SEMIoTICS
Deliverable D4.7
Implementation of Backend API (Cycle 2)

Deliverable release date	31/12/2019
Authors	1. Arne Broering (SAG), 2. Eftychia Lakka, Emmanouil Michalodimitrakis (FORTH), 3. Konstantinos Fysarakis, Iasonas Somarakis, Michail Smyrlis (STS) 4. Piotr Kowalski, Michał Rubaj, Urszula Stawicka (BS) 5. Felix Klement, Korbinian Spielvogel, Henrich C. Pöhls (UP)
Responsible person	Łukasz Ciechomski (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, Verikoukis Christos) 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	1.0
Dissemination level	Confidential

Table of Contents

Acronyms Table	
Acronym	**Definition**
API	Application Programming Interface
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
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
UC	Use Case
UML	Unified Modeling 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

The project 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 in SEMIoTICS is covering not only the implementation of the necessary algorithms, techniques, and components but also the delivery of an API set giving access to them.

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

From an implementation perspective, the current implementation Cycle 2 (due M23) together with Cycle 1 (delivered M17) and Cycle 3 (final, due M30) are providing 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 the Action, this API will also provide IoT components communication across layers and integration with external systems and partners.

Based on the above, Deliverable 4.7 “Implementation of SEMIoTICS Backend API (Cycle 2)” presented herein, is the output of T4.6 (Implementation of SEMIoTICS backend API), provides the status of the second implementation cycle, describes the implementation approach and re-iterates which backend architectural components (see D2.4) are to be developed in which SEMIoTICS development cycle, focusing on the initial algorithms, techniques, and components specified in D4.1 to D4.5 and the APIs for accessing them.

In more detail, the deliverable D4.7 is structured as follows:

- Section 2 describes the SEMIoTICS implementation approach.
- Section 3 establishes the backend architectural components need to be developed in SEMIoTICS development cycle. Due to slight architectural modifications and project plan changes, the plan of development cycles has been slightly modified in order to reflect the current status.
- Section 4 covers the development status of Cycle 2 and describes the progress of development of each component delivered within cycle 2 in dedicated subsections.
- Section 5 positions the work presented herein in reference to the project’s Objectives and associated KPIs and requirements
- Section 6 features the concluding remarks and pointers to future work and ensuing deliverables.

1.1 PERT chart of SEMIoTICS

Figure 1 below presents the PERT chart of the project, visualizing the links and relationships between the WPs and Tasks. Please note that the PERT chart is kept on task level for better readability.

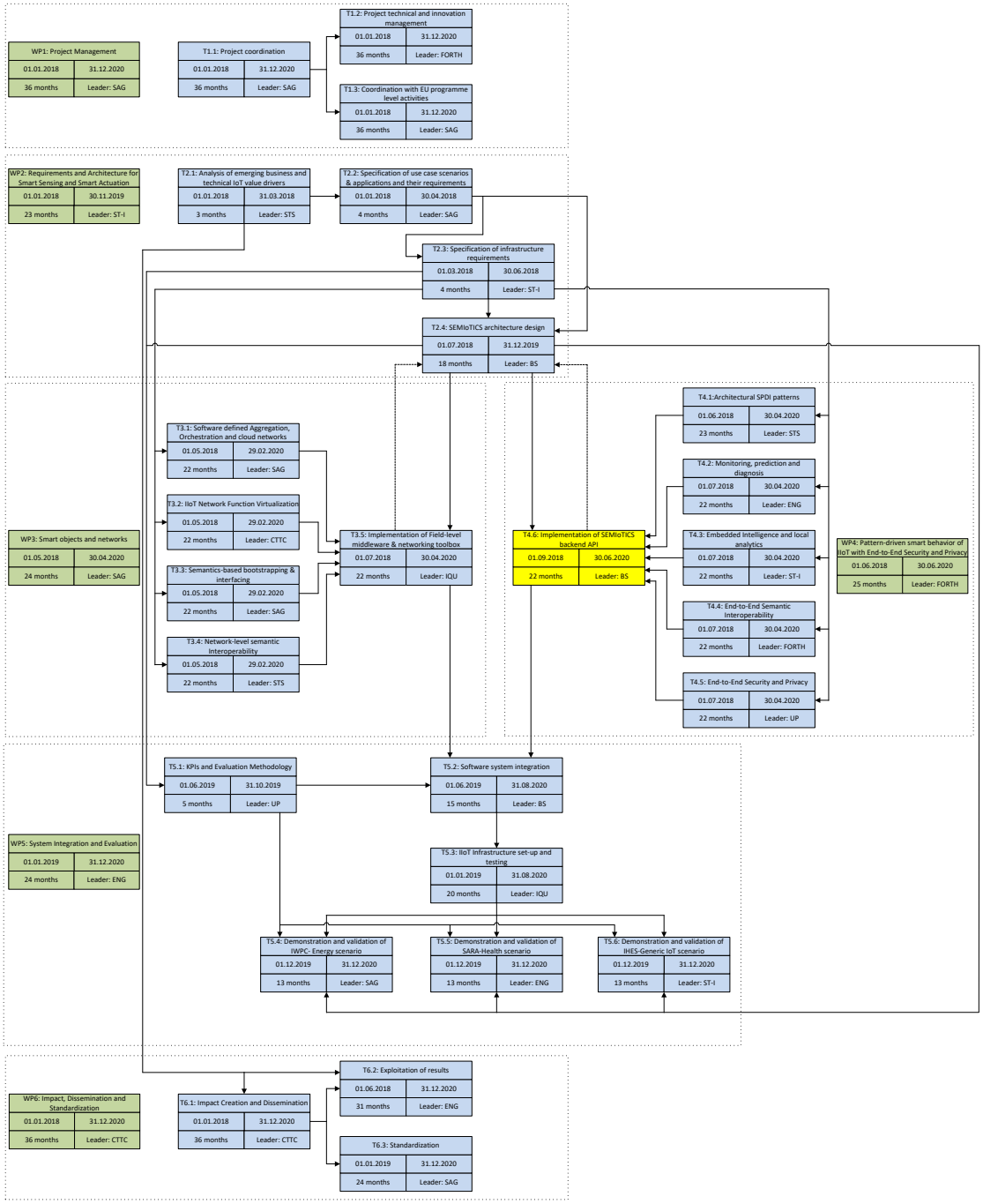

FIGURE 1 PERT CHART

2 IMPLEMENTATION APPROACH

In SEMIoTICS, Task 4.6 is the main implementation task of WP4, which focuses on delivering the SEMIoTICS components specified in the context of WP4's previous tasks (Task 4.1 to 4.5), following incremental release cycles. Implementation has been divided into 3 development cycles as per the Description of the Action and the entire project's work program has been aligned with such an approach.

In the following sections, the software development and release processes are detailed. Moreover, there is a detailed description of the development and release cycles based on an Agile[1] and Continuous Integration (CI) / Continuous Development (CD) best practices which have been proven to be a 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.

[1] https://agilemanifesto.org

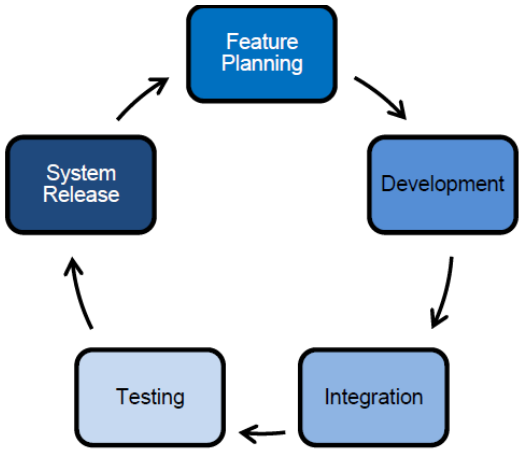

FIGURE 2: SEMIOTICS RELEASE CYCLE

Tentatively, the consortium considers 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 will deliver 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

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.

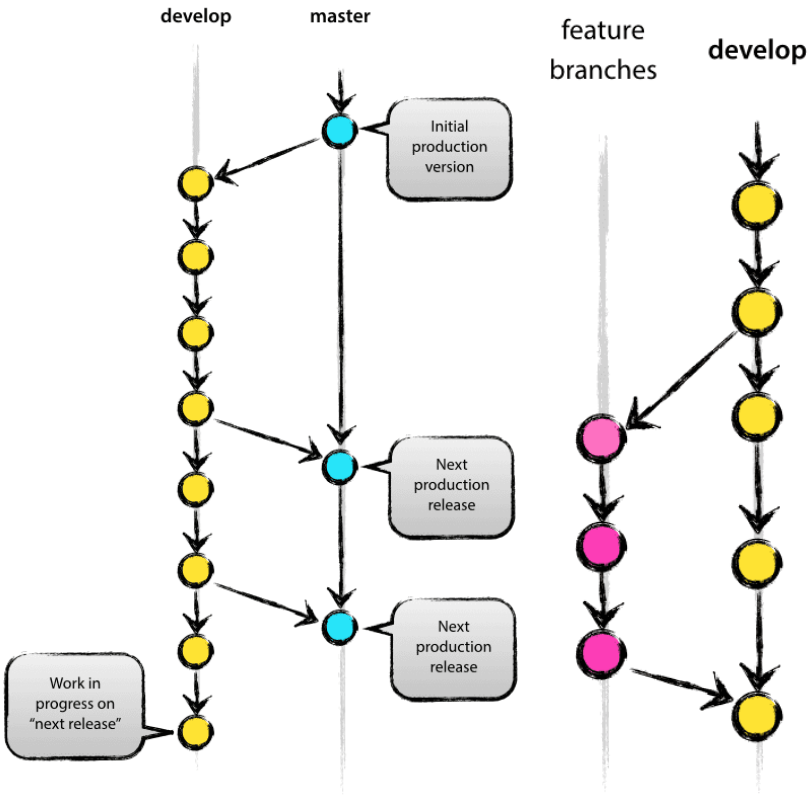

FIGURE 3: SEMIOTICS GIT REPOSITORY BRANCHES

2.2.2 CONTINUOUS INTEGRATION PIPELINE

A Continuous Integration 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.

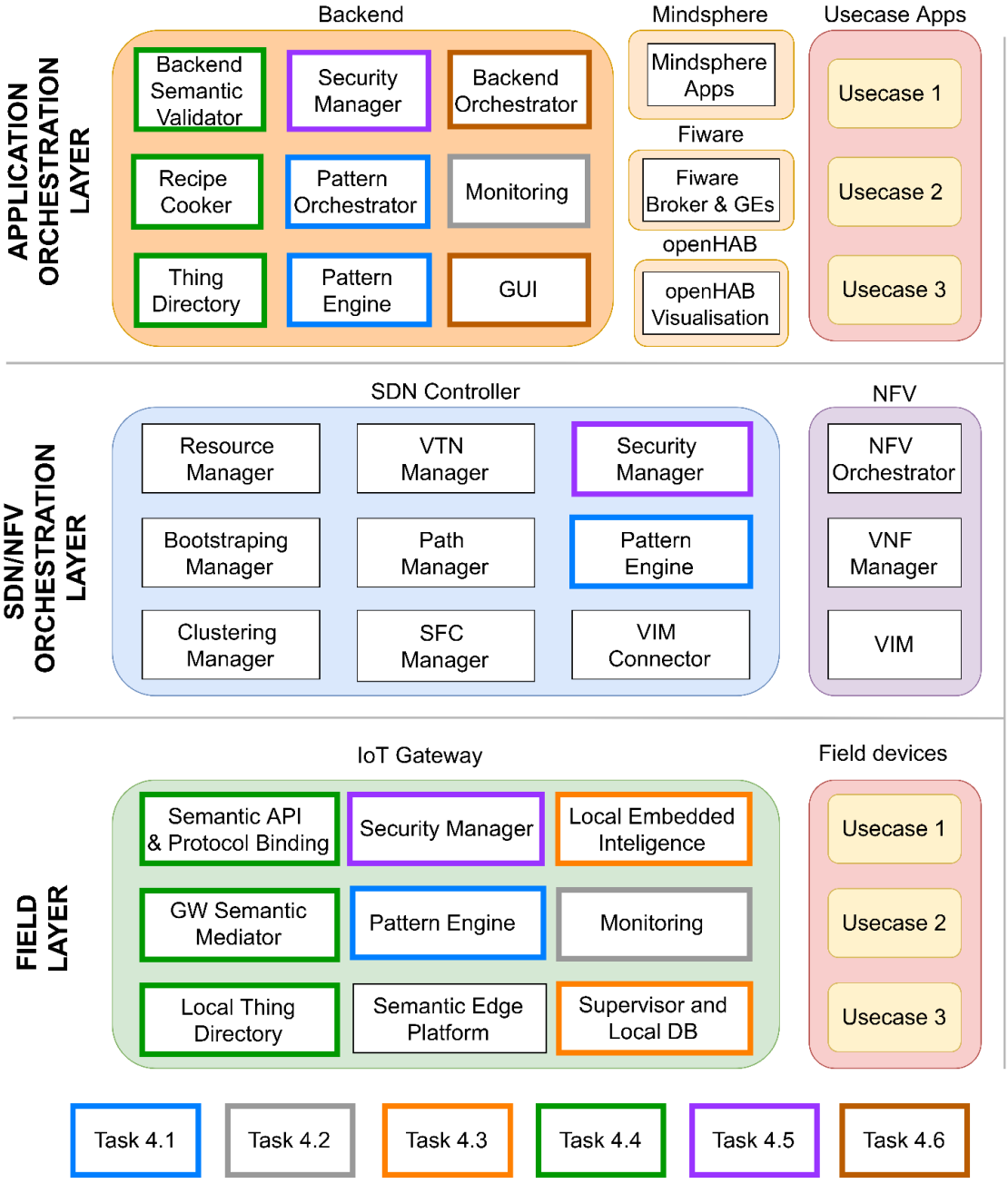

FIGURE 4 SEMIOTICS ARCHITECTURAL FRAMEWORK

Various components from SDN/NFV orchestration layer and field layer are mostly implemented in WP3, thus Table 1 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 (D4.7) covering the second cycle. More details about the individual components can be found in Section 0. While various components from the SDN/NFV orchestration layer and field layer are mostly implemented in WP3, it shows the cycle-assignment of components implemented within WP4 to development cycles. Each component is developed at least in two cycles. This document is a combination of D4.6, D4.7 and D4.13. D4.7 covers the second cycle and all the components are described in Section 4.

TABLE 1 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 0. As per the Agile methodology, the backlog is constantly updated throughout the project.

4 BACKEND COMPONENTS – CYCLE 2

As mentioned above, the implementation of the SEMIoTICS framework solution imposes not only the implementation of the components but also designing suitable interactions between them. Not only is the definition of components API required, but also defining which components are consumers of which component API.

The landscape definition of the component interactions with API definitions has been initiated in Cycle 1 and continued at the early stages of Cycle 2, as it was crucial for further development of the specific components. The subsections below feature implementation details for each of the components comprising the SEMIoTICS backend, as of Cycle 2.

4.1 Graphical User Interface

As described in D4.6, Graphical User Interface 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, 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 2 and few of them have been updated. Also, the architecture of GUI has been enriched with several new components.

TABLE 2 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	To Do
Add routing to other SEMIoTICS' components	Create a bar that allows navigating through other SEMIoTICS' components	Cycle 3	To Do

4.1.1 ARCHITECTURAL CHANGES

Components in the GUI component picture and their roles:

- **GUI: Database** – PostgreSQL database stores configuration data and historical data gathered from registered sensors
- **GUI: Backend** – a mediator between GUI and any application that GUI communicates with
- **GUI: Worker Orchestrator** – a new inner component of GUI: Backend. The role of GUI: Worker Orchestrator is to assign a job to each of the registered workers using a round-robin algorithm.
- **GUI: Thing Worker** – a worker that periodically retrieves data from registered things
- **GUI: Semantic Thing Simulator** – a simulator of a thing that exposes endpoints:
  - to get thing data
  - to interact with a thing

The overall picture of the architecture is presented in Figure 5. With this approach, it is possible to present not only the current state of registered things but also, it is possible to present historical data gathered by Thing Worker.

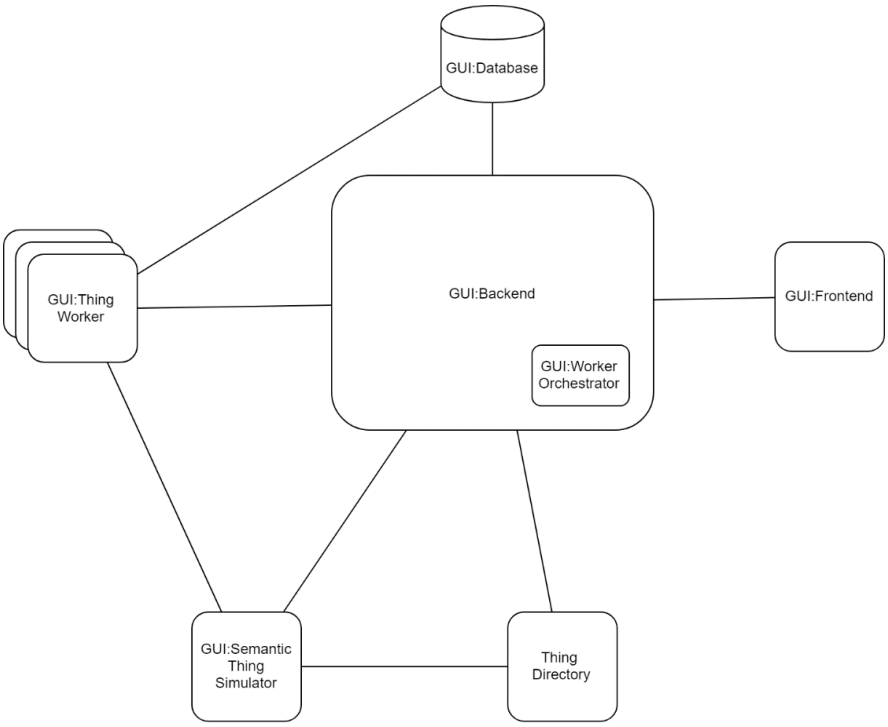

FIGURE 5 GUI ARCHITECTURE COMPONENTS

4.1.2 ACTIONS FLOW

Action sequences are shown in the flow charts below. As shown in Figure 6, after GUI:ThingWorker is initialized, GUI:ThingWorker registers itself with the GUI:Database.

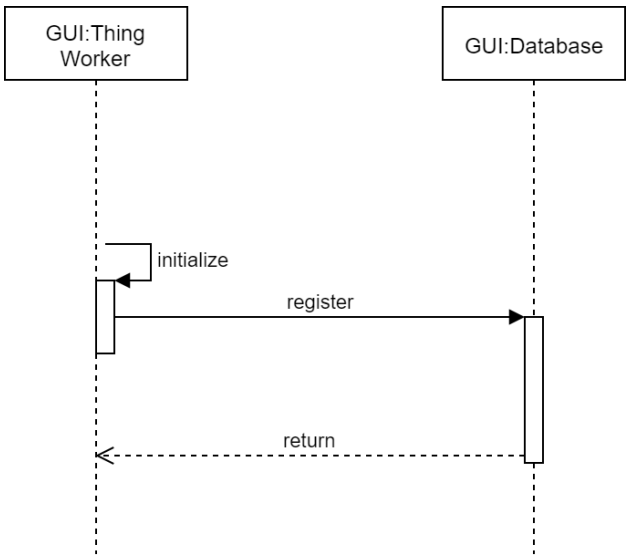

FIGURE 6 THINGWORKER REGISTRATION WITH THE GUI DATABASE

After initialization, GUI:ThingWorker starts cycle job as shown in Figure 7 (with the frequency of the job as a parameter).

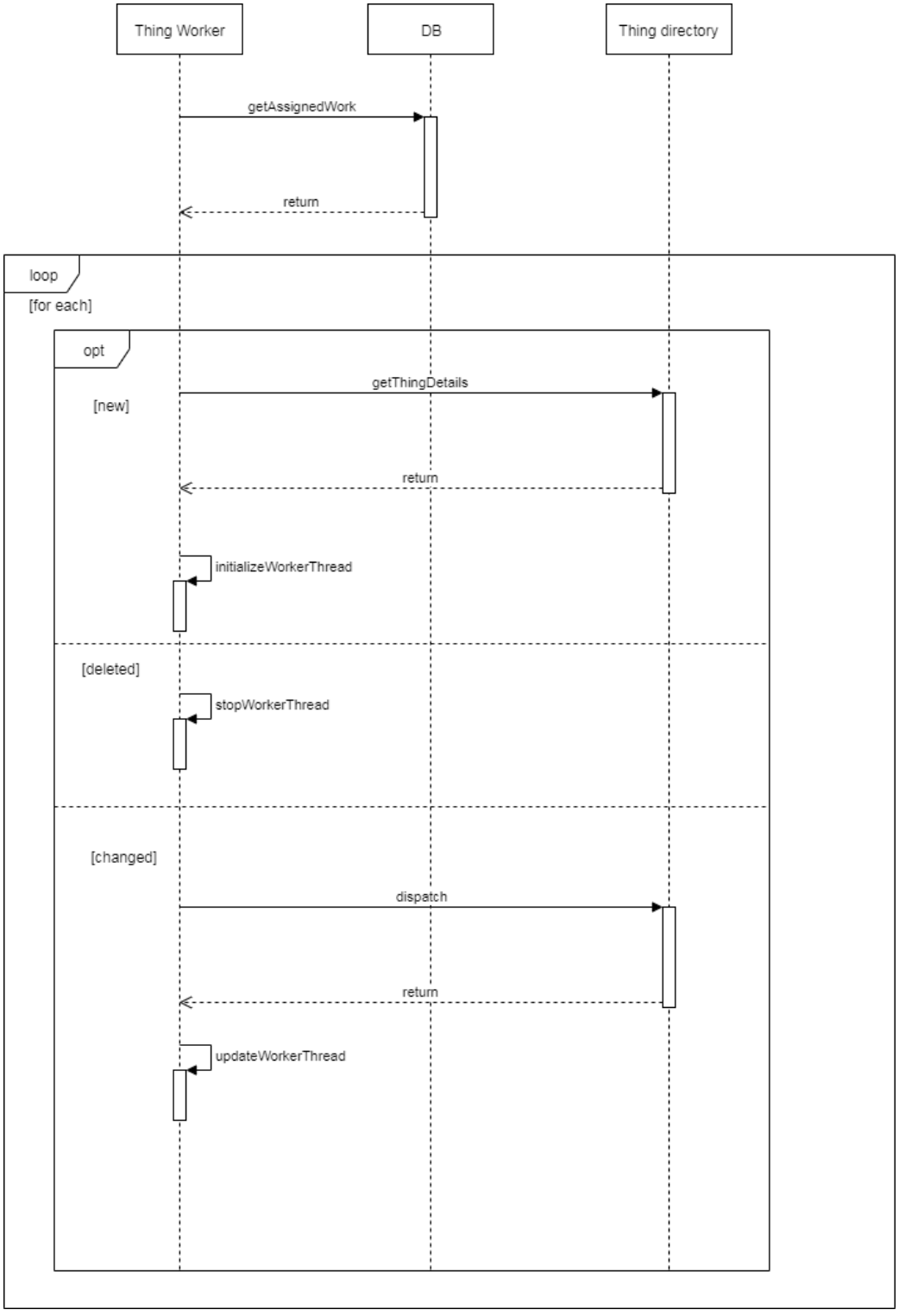

FIGURE 7 GUI:THINGWORKER CYCLE JOB

GUI:ThingWorker's single thread is responsible for sending a request to a thing that was assigned to him to obtain thing's data. The data is then stored in the GUI:Database (as shown in Figure 8).

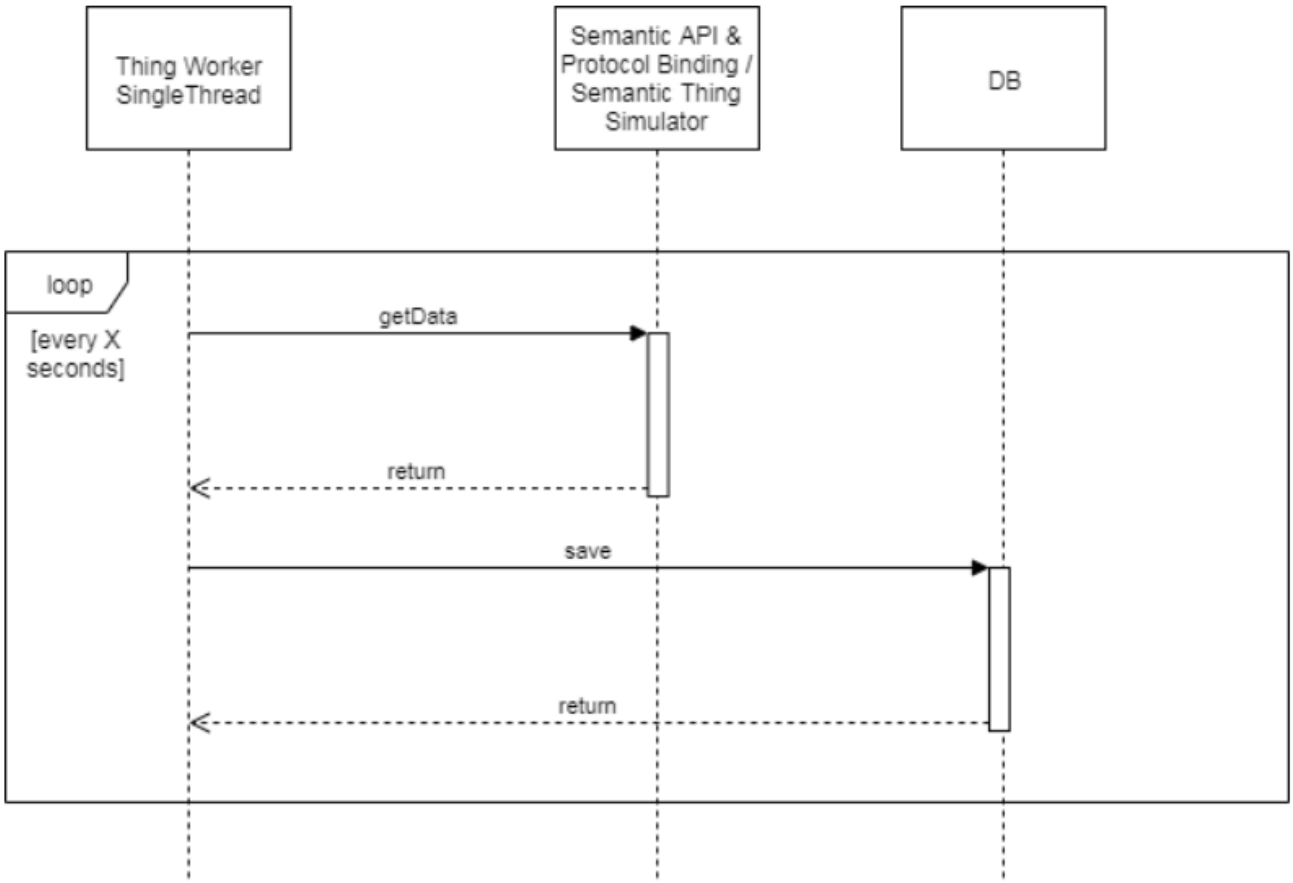

FIGURE 8 GUI:THINGWORKER'S SINGLE THREAD

GUI:WorkerOrchestrator is responsible for checking that all registered workers are working properly. It is done by sending a *checkIfAlive* request to each GUI:ThingWorker. If a worker does not respond or works incorrectly, information about an incident is passed to the GUI:Database and, for security reasons GUI:ThingWorker is destroyed. The described flow of actions is shown in Figure 9.

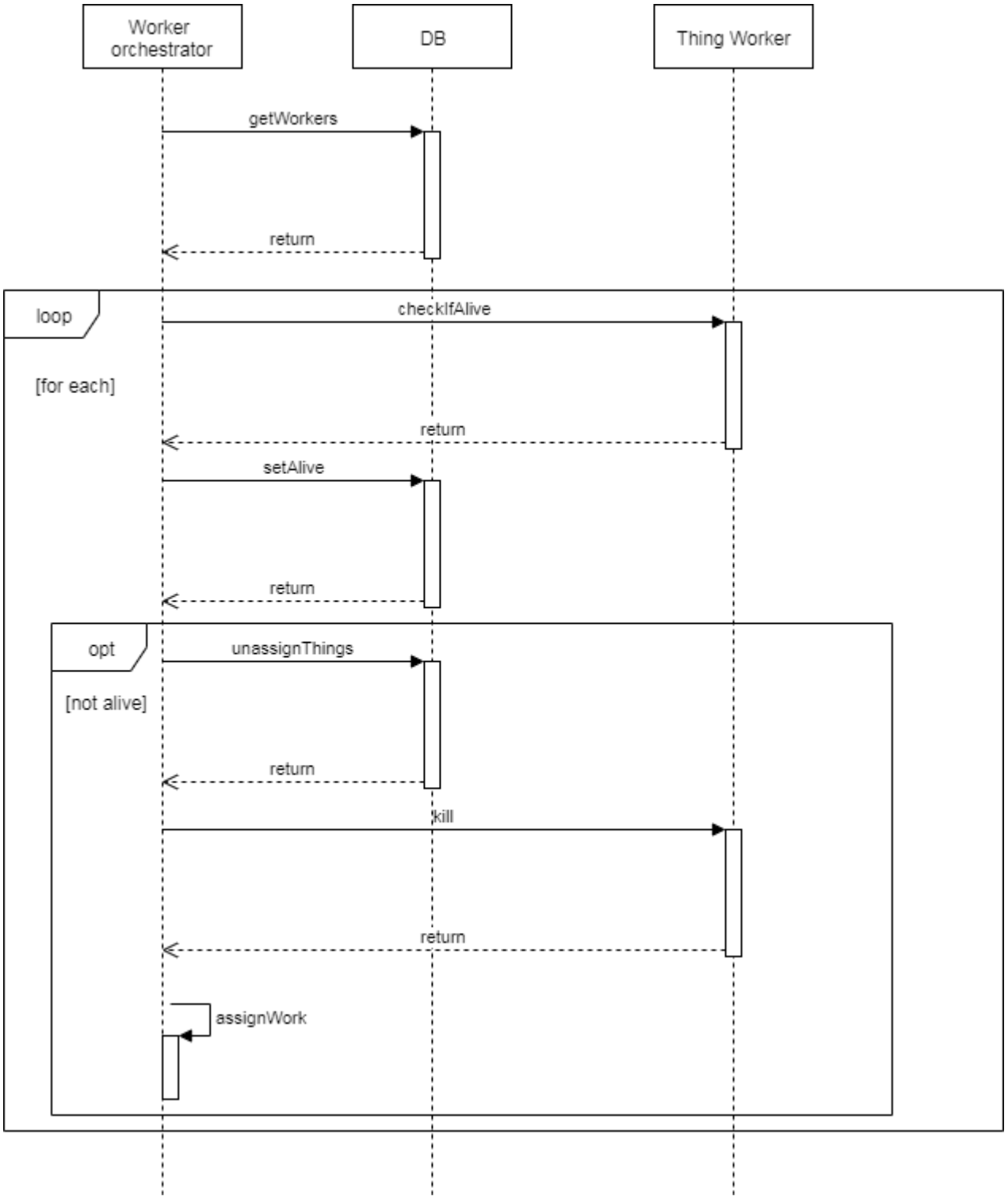

FIGURE 9 GUI:WORKERORCHESTRATOR FLOW

Additionally, GUI:WorkerOrchestrator is responsible for assigning work to every GUI:ThingWorker. In the beginning, GUI:WorkerOrchestrator gets a list of tracked things from GUI:Database. Then, GUI:WorkerOrchestrator gets available workers from GUI:Database. Finally, GUI:WorkerOrchestrator assigns things to every GUI:ThingWorker according to the round-robin algorithm. A sequence diagram of these actions is presented in Figure 10.

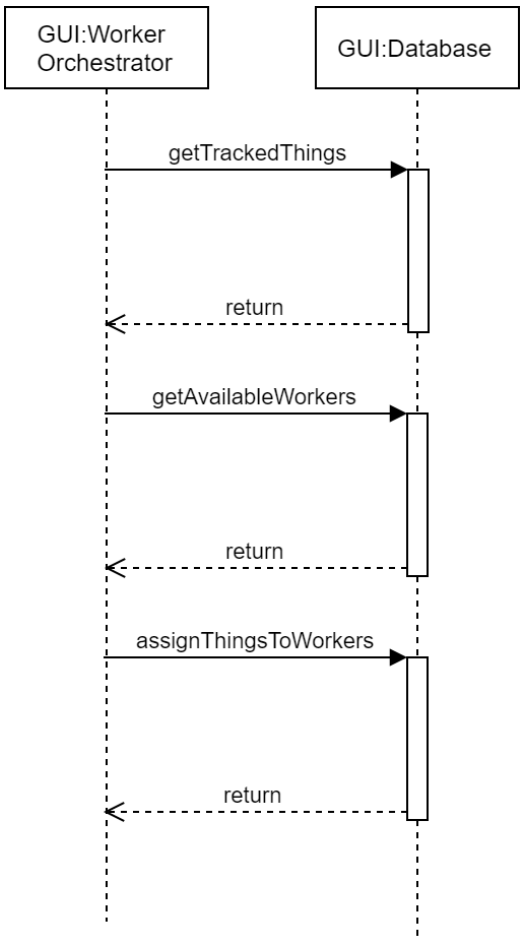

FIGURE 10 GUI:WORKERORCHESTRATOR ASSIGNING FLOW

4.1.3 GUI CHANGES

List view, presented in Figure 11, has been updated with two icons – blue icon navigates to the updated details view (Figure 12), while the green “eye” icon can be used to open a Thing monitoring view that has been developed within this cycle and is described later in this Section. The registration view (Figure 13) does not have any significant changes different than the background change, and light mode used to be consistent with the overall GUI view.

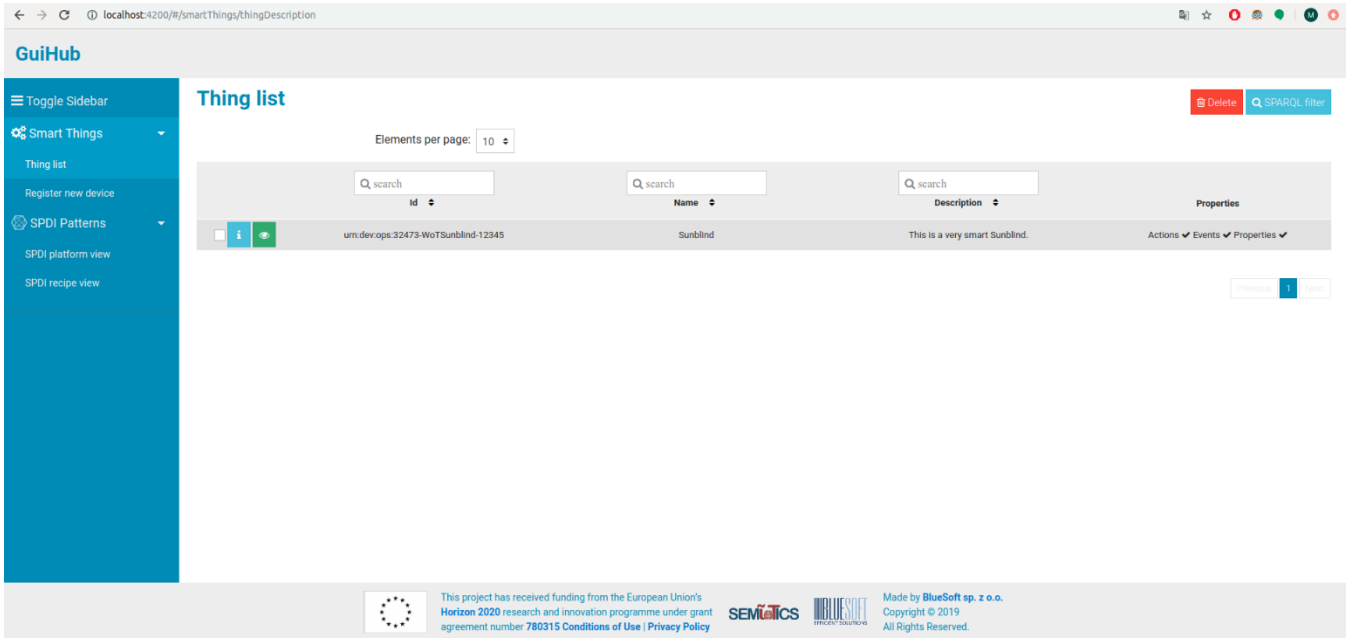

FIGURE 11 UPDATED LIST VIEW

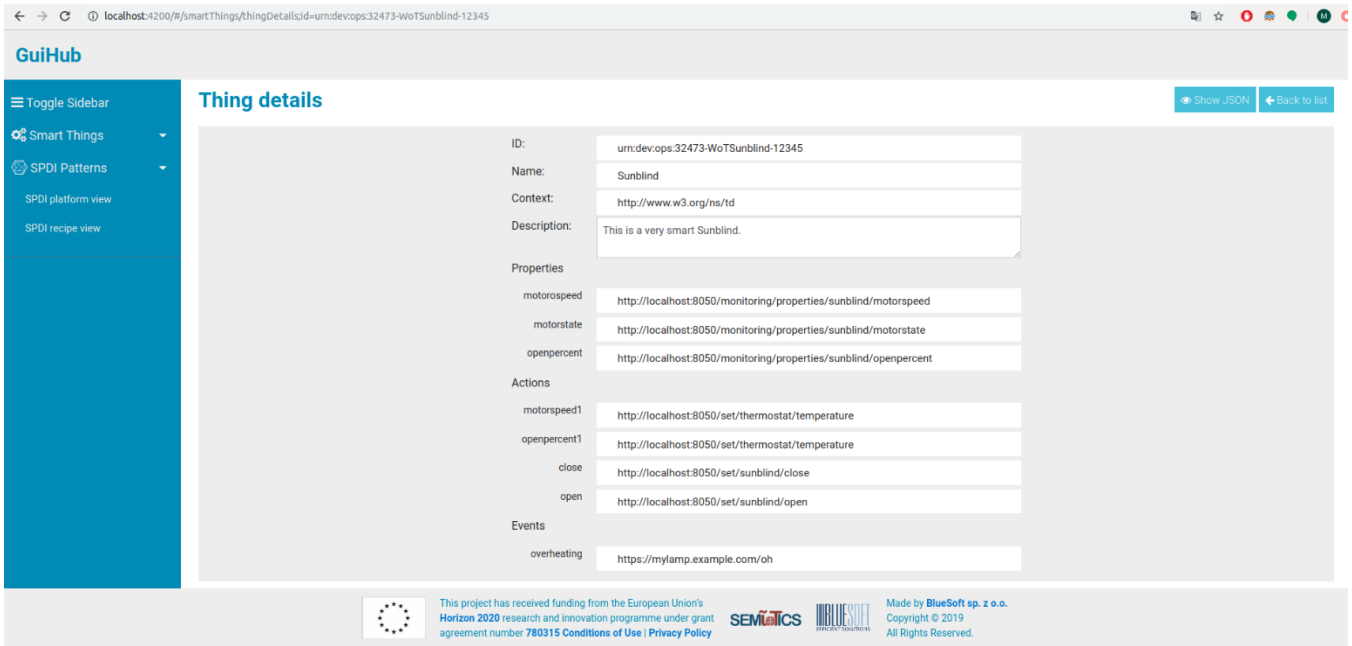

FIGURE 12 UPDATED THING DETAILS VIEW

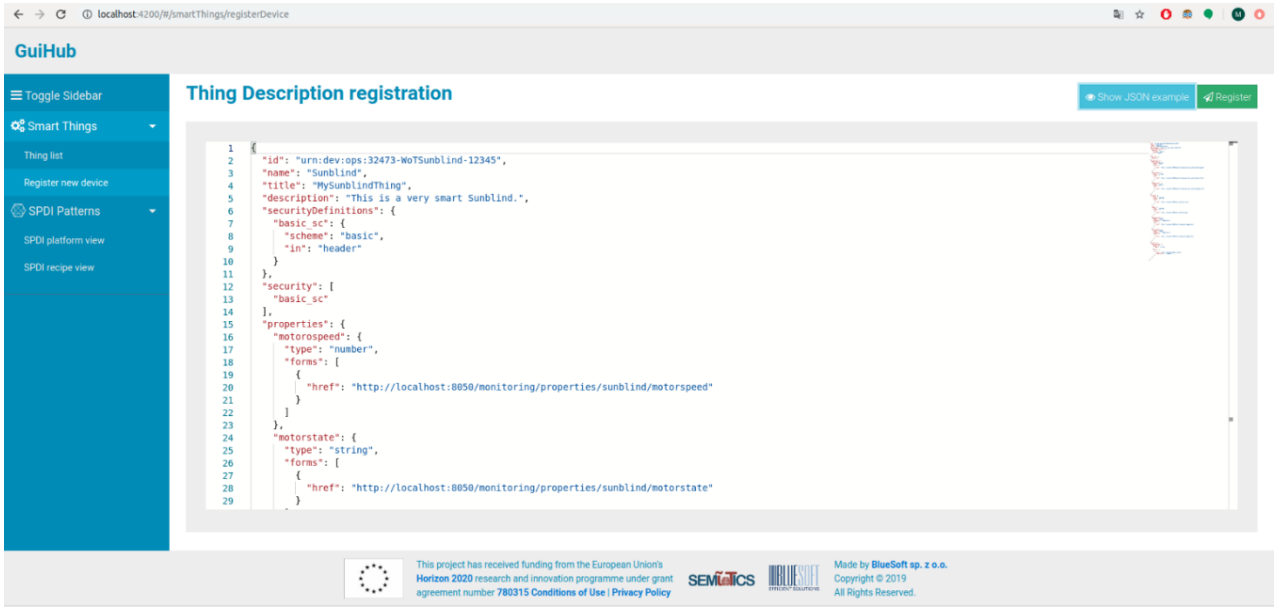

FIGURE 13 UPDATED REGISTRATION VIEW

4.1.4 NEW FEATURES

4.1.4.1 THING MONITORING VIEW

Within cycle 2, the *Thing monitoring view* was developed (presented in Figure 14). Now, it is possible to interact with Things – in this example with a Thing called Sunblind. At the moment, it is possible to set parameters that will be used later to create meaningful results on IoT dashboards in the next development cycle. Now, it is also possible to interact with things – make a simple actuation such as open/close Sunblind or parametrized one e.g. set motor speed value. (As it was mentioned earlier, this view is crucial for the IoT Dashboard coming in the next cycle).

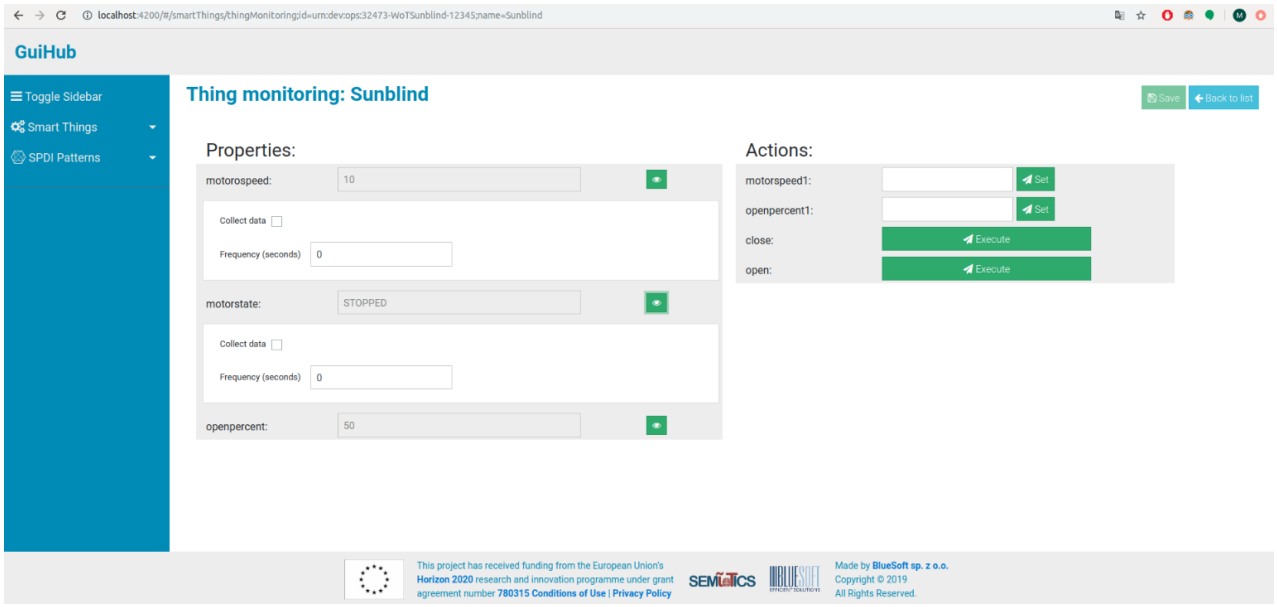

FIGURE 14 THING MONITORING VIEW

4.1.4.2 SPDI PATTERNS

To enable meaningful insights into the platform, the SPDI pattern monitoring view, presented in Figure 15, was developed. One of the objectives of the project was the development of the SPDI patterns, and this view gives a way of presentation of patterns (Security, Privacy, Dependability, and Interoperability), including cross-layer and layer-specific patterns. Patterns are colored according to their state. The icon is green if all patters are satisfied, and it is red if none is satisfied. It is yellow when the pattern is partially met, and it is gray if patterns of one kind aren't defined. After clicking on the "Show Table" button, the SPDI pattern monitoring view can be presented as a table (see Figure 16).

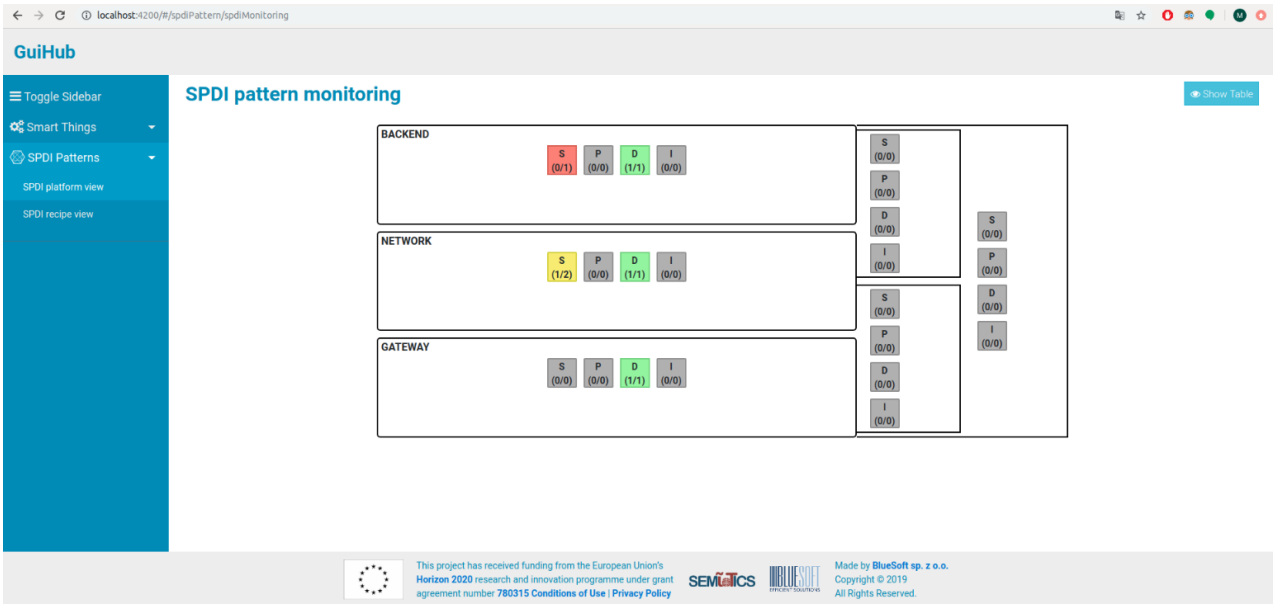

FIGURE 15 SPDI PATTERN MONITORING VIEW

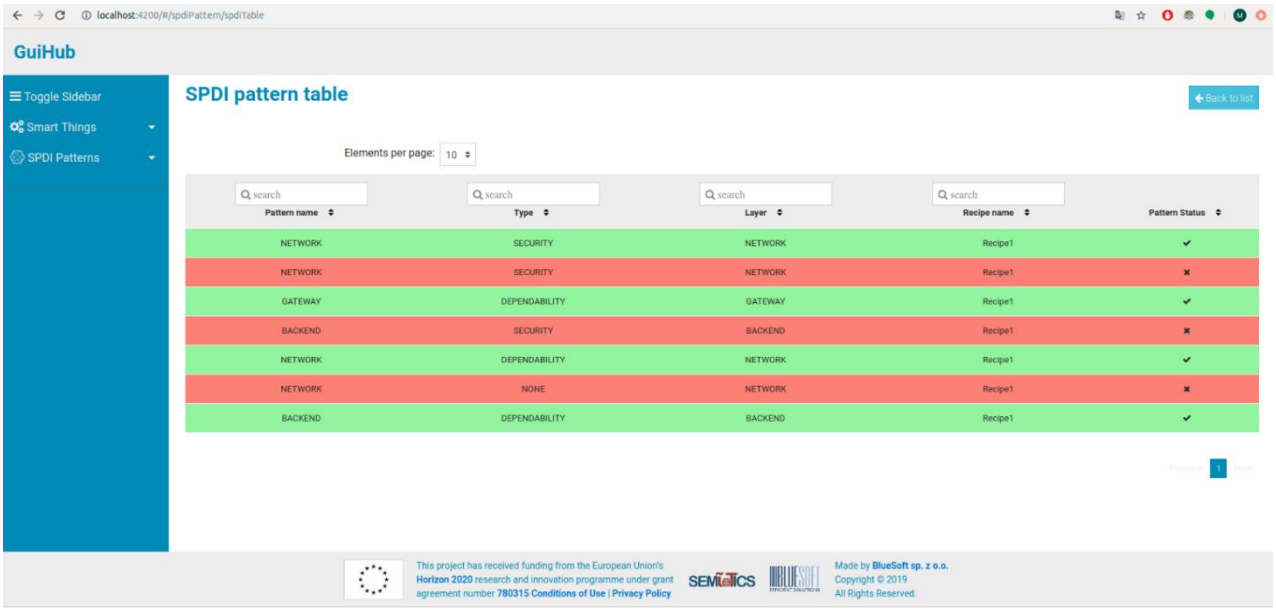

FIGURE 16 SPDI PATTERN TABLE

The pattern details view has been developed as well. It is possible to present all patterns that one kind of pattern consists of (see Figure 17).

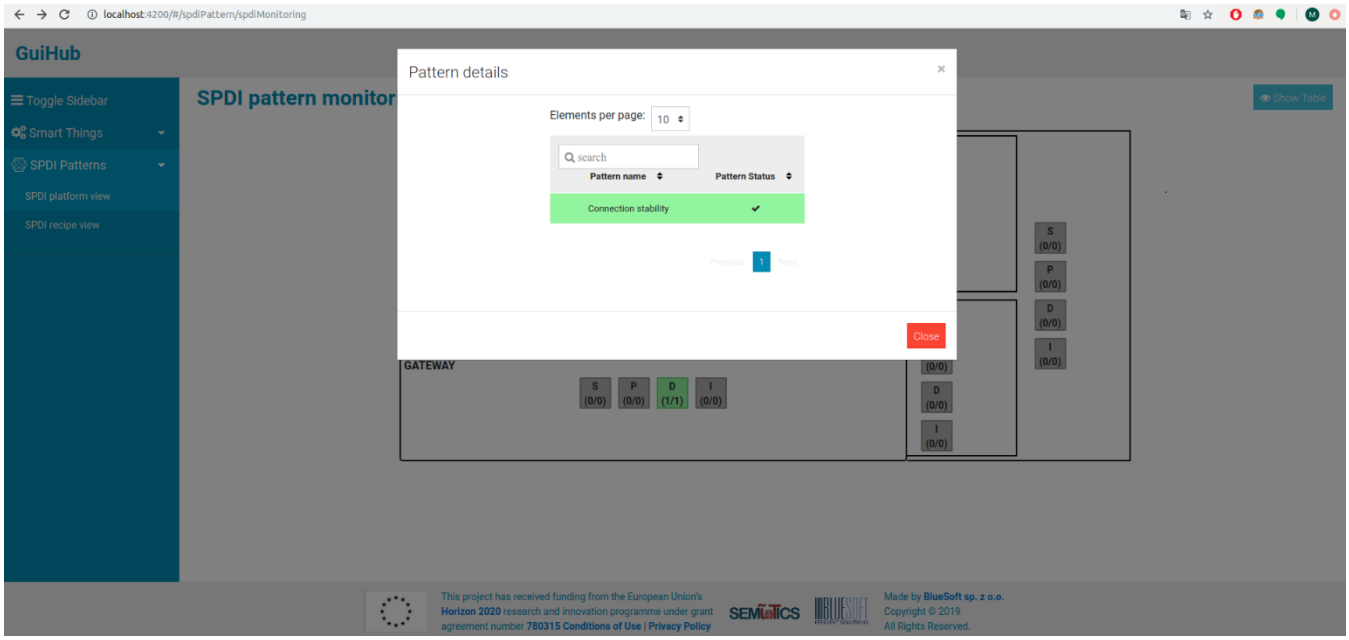

FIGURE 17 PATTERN DETAILS

4.1.4.3 SPDI RECIPE VIEW

SPDI recipe view is the other view that has been developed in this cycle. This view presents recipes obtained from Pattern Orchestrator (they were created before in Recipe Cooker). The given algorithm is presented as a graph. The inner components of sequences are colored with the same color to meaningfully illustrate the recipe. Split nodes are the nodes that have more than one child (e.g. when data from one collector is passed to the two different analytic tools). Merge nodes are the nodes that have at least two parents (e.g. to use two factors to make some decision).

As it is in the “Recipe1” example (see Figure 18), the red one sequence illustrates passing data from *Camera* to the *ObjectDetector*. The green one sequence shows that sound data is passed from the Microphone to the *SoundClassifier*. The output from *ObjectDetector* and the output from *SoundClassifier* are combined in the *DetectIntruder* merge node. This node is responsible for combined analytics with the aim of the detection of an intruder. Based on the output of the DetectIntruder node, the *SendNotification* component can be invoked.

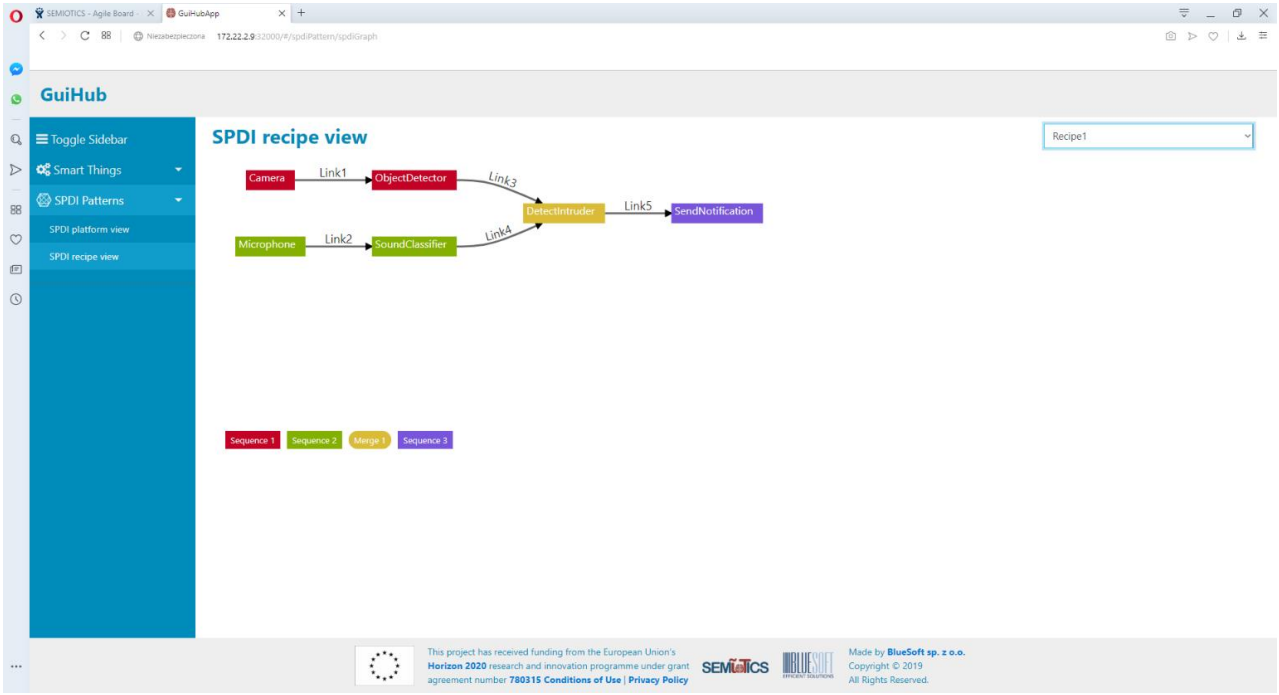

FIGURE 18 SPDI RECIPE VIEW

4.2 Backend Orchestrator

As described in D4.6, the Backend Orchestrator is a component responsible for integrating all backend services and exposing API. Kubernetes[2] has been chosen as a component responsible for the 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 microservices architecture (described widely in D2.4).

The development of the Backend Orchestrator component has been continued within cycle 2.

Within Table 3 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

[2] https://kubernetes.io

TABLE 3 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	Done
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	Done
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	Done
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	In Progress
Develop the scheme of deploying the component	Creating the script of deployment, service and config map for example component.	Cycle 2	Done
Deploying GUI and Thing Simulator on Kubernetes cluster	Developing the structure of deployment files for GUI and Thing Simulator	Cycle 2	Done
Automate the repeatable process of deploying the components.	Installation Jenkins on the machine. Creating the access rules for Jenkins to the BO.	Cycle 2	Done
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	Done

Performing the test of communication between components	Testing internal and external communication between deployed components	Cycle 2	Done
Develop the way of storage and updating the credentials for externals applications	Set of rules about storage and user credentials for GitLabRepository.	Cycle 2	Done
Manual deployment of Thing Directory	Create dockerfile and dockerize the application so it can be later deployed on Kubernetes. Deployment component	Cycle 2	Done
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	Done
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	Done
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	Done
Creating deployment files for AOL components	Create deployment .yml files for Pattern Engine, Recipe Cooker, Backend Semantic Validator, which allows deployment on Kubernetes.	Cycle 3	To Do
Create automatized jobs for deploying components.	Create CI/CD pipeline that allows deploying the following components: Pattern Engine, Recipe Cooker, Backend Semantic Validator on Kubernetes after manual initialization.	Cycle 3	To Do

4.2.1 DEVELOPMENT STATUS

In cycle 2 the Kubernetes instance was installed with ansible scripts on BLS premises (on bare metal). The instance was used to orchestrate a subset of components of SEMIoTICS backend. One of the tasks, which is to prepare a set of access rules for consortium partners to Backend Orchestrator, will be continued in cycle 3. The script with permissions and namespaces has already been prepared but the configuration on Kubernetes is still to be done. Concept of Kubernetes deployment site has changed and there is alternative installation approach considered. Namely, to install BO on the server of one of the partners instead of on BLS. The server will be available in cycle 3 so the final decision and configuration is postponed to this cycle.

Kubernetes uses a declarative approach to manage Kubernetes resources. The ‘semiotics’ namespace resource was created especially for the project, so every application can be deployed in this namespace. This resource was created during Kubernetes setup.

```
apiVersion: v1
kind: Namespace
metadata:
  name: semiotics
```

One of the most popular ways to use Kubernetes API is with kubectl command-line tool. The output of command used to present the current state of all deployed applications within 'semiotics' namespace (created during the installation) is presented in Figure 19 (view from ubuntu terminal).

As it is shown in the presented figures (Figure 19), in the current cycle Kubernetes instance was used to orchestrate following backend applications related to GUI and/or security use case:

- Security Manager
- Security Manager:PEP
- Thing Directory
- GUI:Backend
- GUI:Frontend
- GUI:ThingSimulator
- GUI:ThingWorker
- GUI:Database

In cycle 2, the response from Pattern Orchestrator was mocked by GUI:Frontend application. For every application, a dedicated Kubernetes YAML declarative configuration was prepared. In case of GUI:Frontend application following resources were created:

- Deployment – responsible for ensuring that predefined number of GUI:Frontend applications are ready (see YAML definition in
- Table 6)
- Service – responsible for exposing GUI:Frontend application outside of the cluster (see YAML definition in Table 4)
- ConfigMap – responsible for providing necessary information for GUI:Frontend Deployment (see YAML definition in
-
- Table 5).

TABLE 4 GUI:FRONTEND SERVICE DEFINITION

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-svc
  namespace: semiotics
  labels:
    app: front
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 32000
    protocol: TCP
  selector:
    app: front
```

TABLE 5 CONFIGMAP USED BY GUI:FRONTEND POINTING GUI:BACKEND

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: api-map
  namespace: semiotics
data:
  API_URL: http://172.22.2.9:31000/td
```

TABLE 6 GUI:FRONTEND DEPLOYMENT DEFINITION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    configmap.reloader.stakater.com/reload: "api-map"
  name: front-deployment
  namespace: semiotics
  labels:
    app: front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        app: front
    spec:
      containers:
      - name: front
        image: registry.gitlab.com/semiotics/backend/gui/frontend:jenkins
```

```
      resources:
        requests:
          memory: "20Mi"
          cpu: "200m"
        limits:
          memory: "40Mi"
          cpu: "400m"
      ports:
        - containerPort: 80
      env:
      - name: API_URL
        valueFrom:
          configMapKeyRef:
            name: api-map
            key: API_URL
    imagePullSecrets:
    - name: blue-k8s
```

In case of GUI:Backend application following resources were created:

- Deployment – responsible for ensuring that predefined number of GUI:Backend applications are ready (see YAML definition in Table 7)

SERVICE – RESPONSIBLE FOR EXPOSING GUI:FRONTEND APPLICATION OUTSIDE OF THE CLUSTER (SEE YAML DEFINITION IN

- Table 9)

CONFIGMAP – RESPONSIBLE FOR PROVIDING NECESSARY INFORMATION FOR GUI:FRONTEND DEPLOYMENT (SEE YAML DEFINITION IN

- Table 8)

TABLE 7 GUI:BACKEND DEPLOYMENT DEFINITION

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: back-deployment
  namespace: semiotics
  labels:
    app: back
spec:
```

```
replicas: 1
selector:
  matchLabels:
    app: back
template:
  metadata:
    labels:
      app: back
  spec:
    containers:
    - name: back
      image: registry.gitlab.com/semiotics/backend/gui/backend:jenkins
      resources:
        requests:
          memory: "230Mi"
          cpu: "100m"
        limits:
          memory: "460Mi"
          cpu: "200m"
      imagePullPolicy: Always
      ports:
      - containerPort: 8090
      env:
      - name: THING_DIRECTORY_SERVICE_URL
        valueFrom:
          configMapKeyRef:
            name: backend-map
            key: THING_DIRECTORY_SERVICE_URL
      - name: SPRING_DATASOURCE_URL
        valueFrom:
          configMapKeyRef:
            name: backend-map
            key: SPRING_DATASOURCE_URL
      - name: SPRING_DATASOURCE_USERNAME
        valueFrom:
```

```
          configMapKeyRef:
            name: backend-map
            key: SPRING_DATASOURCE_USERNAME
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: backend-map
              key: SPRING_DATASOURCE_PASSWORD
      imagePullSecrets:
      - name: blue-k8s
```

TABLE 8 CONFIGMAP USED BY GUI:BACKEND (POINTING THING DIRECTORY SERVICE)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-map
  namespace: semiotics
data:
  THING_DIRECTORY_SERVICE_URL: http://tdirectory-svc:8080
  SPRING_DATASOURCE_URL: ***
  SPRING_DATASOURCE_USERNAME: ***
  SPRING_DATASOURCE_PASSWORD: ***
```

TABLE 9 GUI:BACKEND SERVICE DEFINITION KIND: SERVICE

```
apiVersion: v1
metadata:
  name: back-svc
  namespace: semiotics
spec:
  ports:
  - nodePort: 31000
    port: 8090
    protocol: TCP
```

```
  targetPort: 8090
selector:
  app: back
sessionAffinity: None
type: NodePort
```

4.2.2 BACKEND ORCHESTRATOR DASHBOARD SETUP

The deployed Kubernetes dashboard presents the current state (all running) of deployed applications in the browser (Figure 20), while Figure 19 shows the same state using kubectl tool in ubuntu console.

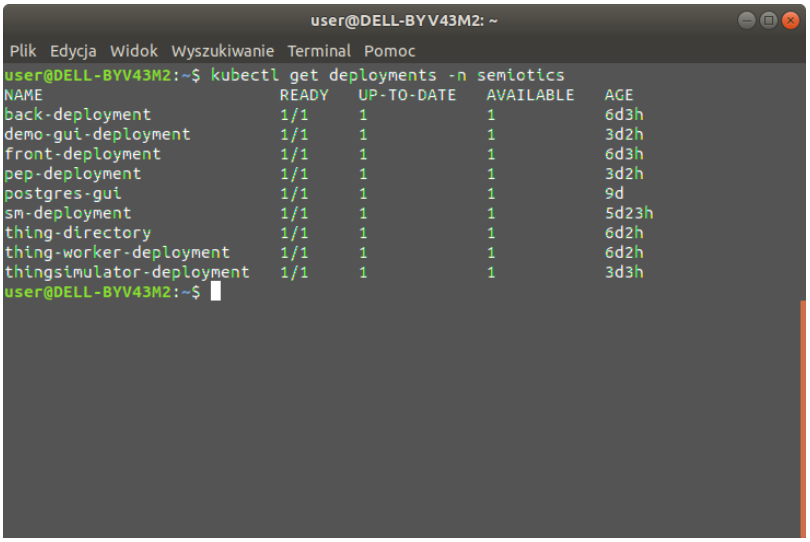

FIGURE 19 CURRENT STATE OF DEPLOYED APPLICATIONS WITHIN SEMIOTICS NAMESPACE

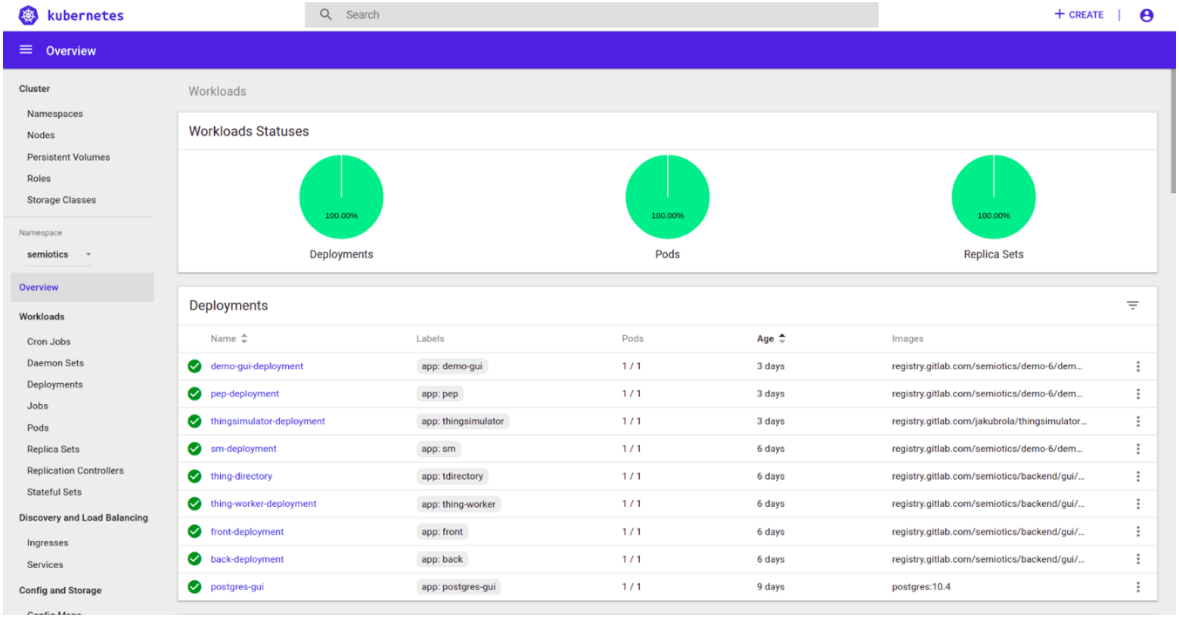

FIGURE 20 STATE OF DEPLOYED APPLICATIONS WITHIN SEMIOTICS NAMESPACE PRESENTED WITH KUBERNETES DASHBOARD

4.3 Pattern Orchestrator

As described in D4.6, Pattern Orchestrator is responsible for the automated configuration, coordination, and management of different patterns and their deployment. In further detail, the Pattern Orchestrator will:

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 2 includes:

- Re-implementation of the Pattern Orchestration interface with REST API
- 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 10 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	In Progress
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	To Do

As already mentioned in Section 4.4 regarding the distribution of patterns to different pattern engines, the decision mechanism of the Pattern Orchestrator, is not yet finalized. 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

Similarly, to the Backend Pattern Engine described in subsection 4.4, the Spring Framework was adopted to build REST web services for the Pattern Orchestrator. Other SEMIoTICS components, such as Recipe Cooker, are able to make REST requests to the Pattern Orchestrator API using REST clients. An excerpt from the source code presents the REST services made available by the Backend Pattern Engine, in Figure 21.

```
@RestController
public class RestGreetingController {

    //private final AtomicLong counter = new AtomicLong();

    @RequestMapping(value = "/insertRecipe", consumes = "application/json", produces = "application/json")
    public String insertRecipe(@RequestBody Recipe recipe) {...
    }

    @RequestMapping(value = "/removeRecipe")
    public String removeRecipe(@RequestParam(value="recipeID") String recipeID) {...
    }

}
```

FIGURE 21: PATTERN ORCHESTRATOR REST API

Recipe Cooker uses the insertRecipe REST request to communicate with Pattern Orchestrator including a recipe description in JSON format. Such a request is depicted in Figure 22. Under “recipeID” a unique string that acts as an identifier is provided, while under “recipe” label lays the recipe description itself. The recipe instance depicted here is very simple and consists of two software components that are placed in sequence, which means that the output of the former is consumed as input by the latter.

```
{
    "recipeID": "Demo2WF1",
    "recipe": "Softwarecomponent(\"f5a474bd4cd818\", \"0\", \"piB\"), Softwarecomponent(\"f86e905a107f1\", \"0\", \"piB\"),
        Sequence(\"Seq0\", \"f86e905a107f1\", \"f5a474bd4cd818\", \"Link0\")"
}
```

FIGURE 22: INSERTRECIPE PAYLOAD IN JSON FORMAT

The removeRecipe request is used once again by the Recipe Cooker for the deletion of a Recipe from the Drools memory in one of the Pattern Engines in the three layers.

Moreover, the SEMIoTICS GUI uses the REST API of Pattern Orchestrator in order to visualize the status of the SPDI and QoS properties of the deployed IoT Service orchestrations. The response of the Pattern Orchestrator is in JSON format. A sample response is depicted below.

```
{
   "recipe": {
      "name": "RecipeName",
      "values": {
         "LinksList": [{
            "ID": "L23",
            "Node1": "P2",
```

```
        "Node2": "P3",
        "layer": "network",
        "properties": [{
            "name": "Pr23",
            "type": "required",
            "category": "qos_bandwidth",
            "value": 1000.0,
            "datastate": "datastate",
            "subject": "L23",
            "satisfied": true,
            "verificationtype": "verificationtype",
            "means": "means"
        }]
    }],
    "NodesList": [{
        "ID": "P1",
        "Name": "P1",
        "layer": "network",
        "properties": [{
            "name": "Pr1",
            "type": "required",
            "category": "qos_bandwidth",
            "value": 10.0,
            "datastate": "datastate",
            "subject": "P1",
            "satisfied": true,
            "verificationtype": "verificationtype",
            "means": "means"
        }]
    }, {
        "ID": "P2",
        "Name": "P2",
        "layer": "network",
        "properties": [{
            "name": "Pr2",
```

```
            "type": "required",
            "category": "qos_bandwidth",
            "value": 10.0,
            "datastate": "datastate",
            "subject": "P2",
            "satisfied": true,
            "verificationtype": "verificationtype",
            "means": "means"
        }]
    }],
    "SequencesList": [{
        "ID": "S23",
        "Name": "S23",
        "Node1": "P2",
        "Node2": "P3",
        "layer": "network",
        "properties": [{
            "name": "PrS23",
            "type": "required",
            "category": "qos_bandwidth",
            "value": 10.0,
            "datastate": "datastate",
            "subject": "S23",
            "satisfied": true,
            "verificationtype": "verificationtype",
            "means": "means"
        }]
    }],
    "MergesList": [{
        "ID": "M1",
        "Name": "M1",
        "Node1": "P1",
        "Node2": "S23",
        "Node3": "S45",
        "layer": "network",
```

```
            "properties": [{
                "name": "PrM1",
                "type": "required",
                "category": "qos_bandwidth",
                "value": 10.0,
                "datastate": "datastate",
                "subject": "M1",
                "satisfied": true,
                "verificationtype": "verificationtype",
                "means": "means"
            }]
        }]
    }
  }
}
```


As we can see the JSON response is actually a list for:

- the components (Nodes) that constitute the orchestration in question
- the Links among these components
- the Sequences in which the Nodes are organized
- the Merges in which the Nodes are organized
- the Choices in which the Nodes are organized
- the Splits in which the Nodes are organized

4.3.2 COMPONENT API INTERACTIONS DESCRIPTION

The key interactions of the Pattern Orchestrator with other components can also be seen at the instantiation and runtime diagrams depicted below. More details regarding the communications of the Pattern Orchestrator with the three Pattern Engines is given in the in D5.2.

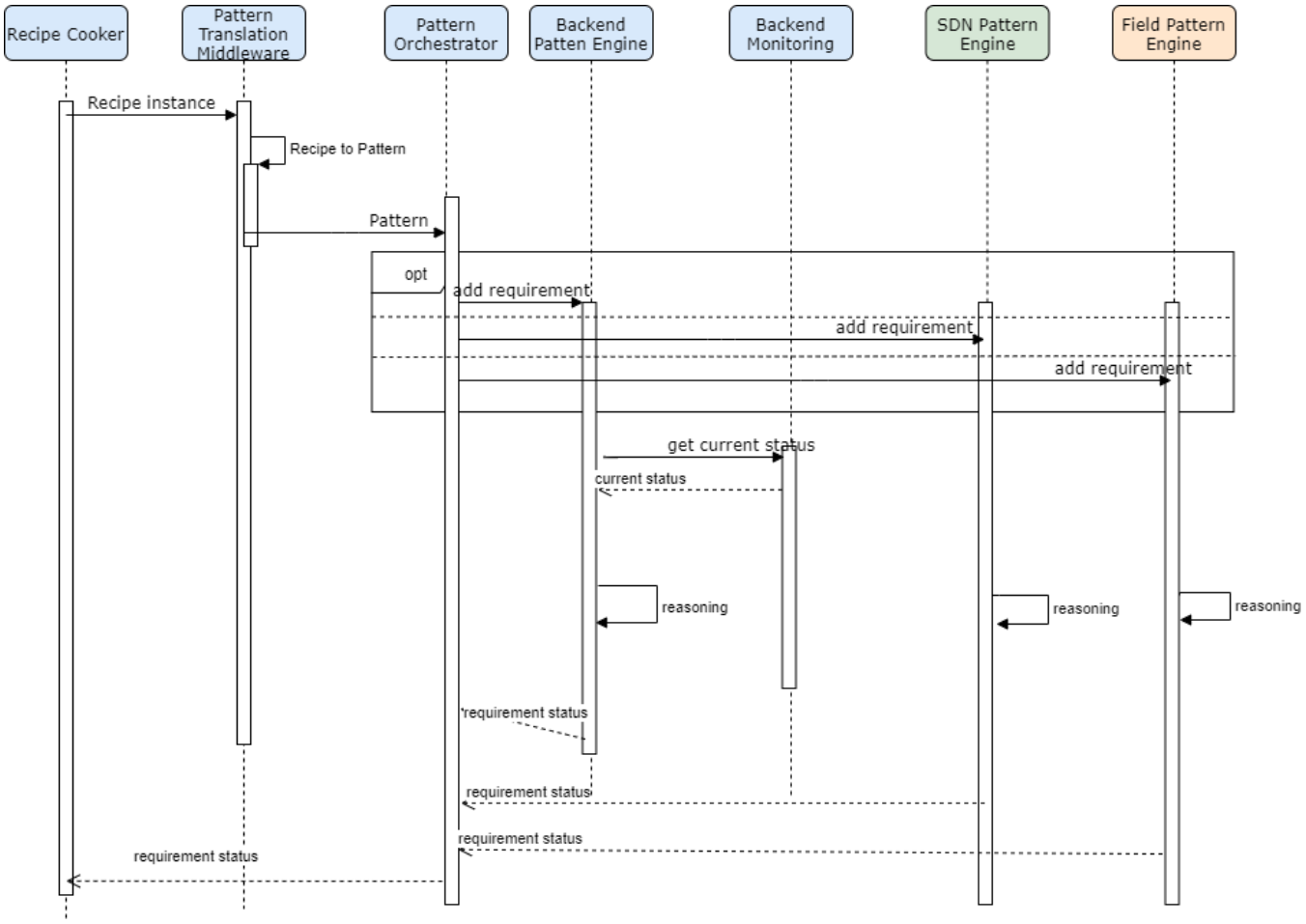

FIGURE 23 PATTERN ORCHESTRATOR INSTANTIATION AND RUNTIME DIAGRAM

4.4 Pattern Engine (backend)

As described in D4.6, the Backend Pattern Engine is a module featuring an underlying semantic reasoner processing Drools rules and facts. Additionally, it allows the insertion, modification, execution, and retraction of patterns either at design time or at runtime, through the Pattern Orchestrator, in the SEMIoTICS backend. Utilizing the Drools rule engine, 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 2 development includes:

- New implementation of API
- Successful communication between the Backend Pattern Engine, the Pattern engines of other layers and the Pattern Orchestrator
- New classes for the instantiation of Drools facts

Please refer to Table 11 for more details.

TABLE 11 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	In progress
Adaptation to maintain desired properties	When the desired property is no longer satisfied, the Backend Pattern Engine must take adaptation actions accordingly.	Cycle 3	To do
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, an adaptation of network to maintain desired properties and used SFC chains.	Cycle 3	To do

Regarding the testing of flow from Recipe Cooker, successful communication has been implemented which delivers the orchestration from the Recipe Cooker to a Pattern Engine through Pattern Orchestrator. The distribution of patterns to different pattern engines, the decision mechanism of the Pattern Orchestrator, is not yet finalized. 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.

4.4.1 DEVELOPMENT STATUS

Regarding the Backend Pattern Engine, the environment that was set up during Cycle 1 included Apache Maven 3.6.1, JBoss Drools7 7.15, and gRPC8 with Protocol Buffers9 Version 3. During Cycle 2, gRPC8 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. An excerpt from the source code presents the REST services made available by the Backend Pattern Engine, in Figure 24.

```
1   package eu.semiotics;
2
3   import com.google.gson.*;…
20
21  @RestController
22  public class GreetingController {
23
24      private KieServices ks;
25      private KieContainer kContainer;
26      private KieSession kSession;
27      private KieFileSystem kfs;
28      public static KieFileSystemExample kfse= new KieFileSystemExample();
29      //private static String FACT_RESOURCE = "../../files/facts/";
30      private static String FACT_RESOURCE = "./facts/";
31      private static String EXTERNAL_DRL_RESOURCE = "./externalrules/";
32
33
34
35      @PostMapping(value = "/addFact", consumes = "application/json", produces = "application/json")
36
37      public String addfact(@RequestBody Fact fact) {…
568
569
570     @RequestMapping(value = "/factRemove", consumes = "application/json", produces = "application/json")
571     public Greeting factRemove(@RequestBody Fact fact){…
061
062     @RequestMapping(value = "/factUpdate", consumes = "application/json", produces = "application/json")
063     public String factUpdate(@RequestBody Fact fact){…
593
594
595     @RequestMapping(value = "/factStatus", consumes = "application/json", produces = "application/json")
596
597     public Greeting factStatus(@RequestBody Fact fact){…
647
648
649     @PostMapping(value = "/insertRule", consumes = "application/json", produces = "application/json")
650     public Greeting insertRule(@RequestBody Rule rl) {…
689
```

FIGURE 24: BACKEND PATTERN ENGINE REST SERVICES

4.4.2 COMPONENT API INTERACTIONS DESCRIPTION

As we can see in Figure 24 above, the main web services exposed from the Backend Pattern Engine API are:

- addFact
- factRemove
- factUpdate
- factStatus
- insertRule

The above corresponds to the creation, retrieval, deletion of facts and creation of rules.

In more detail, the *addFact* REST service is used by the Pattern Orchestrator for the communication of new Drools facts of a new IoT Service orchestration. It can also be used by the Pattern Engines of the other layers (Network and Field) in the case of new fact discovery. In any case, the JSON that is sent is based on the Fact Java class that can be seen in the code snippet below, in Figure 25.

```java
package eu.semiotics;

public class Fact {
    private String id;
    private String from;
    private String message;
    private String type;
    private String recipeId;

    public Fact(String recipe_id,String fact_id, String fact_from, String fact_message, String fact_type) {
        this.id = fact_id;
        this.from = fact_from;
        this.message = fact_message;
        this.type = fact_type;
        this.recipeId=recipe_id;
    }

    }

```

FIGURE 25: FACT JAVA CLASS ATTRIBUTES USED IN REST SERVICES JSON

Moreover, the *factRemove* is used in order for a fact to be deleted from the Drools Memory of the Backend Pattern Engine. The *factUpdate* is used again by the Pattern Orchestrator in case some changes need to be applied to a Drools Fact. The *factStatus* REST service returns the current status of a special type of Drools facts, the instances of Property class. These instances are used to describe SPDI and QoS properties for the components of an IoT Service orchestrator. This REST service could be used for the visualization of the SPDI properties of an orchestrator in the SEMIoTICS GUI. Finally, the *insertRule* REST service is used only by the Pattern Orchestrator to communicate Drools Rules to the Backend Pattern Engine for the reasoning of the SPDI and QoS properties.

Table 12 below shows an aggregation of the API communications between the Backend Pattern Engine and the other SEMIoTICS components.

TABLE 12: BACKEND PATTERN ENGINE API COMMUNICATIONS

Consumer component	Owner	Components that will be used/consumed by this component	A layer of component that will be consumed	Description of interactions
Pattern Orchestrator	STS	Backend Pattern Engine	Backend	CRUD of rules and facts
Backend Pattern Engine	STS	Network Pattern Engine	Network	Fact and Rule Updates
Backend Pattern Engine	STS	Field Pattern Engine	Field	Fact and Rule Updates

4.5 Backend Semantic Validator

As mentioned in the D4.6, the scope of the Backend Semantic Validator (BSV) component is to tackle the semantic interoperability issues that arise in the SEMIoTICS framework (see Deliverable D4.4), at the application orchestration layer. The BSV can receive a request from IoT application for interaction between two Things (i.e. sensor, actuator), which are described with two different TDs (based on W3C Thing Descriptions that are serialized to JSON-LD standard format), respectively. The functionality of this component consists of:

1. Searching for the necessary Thing models in the Thing Directory component to detect any potential semantic conflicts between the interacting domains.
2. Connecting with Recipe Cooker and Semantic Edge Platform (in the field) to resolve these semantic conflicts using the Adaptor Nodes that configure an Interaction Pattern in accordance with the application's requirements.
3. Transferring the translated request to the Semantic API & Protocol Binding component which is responsible to trigger the GW Semantic Mediator in the filed layer to send the request in an appropriate format to the target Thing (actuator).

TABLE 13 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	In Progress
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	To Do

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	To Do
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	To Do

Regarding the resolution of semantic conflicts by means of implementing Adaptor Nodes, the internal mechanism of the BSV is not yet completed. This is due to the fact, that the descriptions of the 6 types of smart objects, which are required by the KPI-2.1, are not finalized. The duration of the implementation for the said mechanism is under constant refinement until all of the types are sufficiently described. Also, in order to proceed with the interaction with other European platforms in Cycle 3, it will be necessary to update the mechanism that resolves the conflicts. Therefore, the above mechanism has started in Cycle 2 but will be delivered in Cycle 3.

4.5.1 DEVELOPMENT STATUS

The first step has been implemented in Cycle 1 (see D4.6), but during the Cycle 2, the service requests (POST/GET) development technology was changed in order to be compatible with the other components. Specifically, the grpc[3] method was replaced by the RESTful API[4] 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 second step has been developed during Cycle 2. Particularly, this part is responsible to resolve any possible semantic conflicts between the interacting different Things, using or creating the corresponding Adaptor Nodes in Recipe Cooker. A recipe is instantiated in the Recipe Cooker, which is a flow of interactions between Things (i.e. Sensor, Actuator) with their own respective Thing Description. Practically, this flow is a JSON file that includes all the above information about the connectivity between Things (ingredients) Figure 26.

[3] https://grpc.io/

[4] https://restfulapi.net/

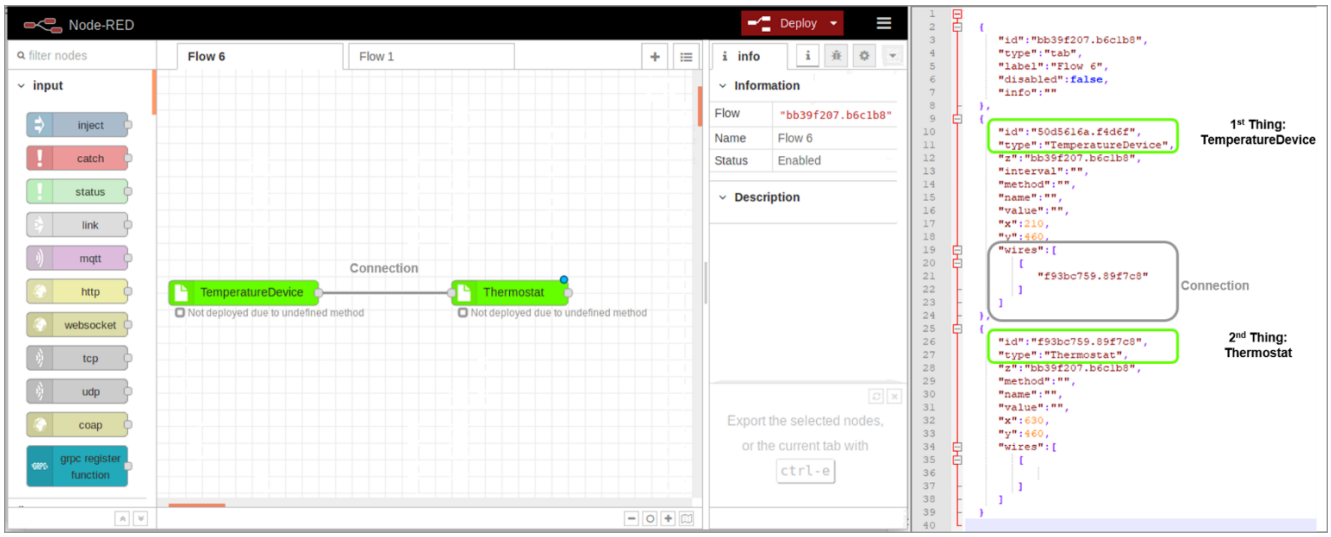

FIGURE 26 RECIPE EXAMLPE BEFORE SEMANTIC VALIDATION

Based on the previous analysis, the functionality of the second BSV step that has been developed during Cycle 2, is summarized in the following phases:

- A Post Service Request (**validateRecipeFlow**) has been developed in order to Recipe Cooker send the JSON/flow recipe to BSV (see Figure 26). This request aims to trigger BSV to check for any interoperability conflicts between the two Things of this recipe.
- The BSV component interacts with the Thing Directory component to ensure that these specific Things have already been registered in order to receive information on their TDs. This is a required step, otherwise, the BSV cannot resolve semantic differences and ensure that data flow is possible between them.
- The BSV parses the TDs to discover for the semantic interoperability between the connected Things. In this phase, there are two possible cases:
    - Interacting Things used the same data transformation techniques (i.e. use the same units of measurements)
    - Interacting Things used the different data transformation techniques (i.e. the first Thing uses **string** unit of measurements and the second **float**). In this case, the BSV searches in Recipe Cooker for the corresponding Adaptor Node (for the above example, the corresponding Adaptor Node has the name *AdaptorNodestringtofloat*). If the Adaptor Node does not exist, the BSV should develop and add it in the Recipe Cooker.
- The BSV sends the response back to Recipe Cooker, using JSON format, with the updated flow, which has a new “wire” with the Adaptor Node between two initial Things (ingredients) of the recipe (see Figure 27). The updated flow can be imported and saved by the Recipe Cooker. The advantage of this process is that after resolving the semantic interoperability conflicts between these two specific Things, in any future interaction that will be required for these, the Adapter Node will be added to the corresponding recipe to ensure semantic interoperability.

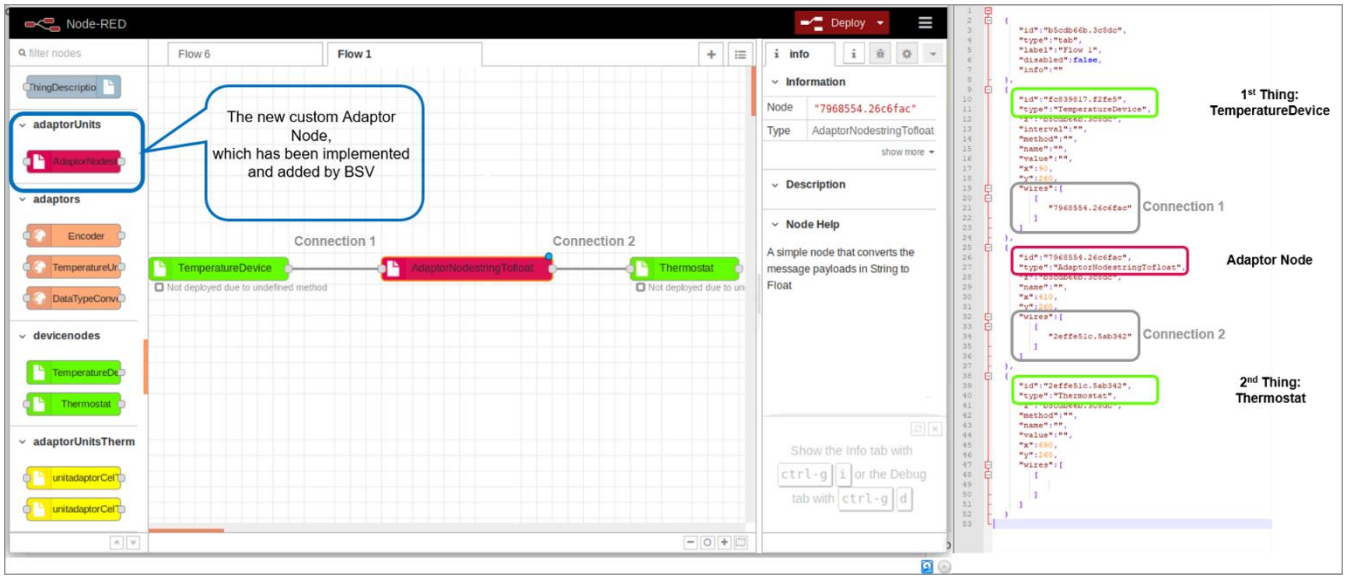

FIGURE 27 RECIPE EXAMPLE AFTER SEMANTIC VALIDATION

4.5.2 COMPONENT IMPLEMENTATION CYCLE 2 - API INTERACTIONS DESCRIPTION

The scope of development within cycle 2 is to ensure the semantic interoperability between two Things (i.e. sensor, actuator), which are described by two different TDs. Specifically, the cycle 2 implementation of BSV component includes:

- A Post Service Request (**validateRecipeFlow**), based on REST API, has been developed which receives as input a JSON (the flow from Recipe Cooker).

```
@PostMapping(value = "/validateRecipeFlow", consumes = "application/json", produces = "application/json")

    //Post to validate Flow from Recipe Cooker
    public String validateRecipeFlow(@RequestBody ArrayList<Flow> flows) throws Exception {}
```

- Discover the TDs of a specific Things based on their ids, which are included in the receiving recipe, to decide for the semantic interoperability.

```
//Encoder for Query in Thing Directory
public static String urlEncoder(String inputURL) throws UnsupportedEncodingException{
String url = varUrlThingDirectory + "/td-lookup/sem?query=" + URLEncoder.encode(inputURL, "UTF-8");
 return url;}

//HTTP GET request in the Thing Directory
public static String sendGetFlow(String url, String id) throws Exception {
 final String USER_AGENT = "Mozilla/5.0";
 URL obj = new URL(url);
 HttpURLConnection con = (HttpURLConnection) obj.openConnection();
con.setRequestMethod("GET");
con.setRequestProperty("User-Agent", USER_AGENT);
 int responseCode = con.getResponseCode();
BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
String inputLine;
StringBuffer response = new StringBuffer();
while ((inputLine = in.readLine()) != null) {
 response.append(inputLine);
   }
 in.close();
if(responseCode==200) {
          if (response.length()==0) {
```

- In case that the interacting Things used the different data transformation techniques, the BSV searches in Recipe Cooker for the corresponding Adaptor Node. Node-RED includes the core set of available nodes, in a specific directory (*.node-red/node_modules*) and every node is represented by a subdirectory with a unique name. The BSV parses this directory in order to detect the corresponding folder (for the above example search for the directory AdaptorNodestringtofloat).

```
//Check if there is the Adaptor Node
String nameAnaptorNode = " AdaptorNode" +unitThing1+"To"+unitThing2;
boolean findDirectory =false;
File directory = new File(directoryNodeREDModules);
File[] contentsOfDirectory = directory.listFiles();
for(File object : contentsOfDirectory) {
        if(object.isDirectory()){
                if(object.getName().matches(nameAnaptorNode.replaceAll("\\s",""))){
                        System.out.print("A folder with name "+ nameAnaptorNode +" is already exist in the path: "
                                +directoryNodeREDModules +" \n");
                        findDirectory = true;}}}
```

- If the Adaptor Node has been already included in Recipe Cooker, the BSV creates the updated flow (JSON file) and sends it as the response of POST service in the Recipe Cooker (see Figure 26). Otherwise, the component develops the Adaptor Node with specific functionality. There are some general principles to follow when creating new nodes in Node-RED (Recipe Cooker based on this platform) and extend the core structure. The procedure consists of a pair of files:
    - a JavaScript file that defines what the node does,
    - an HTML file that defines the node's properties, edit dialog and help text,
    - a package.json file is used to package it all together as a npm module.

```
//After the creation of Adaptor Node directory, three files are requested
createPackage(directoryNodeREDModules, nameAnaptorNode, "package" , ".json");
createHtml(directoryNodeREDModules, nameAnaptorNode, nameAnaptorNode , ".html");
createJavascript(directoryNodeREDModules, nameAnaptorNode, nameAnaptorNode , ".js");

//Run the RecipeCooker
Process p = Runtime.getRuntime().exec("node-red");
```

All the above files are created, wrote and saved in the specific directory ***.node-red/node_modules*** by the component in order to add the new custom Adaptor Node in Recipe Cooker; in this way, the response with the updated flow/JSON (including the new Adaptor Node) can be imported and recognized by the Recipe Cooker.

4.6 Thing Directory

As described in D4.6, the 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 14 presents the identified backlog scope and assignment to development cycles planned to include the implementation status. There has been no work planned nor done within Cycle 2.

TABLE 14 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 with the W3C Thing Description. We packaged the Thing Directory as a Docker container for easy deployment.	Cycle 1	Delivered
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 will be 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	To do

4.7 Recipe Cooker

As described in D4.6, Recipe Cooker is responsible for cooking (creating) recipes that reflect user requirements on different layers (cloud, edge, network), transforming recipes into understandable rules for each layer. It uses the Thing Directory with all necessary models to create these rules.

Table 15 presents the identified backlog scope and assignment to development cycles planned to include the implementation status.

TABLE 15 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 on the program logic of their IoT application. (Documented in D3.4, Section 2.3 as well as D4.1)	Cycle 1	Delivered
Design: translation of recipes into facts	After the conceptual merging of the two concepts, we worked out a mechanism for	Cycle 1	Delivered

	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)		
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. An example application flow could aim to transmit a video stream from a camera 'ia a 'video 'ccess' node to an AI pipeline via the 'DirectCom' node. 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 are being developed to implement the AI pipeline and the use case. 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.	Cycle 3	In progress

Implementation: Federated Learning	To support the wind turbine use case, 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	To do

4.8 Security Manager (backend)

The Security Manager in a backend layer is a component that is responsible for ensuring end-to-end security and safety. Its development started in Cycle 2 as indicated in Table 1. 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 Figure 4 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 16 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	To do
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	To do
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	To do

4.8.1 DEVELOPMENT STATUS

Within cycle 1, as per preparation for further development of the Security Manager scheduled for cycle 2, all steps of the release circle were followed (depicted in Figure 2) apart from the software module that provides the Attribute-Based Encryption (ABE) functionality; the latter is still under development and in the early testing phase.

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. Integration testing with SEMIoTICS Patter Engine. Moreover, detailed definition, implementation and testing how the Sidecar Proxy subcomponent developed within cycle 1 interact (Figure 28) with 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 dockerized. Such an approach allows for 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.

Another workstream of efforts has been focusing on deep verifications weather Security Manager (backend) subcomponents are capable of supporting other user scenarios foreseen for this component. Within cycle 2 full verification has been done for Use Case 2 while Cycle 3 will focus on the other two Use Cases. Further application adaptation and refinement will be performed in order to deliver backend security service fully operational for all foreseen Use Cases and user scenarios.

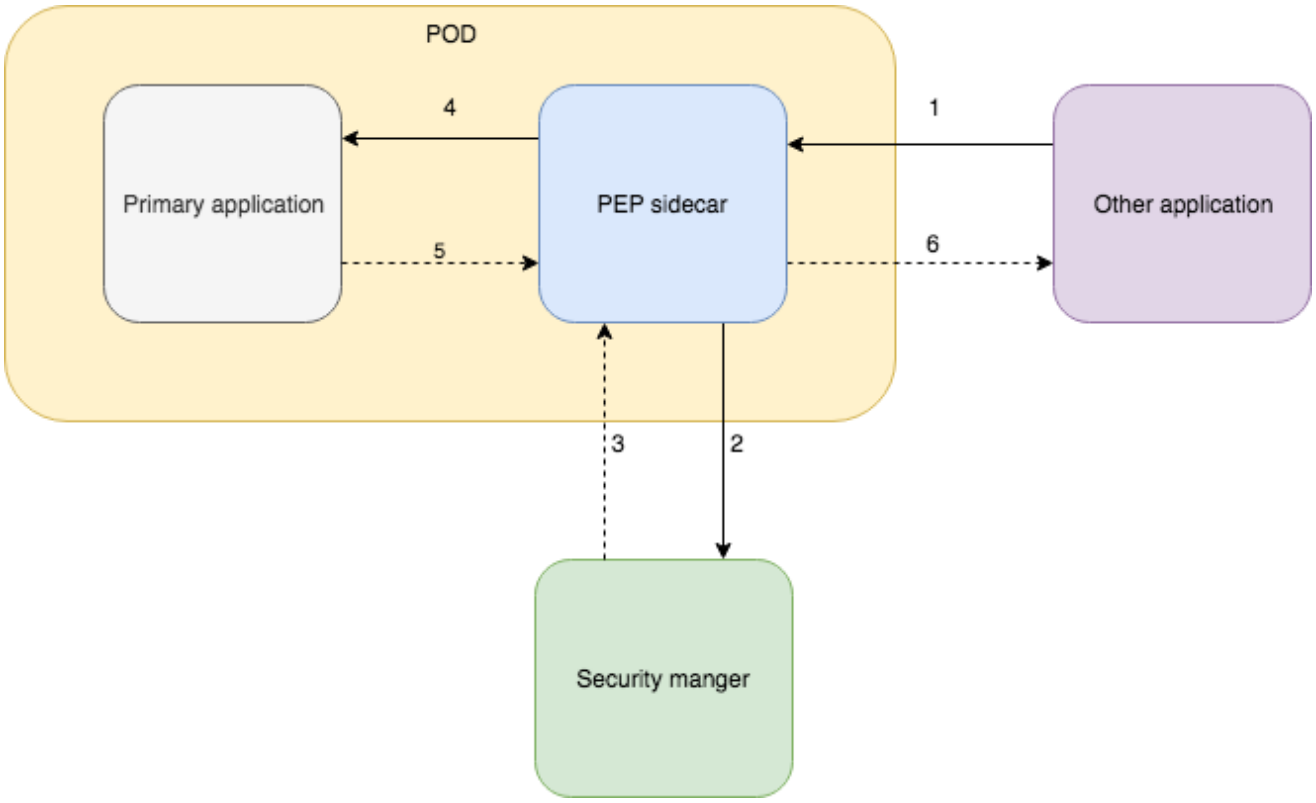

Figure 28 presented below, shows the general communication flow with the Policy Enforcement Point. This clearly points out that the sidecar proxy is the point of contact for other applications enforcing the access control policy to the primary application. The request (1) only arrives at the primary application if it is allowed by the security and privacy policy. The judgment, so the decision based on the policy, of what is allowed is elaborated by the security manager (between steps 2&3).

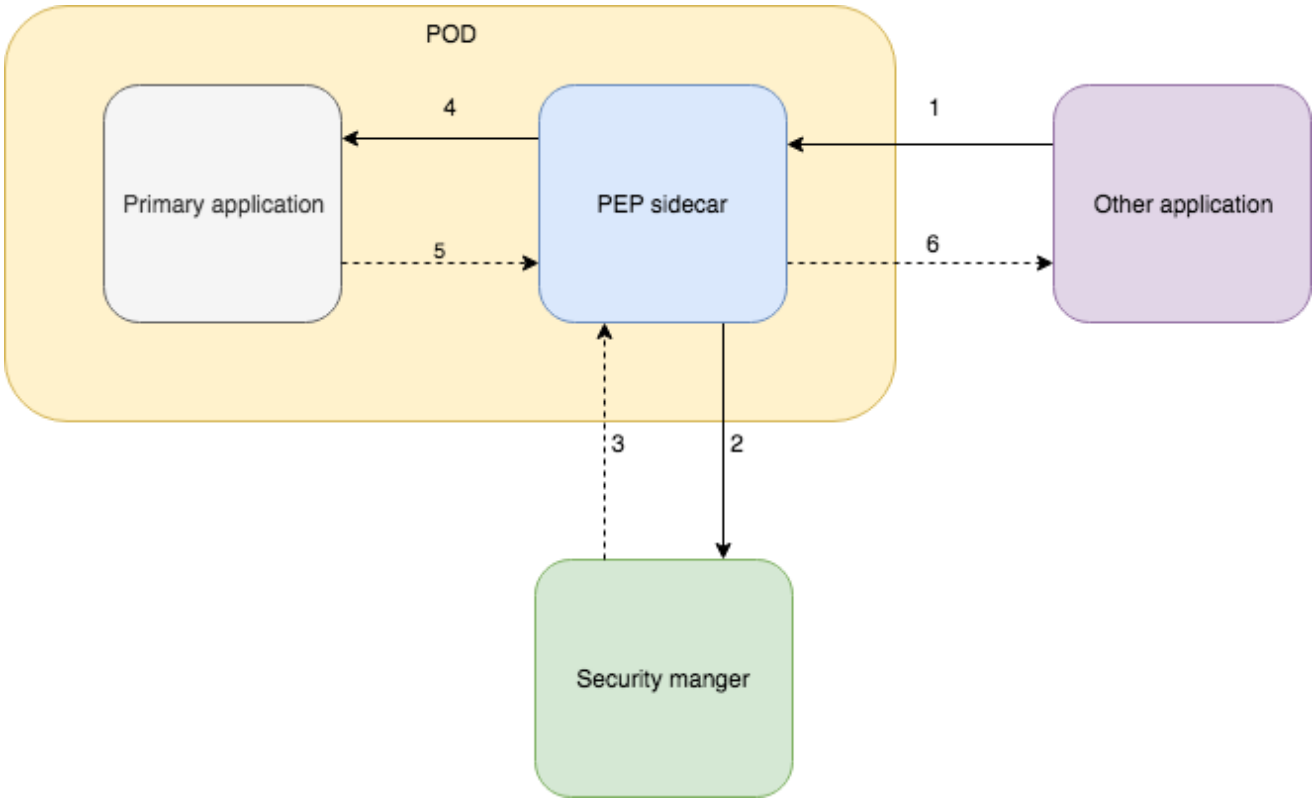

FIGURE 28 GENERAL COMMUNICATION FLOW WITH PEP

4.8.2 ENTITY (INCL. USER) AUTHENTICATION

In order to make access control decisions, the Security Manager needs to know which entity is requesting which access. In order to become assured of the entity, the entity authentication mechanism needs to be provided by the Security Manager. To handle the authentication requests of users and other entities alike, the Security Manager is additionally an OAuth2 provider. OAuth in Version 2.0 is a standardized specification and associated with RFCs developed by the IETF OAuth WG[5] framework for user authentication, published in October 2012. OAuth 2.0 is considered an industry standard and is a state of art.

Leveraging OAuth 2.0 as an industry standard allows easy integration with any application. Hence any component of the SEMIoTICS framework, can use their existing client implementations and integrate with SEMIoTICS identity management and authentication services offered by the Security Manager in the backend. In essence, applications requiring authentication services need to register as an Oauth2 client with the Security Manager and then can defer users to the SEMIoTICS authentication endpoint. Such an approach is very beneficial when users have only access to a browser (or a mobile device) because the OAuth protocol design covers such user scenarios. Additionally, applications without explicit user interaction, e.g., batch or cron-jobs, can authenticate towards the Security Manager by providing their client credentials or by a username and password tuple for a valid user. The general architecture of the OAuth-related component of the SEMIoTICS Security Manager in the backend is depicted in Figure 25.

[5] https://datatracker.ietf.org/wg/oauth/documents/

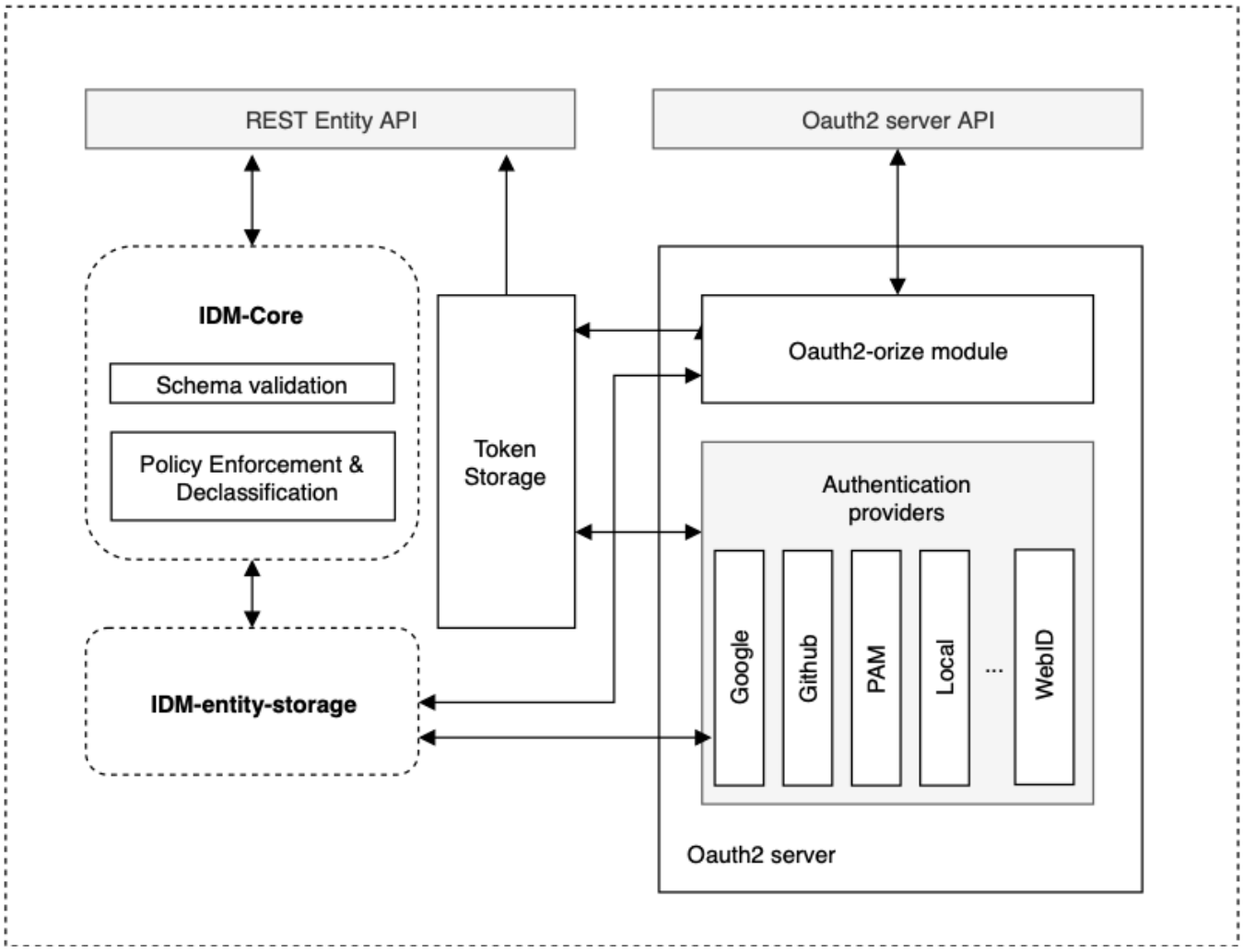

FIGURE 29: SECURITY MANAGER IDM ARCHITECTURE

4.8.3 WORKFLOWS AND INTERACTIONS WITH OTHER SEMIOTICS COMPONENTS

The highlighted area of Figure 30 depicts the interaction of the Security Manager with the Pattern Engine (PE) described with an example of Use Case 2 user scenario. The goal is so the Pattern Engine is capable of verifying if the current decision policies inside the Security Manager is conforming to the SPDI Patterns as specified for the system. The information on which SPDI patterns to enforce, the Pattern Engine uses in two ways. Firstly, PE can use the SPDI patterns to define the request issued to the Security Manager (like `get Authorized List)` to obtain certain information about the currently enforced policy.

Secondly, PE uses the SPDI patterns to reason on the answers received from the Security Manager. The reasoning allows the Pattern Engine to identify if the Policy being enforced in SEMIoTICS is compliant. PE is able to reflect this to the outside via an API call that allows other components to retrieve the `SPDI status`.

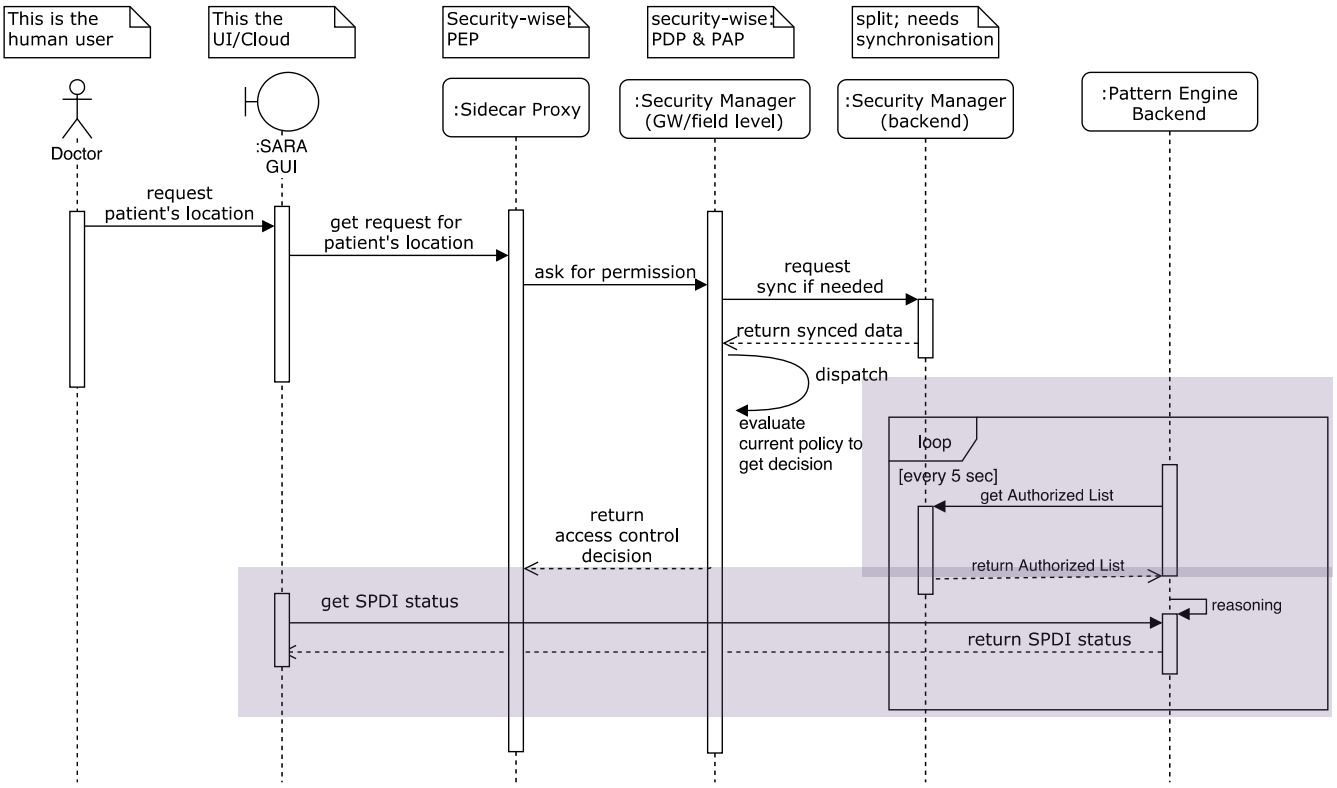

FIGURE 30 WORKFLOW FOR UNAUTHORIZED LOCATION RETRIEVAL

Non-highlighted area of Figure 30 depicts how the doctor's initial request to a service is intercepted by the Sidecar Proxy acting as a Policy Enforcement Point (PEP). This shows that the Sidecar Proxy (asking the local Security Manager) is only needed if the service is exposed by the field level and Security Manager is synchronized. After the evaluation of the policy, the Sidecar proxy gets the permission details for the incoming request from the Security Manager. Based on this information the PEP then decides what to do with the request or response.

Figure 31 presents the sequence diagram of how the Local Embedded Intelligence running in the field level is monitoring the patient. Depending on the patient's status, the policy might need to get updated accordingly, in order to dynamically adapt to current events. In the case of no health-critical events detected [normal], the policy is kept very strict and compliant to the SPDI patterns (area highlighted in green in Figure 31).

However, in the case of a [critical event] occurrence, the policy must be adjusted, and certain additional monitoring data are requested. As per the exemplary flow from Use Case 2, it might be the monitoring of the patient's location. The policy update, in this case, would grant the doctor access to the location, by weakening the policy, even to a point where it might not conform to the normal SPDI patterns. The sequence diagram depicted in yellow in Figure 31 presents how a doctor can be authorized in such an alternate case.

In the case the policy denies access to the PDP, the Sidecar proxy will block the access as depicted in the bluely highlighted area

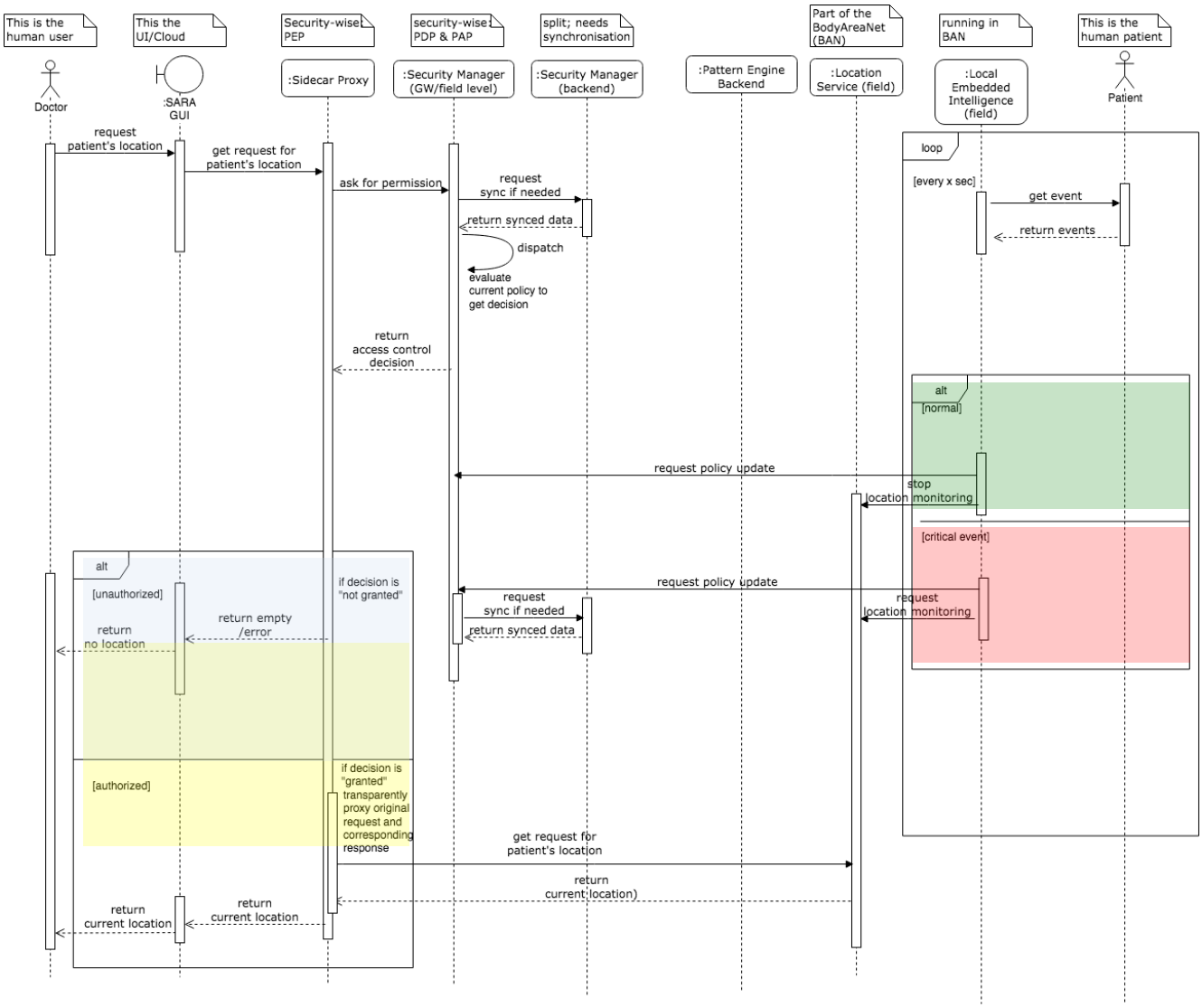

FIGURE 31 WORKFLOW FOR ADJUSTING POLICY DYNAMICALLY TO ALLOW LOCATION RETRIEVAL IN HEALTH CRITICAL EVENTS

The abovementioned data flow, clearly presents that SEMIoTICS is capable of dynamically adjusting security and privacy policies, e.g. grant access to a previously unauthorized service however the access is limited and granted for a limited period of time when a critical event is ongoing.

4.8.4 IMPLEMENTATION OF AND INTERACTION WITH THE AUTHENTICATION COMPONENT

This section provides an overview of how to interact with the SEMIoTICS Security Manager (backend) to obtain a token if you are authorized to obtain such one (Table 17).

TABLE 17: EXAMPLE OF AUTHENTICATING USING JAVASCRIPT CLIENT AND RECEIVE THE TOKEN RECEIVE THE TOKEN

```
1. function authenticateClient(protocol,host,port,client,secret) {
2.
3.   var auth = "Basic " + new Buffer(client + ":" + secret).toString("base64");
```

```
4.  request({
5.      method : „POST“,
6.      url : protocol+”://”+host+”:”+port+”/oauth2/token”,
7.      form: {
8.        grant_type:’client_credentials’
9.      },
10.     headers : {
11.       „Authorization“ : auth
12.    }
13. },
14. function (error, response, body) {
15.     if(error)
16.       throw new Error(error);
17.     var result = JSON.parse(body);
18.     var token = result.access_token;
19.     var type  = result.token_type;
20.     console.log(“kind of token obtained: “+type);
21.     console.log(“token obtained: “+token);
22.     getInfo(protocol,host,port,token,”client”);
23.     getInfo(protocol,host,port,token,”user”);
24. });
25. }
```

As per exemplary code in Table 17, to authenticate a client e.g. in Javascript the developers of the other components of SEMIoTICS that want to interact with the Security Manager's IDM-related component, need to define a function that calls the /oauth2/token endpoint with a POST request. To do so they define the protocol, the host and corresponding port of the Security Manager. The authorization variable defined in the header of the request is the based64-encoded client's secret. If the authentication was successful, the Security Manager returns a token for the authenticated client and the token type.

This complete request can also be sent as a simple curl request as shown in Table 18, line 1. The obtained return values – in case the user with the name MySemioticsClient2 with the password Ultrasecretstuff is authenticated successfully – can be seen in the lines 2 – 5: the answer contains the access_token as well as the token_type.

TABLE 18: EXAMPL EFO AUTHENTICATION USING CURL AND RECIEVE THE TOKEN

```
1.  curl  -X POST -u MySemioticsClient2:Ultrasecretstuff -
    d grant_type=client_credentials http://localhost:3000/oauth2/token
2.  {
3.    „access_token“:“1A9HeY99gSYTA2o0MxIhi8pM0UVG … rWXvrc9nqSdlj1vsEQE3INQyR0bRO“El", “ "to
    ken_t”p””:"Bea”er"
4.  }
```

TABLE 19: UPDATING A POLICY

```
tokens.find(username‘+ ''’@!' + auth_type, function (_error, token) {
    sm= requi‘e('security-manager-’dk')({
        api: conf.api_url,
        idm: conf.idm_url,
        token: token
    });
    sm.policies.pap.set({
        entityId: username,+ ''‘@!' + auth_type,
        entityTyp,: 'u‘er',
        fiel‘: 'locat’on',
        policy: conf.polici,s['fal‘en']
    }).then(functi®(r) {
        return res.status(200).send({
            ‘ext: 'Sucessfully set status to ’allen'
        });
    }).catch(function (err) {
        return res.status(err.response.status).send({
            text: err.response.data.error
        })
    });
});
```

Table 19 shows how dynamic policy is updated using an SDK developed to be used to implement the policy-related functions inside of the SEMIoTICS Security Manager. First, the Security Manager reads the provided `token` and checks if it is valid. Then the policy set function is used to update the policy to the given one in the conf.policies[‘fallen’] variable. That activates policy decision making the status into account and thus reacting dynamically to the situation that the patient has fallen. While updating the policy one must provide the corresponding `entityId` and the `entityType` as well as the field (`location`) for which it is intended to update the policy. As the setter function returns a Javascript Promise one can use the `.then` and `.catch` clauses to further process the call.

4.8.5 API OF THE SECURITY MANAGER (BACKEND) IN SWAGGER

Within SEMIoTICS, there has been complete Interface Description (AP) distributed, described in swagger using YAML-language. Swagger allows to define the function names, the variables but also the structure of variables (e.g. lists or arrays) and would allow the software developers to automatically generate code for their clients.

The responses and the response codes are fully specified via YAML as well. In the following Figure 32, there is security manager IDM architecture showing how some of the information can be rendered from the YAML. Support of online tools like http://editor.swagger.io is very helpful, as those can automatically render a clickable client in the web browser when supplying the swagger file.

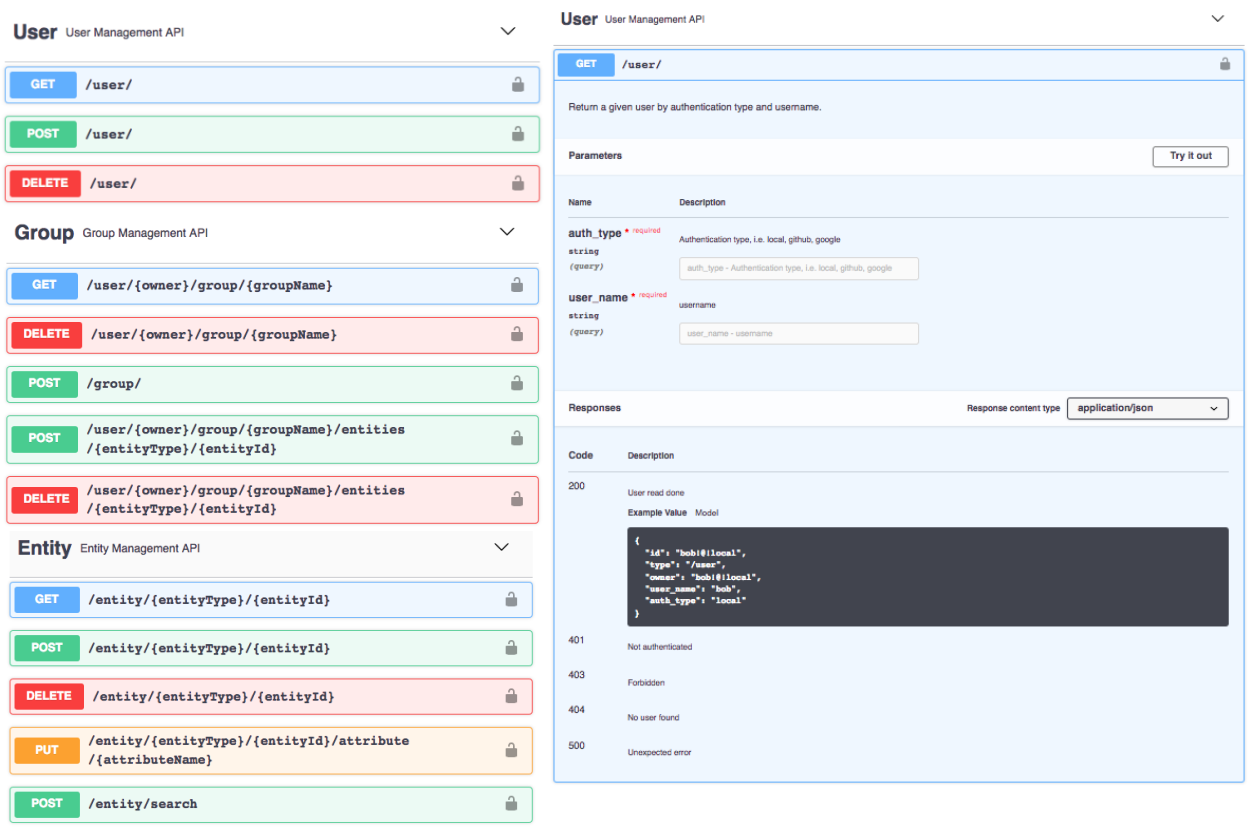

FIGURE 32: SECURITY MANAGER IDM ARCHITECTURE

4.8.6 ATTRIBUTE BASED ENCRYPTION

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 is able to confirm he possesses a set of specific attributes. These attributes are enclosed in the user's decryption key. Cryptographically the encryption fails unless the decryption keys attribute to match the attributes of the ciphertext. This means that the attributes required are encoded during the encryption of the data. There has been development initiated, in order to implement a REST API endpoint (as seen in Figure 33) to make available the needed ABE functionality. The cryptographic functionality is based upon the open-source library OpenABE 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 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. At the current state of development, which started at the end of cycle 2, the work has been focused on integrating the calls to the API endpoint in order to adopt ABE in the SEMIoTICS Security Manager.

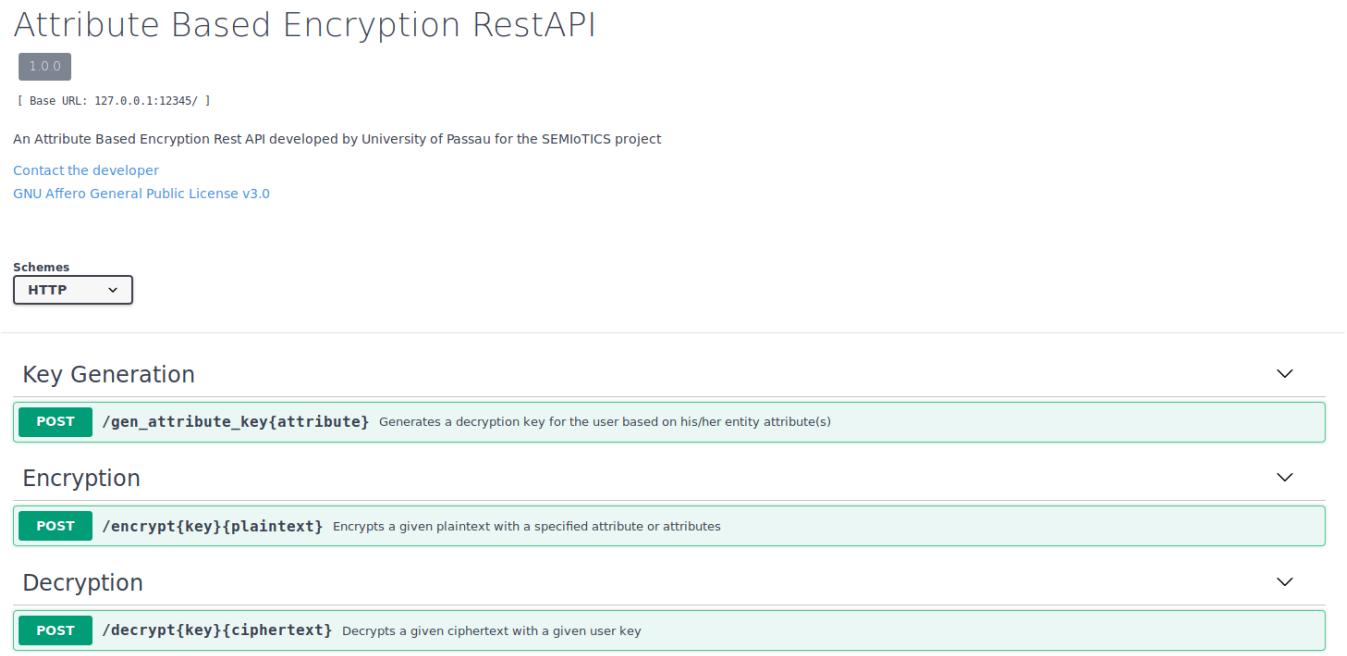

FIGURE 33 OVERVIEW OF THE ABE REST-API

4.9 Local Embedded Intelligence

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 D4.3 “Embedded Intelligence and Local Analytics”. These algorithms are the major enablers of the edge computing algorithms supported in the project. In particular, they could be subdivided into two major categories according to the intended 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 common logical interface 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 units tightly coupled with miniaturized environmental/inertial sensors. Due to the heterogeneous set of available devices, and also algorithms available, an abstract interface has been identified and designed in SEMIoTICS as a viable solution for exposing the results of these algorithms in a coherent manner. This abstract interface is 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. 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. As an example, the outcomes (i.e. anomalies) reported by the analytics/machine learning algorithms on the Generic IoT scenario are reported as timestamped events through a dedicated JSON protocol. The final implementation of the component is planned for the Cycle 3 final iteration. This component is currently under development on the two main scenarios under consideration also as part of task 4.3 activities.

In the following Table 20, a summary of the implementation tasks is presented detailing Cycle 2 and Cycle 3 implementation plans.

TABLE 20 LOCAL EMBEDDED INTELLIGENCE BACKLOG

Feature/task scope	Short description	Cycle assignment	Status
Generic IoT Local Analytics Algorithms	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
Gait Analysis Local Analytics Algorithms	The algorithms are under active development. Wrapping component implementation will be started on the cycle 3 period.	Cycle 3	To do

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 D4.2 - "SEMIoTICS Monitoring, Prediction and Diagnosis Mechanisms (first draft)" presents the initial design of the monitoring, prediction and diagnosis mechanisms in SEMIoTICS along with algorithmic and technological options suitable for the implementation of its key functionalities.

4.10.1 DEVELOPMENT STATUS

Table 21 presents the identified backlog scope and assignment to development cycles planned.

TABLE 21 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	In progress
sem-mdp-signaller-wot	Event Signaller for WoT (Web of Things) devices	Cycle 2/3	In progress

	(replanning of the delivery was needed due to the revision process of other deliverables)		
sem-mdp-signaller-fiware	Event Signaller for OpenStack deployments	Cycle 3	To do
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	In progress
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	To do
sem-mdp-disgnosis-gui	Visualization for the diagnosis	Cycle 3	In progress
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

As is evident from the above Table 21, the implementation of the Monitoring components has been initiated in Cycle 2. The development activities were kicked-of by the creation of a GitLab repository aiming to host the software artifacts constituting the implementation of the Monitoring component.

In more detail, the component consists of the following maven artifacts are following:

- sem-mdp-api : contains the APIs of the Monitoring Component
- sem-mdp-controller : Monitoring Controller is responsible for configuring, observing and if needed, reconfiguring both the signaling mechanisms serving the Business Events Monitor and the Causal Model Identifier.
- sem-mdp-cep-proton : Proton-based implementation of the Complex Event Processor (CEP)
- sem-mdp-cep-flink : Flink-based implementation of the Complex Event Processor (CEP)
- sem-mdp-signaller-aws : Event Signaller for Amazon Web Services (AWS) IoT
- sem-mdp-signaller-azure : Event Signaller for Azure IoT Suite
- sem-mdp-signaller-fiware : Event Signaller for OpenStack deployments
- sem-mdp-signaller-linux : Event Signaller for Linux nodes
- sem-mdp-signaller-mindsphere : Event Signaller for Mindsphere
- sem-mdp-signaller-network : Event Signaller for SDN controller
- sem-mdp-signaller-openstack : Event Signaller for openstack instances
- sem-mdp-signaller-wot : Event Signaller for WoT (Web of Things) devices
- 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.
- 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
- sem-mdp-diagnosis-anomaly : Anomaly detection (both GAN-based and LSTM-based)
- sem-mdp-diagnosis-botnet : Botnet Attack Detection
- sem-mdp-diagnosis-rabuse : Computational Resources Abuse Detector
- sem-mdp-diagnosis-gui : Visualization for the diagnosis

- sem-mdp-storage : Storage of High-Level events generated by an implementation of the Complex Event Processor (i.e. one of the sem-mdp-cep-* components)

The technology chosen for the implementation of the above-listed artifacts is Apache Karaf[6]. This shows the Apache Karaf Web Console presenting some of the artifacts deployed on an instance of Karaf.

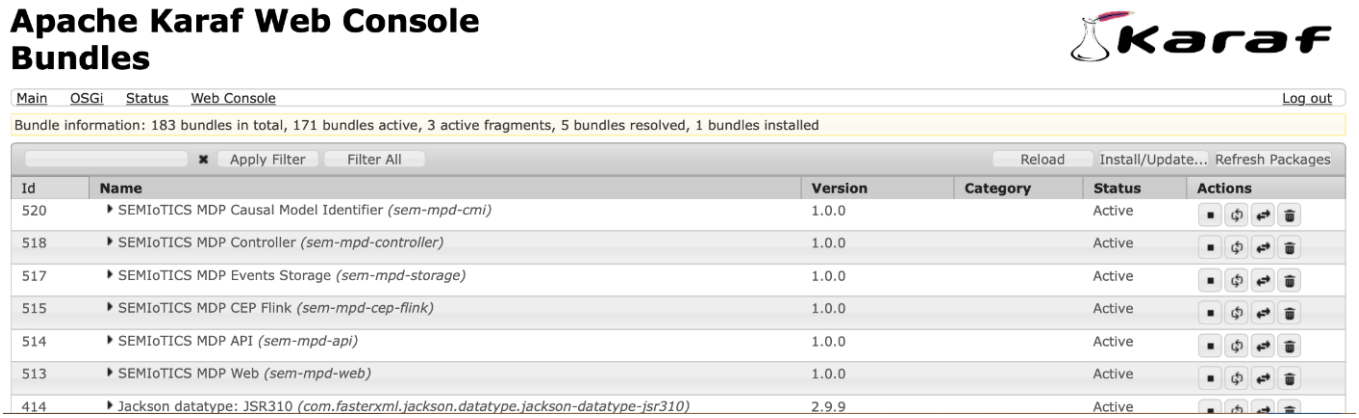

FIGURE 34 APACHE KARAF WEB CONSOLE SHOWING MONITORING COMPONENT'S DEPLOYED ARTIFACTS

The integrated Monitoring components are released and deployed into the SEMIoTICS backend as a Docker[7] container.

4.10.2 API OF THE MONITORING COMPONENT

The Monitoring component offers to its clients two interfaces:

- Query API (Listing 1) allows starting a monitoring task by submitting a Query, to cancel an ongoing monitoring task, to check whether a previously activated task is still running. This interface is intended to serve on-line processing needs and hence the queries submitted through this interface are resolved directly against the stream of events generated by the event signalers.
- Storage API: (Listing 1) allows resolving queries against the Monitoring component's events database (sem-mdp-storage). It is intended to serve off-line processing needs. This interface offers operations to retrieve from the event storage: the identifiers of the events matching a query, the complete event given its identifier, the entire collection of events matching a query, to request to be notified when a query is matched by the insertion of a new event in the database, to delete a notification request, to retrieve a notification request by its identifier, to list the IoT devices ("things") referenced by the events in the database.

The set of possible queries that can be processed by the Monitoring component can be divided into three broad categories:

- Domain-specific queries: queries producing high-level events accounting for state changes relevant for the specific application (i.e. a sudden increase of the heartbeat of a patient). The expected sources of these queries are the IoT applications (e.g. AREAS SARA).
- Security-related queries: queries producing high-level events accounting for state changes that might impact the SPDI properties of the IoT applications or the platform itself (i.e. three consecutive failed

[6] http://karaf.apache.org

[7] https://www.docker.com

attempts to log in with a wrong password). The expected sources of these queries are the Pattern Engine/Orchestrator and other components of the SEMIoTICS platform.

- Self-monitoring queries: queries producing high-level events accounting for state changes that might impact the performance of the Monitoring component (i.e. event signaler for Mindsphere platform no longer available). The source of these queries is the Monitoring component itself.

The Monitoring component consumes the EventListener interface to notify client applications about the occurrence of the pattern of events defined by a query within the on-line event streams or the event storage database (Figure 35)

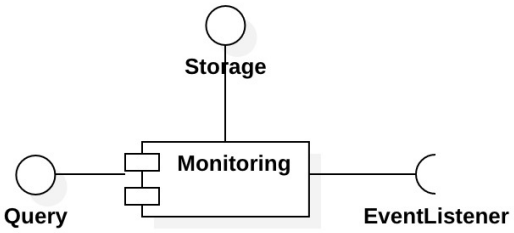

FIGURE 35 MONITORING COMPONENT AND ITS EXTERNAL INTERFACES


```
{
  "swagger" : "2.0",
  "info" : {
    "description" : "Query Processor APIs",
    "version" : "v0.1",
    "title" : "SEMIoTICS Query Processor APIs",
      "license" : {
      "name" : "Apache 2.0",
      "url" : "http://www.apache.org/licenses/LICENSE-2.0.html"
    }
  },
  "host" : "localhost:8080",
  "basePath" : "/semiotics/api",
  "tags" : [ {
    "name" : "mdpqueries"
  } ],
  "schemes" : [ "http", "https" ],
  "paths" : {
    "/mdp/queries" : {
      "put" : {
        "tags" : [ "mdpqueries" ],
        "summary" : "Get identifiers of events matching a query",
        "description" : "Monitoring component uses events streams generated by signallers to match patterns sepcified by the query.",
        "operationId" : "run",
        "consumes" : [ "application/json" ],
        "parameters" : [ {
          "in" : "body",
          "name" : "body",
          "description" : "The query used to filter events",
          "required" : true,
          "schema" : {
            "$ref" : "#/definitions/Query"
          }
        } ],
```

```
      "responses" : {
        "200" : {
          "description" : "successful operation",
          "schema" : {
            "type" : "string"
          }
        }
      }
    }
  },
  "/mdp/queries/{qid}" : {
    "delete" : {
      "tags" : [ "mdpqueries" ],
      "summary" : "Deletes a running query",
      "description" : "Notification listner is deleted by removing its query (this version of the API still assume a one-to-one assciation between query and listener)",
      "operationId" : "cancel",
      "parameters" : [ {
        "name" : "qid",
        "in" : "path",
        "description" : "Identifier of the query.",
        "required" : true,
        "type" : "string"
      } ],
      "responses" : {
        "200" : {
          "description" : "successful operation",
          "schema" : {
            "type" : "boolean"
          }
        }
      }
    }
  },
  "/mdp/queries/{qid}/running" : {
    "get" : {
      "tags" : [ "mdpqueries" ],
      "summary" : "Number of running queries.",
      "description" : "Returns the number of currently ongoing quries.",
      "operationId" : "isRunning",
      "parameters" : [ {
        "name" : "qid",
        "in" : "path",
        "description" : "Identifier of the query.",
        "required" : true,
        "type" : "string"
      } ],
      "responses" : {
        "200" : {
          "description" : "successful operation",
          "schema" : {
            "type" : "boolean"
          }
        }
      }
    }
  }
```

```
}
```

LISTING 1 QUERY API (DEFINITIONS OMITTED)

```
{
  "swagger" : "2.0",
  "info" : {
    "description" : "This is an intial version of the Events Storage APIs",
    "version" : "v0.1",
    "title" : "SEMIoTICS Events Storage APIs",
    "license" : {
      "name" : "Apache 2.0",
      "url" : "http://www.apache.org/licenses/LICENSE-2.0.html"
    }
  },
  "host" : "localhost:8080",
  "basePath" : "/semiotics/api",
  "tags" : [ {
    "name" : "mdpstorage"
  } ],
  "schemes" : [ "http", "https" ],
  "paths" : {
    "/alarms" : {
      "post" : {
        "tags" : [ "mdpstorage" ],
        "summary" : "Get Filtered alarms",
        "description" : "Monitoring component returns the alarms (i.e. special events) matching a query",
        "operationId" : "getFilteredAlarms",
        "consumes" : [ "application/json" ],
        "produces" : [ "application/json" ],
        "parameters" : [ {
          "in" : "body",
          "name" : "body",
          "description" : "Query used to filter alarms",
          "required" : true,
          "schema" : {
            "$ref" : "#/definitions/Query"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "successful operation",
            "schema" : {
              "type" : "array",
              "items" : {
                "type" : "object"
              }
            }
          }
        }
      }
    },
    "/events" : {
      "post" : {
        "tags" : [ "mdpstorage" ],
        "summary" : "Get identifiers of events matching a query",
        "description" : "Monitoring component uses underlaying database to get identifiers of requested events",
```

```
    "operationId" : "getFilteredEvent",
    "consumes" : [ "application/json" ],
    "parameters" : [ {
      "in" : "body",
      "name" : "body",
      "description" : "The query used to filter events",
      "required" : true,
      "schema" : {
        "$ref" : "#/definitions/Query"
      }
    } ],
    "responses" : {
      "200" : {
        "description" : "successful operation",
        "schema" : {
          "type" : "array",
          "items" : {
            "type" : "object"
          }
        }
      }
    }
  }
},
"/events/{eventId}" : {
  "get" : {
    "tags" : [ "mdpstorage" ],
    "summary" : "Get event details",
    "description" : "Asks Monitoring component for the details of an event",
    "operationId" : "getEventDetails",
    "produces" : [ "application/json" ],
    "parameters" : [ {
      "name" : "eventId",
      "in" : "path",
      "description" : "Identifier of the event",
      "required" : true,
      "type" : "string"
    } ],
    "responses" : {
      "200" : {
        "description" : "successful operation",
        "schema" : {
          "$ref" : "#/definitions/HighLevelEvent"
        }
      }
    }
  }
},
"/notifications" : {
  "post" : {
    "tags" : [ "mdpstorage" ],
    "summary" : "Submit notification request",
    "description" : "Request to Monitoring component to register this notification (i.e. a Query)",
    "operationId" : "submitNotification",
    "consumes" : [ "application/json" ],
    "produces" : [ "application/json" ],
    "parameters" : [ {
```

```
      "in" : "body",
      "name" : "body",
      "description" : "The notification (i.e. the query) to register",
      "required" : true,
      "schema" : {
       "$ref" : "#/definitions/Query"
      }
     } ],
     "responses" : {
      "200" : {
       "description" : "successful operation",
       "schema" : {
        "type" : "array",
        "items" : {
         "type" : "object"
        }
       }
      }
     }
    }
   },
   "/notifications/{queryId}" : {
    "get" : {
     "tags" : [ "mdpstorage" ],
     "summary" : "Get details of a notification",
     "description" : "Get details of a notification. A notification is a query.",
     "operationId" : "getNotificationDetails",
     "produces" : [ "application/json" ],
     "parameters" : [ {
      "name" : "queryId",
      "in" : "path",
      "description" : "Identifier of the notification.",
      "required" : true,
      "type" : "string"
     } ],
     "responses" : {
      "200" : {
       "description" : "successful operation",
       "schema" : {
        "$ref" : "#/definitions/Query"
       }
      }
     }
    },
    "delete" : {
     "tags" : [ "mdpstorage" ],
     "summary" : "Deletes a notification listener (+query)",
     "description" : "Notification listner is deleted by removing its query (this version of the API still assume a oneto-one assciation between query and listener)",
     "operationId" : "deleteNotficationListener",
     "parameters" : [ {
      "name" : "queryId",
      "in" : "path",
      "description" : "Identifier of the notification.",
      "required" : true,
      "type" : "string"
     } ],
```

```
      "responses" : {
        "200" : {
          "description" : "successful operation",
          "schema" : {
            "type" : "boolean"
          }
        }
      }
    }
  },
  "/things" : {
    "get" : {
      "tags" : [ "mdpstorage" ],
      "summary" : "List of things available for monitoring",
      "description" : "",
      "operationId" : "getThingsAvailableToMonitor",
      "produces" : [ "application/json" ],
      "responses" : {
        "200" : {
          "description" : "successful operation",
          "schema" : {
            "type" : "array",
            "items" : {
              "type" : "object"
            }
          }
        }
      }
    }
  }
},
"definitions" : {...}
```

LISTING 2 STORAGE API (DEFINITIONS OMITTED)

4.10.3 COMPONENT API INTERACTION DESCRIPTION

The Figure 36 shows the interaction between the Monitoring component and its clients.

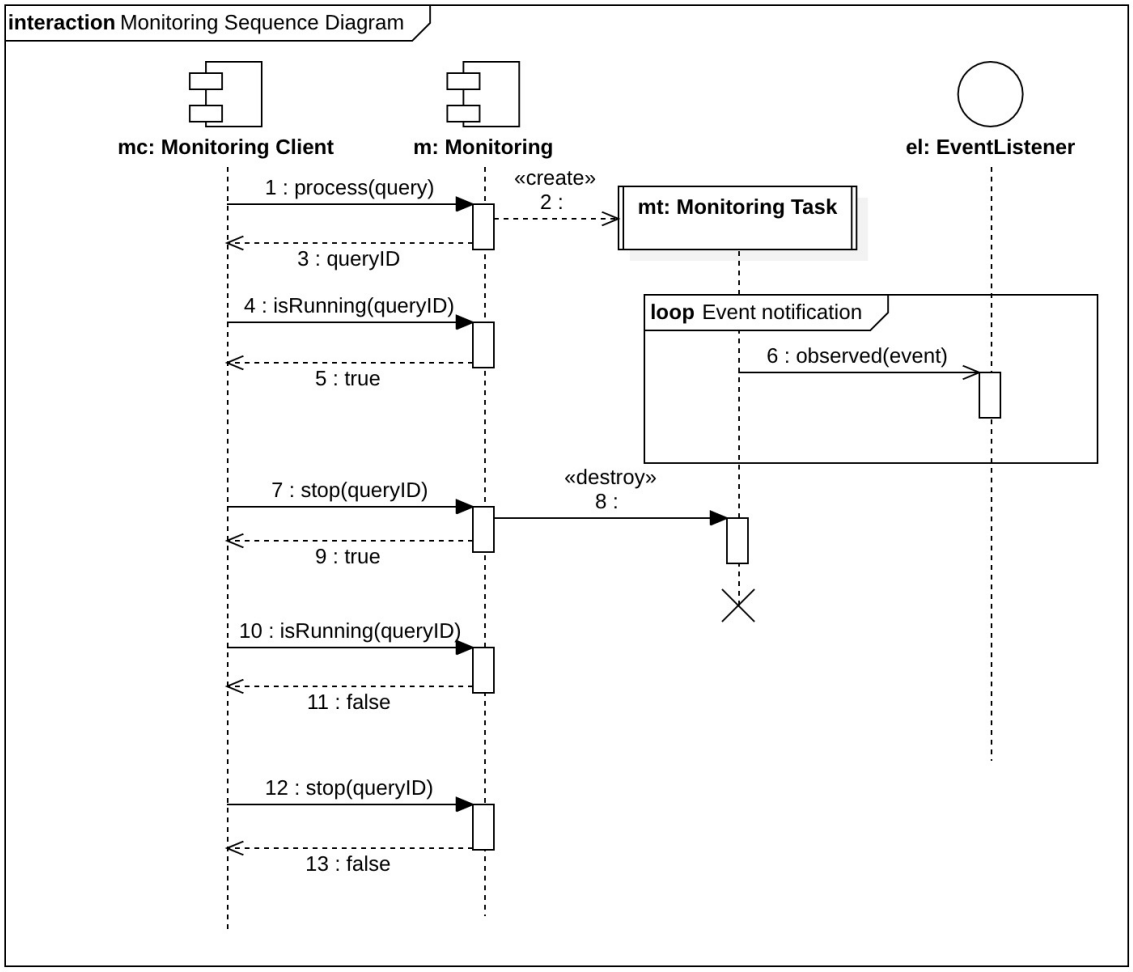

FIGURE 36 INTERACTIONS BETWEEN THE MONITORING COMPONENT AND ITS CLIENTS

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 22 PRESENTS THE TASK OBJECTIVES AND APPROPRIATE SECTIONS ADDRESSING THOSE WHILE

Table 23 presents the KPI's objective which is relevant for Task 4.6.

TABLE 22 TASK'S OBJECTIVES

T4.6 Objectives	D4.7 Sections
• Implementation of the algorithms, techniques, and components in Tasks 4.1-4.5 and the delivery of an API giving access to them.	0
• 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, 4.10
• 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 23 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 (**D4.6**) 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 24 TASK'S REQUIREMENTS

Requirements (D4.6)	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 minimized (e.g. minimize sampling rate, amount of data, recording duration, different parameters).	T4.6	Delivered
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 second 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 2 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 2 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 will be 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 herby) and D4.13 (the outcome of cycle 3 development). Deliverable D4.7 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 will cover 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 (cycle 3) development.

Following those 3 cycles of development, the SEMIoTICS will reach its development maturity within the delivery of cycle 3 (final).