the Security Manager API and shows how to integrate and interact with the Security Manager. It can also be used as a fully working configuration service. All the functionality of the API of the Security Manager is mapped there via the GUI and can be easily accessed via the interface. This removes the hurdle of having to write complicated configuration files to use the Security Manager initially.

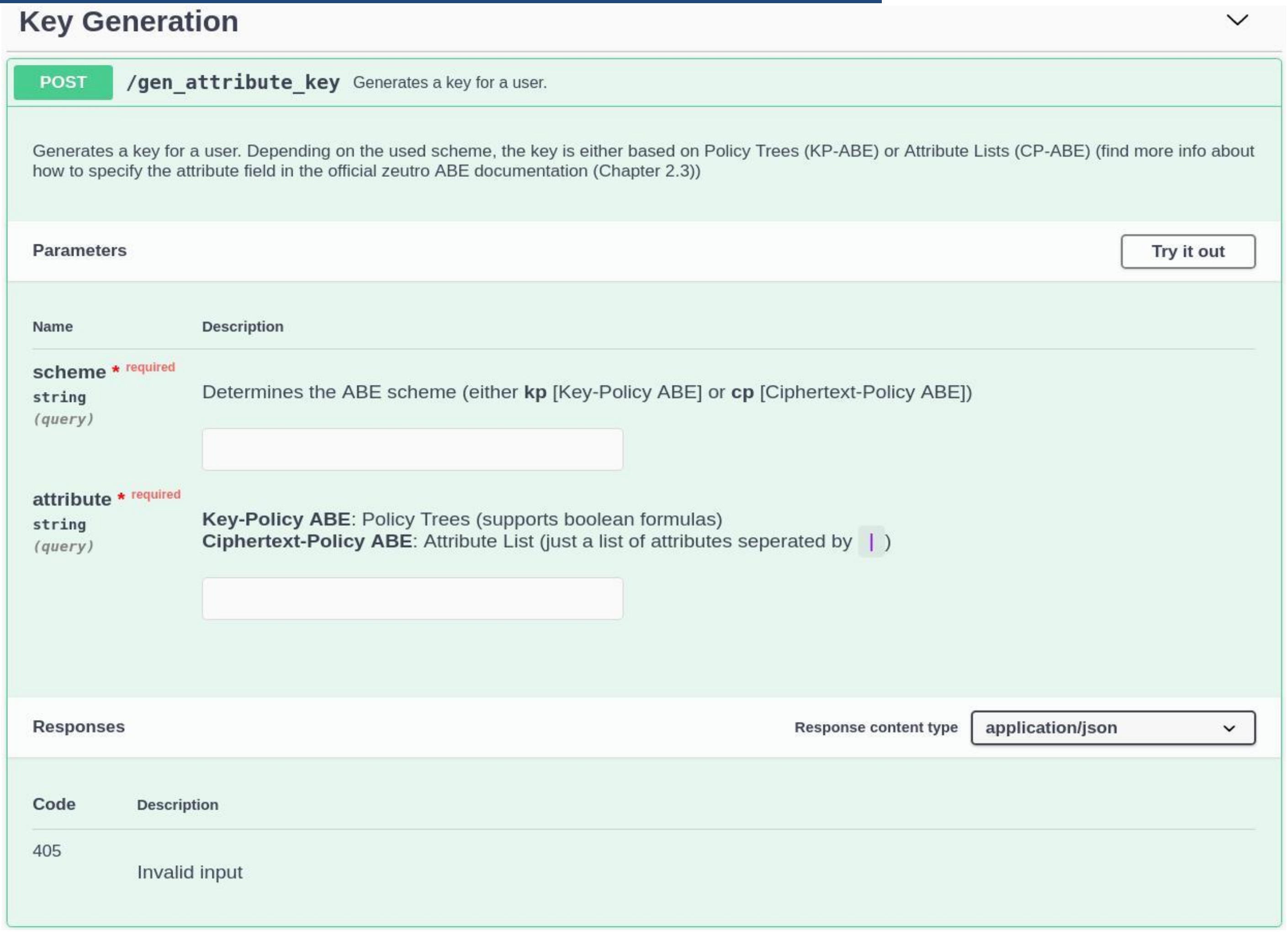
4.8.1.2 ATTRIBUTE BASED ENCRYPTION – REST API

Attribute-Based Encryption (ABE) determines the authorization of a user to decrypt encrypted data based on the user‘s attributes. That means that the decryption of a ciphertext is only possible if the user can present that the user possesses a set of attribute; these attributes are enclosed in the user’s decryption key. Cryptographically the encryption fails unless the decryption keys attributes match the attributes of the ciphertext. This means that the attributes required are encoded during the encryption of the data (for more information we refer to Deliverable 4.12).

A REST API endpoint (Figure 40, Figure 41, Figure 42) was implemented, to make available the needed ABE functionality; the cryptographic functionality is based upon the open source library OpenABE library[7] that provides a variety of attribute-based encryption algorithms. With this API, SEMIoTICS is enabled to seamlessly incorporate ABE technology into the Security Manager. This can then be used where appropriate, to secure information. Also, it ensures that the information can only be accessed by a certain entity or by a group of entities with the requested set of attributes, e.g. only entities with the attribute "doctor" are able to access encrypted medical data.

During Cycle 3, the implementation & testing of the ABE REST API was finalized. This component is an important part for establishing security & privacy for sensitive data in the Use Case 2 demo.

[7] https://github.com/zeutro/openabe



Key Generation

POST `/gen_attribute_key` Generates a key for a user.

Generates a key for a user. Depending on the used scheme, the key is either based on Policy Trees (KP-ABE) or Attribute Lists (CP-ABE) (find more info about how to specify the attribute field in the official zeutro ABE documentation (Chapter 2.3))

Parameters

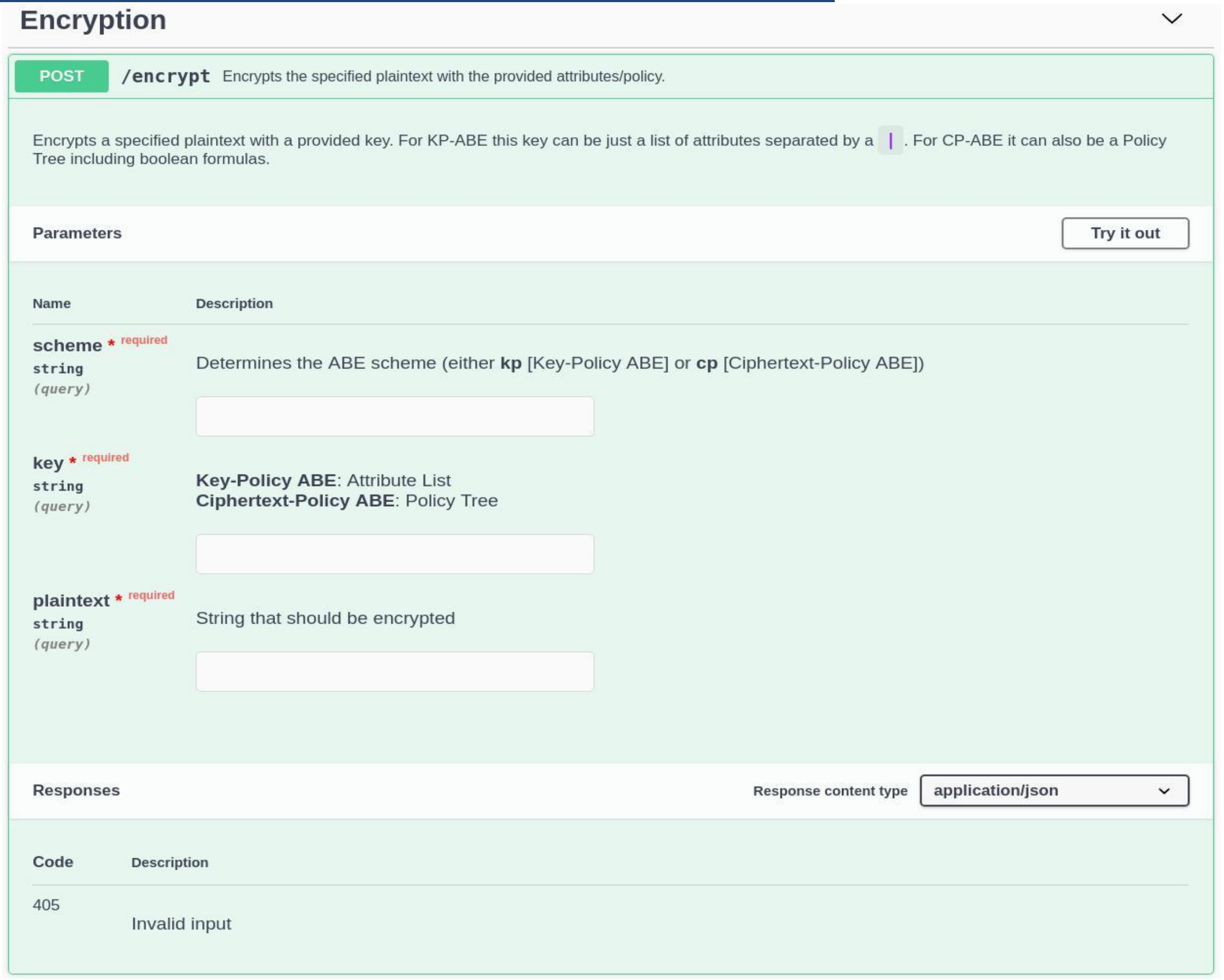
Name	Description
scheme * required `string` *(query)*	Determines the ABE scheme (either **kp** [Key-Policy ABE] or **cp** [Ciphertext-Policy ABE])
attribute * required `string` *(query)*	**Key-Policy ABE**: Policy Trees (supports boolean formulas) **Ciphertext-Policy ABE**: Attribute List (just a list of attributes seperated by `|`)

Responses

Response content type: application/json

Code	Description
405	Invalid input

FIGURE 40 ATTRIBUTE BASED ENCRYPTION – REST API - KEY GENERATION



Encryption

POST /encrypt Encrypts the specified plaintext with the provided attributes/policy.

Encrypts a specified plaintext with a provided key. For KP-ABE this key can be just a list of attributes separated by a `|`. For CP-ABE it can also be a Policy Tree including boolean formulas.

Parameters

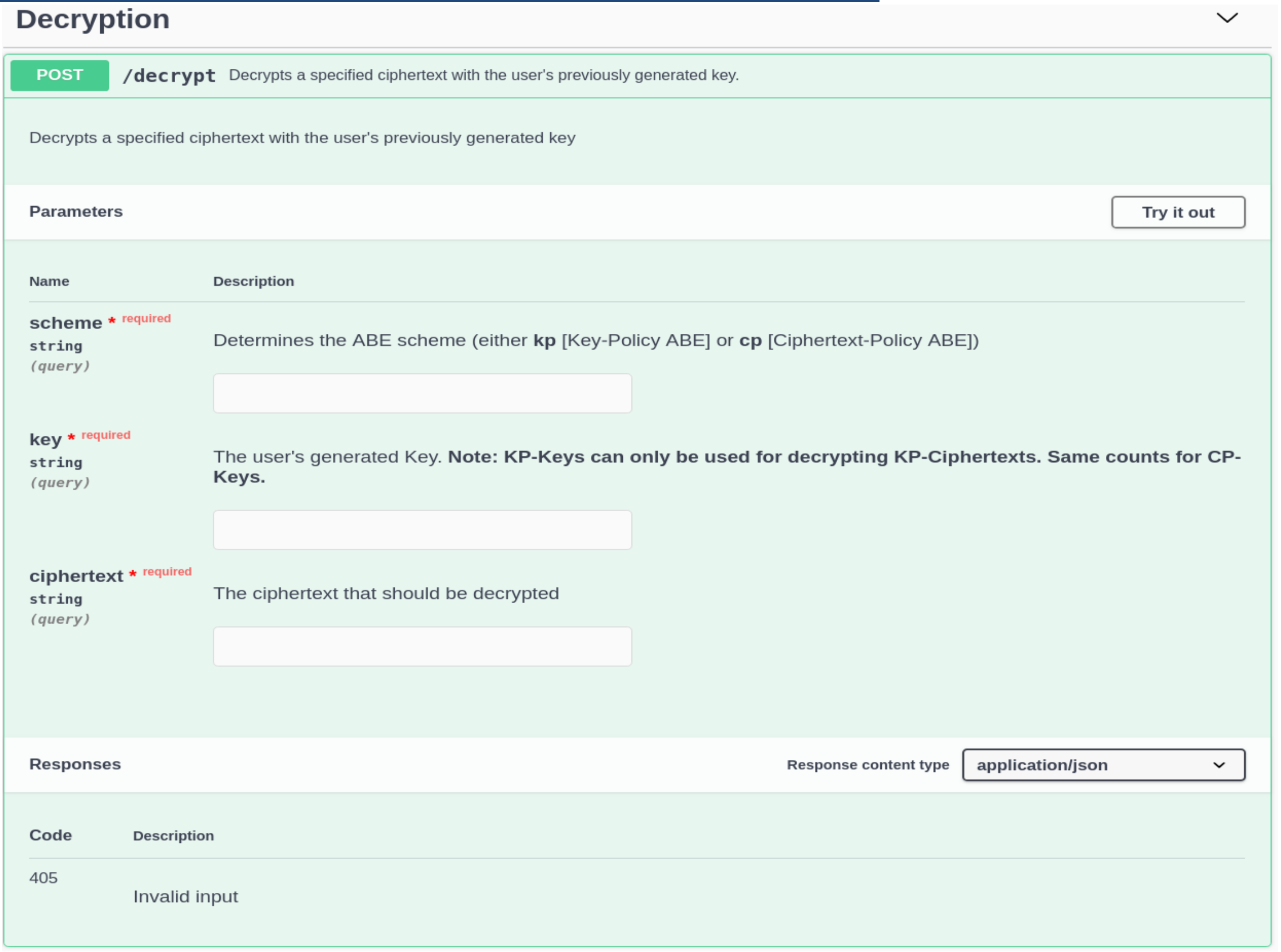
Name	Description
scheme * required string *(query)*	Determines the ABE scheme (either **kp** [Key-Policy ABE] or **cp** [Ciphertext-Policy ABE])
key * required string *(query)*	**Key-Policy ABE**: Attribute List **Ciphertext-Policy ABE**: Policy Tree
plaintext * required string *(query)*	String that should be encrypted

Responses

Response content type: application/json

Code	Description
405	Invalid input

FIGURE 41 ATTRIBUTE BASED ENCRYPTION – REST API - ENCRYPTION


Decryption

POST /decrypt Decrypts a specified ciphertext with the user's previously generated key.

Decrypts a specified ciphertext with the user's previously generated key

Parameters

Try it out

Name	Description
scheme * required string (query)	Determines the ABE scheme (either **kp** [Key-Policy ABE] or **cp** [Ciphertext-Policy ABE])
key * required string (query)	The user's generated Key. **Note: KP-Keys can only be used for decrypting KP-Ciphertexts. Same counts for CP-Keys.**
ciphertext * required string (query)	The ciphertext that should be decrypted

Responses

Response content type: application/json

Code	Description
405	Invalid input

FIGURE 42 ATTRIBUTE BASED ENCRYPTION – REST API - DECRYPTION

4.8.2 COMPONENT API INTERACTONS DESCRIPTION

API	Status	API access type	Additional comments
postBatchEvaluation	Deployed	External	Evaluates whether the authenticated user can excute a set of actions or can read attributes from entity.
getEntityPolicy	Deployed	External	Returns the policy for an entity.
getEntityPolicyOfField	Deployed	External	Returns the policy for a field within a policy structure for an entity.
deleteEntityPolicyOfField	Deployed	External	Returns the resulting policy structure for the entity after deletion.
getToken	Deployed	External	See Authorize button in swagger api
deleteActionEvaluation	Deployed	External	Evaluates whether the authenticated user can delete an action of an entity.

postActionEvaluation	Deployed	External	Evaluates whether the authenticated user can post an action of an entity.
putActionEvaluation	Deployed	External	Evaluates whether the authenticated user can update an action of an entity.
getActionEvaluation	Deployed	External	Evaluates whether the authenticated user can read an action of an entity.
deleteAttributeEvaluation	Deployed	External	Evaluates whether the authenticated user can delete attribute of an entity.
postAttributeEvaluation	Deployed	External	Evaluates whether the authenticated user can update attribute of an entity.
putAttributeEvaluation	Deployed	External	Evaluates whether the authenticated user can update an attribute of an entity.
getAttributeEvaluation	Deployed	External	Evaluates whether the authenticated user can read an attribute of an entity.

4.8.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

The development of the Security manager started in Cycle 2 as planned. This allowed to base the development on the requirements, thus we followed the security-by-design approach.

During cycle 2, the implementation of the Backend Security Manager was started according to the steps of the release circle except for the software module/component that provides the attribute-based encryption (ABE) functionality. The latter, was at this point still under development and in early testing phases. In Cycle 2 we also started checking that the components are generic enough to support all other use cases that are foreseen for the Backend Security Manager.

During cycle 3, the ABE-API was finished and fully tested. It now supports all functionalities (Encryption, Decryption & Key-Generation) for Ciphertext-Policy ABE (CP-ABE) as well as for Key-Policy ABE (KP-ABE). Furthermore, all of the developed components were dockerized, so that the integration to the upcoming Use Cases, specifically Use Case 2, will be much easier.

At the end of cycle 3 we can state that we completely developed and tested the developed the security manager's subcomponents with full API documentation and thus fully offer the planned functionality for Attribute based Encryption (ABE) and Access Control, and Dynamic Policy Management for which the Security Manager in the Backend was designed.

Now that the core development has ended UP will oversee and aid with the integration of the Security Manager in Backend components into the Use Cases, and if necessary update or bug-fix them.

4.8.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO SECURITY MANAGER

SEMIoTICS Requirement		Evaluation	Reference
Req. ID	**Description**		
R.BC.15	Secure communication among the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules)	Generally, the security manager's API is bound to specific ports that can be firewalled. Furthermore, the interfaces of the security manager can be secured by enabling TLS for those services	
R.S.1	The confidentiality of all network communication MUST be protected	SDN connection with ovs switches by enabling SSL, +	D3.7 and D4.12 Section 3.2.2 + Section 3.4.2

	using state-of-the-art mechanisms	Communication between Pattern Orchestrator and Pattern Engines with SSL	
R.P.3	Storage of data MUST be minimized.	This can be technically supported by mongo DB's compression options as well as Level DB's MCPE extension.	Will be evaluated in D5.5
R.P.4	A short data retention period MUST be enforced, and maintaining data for longer than necessary avoided.	In order to provide a mechanism to support a short data retention period we leverage for all current supported databases a TTL parameter. Furthermore, ABE can also be used to make the encrypted information only accessible for a certain timeframe by inserting a timestamp as an additional attribute into the encryption and decryption process.	Will be evaluated in D5.5
R.P.5	As much data as possible MUST be processed at the edge in order to hide data sources and not reveal user related information to adversaries (e.g. user's location).	Replica of Security Manager can be deployed in the field layer, i.e. running on the gateway which is at the edge of the network. Having the replica running at the gateway, there is no need to send requests/answers to/from the backend that might leak information about access being requested or performed. Thus, the replication helps to reduce this attack surface. See D4.12 Section 3.4.1	Will be evaluated in D5.5
R.P.8	Data MUST be stored in encrypted form.	Attribute Based Encryption (ABE)	Will be evaluated in D5.5

		allows to encrypt data before it is stored such that the protected data can only be decrypted by authorised users that posses the required attributes. ABE is described in Section 4.8.1.2 and D4.12 Section 3.2.1.4 can be of aid	
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised.	Security Manager supports different log levels.	Will be evaluated in D5.5
R.GSP.9	The SARA system SHALL provide robust mechanisms to protect Patient-related data.	ABE is used for encryption but with our library we can also facilitate access control with it. ABE as described in Section 4.8.1.2 and D4.12 Section 3.2.1.4 can be of aid	Will be evaluated in D5.5

4.9 Local Embedded Intelligence (Field Layer)

The Local Embedded Intelligence Component in SEMIoTICS aims to provide a logical interface for exposing to the SEMIoTICS ecosystem the complete set of analytics algorithms developed within the project and described in D4.10 “Embedded Intelligence and Local Analytics (final draft)”. These algorithms are the major enablers of the edge computing algorithms supported in the SEMIoTICS project. In particular, they could be subdivided into some further category according to the intended main usage scenario.
The 1st set of algorithms enables the gait analysis on the SARA Healthcare scenario (i.e. UC2), whereas the 2nd set of algorithms will support the Smart sensing use case (Generic IoT horizontal) that will be demonstrated mainly in UC3 final demo. The need for a coherent integration logic driven by SEMIoTICS SPDI pattern approach with other SEMIoTICS components is enforced by the fact that these algorithms will be deployed on different types of field devices, with different legacy middleware constraints. As an example, the role of smart sensing units, within UC3, is played by small microcontroller (smart sensing) units tightly coupled with miniaturized environmental/inertial sensors. Due to the heterogeneous set of available devices, and also very heterogeneous set algorithms available, an integration methodology has been identified and designed in

SEMIoTICS as a viable solution for exposing the results of these algorithms in a coherent manner within each specific UC need / requirement. This methodology implies the definition of an abstract interface, used to wrap conveniently the heterogeneous set of algorithms developed within Task 4.3 activities in order to make their outputs available in the field device level of SEMIoTICS. In this deliverable an update related to the status of this aspect within SEMIoTICS will be provided according to what has been already reported in D4.10 Local Embedded Analytics (final draft) where the complete set of algorithms has been fully characterized. The outputs of the local analytics algorithms are described as event messages that are sent to the SEMIoTICS field level infrastructure in a semantically interoperable manner by instantiating specific scenario driven patterns through the some of the components available in the SEMIoTICS field level infrastructure (i.e. the local pattern engine, the Semantic GW mediator and the Semantic Edge Platform). As an example, the outcomes (i.e. anomalies) reported by the analytics/machine learning algorithms on the Generic UC3 IoT scenario are reported to SEMIoTICS field level at a bottom side as timestamped events through a dedicated JSON protocol published over a MQTT infrastructure. The final integration of the component is planned for the Cycle 3 final iteration where those events will be integrated and notified by translated them into dedicated patterns thanks to the pattern engine. Similar integration logic is currently under development on all the three main scenarios under consideration as part of WP5 Task 5.4 to Task 5.6 activities.
In the following Table, a summary of the implementation tasks is presented detailing Cycle 2 and updated Cycle 3 implementation plans.

TABLE 15 LOCAL EMBEDDED INTELLIGENCE BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
UC3 Generic IoT Local Analytics Algorithms	UC3 Local Embedded Component wrapper deployed on ST X-Nucleo Microcontroller equipped with MQTT Client. The component provides MQTT events regarding anomalies on inertial or environmental real-time acquired data.	Cycle 2	Delivered – see D4.10
UC3 Generic IoT Local Analytics Algorithms	UC3 Local Embedded Component MQTT events mapping to dedicated patterns in pattern engine component	Cycle 3	Ongoing deployment into T5.6
UC3 Generic IoT Local GTW Supervisor Service	UC3 Local GTW Supervisor Service	Cycle 3	Delivered – see D4.10
UC3 Generic IoT Local Analytics Algorithms	UC3 Local Event DB component development	Cycle 3	Delivered – see D4.10
UC3 Generic IoT Local Analytics Algorithms	UC3 Local Event DB component integration	Cycle 3	Ongoing deployment into T5.6
Gait Analysis Local Analytics Algorithms	The algorithms are under active development. Wrapping component implementation will be started on the cycle 3 period.	Cycle 3	Characterized and defined – see D4.10

4.9.1 DEVELOPMENT STATUS

Current development status of the local embedded functionality in SEMIoTICS is reported in Table 15. In particular, during Cycle 3 integration the majority of functionalities of the local embedded analytics has been implemented through dedicated algorithms mapped to specific field layer components according to the specific use case scenario. All the algorithms has been identified and characterized (please refer to D4.10 for a complete detailed presentation) and currently their deployment and component mapping into the SEMIoTICS architecture is taking place as part to WP5 integration activities within each use case (i.e. from task 5.4 to task 5.6 activities).

4.9.2 COMPONENT API INTERACTIONS DESCRIPTION

An example of interaction APIs implemented and documented in D4.10 is shown in Table 15. It provides an abstract interaction interface that is used by some SEMIoTICS FL components deployed at the IoT gateway (i.e. the pattern engine, the IoT Generic Supervisor Service component, etc.) to implement semantic bridging vs use case specific UC3 patterns. Each of those components have a built-in MQTT client able to interact with the Local Embedded Analytics deployed into the Sensing Unit mapped onto the STM32 Microcontroller HW board.

API	Status	API access type	Additional comments
pushRawData	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Push raw sensor data readings (publish)
getRawData	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Get raw sensor data readings (subscribe)
getMsgEvent	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Get analytics results / events (subscribe)
pushMsgEvent	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Push analytics results / events (publish)
reset	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Node device SW reset event (publish)
reconfigure	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Node device reconfigure event (publish). See R.UC3.5 on section 4.9.4
pushKeepAlive	Implemented	Internal (MQTT)	As JSON payload message reported in D4.10. Node keepalive message to detects faulty units and implements dependability pattern in UC3

4.9.3 COMPONENT DEVELOPMENT SUMMARY AFTER ALL CYCLES

The majority of the SEMIoTICS components needed by the local embedded analytics have been developed and are currently moving to the integration / deployment phase as part of WP5 activities. Three specific components have been specifically developed for the UC3 Generic IoT scenario in order to accomplish the task of providing smart artificial intelligence enable sensing devices units, whereas for other use case scenario the local embedded intelligence have been designed to interact at the IoT gateway level with existing components hereby deployed (i.e. the pattern engine, the Semantic Edge platform, the local thing directory, etc.)

4.9.4 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO LOCAL EMBEDDED INTELLIGENCE

The following table summarizes the complete set of requirements considered during the design and implementation of the local embedded analytics component in SEMIoTICS:

SEMIoTICS Requirement		Local Analytics considerations	Reference
Req. ID	Description		
R.GP.2	Scalable infrastructure due to the fast-paced growth of IoT devices	This is a set of generic requirements in SEMIoTICS that motivates the need for layered distributed intelligence in the project integrated into the platform exploiting the SPDI pattern approach.	Patterns has been defined and characterized in SEMIoTICS in many WP3 deliverables, i.e. in task 3.3 (Patterns), task 3.4 (Network-level semantic Interoperability) and task 3.5 (Implementation of Field-
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern		

R.NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities		level middleware & networking toolbox)
R.FD.1	Field devices SHOULD be able to get data from the environment through sensors (sensors).	This is a set of specific field layer requirements, common to all use case scenarios that affects the definition of the local embedded analytics component.	Local embedded components (i.e. the algorithms used in SEMIoTICS) has been mainly identified during task 4.3 activities in two submitted deliverables. Communication design, interoperability aspects and deployment has been considered in a larger scope within WP4 activities (i.e. task 4.1 for SPDI patterns, task 4.4 for End-to-End Semantic Interoperability and finally in task 4.6 for the interfacing APIs) and for integration / deployment aspects in WP5 as part of the use case specific demonstrators (i.e. form T5.4 to task 5.6)
R.FD.2	Field devices SHOULD be able to process data in near real time (process units).		
R.FD.4	Field devices SHOULD use a global clock for time synchronization.		
R.FD.5	Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components		
R.FD.6	Field devices MUST interoperate using a standard communication protocol like Rest APIs, COAP, MQTT.		
R.FD.7	Field devices MUST use standardize interoperable message format (e.g. JSON, etc.).		
R.FD.9	Field devices MUST be able to communicate with the IIoT Gateway / other architectural components.		
R.FD.10	Field devices SHOULD minimize data traffic.		
R.FD.11	Field devices SHOULD minimize energy consumption.		
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).		
R.FD.14	The field layer must feature SPDI pattern reasoning local embedded intelligence capabilities		
R.FD.15	The field layer must aggregate intra-layer information to enable local intelligence reasoning and adaptation		

R.P.3	Storage of data MUST be minimized.	This is a set of specific privacy requirements, common to all use case scenarios where the local embedded analytics could support through its distributed local data processing the concept of data retention and data anonymization.	In D4.10 all algorithms used in SEMIoTICS for the local embedded analytics has been presented. In this deliverable it is presented for each specific use case under consideration the impact on the architecture where the benefits of the data anonymization and customizable data retention are shortly discussed.
R.P.4	A short data retention period MUST be enforced and maintaining data for longer than necessary avoided		
R.P.5	As much data as possible MUST be processed at the edge in order to hide data sources and not reveal user related information to adversaries (e.g. user's location).		
R.P.6	Data MUST be anonymized wherever possible by removing the personally identifiable information in order to decrease the risk of unintended disclosure		
R.GSP.3	IoT gateway SHALL be able to estimate abnormal detection based on (un)-supervised model.		
R.UC1.10	Local analytical capability of IIoT Gateway to run machine learning algorithms (e.g. specific to 2 specific sub-use cases)	This is the complete list of use case specific requirements that has been considered for the development of the local analytics components in the main three use case scenarios in SEMIoTICS. They have been used to identify / develop / characterize and later on deploy all the embedded local analytics algorithms in SEMIoTICS	Please refer to D4.10 for an in-depth discussion of all the local embedded analytics algorithms developed within SEMIoTICS project.
R.UC2.6	The SEMIoTICS platform SHOULD allow the SARA solution to retrieve the resources exposed by registered devices via their object model (i.e. a data structure wherein each element represents a resource, or a group of resources, belonging to a device). The SEMIoTICS platform SHOULD support at least the OMA LWM2M object model.		
R.UC3.1	IoT Sensing unit shall be able to embed environmental (e.g. temperature, pressure, humidity, light) and inertial sensors (accelerometer, gyroscope).		
R.UC3.2	IIoT Sensing unit shall be able to interface to the IIoT Sensing gateway in order to coordinate with it. A standard IP based (i.e. TCP transport) 1 to many M2M communication protocol		

	must be adopted to properly handle node communication with components in the gateway.		
R.UC3.3	IIoT Sensing unit shall be able to learn a model from observed data in an unsupervised manner. In particular IoT Sensing unit shall be equipped with a low power (tens/hundreds of mW range) 32 bits MCU to support unsupervised learning and unsupervised statistical processing.		
R.UC3.4	IIoT Sensing unit shall be able to detect relevant changes from the learned model and report them to IIoT Sensing gateway.		
R.UC3.5	IIoT Sensing unit shall be able to adapt to a new model if IIoT sensing gateway requires this.		
R.UC3.6	IIoT Sensing gateway shall be able to coordinate a set of IIoT sensing units by finding any correlation btw them according to observed data, models		
R.UC3.7	IIoT Sensing gateway shall be able aggregate relevant events (i.e. changes) coming from whichever of connected IIoT sensing units deciding if they are global or local changes		
R.UC3.8	IIoT Sensing gateway may have the capability to exchange relevant information (i.e. events) between itself, the cloud and the sensing units with some connectivity capabilities		
R.UC3.11	IoT Sensing unit shall be able to run Artificial neural networks on the MCU in real time at the sensor data rate of choice.		
R.UC3.12	IoT Sensing unit shall be able to run lightweight statistical model analysis algorithms on the MCU not in real time at the sensor data rate of choice.		
R.UC3.14	MCU IoT Sensing unit shall be able to run neural network online training at the sensor data rate of choice.		

R.UC3.15	IoT Sensing gateway shall support 1 to many standard IP based (i.e. TCP transport) M2M communication protocol to interface a number N of connecting Sensing units (e.g. broadcast type).		
R.UC3.19	IoT Sensing gateway should be able to support http and standard protocols for cloud interfacing.		
R.UC3.20	The specific M2M protocol adopted on UC3 is based on MQTT. A MQTT broker service will be available to dispatch messages between the coordinating Sensing gateway and its associated Sensing units.		
R.UC3.21	A use case specific serialized message protocol is required to coordinate the gateway and its associated units and exchange data / events / anomalies between them. JSON will be the preferred serialization format adopted.		
R.UC3.22	Each connected IHES sensing unit should send to the gateway a keep alive signal on a specified period (e.g. few seconds) to notify the gateway it is correctly working. The sensing gateway should detect by this mean any non-working sensing unit and reconfigure the system accordingly.		
R.UC3.23	Sensing units and sensing gateway should share a common clock (i.e. global reference time), precise up to milliseconds, to properly classify events and data acquired during the processing. This global reference time will be negotiated when a sensing unit node will join a given gateway. Internally the system will work scheduling activities according to this global reference time.		
R.UC3.24	Sensing units may be equipped with dedicated FW to detect relevant sensors		

	malfunctioning and report that to the gateway		

4.10 Monitoring

The objectives of the SEMIoTICS Monitoring component are twofold:

- To generate specific messages in response to the reception of a set of messages generated by the components of an IoT application and matching some condition specified in the monitoring component by a client application (Monitoring requirement).
- To guarantee that the messages needed to decide whether to generate a message can be produced by an IoT application and received by the monitoring component (Observability property).
- The project's deliverable D4.9 - "SEMIoTICS Monitoring, Prediction and Diagnosis Mechanisms (final)" presents the final design of the monitoring, prediction and diagnosis mechanisms in SEMIoTICS along with algorithmic and technological options choose for the implementation of its key functionalities.

Table 22 presents the identified backlog scope and assignment to development cycles planned.

TABLE 22 MONITORING COMPONENT BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
sem-mdp-api	Create a library for Monitoring API	Cycle 2	Delivered
sem-mdp-controller	The first version of the Monitoring Controller	Cycle 2	Delivered
sem-mdp-web	Bundle making available controller as a REST service	Cycle 2	Delivered
sem-mdp-cep-flink	Flink-based implementation of the Complex Event Processor (CEP) (replanning of the delivery was needed due to the revision process of other deliverables)	Cycle 2/3	Delivered
sem-mdp-signaller-wot	Event Signaller for WoT (Web of Things) devices (replanning of the delivery was needed due to the revision process of other deliverables)	Cycle 2/3	Delivered
sem-mdp-signaller-fiware	The FIWARE Signaler implements the *Signaler* interface offering the operations to read, write or subscribe attributes of NGSIv2 entities.	Cycle 3	Delivered
sem-mdp-signaller-network	The network signaler is responsible for monitoring the status of the network topology of the SEMIoTICS use cases.	Cycle 3	Delivered

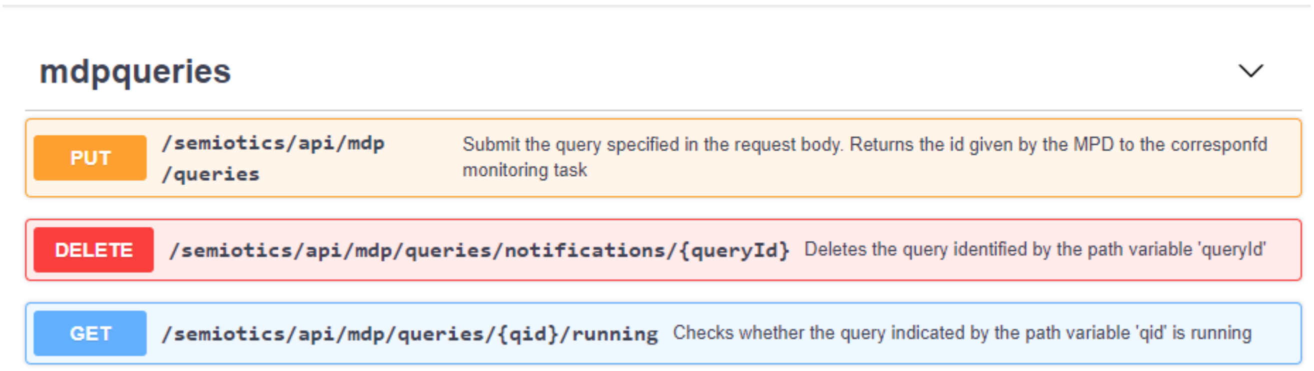
sem-mdp-signaller-kubernetes	The Backend Orchestrator exposes API to get the events as they have occurred in the Kubernetes cluster.	Cycle 3	Delivered
sem-mdp-cmi	Causal Model Identifier has the role to build the causal models. These models are created using as input both the (Re)configuration commands emitted by the Monitoring Controller and the events generated by the Business Event Monitor	Cycle 3	Delivered
sem-mdp-epredictor	The Event Predictor uses to Causal Model learned by the Causal Model Identifier to infer events not directly observable through the Events Signalers	Cycle 3	Delivered
sem-mdp-disgnosis-gui	Visualization for the diagnosis	Cycle 3	Delivered
sem-mdp-storgae	Storage of High-Level events generated by an implementation of the Complex Event Processor (i.e. one of the sem-mdp-cep-* components)	Cycle 2	Delivered

4.10.1 DEVELOPMENT STATUS

During cycle 3 of development the following modules were delivered:

- sem-mdp-cep-flink
- sem-mdp-signaller-wot
- sem-mdp-signaller-fiware
- sem-mdp-signaller-network
- sem-mdp-signaller-kubernetes
- sem-mdp-cmi
- sem-mdp-epredictor
- sem-mdp-disgnosis-gui

4.10.2 COMPONENT API INTERACTIONS DESCRIPTION


mdpqueries

PUT `/semiotics/api/mdp/queries` Submit the query specified in the request body. Returns the id given by the MPD to the corresponfd monitoring task

DELETE `/semiotics/api/mdp/queries/notifications/{queryId}` Deletes the query identified by the path variable 'queryId'

GET `/semiotics/api/mdp/queries/{qid}/running` Checks whether the query indicated by the path variable 'qid' is running

API	Status	API access type	Additional comments
submitQuery	Delivered	External	-
cancelQuery	Delivered	External	-
checkQueryStatus	Delivered	External	-

4.10.3 SEMIOTICS REQUIREMENTS IMPLEMENTATION MAPPED TO MONITORING

SECTION 1.1 OF DELIVERABLE D4.9 - "SEMIoTICS Monitoring, prediction and diagnosis mechanisms (FINAL)" shows which are the SEMIoTICS Requirements addressed by the Monitoring component as detailed in the following Table.

TABLE 16 REQUIREMENTS ADDRESSED BY THE MPD

SEMIoTICS Requirement		Evaluation	Reference
Req. ID	Description		
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern	The MPD allows detecting Network level events thanks to the availability of adapters able to capture the events generated by the SDN Controller and Virtual Infrastructure Manager (VIM).	Section 2.2.7 in D4.9
R.P.4	A short data retention period MUST be enforced, and maintaining data for longer than necessary avoided.	The MPD uses Complex Event Processing technology to aggregate data. In fact, CEP technology allows detecting events patterns directly in the stream of events without the need to store the events in a database for subsequent processing.	Section 2.3 in D4.9
R9.4	The cloud platform SHALL to be able to monitor the execution of an app, in particular its interactions with other apps, the network interface, and APIs.	The MPD provides adapters that enable to monitor the execution of apps by means of the native monitoring capabilities of Cloud and IoT platforms	Section 2.2 in D4.9
R.BC.20	The backend layer must aggregate intra-layer as well as inter-layer SPDI status information to enable local and global intelligence reasoning and adaptation.	The MPD provides adapters to capture events generated by the backend layer. The MPD aggregates events using CEP technology. MPD defines strategies to translate SPDI pattern into monitoring policies.	Section 1.2, Section 2.5, Section 2.6 in D4.9
R.NL.13	The network layer must aggregate intra-layer monitored information to enable local intelligence reasoning and adaptation.	The MPD provides adapters to capture events generated by network layer. The MPD aggregates events using CEP technology. MPD defines strategies to translate SPDI patterns into monitoring policies.	Section 1.2, Section 2.5, Section 2.6 in D4.9
R.FD.15	The field layer must aggregate intra-layer monitored information to enable local intelligence	The MPD provides adapters to capture events generated by field devices. The MPD aggregates events using CEP technology. MPD defines	Section 1.2, Section 2.5, Section 2.6 in D4.9

	reasoning and adaptation.	strategies to translate SPDI pattern into monitoring policies.	
R.UC2.10	The SEMIoTICS platform SHOULD allow the SARA components (e.g. SARA Hubs) to query and aggregate (e.g. to average) the values of a resource (e.g. current measured temperature) hosted by a group of field devices. The SARA solution defines a group of devices by specifying filtering criteria over the set of registered devices.	The MPD provides adapters to capture events generated by field devices. The Query language of the MPD provides means to express filtering conditions over the sources of events.	Section 2.2.8, Section 2.5 in D4.9
R.UC2.12	The SEMIoTICS platform SHOULD allow SARA components to delegate to the platform the computation of complex functions over the data received by field devices. These computations may result either in the generation of higher-level observation events (e.g. significant Patient events abstracted form sensor data) towards the ACS or in sensors configuration parameters (including actuators command).	The MPD provides adapters to capture events generated by field devices. Moreover, The Query language of the MPD provide business IoT applications (e.g. SARA) with means to specify a high-level observation event as the occurrence of a specific pattern of events within the stream of events generated by field devices.	Section 2.2.8, Section 2.5 in D4.9

5 VALIDATION

This Section describes the validation features of SEMIoTICS that are related to the implementation of backend components and the rest topics that are presented in this document.

5.1 Related Project Objectives and Key Performance Indicators (KPIs)

Table 17 presents the task objectives and appropriate sections addressing those while Table 18 presents the KPI's objective which is relevant for Task 4.6.

TABLE 17 TASK'S OBJECTIVES

T4.6 Objectives	D4.13 Sections
• Implementation of the algorithms, techniques, and components in Tasks 4.1-4.5 and the delivery of an API giving access to them.	4.2, 4.3, 4.4, 4.8, 4.9, 4.10
• Providing IoT components communication across layers and integration with external systems and partners.	4.2
• Receiving messages from sensors and resource provisioning as a result of analytics computing.	4.1
• Implementation of appropriate security levels for each connection type, in order to ensure the coherence of data and minimal latency in data transmission.	4.2, 4.8
• Using semantic communication metadata to enable negotiation and interoperability between components.	4.5, 4.8
• Registration of SPDI pattern, which will include the SPDI patterns known to the infrastructure and their currently deployed instances in the IoT applications managed by the infrastructure.	4.3, 4.4, 4.6, 4.7
• Dashboard providing administrators of such applications with access to runtime IoT application management information.	4.2
• Component supporting different types of horizontal and vertical runtime of proactive and reactive adaptation.	4.2, 4.3, 4.9, 4.10

Because task 4.6 is closely related to Tasks 4.1-4.5 and provides an implementation of the algorithms, techniques, and components described in these tasks, hence is correlated with the project's requirements from the entire WP4. The **KPI's objectives** for T4.6 are presented below:

TABLE 18 KPI'S AND OBJECTIVES

Objective		KPI-ID	Description	Related task
1	SPDI Patterns	KPI-1.1	Number of SPDI Patterns	T4.1
1	SPDI Patterns	KPI-1.2	Deployment of a multi-domain SDN orchestrator	T4.1
2	Semantic Interoperability	KPI-2.1	Semantic descriptions for 6 types of smart objects	T4.1,T4.4
2	Semantic Interoperability	KPI-2.2	Data type mapping and ontology alignment	T4.4
2	Semantic Interoperability	KPI-2.3	Semantic interoperability with 3 IoT platforms	T4.4
3	Monitoring Mechanisms	KPI-3.1.1	Generating monitoring strategies in the 3 targeted IoT platforms	T4.1, T4.2

3	Monitoring Mechanisms	KPI-3.1.2	Fuse results from these monitors	T4.1, T4.2
3	Monitoring Mechanisms	KPI-3.1.3	Performing predictive monitoring with an average accuracy of 80%	T4.1, T4.2
3	Monitoring Mechanisms	KPI-3.2	Delivery of a monitoring language	T4.1, T4.2
4	Multi-layered Embedded Intelligence	KPI-4.1	Delivery of lightweight ML algorithms	T4.3
4	Multi-layered Embedded Intelligence	KPI-4.2	Delivery of mechanisms with adaptation time of 15ms	T4.1, T4.2, T4.3
4	Multi-layered Embedded Intelligence	KPI-4.3	Delivery of adaptations mechanisms enabling improvement by at least 20%	T4.2, T4.3
4	Multi-layered Embedded Intelligence	KPI-4.4	Detection time of less than 10 ms	T4.3
4	Multi-layered Embedded Intelligence	KPI-4.6	Development of new security mechanisms/controls	T4.1, T4.5
5	IoT-aware Programmable Networks	KPI-5.2	Service Function Chaining (SFC) of a minimum 3 VNFs	T4.1
6	Development of a Reference Prototype	KPI-6.1	Reduce Required Manual Interventions	T4.1
6	Development of a Reference Prototype	KPI-6.3	Delivery of 3 prototypes of IIoT/IoT applications	T4.6

5.2 SEMIoTICS implementation requirements

The general SEMIoTICS' requirements that are covered by the presented implementation of SEMIoTICS components are summarized in the next table.
For the sake of easier readability, here we present only the requirements directly related to Task 4.6 and logical components belonging only to this task, while all requirements related to Tasks 4.1 to T4.5 are presented in respective deliverables. The full scope of requirements mapping is available in D2.4

TABLE 19 TASK'S REQUIREMENTS

Requirements	Description	Related task	Status
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	T4.6	Delivered
R.GP.2	Scalable infrastructure due to the fast-paced growth of IoT devices	T4.6	Delivered
R.BC.15	Secure communication among the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules)	T4.6	Delivered
R.P.1	The collection of raw data MUST be minimized.	T4.6	Delivered
R.P.2	The data volume that is collected or requested by an IoT application MUST be	T4.6	Delivered

	minimized (e.g. minimize sampling rate, amount of data, recording duration, different parameters).		
R.P.3	Storage of data MUST be minimized.	T4.6	Delivered
R.P.4	A short data retention period MUST be enforced and maintaining data for longer than necessary avoided.	T4.6	In progress
R.P.9	Repeated querying for specific data by applications, services, or users that are not intended to act in this manner SHALL be blocked.	T4.6	In progress

6 CONCLUSION

Within this deliverable, the details of the WP4 developed components of the final cycle of implementation Task 4.6 are presented. The progress of work advancement has been tracked using GitLab, which is the main code repository of the development monitoring and tracking. Based on the open issues tracked in Gitlab, weekly technical meetings have been held for the status and any risk tracking.

All work delivered within cycle 3 has been focusing on the variety of key aspects of SEMIoTICS. The development, distributed across involved partners, was delivered separately while the integration part has been reserved for the future cycle and mainly for the WP5. Planning and implementation of cycle 3 have been performed within five subjective streams as follows:

- The first workstream is focusing on SPDI patterns, going from Recipe Cooker where the distributed execution of recipes was developed. Moreover, storing the patterns in the backend repository of Pattern Engine has been delivered along with the classification and distribution of the patterns from Pattern Orchestrator to Pattern Engines. Finally, the visualization of patterns in the SEMIoTICS platform has been delivered within the GUI component.
- Within the second workstream, the effort has been put into the delivery of semantic interoperability. Communication between Recipe Cooker and BSV has been established successfully. The BSV's endpoints were reimplemented using RESTful services instead of gRPC and the work on resolving semantic conflicts using the Adaptor Nodes has been started and was continued in cycle 3.
- The third workstream was focusing on the security aspects. PEP, AEP and Proxy mechanisms.
- The fourth workstream has been focusing on the Backend Orchestrator implementation and proper configuration along with further development of one central GUI for user interaction with the framework.
- The last workstream was focusing on the monitoring and local embedded intelligence aspects. The Monitoring component has identified two interfaces (Query API and Storage API) and 3 possible domains of queries: domain-specific, security-related and self-monitoring. Local embedded intelligence efforts have been focusing on the generic local IoT analytic algorithms.

According to the description provided in Section 3, Task 4.6 delivers the implementation of components defined within WP4, the backend API and the integration of the respective components that are also related to the outputs of the tasks as depicted in Figure 4. The outcome of the task T4.6 are deliverables D4.6 (presented in June 2019), D4.7 (presented December 2019) and D4.13 (the outcome of cycle 3 development). Deliverable D4.13 has provided development status for Graphical User Interface, Backend orchestrator, Patter Orchestrator, Pattern Engine (backend), Backend Semantic Validator, Thing Directory, Recipe Cooker, Security Manager (backend), Local Embedded Intelligence and Monitoring.

Deliverable D4.13 has covered the finalization of the development of all components involved within WP4. While the interaction between all the architectural components is defined within D2.5 (Deliverable 2.5 “SEMIoTICS high-level architecture (final)”), the detailed specifications of the API area partially the outcome of D4.7 (cycle 2) and D4.13 (Final Cycle) development. Following those 3 cycles of development, the SEMIoTICS has reached its development maturity within the delivery of cycle 3 (final).