SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Project Number:	**FP7–SMARTCITIES–2013(ICT)**
Project Acronym:	**VITAL**
Project Number:	**608682**
Project Title:	**Virtualized programmable InTerfAces for innovative cost-effective IoT depLoyments in smart cities**

D3.2.3 Specification and Implementation of Virtualized Unified Access Interfaces V3

Document Id:	VITAL-D323-250216-Draft
File Name:	VITAL-D323-250216-Draft.pdf
Document reference:	Deliverable 3.2.3
Version :	Draft
Editor :	John Soldatos, Katerina Roukounaki
Organisation :	AIT
Date :	25 / 02 / 2016
Document type:	Deliverable
Security:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V10	John Soldatos	AIT	22/11/2015	Table of Contents, Structure and Enhancement to previous versions
V11	John Soldatos	AIT	22/11/2015	Updates to Introduction
V12	John Soldatos	AIT	25/11/2015	Section 2
V13	Salvatore Guzzo Bonifacio	INRIA	21/12/2015	Updates to Service Discovery and Filtering
V14	Lorenzo Bracco, Paola Dal Zovo	Reply	22/12/2015	Updates to 6.5 (Hi Reply PPI implementation) and Security in Section 5
V15	Aqeel Kazmi	NUIG	22/12/2015	Updates to Section 4.1 (DMS)
V16	Elisa Herrmann	ATOS	12/01/2015	Updates to Section 4.4 (CEP)
V17	Lorenzo Bracco, Paola Dal Zovo	Reply	15/01/2016	Updates to data access control
V18	Angelos Lenis	SiLo	16/01/2016	Updates to Section 4.5
V19	Katerina Roukounaki	AIT	27/01/2016	Updates to Sections 3 and 6
V20	Elisa Herrmann	Atos	28/01/2016	Updates to section 4.4
V21	Riccardo Petrolo	INRIA	28/01/2016	Updates to section 4.2 and 4.3
V22	Katerina Roukounaki	AIT	30/01/2016	Architecture and sequence diagrams
V23	Aqeel Kazmi	NUIG	02/02/2016	Technical Review
V24	Katerina Roukounaki	AIT	08/02/2016	Make changes to all services
V25	Katerina Roukounaki	AIT	11/02/2016	Make changes to section 6.3
V26	Aqeel Kazmi	NUIG	16/02/2016	Technical Review and updates to DMS section, circulated for approval
Draft	Martin Serrano	NUIG	25/02/2016	Ready for EC submission.

OVERVIEW OF UPDATES/ENHANCEMENTS OVER D3.2.2

Section	Description
Section 1	Introductory section. Updates to the description of the summary and scope of the final version of the deliverable.
Section 2	Alignment of the section with the final version of the VITAL architecture.
Section 3	Updates to the PPI specification.
Section 4	Finalisation of all interfaces.
Section 5	Description of the security architecture.
Section 6	Description of PPIs for various systems.

TABLE OF CONTENTS

DOCUMENT HISTORY **1**

OVERVIEW OF UPDATES/ENHANCEMENTS OVER D3.2.2 ..... **2**

1 INTRODUCTION **8**

1.1 Scope 8
1.2 Audience 9
1.3 Summary 9
1.4 Structure 10

2 VIRTUALIZED UNIFIED ACCESS INTERFACES **11**

2.1 Purpose and Positioning within the VITAL Architecture 11
2.2 Types of VUAIs and Functional Scope 12

3 PLATFORM PROVIDER INTERFACE SPECIFICATION **14**

3.1 Overview 14

3.2 PPI Specification 14
3.2.1 Access to IoT System Metadata 15
3.2.2 Access to IoT Service Metadata 17
3.2.3 Access to Sensor Metadata 19
3.2.4 Access to Sensor Observations 22
3.2.4.1 Pull-based mechanism 22
3.2.4.2 Push-based mechanism 23
3.2.5 Configuration 25
3.2.6 Monitoring 27
3.2.6.1 Performance Metric Monitoring 28
3.2.6.2 SLA parameter monitoring 30

3.3 IoT Data Adapter 32
3.4 Standalone Sensors 36

4 VIRTUALIZED ACCESS TO VITAL MODULES **38**

4.1 Interfaces to Data Management Service 38
4.1.1 Store Metadata and Observations 39
4.1.2 Query Metadata and Observations 43

4.2 Interfaces to Service Discovery 46
4.3 Interface to Filtering 50
4.4 Interfaces to CEP Module 53
4.5 Interfaces to Workflow Management 59

5 SECURITY OF VIRTUALIZED UNIFIED ACCESS INTERFACES 70

5.1 Overview of VUAI Security Architecture and Implementation 71
5.1.1 Security Modules and Libraries 71
5.1.2 Authentication 74
5.1.3 Authorization 75

5.2 Example: PPI Authentication and Authorization 79
5.3 Example: VUAI Authentication and Authorization 80

6 PPI PROTOTYPE IMPLEMENTATIONS 81

6.1 PPI Implementation for Open Data Sources 81
6.1.1 PPI Implementation for Footfall Data Feed (Camden) 81
6.1.2 PPI Implementation for Bus Arrival Data Feed 82

6.2 PPI Implementation for X-GSN 83
6.3 PPI Implementation for Xively 84
6.4 PPI Implementation for FIT/IoT-LAB 85
6.4.1 Implementation Overview 85
6.4.2 Examples 86

6.5 PPI Implementation for Hi Reply 86
6.5.1 Implementation Overview 86
6.5.1.1 Hi Reply Client 87
6.5.1.2 Core & Rest API 88
6.5.2 Examples 89

7 CONCLUSIONS 90

8 REFERENCES 90

LIST OF FIGURES

FIGURE 1: OVERVIEW OF THE VITAL ARCHITECTURE. 11
FIGURE 2: TWO MAIN OPTIONS OFFERED BY THE VITAL VIRTUALIZATION LAYER (VUAIS) REGARDING IOT DATA ACCESS. . 12
FIGURE 3: OVERVIEW OF THE VUAIS PROVIDED BY THE VITAL PLATFORM. 13
FIGURE 4: CLASSIFICATION OF THE TYPES OF INFORMATION ACCESSED VIA PPI. 14
FIGURE 5: IOT DATA ADAPTER POSITION IN THE VITAL ARCHITECTURE. 33
FIGURE 6: IOT PROVIDER REQUESTS FROM THE IOT DATA ADAPTER TO REFRESH ALL RELATED METADATA. 35
FIGURE 7: IOT DATA ADAPTER PULLS PERIODICALLY OBSERVATIONS MADE BY A SENSOR. 36
FIGURE 8: IOT PUSHES NEW OBSERVATIONS MADE BY A SENSOR TO VITAL. 36
FIGURE 9: DMS POSITION IN THE VITAL ARCHITECTURE. 38
FIGURE 10: SERVICE DISCOVERY POSITION IN THE VITAL ARCHITECTURE. 47
FIGURE 11: FILTERING POSITION IN THE VITAL ARCHITECTURE. 50
FIGURE 12: SAMPLE INTERACTION WITH FILTERING. 51
FIGURE 13: CEP POSITION IN THE VITAL ARCHITECTURE. 54
FIGURE 14: SAMPLE INTERACTION WITH CEP 58
FIGURE 15: ORCHESTRATOR POSITION IN THE VITAL ARCHITECTURE. 59
FIGURE 16: BASIC FLOW OF AN SAML 2.0 SSO THROUGH A WEB BROWSER USING HTTP. 70
FIGURE 17: OPENAM IN THE OAUTH 2.0 FLOW. 72
FIGURE 18: POLICY AGENT & OPENAM INTEGRATION. 75
FIGURE 19: POLICY AGENT & OPENAM AUTHENTICATION/AUTHORIZATION FLOW. 76
FIGURE 20: VUAI SECURITY ARCHITECTURE OVERVIEW. 77
FIGURE 21: PPI SECURITY ARCHITECTURE OVERVIEW. 77
FIGURE 22: FIT IOT-LAB REPRESENTATION 86
FIGURE 23: PPI IMPLEMENTATION FOR THE HI REPLY PLATFORM. 87
FIGURE 24: EXAMPLE FLOW. 89

LIST OF TABLES

TABLE 1: ACCESS TO GENERAL INFORMATION ABOUT AN IOT SYSTEM. 15
TABLE 2: ACCESS TO CURRENT IOT SYSTEM STATUS. 16
TABLE 3: ACCESS TO METADATA ABOUT THE PROVIDED IOT SERVICES. 17
TABLE 4: ACCESS TO METADATA ABOUT THE MANAGED SENSORS. 19
TABLE 5: ACCESS TO CURRENT SENSOR STATUS. 20
TABLE 6: METADATA SAMPLE FOR AN OBSERVATIONSERVICE THAT SUPPORTS THE PULL-BASED MECHANISM. 22
TABLE 7: ACCESS TO SENSOR OBSERVATIONS 22
TABLE 8: METADATA SAMPLE FOR AN OBSERVATIONSERVICE THAT SUPPORTS THE PUSH-BASED MECHANISM. 24
TABLE 9: SUBSCRIPTION TO AN OBSERVATION STREAM. 24
TABLE 10: CANCELLATION OF A SUBSCRIPTION TO AN OBSERVATION STREAM. 25
TABLE 11: SAMPLE CONFIGURATIONSERVICE METADATA. 25
TABLE 12: ACCESS TO CURRENT IOT SYSTEM CONFIGURATION. 26
TABLE 13: CHANGES TO CURRENT IOT SYSTEM CONFIGURATION. 26
TABLE 14: MONITORINGSERVICE METADATA SAMPLE. 27
TABLE 15: ACCESS TO SUPPORTED PERFORMANCE METRICS. 28
TABLE 16: ACCESS TO CURRENT VALUES OF PERFORMANCE METRICS. 29

TABLE 17: ACCESS TO SUPPORTED SLA PARAMETERS. 30
TABLE 18: ACCESS TO CURRENT VALUES OF SLA PARAMETERS. 31
TABLE 19: ACCESS TO REGISTERED IOT SYSTEMS. 34
TABLE 20: SAMPLE STANDALONE SENSOR METADATA. 37
TABLE 21: INSERT SYSTEM. 39
TABLE 22: INSERT SERVICE. 40
TABLE 23: INSERT SENSOR. 41
TABLE 24: INSERT OBSERVATION. 42
TABLE 25: QUERY SYSTEM. 43
TABLE 26: QUERY SERVICE. 44
TABLE 27: QUERY SENSOR. 44
TABLE 28: QUERY OBSERVATION. 45
TABLE 29: GET SYSTEMS. 47
TABLE 30: GET ICOS. 48
TABLE 31: GET SERVICES. 49
TABLE 32: THRESHOLD. 51
TABLE 33: RESAMPLE. 52
TABLE 34: FIELDS IN THE CREATECEPICO INTERFACE. 54
TABLE 35: GET CEPICOS. 55
TABLE 36: GET CEPICO. 55
TABLE 37: CREATE CEPICO. 57
TABLE 38: DELETE CEPICO. 58
TABLE 39: GET OPERATIONS. 60
TABLE 40: GET OPERATION. 60
TABLE 41: CREATE OPERATION. 61
TABLE 42: UPDATE OPERATION. 61
TABLE 43: DELETE OPERATION. 62
TABLE 44: TEST OPERATION EXECUTION. 62
TABLE 45: GET WORKFLOWS. 63
TABLE 46: GET WORKFLOW. 63
TABLE 47: CREATE WORKFLOW. 64
TABLE 48: UPDATE WORKFLOW. 65
TABLE 49: DELETE WORKFLOW. 66
TABLE 50: TEST WORKFLOW EXECUTION. 66
TABLE 51: GET META-SERVICES. 67
TABLE 52: GET META-SERVICE. 68
TABLE 53: DEPLOY META-SERVICE. 68
TABLE 54: UN-DEPLOY META-SERVICE. 69
TABLE 55: EXECUTE SERVICE. 69
TABLE 56: REQUEST FOR AUTHENTICATION TOKEN. 79
TABLE 57: REQUEST FOR A PROTECTED RESOURCE. 80
TABLE 58: TECHNOLOGIES USED IN FOOTFALL DATA FEED PPI IMPLEMENTATION. 82
TABLE 59: TECHNOLOGIES USED IN LIVE BUS ARRIVAL DATA FEED. 83
TABLE 60: TECHNOLOGIES USED IN X-GSN PPI IMPLEMENTATION. 84
TABLE 61: TECHNOLOGIES USED IN XIVELY PPI IMPLEMENTATION. 85
TABLE 62: TECHNOLOGIES USED IN HI REPLY PPI IMPLEMENTATION. 87

TERMS AND ACRONYMS

API	Application Programming Interface
ARIMA	AutoRegressive Integrated Moving Average
CEP	Complex Event Processing
FCAPS	Fault, Configuration, Accounting, Performance, Security
FIT	Future Internet of Things
GSN	Global Sensor Networks
ICO	Internet Connected Object
IoT	Internet-of-Things
JAXB	Java Architecture for XML Binding
JSON-LD	JavaScript Object Notation for Linked Data
LDAP	Lightweight Directory Access Protocol
LSM	Linked Sensor Middleware
OIDC	OpenID Connect
PADA	Platform Access and Data Acquisition
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PPI	Platform Provider Interface
REST	Representational State Transfer
SAML	Security Assertion Markup Language
SSN	Semantic Sensor Network
SSO	Single Sign-On
TfL	Transport for London
VUAI	Virtualized Unified Access Interface
X-GSN	eXtended Global Sensor Networks

1 INTRODUCTION

1.1 Scope

The third work package of the VITAL project (WP3) deals with the specification and implementation of models and interfaces that could enable virtualization of diverse IoT systems. This virtualization is a key to implement IoT platform-agnostic functionalities as part of the VITAL platform, thereby enabling development of integrated IoT applications for smart cities, i.e. applications leveraging data and services from multiple underlying IoT systems. Key elements of the VITAL virtualization infrastructure include:

- A range of platform-agnostic data models (data schemas, ontologies and databases), along with systems for managing their data elements. At the heart of these data models lies the VITAL ontology, which provides a model for representing data for IoT applications in smart cities regardless of the IoT systems used to capture and/or process these data. The VITAL ontology is already specified as part of deliverable D3.1 of the project.
- A range of virtualized interfaces, which enable platform-agnostic access to the IoT services of diverse heterogeneous IoT systems. These interfaces boost VITAL's integrated virtualized application development paradigm, which promotes a «learn-once-and-use-across-IoT-systems» approach to IoT application development in smart cities.

The purpose of the present deliverable is to provide the specification of the virtualized interfaces of the VITAL platform. In particular, the following types of virtualized interfaces are specified:

- Interfaces to the value-added functionalities of the VITAL platform, such as Complex Event Processing (CEP) and Service Discovery functionalities. These functionalities typically use the VITAL ontology as a means for expressing their results in a way that is independent of the initial data sources (e.g. of the underlying platforms that provide the discovered services).
- Abstract, virtualized interfaces to the functionalities of the underlying IoT systems (platforms and applications), which are classified as PPIs (Platform Provider Interfaces). The notion and the role of PPIs have already been introduced as part of the VITAL architecture specification (described in deliverable D2.3).

All the above interfaces can be classified as VUAIs (Virtualized Unified Access Interfaces). The present version (D3.2.3) of the deliverable is the final one and enhances the first (D3.2.1) and the second (D3.2.2) versions of the same deliverable. This version presents the final version of the VITAL architecture, which reflects updates to the structure and operation of the VUAIs. It describes the changes to the PPI specification, so that the latter allows sensing systems and devices to be directly connected to VITAL (rather than through an IoT system). It also documents several examples of PPI implementations, in order to demonstrate the versatility, the platform-agnostic nature and the functionality of the PPI abstraction. Furthermore, the implementation of PPIs over open data sources is illustrated.

1.2 Audience

This deliverable is addressed to the following audiences:

- **IoT applications developers and solution providers**, notably solution providers emphasizing on smart city applications using the IoT paradigm. Application developers and solution providers are expected to be interested into the project's general-purpose interfaces for accessing IoT systems, especially given the fact that the VITAL VUAI specifications attempt to virtually address any IoT system. Furthermore, VUAIs provide a first approach to implement the very topical target of IoT/BigData convergence, since they include a range of abstract data processing functions over platform-agnostic IoT services.
- **IoT researchers,** notably researchers working on abstract data models and service models for IoT applications. To these researchers, VUAIs may serve as a source of information for their research.
- **VITAL developers,** notably individuals engaging in the development of the VITAL added-value functionalities and of the VITAL applications. The former will need to understand the VUAIs in order to make sure that their components/modules support them, while the latter need to gain insights on the VUAIs in order to use them properly as part of their smart city application development tasks.

As already outlined, the present deliverable will be consulted by VITAL developers working in WP4, WP5 and WP6 activities, which will be using PPIs and VUAIs for accessing IoT data and services in a platform-agnostic manner.

1.3 Summary

As already outlined, the VITAL virtualized interfaces can be classified in two different types, namely: (a) abstract interfaces to IoT systems, IoT services, and ICOs, and (b) abstract Interfaces to the added-value functionalities of the VITAL platform. This deliverable includes dedicated parts to the specification of each of the above types of interfaces. Each dedicated part describes the interfaces and their use in the scope of the VITAL platform, i.e. when and how they can be used by «client» applications accessing the VITAL platform. The specification of each interface (regardless of its type) includes a description of the functionalities it provides, as well as a description of the input it expects and the output it produces.

In addition to providing the specifications of the virtualized interfaces, the deliverable elaborates on their positioning and use in the scope of the VITAL architecture. It therefore identifies the consumers of the interface functionalities. For instance, PPIs provide a low-latency interface for accessing data streams and capabilities of the underlying IoT systems. Hence, PPIs are very handy for VITAL application developers, notably developers that are engaging in the integration of real-time or semi real-time applications. As another example, complex event processing interfaces are handy for solution developers focusing on data-intensive applications within the smart city.

Note that several aspects, such as security, are handled within the specification of each interface, rather than horizontally i.e. in a way that transcends all specified interfaces. This is intentional and due to the different scopes and time-scales of operations of the various interface types.

The present final release of the deliverable describes the implementation of VUAIs over the VITAL data management services, as well as the prototype implementation of the PPIs over different platforms that have been selected for practical adaptation to the VITAL architecture and semantically interoperability paradigm. Moreover, the implementation of PPI interfaces to sensors and open data sources is presented.

1.4 Structure

The deliverable is structured as follows:

- Section **Error! Reference source not found.**, following this introductory section, illustrates the scope and purpose of the various types of VUAIs. It also discusses their positioning within the VITAL architecture.
- Section 3 is devoted to the PPI specification. The specification includes several updates compared to the earlier versions of the deliverable. The updates have been produced as a result of the need to allow sensing systems and devices to connect directly to VITAL (without the need to be part of an IoT system).
- Section 4 focuses on the specifications of VUAIs for accessing the added-value functionalities of the VITAL platform.
- Section 5 illustrates the architecture and implementation of the security mechanisms used to authorize and authenticate interactions through the VUAIs.
- Section 6 describes the PPI implementations for IoT platforms, sensors/devices and open data sources.
- Section 7 is the final and concluding section of this deliverable, which provides also an outlook for the development of the next (and final) release of the latter.

2 VIRTUALIZED UNIFIED ACCESS INTERFACES

2.1 Purpose and Positioning within the VITAL Architecture

The purpose of VUAIs is to facilitate virtualized platform-agnostic access to sensor data and IoT services for smart cities. They are abstract interfaces enabling developers to access data streams and services provided by multiple IoT systems, without the need to deal with the low-level details of the underlying systems. The concept of VUAIs has already been introduced in deliverable D2.3 as part of the presentation of the VITAL architecture.

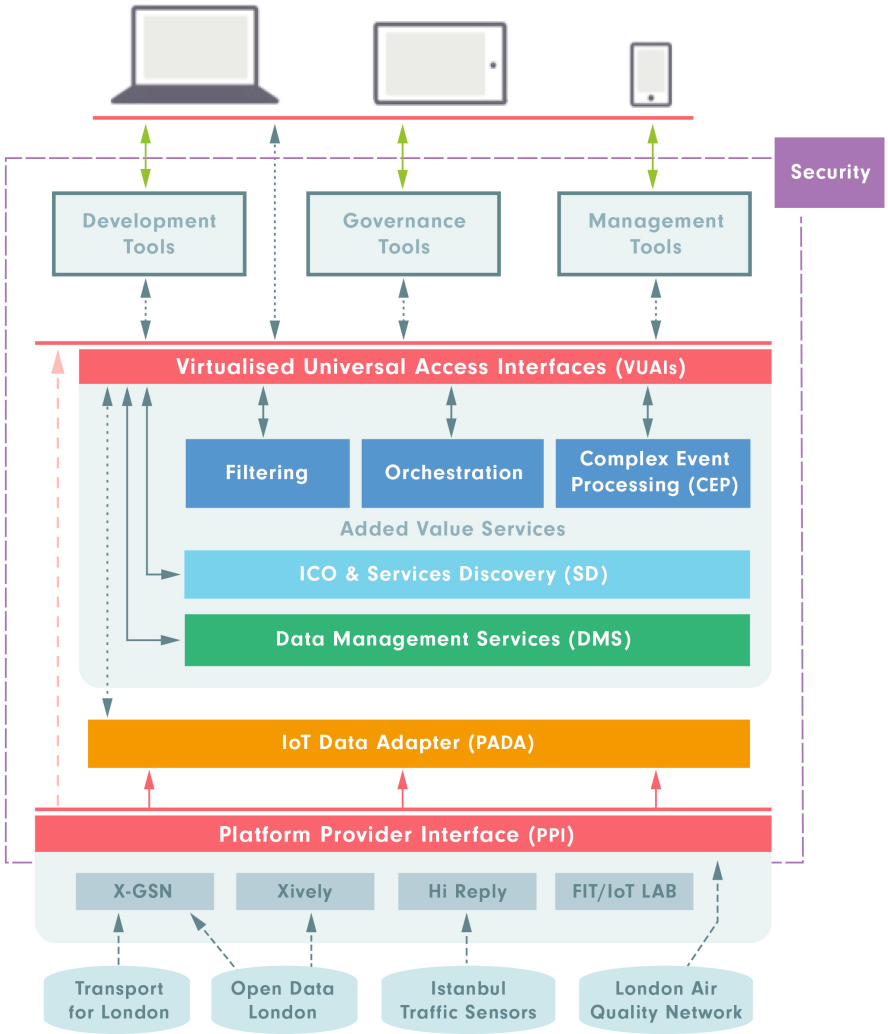

Figure 1: Overview of the VITAL architecture.

Figure 1 illustrates a slightly enhanced version of the (high-level) VITAL architecture provided in deliverable D2.3. It depicts the role of VUAIs as abstract interfaces residing at the top layer of the VITAL architecture, thereby enabling access to added-value data processing and process management functionalities, including CEP, service discovery, filtering, and other functionalities. As already outlined, it also illustrates additional concepts regarding high-performance access to platforms and data sources using PPIs. These concepts are implied, but not detailed in deliverable D2.3. In particular:

- **PPI as high-performance VUAI**: The architecture specifies the ability of accessing IoT data both through VUAIs accessing the VITAL data management services and through direct access to PPIs. The latter option is required in cases where high-performance, low-overhead access to data provided by IoT systems is required (e.g. (near) real-time applications). In this context, the PPI, i.e. the platform agnostic interface for accessing IoT systems, can be considered as a high-performance VUAI for data access. Overall, the two virtualized platform-agnostic options for accessing IoT systems and IoT data through VUAIs are explained in Figure 2.
- **PPI as an interface for accessing individual data sources**: VITAL enables the federation of IoT platforms (such as the four platforms selected in deliverable D2.2, namely FIT/IoT-LAB, X-GSN, Xively and Hi Reply). Nevertheless, it also acknowledges the possibility of exploiting individual (IoT-related) data sources and datasets (e.g. live feeds stemming from TfL open dataset), based on direct access to their data. Access to individual data sources can be also carried out through a PPI implementation, and more specifically based on a light implementation that does not implement optional functionalities. This concept is also illustrated in Figure 1.

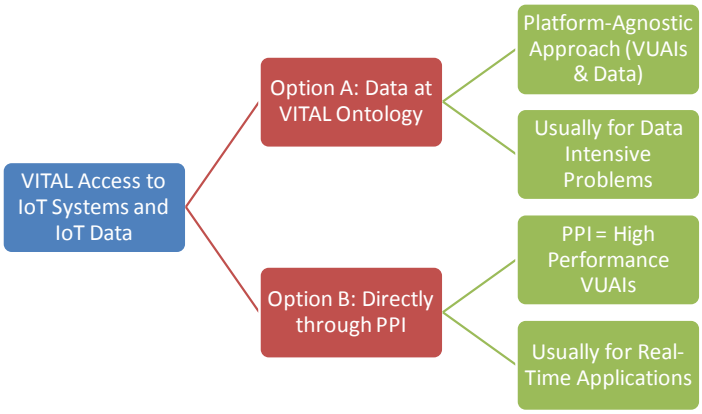

Figure 2: Two main options offered by the VITAL Virtualization Layer (VUAIs) regarding IoT data access.

In addition to virtualized access to IoT data and services, as well as to added-value functionalities (like discovery, filtering and CEP), the high-level VITAL architecture (depicted in the previous figure) prescribes that VUAIs will be accessed by the VITAL development environment and tool. The tool is expected to facilitate the development of IoT applications that are based on data processing functionalities (notably data intensive problems). It is therefore envisaged that VUAIs should include interfaces for executing data processing and mining methods (such as classification and clustering).

2.2 Types of VUAIs and Functional Scope

The previous paragraph has introduced the concept of VUAIs and their positioning in the VITAL architecture. It has also illustrated the fact that PPIs can be considered VUAIs. As a result, VUAIs can be classified in two main categories:

- **VUAIs for direct data access leveraging PPIs**: These are interfaces for platform access, which enable a «learn-once-and-use-across-IoT-systems» discipline for

accessing IoT data. They facilitate developers and solution providers to access any VITAL compliant platform via a unified interface. PPIs can also be used to access individual data sources. As illustrated in the following section, the PPI specification includes a range of both mandatory and optional methods, which correspond to the different spectrum of functionalities that are required to access individual data sources or integrated fully fledged IoT platforms.

- **VUAIs accessing VITAL modules**: These are interfaces for accessing the modules residing at the added-value layer of the VITAL architecture, notably the filtering, CEP and service discovery modules, as well as the data from the VITAL data management services (following the VITAL ontology initially specified in deliverable D3.1).

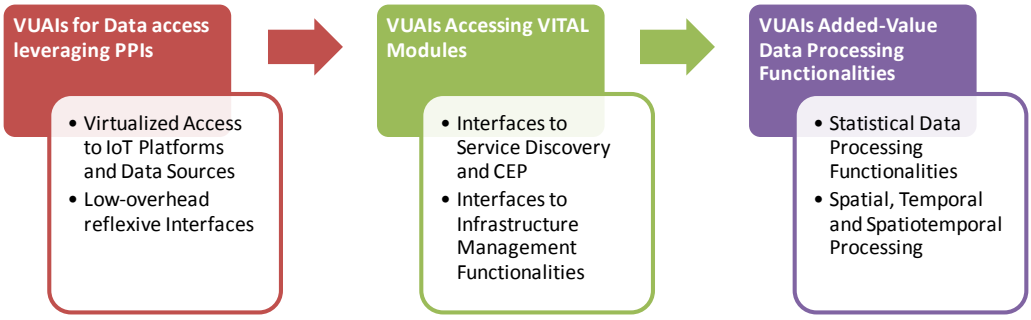

Figure 3: Overview of the VUAIs provided by the VITAL platform.

The two types of VUAIs outlined above are specified in the following paragraphs. The specifications are driven by other VITAL results produced as part of earlier deliverables in particular:

- The VITAL ontology and related data models, which drive the specification of the data elements that are exchanged via the VUAIs. Indeed, we do not specify new data models in this deliverable; rather we rely on the data models prescribed in the final version of the VITAL data modelling deliverable (D3.1.2).
- The VITAL architecture introduced in deliverable D2.3. The same deliverable has also illustrated requirements associated with the data elements that should be exchanged over VUAIs.

Overall, the present deliverable is in-line with background results contained in deliverables D3.1 and D2.3. The readership is advised to consult these documents for more details on data models and architectural concepts that underpin the VUAI specifications in the present document.

It should also be noted that the interfaces (VUAIs) specified in this document will provide valuable inputs to other technical activities of the project, notably activities that will produce technical modules that will consume/use the VUAIs. Several such modules (e.g. CEP, discovery, filtering, data access for FCAPS management) are currently being developed in WP4 and WP5 of the project.

3 PLATFORM PROVIDER INTERFACE SPECIFICATION

3.1 Overview

The Platform Provider Interface (PPI) is a set of RESTful web services, marked as either mandatory or optional, that enable access to IoT systems, as well as to ICOs managed and IoT services provided by those systems. All VITAL compliant IoT systems are expected to implement and expose at least the web services that are designated as mandatory. PPI also allows ICOs to be connected to VITAL in a standalone mode (rather than as part of an IoT system).

3.2 PPI Specification

PPI is defined as a set of RESTful web services that IoT systems to be integrated into VITAL should expose, and that the VITAL platform can then use in order to retrieve:

- Information about the IoT systems (e.g. their status)
- Information about the IoT services that an IoT system exposes (e.g. how to access them)
- Information about the sensors that an IoT system manages (e.g. what they observe)
- Observations made by the sensors that an IoT system manages

In its final version, PPI has been extended in order to allow standalone sensors (i.e. sensors that are not managed by an IoT system) to implement it and connect to VITAL.

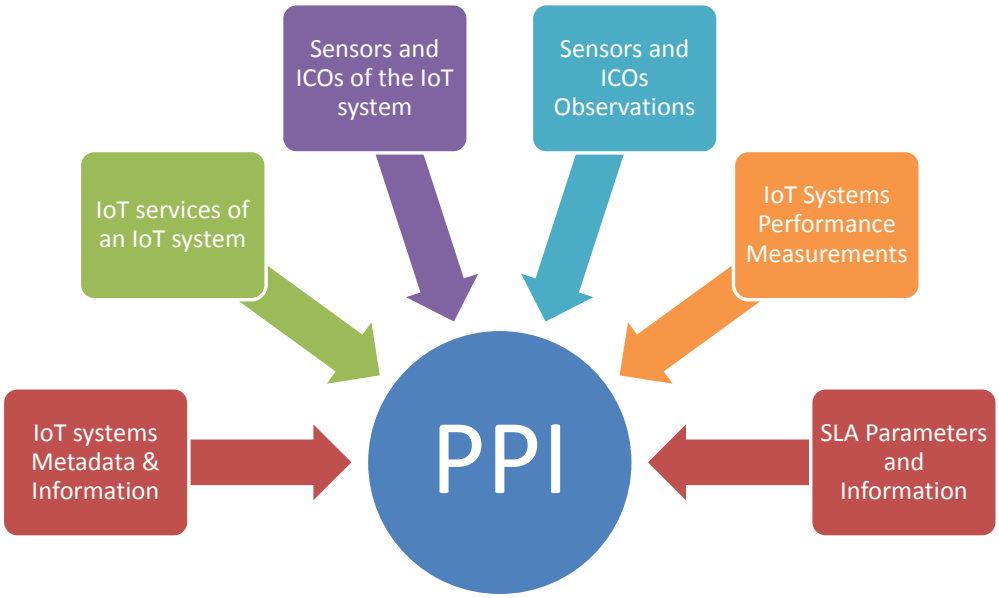

Figure 4: Classification of the types of information accessed via PPI.

A detailed description of these web services is given in the following paragraphs and is in-line with information contained in earlier deliverable D2.3. Specifically, in D2.3 we have already provided an overview of data and services that the PPI is expected to provide/expose, and we have also classified these data and services as mandatory or optional. Overall, the detailed PPI specifications contained in the following paragraphs build on earlier specifications and requirements specified as part of WP2 of the project.

3.2.1 Access to IoT System Metadata

This section describes the web services that any VITAL compliant IoT system must expose, so that the VITAL platform can use them to obtain metadata about the system. IoT system metadata include:

- general information about the IoT system (e.g. its operator)
- information about any configuration functionalities that the IoT system might provide (e.g. services that allow to retrieve the current configuration of the system and probably change it)
- the current status of the IoT system

The purpose of the web service described in Table 1 is to provide access to general information about the IoT system (e.g. its service area), as well as to the list of the services it provides and the list of the sensors it manages. Access to that information is described as mandatory in D2.3, and thus all VITAL compliant IoT systems must implement and expose this web service.

Table 1: Access to general information about an IoT system.

	Get IoT system metadata	
Description	This service can be used to access metadata about the IoT system.	
URL	PPI_BASE_URL/metadata	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	{
	}	

Response headers	Content-Type	application/ld+json or application/json
Response body	**Example** { "@context": "http://vital-iot.eu/contexts/system.jsonld", "id": "http://example.com", "type": "vital:VitalSystem", "name": "Sample IoT system", "description": "This is a VITAL compliant IoT system.", "operator": "http://example.com/people#john_doe", "serviceArea": "http://dbpedia.org/page/Camden_Town", "sensors": ["http://example.com/sensor/1", "http://example.com/sensor/2"], "services": ["http://example.com/service/1", "http://example.com/service/2",	

	<pre>"http://example.com/service/3"
	],
	"status": "vital:Running"
	}</pre>

Mandatory	Yes
Notes	• At the moment, the request body is an empty JSON object with no fields, but it can be later used for filtering the returned metadata. • The context of the response body is the JSON-LD context for systems described in Section 5.1 of D3.1.2.

PPI_BASE_URL denotes the URL where the PPI implementation of the underlying IoT system can be accessed. Thus, if the PPI implementation for a particular IoT system can be accessed through http://example.com/ppi, then its metadata can be retrieved by issuing a POST request to http://example.com/ppi/metadata.

Note that the above web service returns only the IDs of the services provided and of the sensors managed by the IoT system. Sections 3.2.2 and 3.2.3 describe how to retrieve more information about them.

If an IoT system intends to expose any configuration functionality to the VITAL platform, it must provide a service of type **ConfigurationService**, as described in Section 3.2.5. Note that, since the configuration service is in fact one of the IoT services that the corresponding IoT system provides, its ID must be listed in the result of the web service described in Table 1, and its metadata must be available in the same way that the metadata of all IoT services that an IoT system provides are exposed, as described in Section 3.2.2.

Finally, since the status of an IoT system might change throughout its lifecycle, we decided to represent its values as observations of a virtual sensor. Thus, VITAL compliant systems that want to expose their current operational state must manage a virtual sensor of type **MonitoringSensor**. More details about that sensor can be found in Section 3.2.6. As an alternative way of exposing its current status, an IoT system may provide an IoT service of type **MonitoringService** with (at least) an operation of type **GetSystemStatus**. The result of that operation is the current status of the system, and is equivalent to retrieving the last observation made by the sensor of type MonitoringSensor for the property **OperationalState** of the feature of interest http://example.com (namely the IoT system). Refer to Table 14 for metadata about a monitoring service, and to Table 2 for details about the implementation requirements for the GetSystemStatus operation.

Table 2: Access to current IoT system status.

	Get IoT system status	
Description	This service can be used to access the current status of the IoT system.	
Method	POST	
Response headers	Content-Type	application/json

Response body	**Example**
	<pre>{
	"@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	"id": "http://example.com/sensor/1/observation/1",
	"type": "ssn:Observation",
	"ssn:observationProperty":
	{</pre>	

	<pre>"type": "vital:OperationalState"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest": "http://example.com",
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "vital:Running"
	}
	}
	}</pre>

Mandatory	No
Notes	• The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2.

Based on D2.3, the provision of lifecycle information, part of which are the current status and any configuration functionality, is optional for IoT systems to be integrated into VITAL. Thus, the provision of the necessary operations as part of a configuration and a monitoring service are both optional.

3.2.2 Access to IoT Service Metadata

IoT systems that want to connect to the VITAL platform can expose to the latter metadata about any IoT services they provide. In order to do that, they can implement and expose the web service described in Table 3, which returns metadata about the provided IoT services that have a specific ID or are of a specific type. Based on D2.3, the provision of the following web service is optional for the IoT systems that want to be PPI compliant.

Table 3: Access to metadata about the provided IoT services.

	Get IoT service metadata	
Description	This service can be used to access metadata about IoT services that the IoT system provides.	
URL	PPI_BASE_URL/service/metadata	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	<pre>{
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	<pre>[
	{
	"@context": "http://vital-iot.eu/contexts/service.jsonld",
	"id": "http://example.com/service/1",
	"type": "vital:ConfigurationService",
	"operations":</pre>	

<table>
<tr><td></td><td>
<pre>
    [
      {
        "type": "vital:GetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:SetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  },
  {
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/2",
    "type": "vital:MonitoringService",
    "msm:hasOperation":
    [
      {
        "type": "vital:GetSystemStatus",
        "hrest:hasAddress": "http://example.com/system/status",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSensorStatus",
        "hrest:hasAddress": "http://example.com/sensor/status",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSupportedPerformanceMetrics",
        "hrest:hasAddress": "http://example.com/system/performance",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:GetPerformanceMetrics",
        "hrest:hasAddress": "http://example.com/system/performance",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSupportedSLAParameters",
        "hrest:hasAddress": "http://example.com/system/sla",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:GetSLAParameters",
        "hrest:hasAddress": "http://example.com/system/sla",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  },
  {
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/3",
    "type": "vital:ObservationService",
    "operations":
    [
      {
        "type": "vital:GetObservations",
        "hrest:hasAddress": "http://example.com/sensor/observation",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  }
]
</pre>
</td></tr>
<tr><td>Mandatory</td><td>No</td></tr>
<tr><td>Notes</td><td>• The request body is a JSON object with the following fields: id, whose value</td></tr>
</table>

	is an array of IoT service IDs (e.g. http://example.com/service/1), **type**, whose value is an array of IoT service type IDs (e.g. http://vital-iot.eu/ontology/ns/MonitoringService). Both fields are optional and can be used to filter the IoT services to retrieve metadata for. • The context of the response body is the JSON-LD context for services described in Section 5.2.2 of D3.1.2.

3.2.3 Access to Sensor Metadata

IoT systems must expose to VITAL information about the sensors they manage (e.g. their type or their location). In order to do that, VITAL compliant IoT systems are expected to implement the RESTful web service that is presented in Table 4. The purpose of this service is to return metadata about the managed sensors that satisfy certain criteria. The current version of the PPI specification supports filtering of the managed sensors based on their ID and type.

Table 4: Access to metadata about the managed sensors.

	Get sensor metadata	
Description	This service can be used to access metadata about sensors that the IoT system manages.	
URL	PPI_BASE_URL/sensor/metadata	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`}`	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	`[`
	`  {`
	`  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",`
	`  "id": "http://example.com/sensor/1",`
	`  "type": "vital:MonitoringSensor",`
	`  "name": "System Monitoring Sensor",`
	`  "description": "A virtual sensor that monitors the operational state of the system, as well as its services and sensors.",`
	`  "status": "vital:Running",`
	`  "ssn:observes":`
	`  [`
	`  {`
	`  "type": "vital:OperationalState",`
	`  "id": "http://example.com/sensor/1/operationalState"`
	`  },`
	`  {`
	`  "type": "vital:SysUptime",`
	`  "id": "http://example.com/sensor/1/sysUptime"`
	`  },`
	`  {`
	`  "type": "vital:SysLoad",`
	`  "id": "http://example.com/sensor/1/sysLoad"`
	`  },`
	`  {`
	`  "type": "vital:Errors",`
	`  "id": "http://example.com/sensor/1/errors"`
	`  }`
	`  ]`	

	<pre> },
	  {
	  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	  "id": "http://example.com/sensor/2",
	  "name": "A sensor.",
	  "type": "vital:VitalSensor",
	  "description": "A sensor.",
	  "hasLastKnownLocation":
	  {
	  "type": "geo:Point",
	  "geo:lat": 53.2719,
	  "geo:long": -9.0849
	  },
	  "ssn:observes":
	  [
	  {
	  "type": "openiot:Temperature",
	  "id": "http://example.com/sensor/2/temperature"
	  }
	  ]
	  }
	]</pre>

Mandatory	Yes
Notes	• The request body is a JSON object with the following fields: **id**, whose value is an array of sensor URIs (e.g. http://example.com/sensor/1), **type**, whose value is an array of sensor type IDs (e.g. http://vital-iot.eu/ontology/ns/VitalSensor). Both fields are optional and can be used to filter the sensors to retrieve metadata for. • The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.2.

Since access to sensor metadata is marked as mandatory in deliverable D2.3, all VITAL compliant IoT systems are expected to implement and expose a RESTful web service that satisfies all requirements outlined in Table 4.

Apart from sensor metadata that are **static** (e.g. their name or the properties they observe), there are also **dynamic** sensor metadata that include:

- the current status of the sensor
- the current location of the sensor, in case the sensor is mobile

IoT systems are expected to expose the current status of a sensor they manage as the last observation of the **MonitoringSensor** (described in Section 3.2.6) for the property **OperationalState** of the managed sensor (i.e. the sensor is the feature of interest). As an alternative way of exposing the current status of the sensors it manages, an IoT system may provide an IoT service of type **MonitoringService** with (at least) an operation of type **GetSensorStatus**. Refer to Table 14 for metadata about a monitoring service, and to Table 5 for details about the implementation requirements for the GetSensorStatus operation.

Table 5: Access to current sensor status.

	Get sensor status	
Description	This service can be used to access the current status of the sensors managed by the IoT system.	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example**	

	<pre>{
	  "id":
	  [
	  "http://example.com/sensor/2"
	  ]
	}</pre>

Response headers	Content-Type: application/ld+json or application/json

Response body	**Example**
	<pre>[
	  {
	  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	  "id": "http://example.com/sensor/1/observation/2",
	  "type": "ssn:Observation",
	  "ssn:observationProperty":
	  {
	  "type": "vital:OperationalState"
	  },
	  "ssn:observationResultTime":
	  {
	  "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	  },
	  "ssn:featureOfInterest": "http://example.com/sensor/2",
	  "ssn:observationResult":
	  {
	  "type": "ssn:SensorOutput",
	  "ssn:hasValue":
	  {
	  "type": "ssn:ObservationValue",
	  "value": "vital:Running"
	  }
	  }
	  }
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with the following fields: **id**, whose value is an array of sensor IDs (e.g. http://example.com/sensor/1), **type**, whose value is an array of sensor type IDs (e.g. http://vital-iot.eu/ontology/ns/VitalSensor). Both fields are optional and can be used to filter the sensors to retrieve the current status of. • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2.

Finally, an IoT system can expose the current status of a sensor in the value of the status field in the metadata retrieved for that sensor through the service described in Table 4.

As far as the current location of a mobile sensor is concerned, there are two ways to obtain it (as described in Section 6.5.1 of D3.1):

- through the IoT service that the **hasLocalizer** property contained in the sensor metadata points or describes
- through the **hasLastKnownLocation** property contained in the sensor metadata

The first way is always preferred to the second, since the value of the hasLastKnownLocation property might be outdated.

3.2.4 Access to Sensor Observations

It is prescribed that the VITAL platform can use both a pull and a push-based mechanism in order to obtain observations made by a sensor. An IoT system can support one or both of these mechanisms by providing an IoT service of type **ObservationService** with the necessary operations.

Since access to the sensor data of an IoT system is mandatory, all VITAL compliant IoT systems must support at least one of these two mechanisms. This essentially means that IoT providers should implement and expose one RESTful web service for each operation that is required by the mechanism (or the mechanisms) they want to support, as described in the following paragraphs.

3.2.4.1 Pull-based mechanism

In order to support the pull-based mechanism (i.e. in order for VITAL to be able to pull observations from the IoT system), the **ObservationService** must have (at least) an operation of type **GetObservations**. An example for the description of an IoT service of type ObservationService that supports the pull-based mechanism is shown in Table 6.

Table 6: Metadata sample for an ObservationService that supports the pull-based mechanism.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/3",
  "type": "vital:ObservationService",
  "operations":
  [
    {
      "type": "vital:GetObservations",
      "hrest:hasAddress": "http://example.com/sensor/observation",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As shown in Table 7, the GetObservations operation is expected to return observations that satisfy certain criteria as JSON-LD documents that follow the contexts described in deliverable D3.1.2 (see Section 3.4).

Table 7: Access to sensor observations.

	Get observations	
Description	This service can be used to access observations made by sensors.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "sensor":`
	`  [`
	`  "http://example.com/sensor/2"`
	`  ],`
	`  "property": " http://lsm.deri.ie/OpenIot/Temperature"`
	`  "from": "2014-11-17T09:00:00+02:00",`
	`  "to": "2014-11-17T11:00:00+02:00"`	

<table>
<tr><td></td><td colspan="2">}</td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example

<pre>
[
  {
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "id": "http://example.com/sensor/2/observation/1",
    "type": "ssn:Observation",
    "ssn:observedBy": "http://example.com/sensor/2",
    "ssn:observationProperty":
    {
      "type": "openiot:Temperature"
    },
    "ssn:observationResultTime":
    {
      "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
    },
    "dul:hasLocation":
    {
      "type": "geo:Point",
      "geo:lat": "55.701",
      "geo:long": "12.552",
      "geo:alt": "4.33"
    },
    "ssn:observationResult":
    {
      "type": "ssn:SensorOutput",
      "ssn:hasValue":
      {
        "type": "ssn:ObservationValue",
        "value": "21.0",
        "qudt:unit": "qudt:DegreeCelsius"
      }
    }
  }
]
</pre></td></tr>
<tr><td>Mandatory</td><td colspan="2">No</td></tr>
<tr><td>Notes</td><td colspan="2">• The request body is a JSON object with the following fields: sensor, whose value is an array of sensor IDs (e.g. http://example.com/sensor/1), property, whose value is a property type (e.g. http://lsm.deri.ie/OpenIoT/Temperature), and from and to that define a time interval. sensor and property fields are mandatory and determine the sensor and the property, respectively, to return observations for. from and to determine the time interval, when the observations to return were taken. Both to and from are optional. If to is omitted, then all observations taken after from are returned. If both from and to are omitted, then the last observation taken from the specified sensor for the specified property is returned.
• The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2.</td></tr>
</table>

3.2.4.2 Push-based mechanism

In order to support the push-based mechanism (i.e. in order for the IoT system to be able to push observations to the VITAL platform), the **ObservationService** should have (at least) the following operations:

- an operation of type **SubscribeToObservationStream** that creates a subscription to a specific stream of observations
- an operation of type **UnsubscribeFromObservationStream** that cancels the subscription with a specific ID.

An example for the description of an IoT service of type ObservationService that supports the push-based mechanism is shown in Table 8.

Table 8: Metadata sample for an ObservationService that supports the push-based mechanism.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/3",
  "type": "vital:ObservationService",
  "operations":
  [
    {
      "type": "SubscribeToObservationStream",
      "hrest:as Address": "http://example.com/observation/stream/subscribe",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "UnsubscribeFromObservationStream",
      "hrest:as Address": "http://example.com/observation/stream/unsubscribe",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As described in Table 9, the SubscribeToObservationStream operation is expected to return a subscription ID. As a result of that operation, VITAL periodically receives at the URL, which was specified in the body of the SubscribeToObservationStream request, observations that match the specified criteria (i.e. observations that were obtained by the specified sensor for the specified property). The URL essentially indicates another RESTful web service – one that is implemented and exposed by the VITAL platform – that supports the HTTP POST method, and expects a set of observations in JSON-LD format based on the JSON-LD context for measurements described in Section 3.4 of D3.1.2.

Table 9: Subscription to an observation stream.

	Subscribe to observation stream	
Description	This service can be used to subscribe to an observation stream.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	{
	"sensor": "http://example.com/sensor/1",
	"property": " http://lsm.deri.ie/OpenIot/Temperature",
	"url": "http://vital-iot.eu/observation/push"
	}	

Response headers	Content-Type	application/json

Response body	**Example**
	{
	"subscription": "d670460b4b4aece5915caf5c68d12f560a9fe3e4"
	}	

Mandatory	No
Notes	• The request body is a JSON object with the following fields: **sensor**, whose value is a sensor ID (e.g. http://example.com/sensor/1), **property**, whose value is a property type (e.g. http://lsm.deri.ie/OpenIoT/Temperature), and **url**, whose value is a URL. All fields are mandatory. **sensor** and **property** fields determine the sensor and the property, respectively, to receive observations for. **url** determines the URL where the IoT system is expected to push to the subscriber any new observations made by the sensor for the property.

As described in Table 10, the UnsubscribeFromObservationStream operation gives back no response, and results in the cancellation of the subscription with the specified ID.

Table 10: Cancellation of a subscription to an observation stream.

	Unsubscribe from observation stream	
Description	This service can be used to cancel a currently active subscription to an observation stream.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "subscription": "d670460b4b4aece5915caf5c68d12f560a9fe3e4"`
	`}`	

Mandatory	No	
Notes	• The request body is a JSON object with a single field, **subscription**. The subscription-id field is mandatory, and its value is the ID of the subscription to cancel.	

3.2.5 Configuration

IoT systems connected to the VITAL platform may provide configuration functionalities to the latter. For that purpose, an IoT system can provide an IoT service of type **ConfigurationService**. If VITAL is allowed to retrieve the current configuration of the IoT system, then the service must have an operation of type **GetConfiguration**. If VITAL is also allowed to modify the current configuration of the system, then the service must also contain a second operation of type **SetConfiguration**. Table 11 presents an example for the description of such a service.

Table 11: Sample ConfigurationService metadata.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/1",
  "type": "vital:ConfigurationService",
  "msm:hasOperation":
  [
    {
      "type": "vital:GetConfiguration",
      "hrest:hasAddress": "http://example.com/service/1",
      "hrest:hasMethod": "hrest:GET"
    },
    {
```

```
      "type": "vital:SetConfiguration",
      "hrest:hasAddress": "http://example.com/service/1",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As described in Table 12, the GetConfiguration operation is expected to return the current configuration of the IoT system. For each configuration parameter, this operation is expected to return its name, its current value, its type, and the permissions that the VITAL platform has on it.

Table 12: Access to current IoT system configuration.

	Get configuration	
Description	This service can be used to access the current configuration of the IoT system.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	"parameters":
	[
	{
	"name": "c1",
	"value": "1",
	"type": "http://www.w3.org/2001/XMLSchema#int",
	"permissions": "rw"
	},
	{
	"name": "c2",
	"value": "v2",
	"type": " http://www.w3.org/2001/XMLSchema#string",
	"permissions": "rw"
	}
	]
	}	

Mandatory	No	
Notes	• The response contains the key-value pairs for all configuration parameters exposed to VITAL, along with the value types and the associated permissions (rw, r).	

Table 13 contains the functional requirements for the SetConfiguration operation, which receives one or more new values for one or more (writable) configuration parameters, and is expected to update the IoT system configuration accordingly.

Table 13: Changes to current IoT system configuration.

	Change configuration	
Description	This service can be used to change the current value of one or more configuration parameters of the IoT system.	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example** { "parameters": [	

	<pre>{
	"name": "c1",
	"value": "2"
	},
	{
	"name": "c2",
	"value": "v3"
	}
	]
	}</pre>

Mandatory	No
Notes	• The status code of the request can be: ◦ **200**, if the IoT system configuration has been successfully updated ◦ **403**, if at least one the specified parameters is read-only ◦ **404**, if at least one of the specified parameters does not exist • Note that if among the specified parameters there is at least one that does not exist or is not writable, then none of the required configuration parameter value changes will be performed.

Note that the configuration service is one of the IoT services that the corresponding IoT system provides, which implies that its metadata must be available in the same way that the metadata of all IoT services that an IoT system provides are exposed, as described in Section 3.2.2.

3.2.6 Monitoring

The VITAL platform may be able to monitor various aspects of a registered IoT system, depending on whether the latter has exposed any monitoring functionalities to VITAL. More specifically, VITAL can monitor:

- the status of the IoT system
- the status of the sensors that the IoT system manages
- various performance metrics related to the IoT system (e.g. its uptime)
- various SLA parameters related to the IoT system (e.g. the response time for successful requests)

An IoT system can allow VITAL to access and monitor any or all of the above listed information by providing an IoT service of type **MonitoringService**. Table 14 contains an example for metadata about a monitoring service. We have already described how an IoT system can expose its current status and the current status of the sensors it manages in Sections 3.2.1 and 3.2.3, respectively. The rest of the section focuses on the performance- and SLA-related actions of MonitoringService.

Table 14: MonitoringService metadata sample.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/2",
  "type": "vital:MonitoringService",
  "msm:hasOperation":
  [
    {
      "type": "vital:GetSystemStatus",
      "hrest:hasAddress": "http://example.com/system/status",
```

```
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSensorStatus",
      "hrest:hasAddress": "http://example.com/sensor/status",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSupportedPerformanceMetrics",
      "hrest:hasAddress": "http://example.com/system/performance",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "type": "vital:GetPerformanceMetrics",
      "hrest:hasAddress": "http://example.com/system/performance",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSupportedSLAParameters",
      "hrest:hasAddress": "http://example.com/system/sla",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "type": "vital:GetSLAParameters",
      "hrest:hasAddress": "http://example.com/system/sla",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

3.2.6.1 Performance Metric Monitoring

In order for VITAL to be able to monitor various performance metrics about an IoT system, the latter must provide a MonitoringService with (at least) an operation of type **GetSupportedPerformanceMetrics** that returns a list of the performance metrics that the IoT system supports. Table 15 contains the specifications for that operation. A list of the performance metrics that IoT systems may support is included in Section 6.7.1 of D3.1 and in Section 3.2.1 of D5.1.

Table 15: Access to supported performance metrics.

	Get supported performance metrics	
Description	This service can be used to get a list of the performance metrics that the IoT system supports.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	"metrics":
	[
	{
	"type": "vital:SysUptime",
	"id": "http://example.com/sensor/1/sysUptime"
	},
	{
	"type": "vital:SysLoad",
	"id": "http://example.com/sensor/1/sysLoad"
	},
	{
	"type": "vital:Errors",
	"id": "http://example.com/sensor/1/errors"
	}	

	<pre>] }</pre>
Mandatory	No
Notes	• The response body contains all supported performance metrics. For each metric, the type and the ID of the corresponding observed property are provided.

As it is obvious from Table 15, each performance metric corresponds essentially to a property that the **MonitoringSensor** observes. That way an IoT system can expose the current value of a performance metric (e.g. for the system uptime) as the last observation of the MonitoringSensor for the corresponding property (e.g. SysUptime) of the feature of interest that represents the IoT system (e.g. http://example.com).

Alternatively, VITAL compliant IoT systems can expose the current (or last) value of the supported performance metrics by adding an operation of type **GetPerformanceMetrics** to the MonitoringService. Refer to Table 16 for the functional requirements for that operation.

Table 16: Access to current values of performance metrics.

	Get performance metrics	
Description	This service can be used to access the current value of the performance metrics that the IoT system supports.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	<pre>{
	  "metric":
	  [
	  " http://vital-iot.eu/ontology/ns/SysLoad",
	  " http://vital-iot.eu/ontology/ns/SysUptime"
	  ]
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	<pre>[
	  {
	  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	  "id": "http://example.com/sensor/1/observation/3",
	  "type": "ssn:Observation",
	  "ssn:observationProperty":
	  {
	  "type": "vital:SysLoad"
	  },
	  "ssn:observationResultTime":
	  {
	  "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	  },
	  "ssn:featureOfInterest": "http://example.com",
	  "ssn:observationResult":
	  {
	  "type": "ssn:SensorOutput",
	  "ssn:hasValue":
	  {
	  "type": "ssn:ObservationValue",
	  "value": "80",
	  "qudt:unit": "qudt:Percent"</pre>	

	<pre>}
	}
	},
	{
	"@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	"id": "http://example.com/sensor/1/observation/4",
	"type": "ssn:Observation",
	"ssn:observationProperty":
	{
	"type": "vital:SysUptime"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest": "http://example.com",
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "800000",
	"qudt:unit": "qudt:MilliSecond"
	}
	}
	}
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with a single field, **metric**. The value of this field is an array of property type IDs (e.g. http://vital-iot.eu/ontology/ns/SysLoad). • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2.

Since an IoT system is not obliged to provide any configuration-related functionality in order to be VITAL compliant, the provision of a monitoring service, as well as the implementation of any of the operations described in this section are classified as optional.

3.2.6.2 SLA parameter monitoring

For QoS purposes, an IoT system might allow VITAL to monitor certain SLA parameters. An initial version of a list with the SLA parameters that IoT systems may support is included in Section 3.2.4.1 of D5.1. In order for an IoT system to enable VITAL to monitor any SLA parameters about it, the former must provide a MonitoringService with (at least) an operation of type **GetSupportedSLAParameters**, which returns a list of the SLA parameters that the IoT system supports, and which is described in Table 17.

Table 17: Access to supported SLA parameters.

	Get supported SLA parameters	
Description	This service can be used to get a list of the SLA parameters that the IoT system supports.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	"parameters":
	[
	{
	"type": "http://vital-iot.eu/ontology/ns/ResponseTime",
	"id": "http://example.com/sensor/1/responseTime"
	},
	{
	"type": "http://vital-iot.eu/ontology/ns/StatusCode",
	"id": "http://example.com/sensor/1/statusCode"
	}
	]
	}

Mandatory	No
Notes	• The response body contains all supported SLA parameters. For each parameter, the type and the ID of the corresponding observed property are provided.

Each SLA parameter, like each performance metric, corresponds to a property that the **MonitoringSensor** observes. Based on that an IoT system has two alternative ways of exposing the current value of SLA parameters:

- as the last observation of the MonitoringSensor for the corresponding property (e.g. ResponseTime) of the appropriate feature of interest (e.g. HTTP requests made to a specific URL by a specific user)
- by adding an operation of type **GetSLAParameters** to the MonitoringService

More information about the GetSLAParameters operation can be found in Table 18.

Table 18: Access to current values of SLA parameters.

	Get SLA parameters	
Description	This service can be used to access the current value of the SLA parameters that the IoT system supports.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	{
	"parameter":
	[
	"http://vital-iot.eu/ontology/ns/ResponseTime"
	]
	}	

Response headers	Content-Type	application/ld+json or application/json
Response body	**Example** [{ "@context": "http://vital-iot.eu/contexts/measurement.jsonld", "id": "http://example.com/sensor/1/observation/5", "type": "ssn:Observation", "ssn:observationProperty": { "type": "vital:ResponseTime" }, "ssn:observationResultTime":	

	<pre>{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest":
	{
	"http:methodName": "GET",
	"http:absoluteURI": "http://example.com/system/status",
	"madeBy": "http://vital-example.com/users#243",
	},
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "80",
	"qudt:unit": "qudt:MilliSecond"
	}
	}
	}
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with a single field, **parameter**. The value of this field is an array of property type IDs (e.g. http://vital-iot.eu/ontology/ns/ResponseTime). • The context of the response body is the JSON-LD context described in Section 3.4 of D3.1.2.

The enablement of SLA parameter monitoring is not mandatory for the characterization of an IoT system as VITAL compliant. Thus, the provision of the operations described above as part of a service of type MonitoringService is optional.

3.3 IoT Data Adapter

IoT Data Adapter (a.k.a. Platform Agnostic Data Access or PADA) is the component of the VITAL platform that acts as the intermediary between VITAL and PPI implementations of external IoT systems that want to be integrated into VITAL. Figure 5 shows the position of the IoT data adapter within the VITAL architecture.

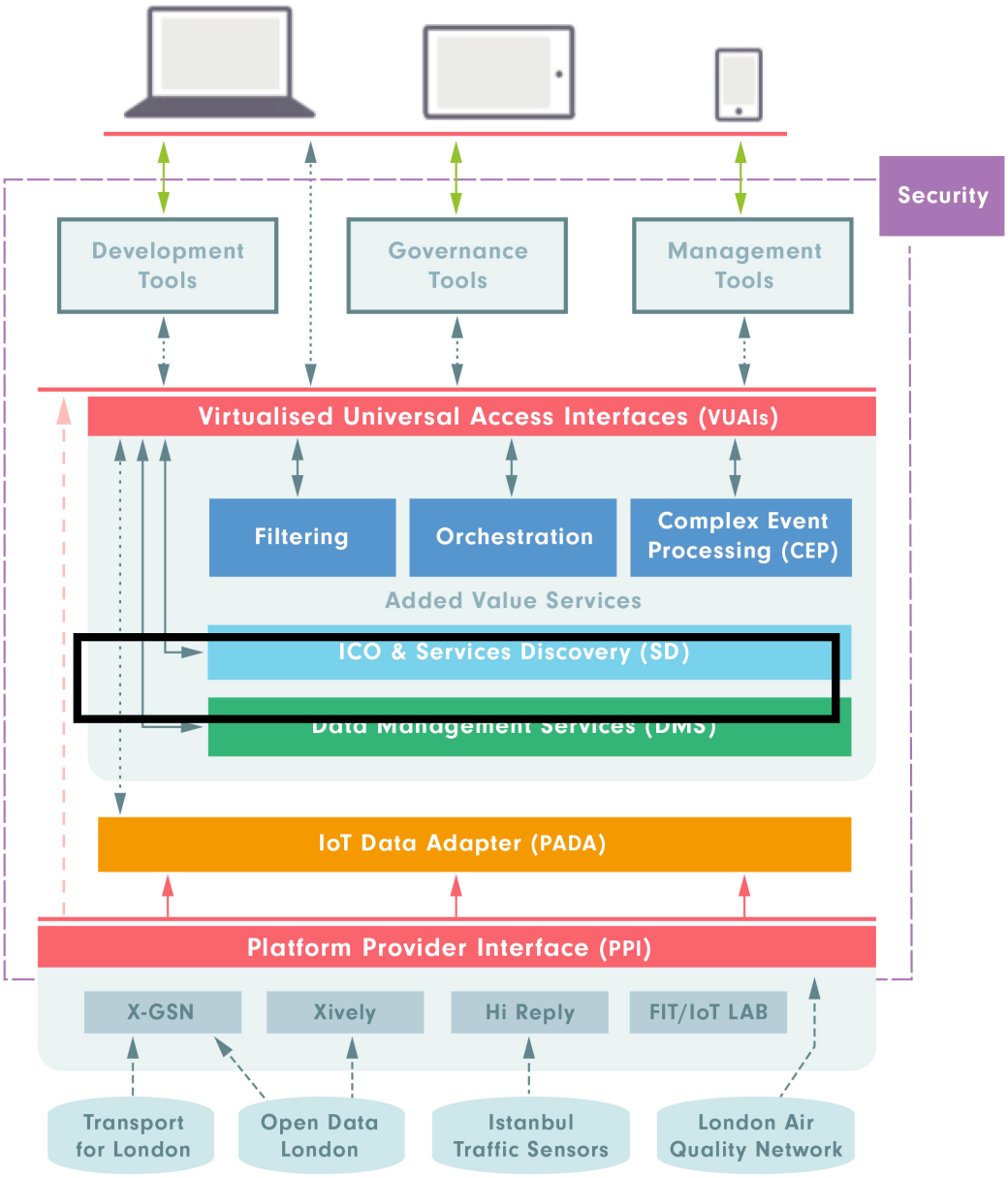

Figure 5: IoT data adapter position in the VITAL architecture.

The purposes of the adapter are:

- to provide a web-accessible user interface that can be used by IoT system providers in order to:
  - register and de-register their systems
  - manage the settings used by VITAL to access their systems (e.g. the URL where the PPI implementation of their system is exposed)
- to collect information from the registered IoT systems, and store this information into a VITAL-specific internal store

As part of the registration and deregistration processes, the IoT system provider is expected to provide:

- the **URI** of the IoT system, which acts as its unique identifier (this is the value of the id field in the IoT system metadata, see Table 1).
- the **base URL**, through which the PPI implementation exposed by the IoT system is accessible.

The adapter provides a web service that can be used to retrieve information about the IoT systems that are currently registered in the VITAL platform. Details about this web service are given in Table 19.

Table 19: Access to registered IoT systems.

	Get registered IoT systems	
Description	This service can be used to obtain information about the IoT systems currently registered in VITAL.	
URL	IOTDA_BASE_URL/system/registered	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`[`
	`  {`
	`  "id": "http://example.com",`
	`  "ppi": "http://example.com/ppi",`
	`  "type": "system"`
	`  }`
	`]`	

Notes	• The response body contains information about all registered systems. For each system, its ID, its base URL and its type (see Section 3.4) are provided.	

IOTDA_BASE_URL denotes the URL that can be used to access the IoT data adapter component of an instance of the VITAL platform. The above web service can be used to construct the value of the **providesSystem** property of a **VitalSystem**, as described in Section 6.1.1 of D3.1.2.

The adapter is responsible for collecting information about and from the registered IoT systems. As soon as a new IoT system is registered to the VITAL platform, the adapter retrieves metadata about the system itself (Section 3.2.1), the IoT services it provides (Section 3.2.2), and the sensors it manages (Section 3.2.3). The retrieved metadata are pushed to DMS through a VUAI the latter exposes for that purpose (Section 4.1).

Once an IoT system has been registered to the VITAL platform, its provider can explicitly request VITAL (through the adapter user interface) to refresh the metadata stored about it. The adapter is then responsible to retrieve a fresh version of the IoT system, IoT service and sensor metadata, and push them to DMS, using the same VUAI described above. This is how the adapter manages information that is exposed by the registered IoT systems and that is considered to be rarely modified (e.g. the name of the system). Figure 6 depicts this sequence of actions.

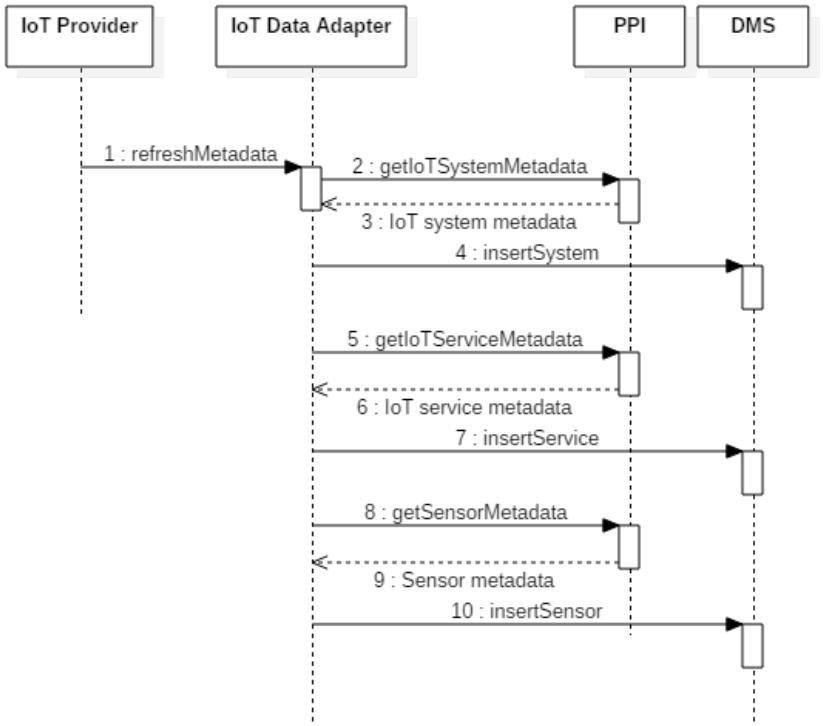

Figure 6: IoT provider requests from the IoT data adapter to refresh all related metadata.

Apart from static information, IoT systems provide also access to dynamic information (i.e. information that changes or is generated during the lifetime of the system). At the moment, this dynamic information includes:

- the current status of the IoT system
- the current status of all sensors managed by the IoT system
- the current locations of all mobile sensors managed by the IoT system
- new observations made by sensors managed by the IoT system

In order to guarantee that VITAL has an almost up-to-date version of this dynamic information, the adapter pulls periodically this information (using the respective PPI primitives described in Sections 3.2.1, 3.2.3 and 3.2.4), and pushes them to DMS. The frequency with which each one of the above pieces of information is pulled from an IoT system is configurable (through the adapter web interface) and might vary from system to system, depending also on business requirements of the targeted smart cities applications. Figure 7 illustrates how the IoT data adapter periodically pulls from the PPI of an IoT system new observations that have been made in the meantime by a sensor.

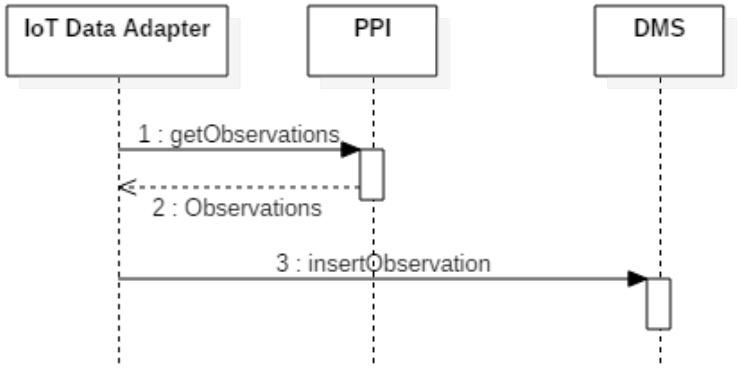

Figure 7: IoT data adapter pulls periodically observations made by a sensor.

Finally, the adapter always checks for the provision of a push-based mechanism that it can probably use to collect any of the above pieces of information, and prefers it over a pull-based mechanism if both are supported by the corresponding PPI implementation. Figure 8 illustrates how the adapter uses the push-based mechanism to get observations from the PPI of an IoT system.

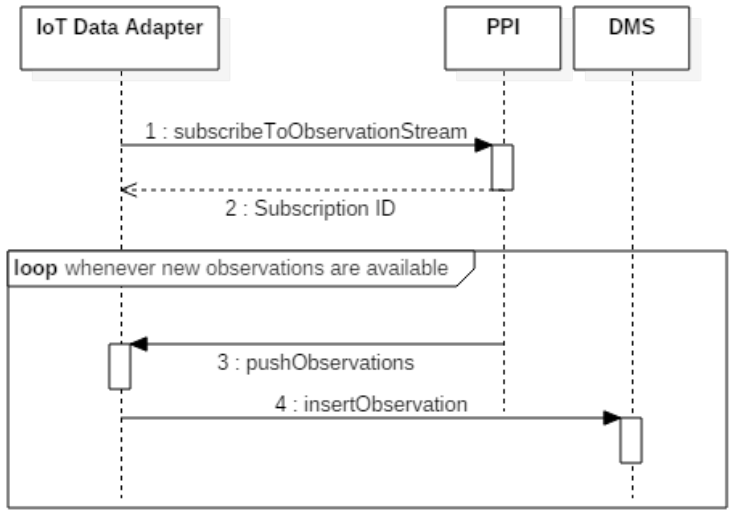

Figure 8: IoT pushes new observations made by a sensor to VITAL.

3.4 Standalone Sensors

Oftentimes, IoT providers may want to connect a single sensor, rather than an entire IoT system, to VITAL. If in those cases the providers are also required to provide a proper PPI implementation (i.e. implement at least the PPI primitives that are marked as mandatory), as described in the previous sections, then that would be an overkill.

In order to support this scenario, we decided to relax the requirements we have set for PPI implementations. More specifically, in such cases, IoT providers are required to implement only the **Get sensor metadata** PPI primitive, in order to provide information about the sensor (or the sensors) they want to connect to the VITAL platform.

The current status of the sensor can be provided either through the **status** property of the sensor or through a sensor of type **MonitoringSensor** or using the **hasMonitor** property of the sensor that is expected to describe a service of type **MonitoringService**.

The current location of the sensor can be provided either through the **hasLastKnownLocation** property of the sensor or using its **hasLocalizer** property that is expected to describe a service of type **LocationService**.

Finally, the observations made by the sensor are accessible using the **hasObserver** property of the sensor that is expected to point to a service of type **ObservationService**. Table 20 contains an example of the metadata for a standalone sensor.

Table 20: Sample standalone sensor metadata.

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
  "id": "http://example.com/sensor",
  "name": "A sensor.",
  "type": "vital:VitalSensor",
  "description": "A sensor.",
  "hasLastKnownLocation":
  {
    "type": "geo:Point",
    "geo:lat": 53.2719,
    "geo:long": -9.0849
  },
  "ssn:observes":
  [
    {
      "type": "openiot:Temperature",
      "id": "http://example.com/sensor/2/temperature"
    }
    ]
  },
  "hasObserver":
  {
    "id": "http://example.com/service",
    "type": "vital:ObservationService",
    "operations":
    [
      {
        "type": "vital:GetObservations",
        "hrest:hasAddress": "http://example.com/sensor/observation",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  },
  "status": "vital:Running"
}
```

Once the PPI implementation for the standalone sensor is ready, the IoT provider registers it using the web interface provided by the IoT data adapter. As part of the registration process, the provider is still expected to specify a unique URI for the sensor (that in this case represents the “system”), and the base URL, through which the PPI implementation is accessible. The IoT data adapter uses the exact same mechanisms described in Section 3.3 to retrieve metadata about and data from the registered sensor.

4 VIRTUALIZED ACCESS TO VITAL MODULES

4.1 Interfaces to Data Management Service

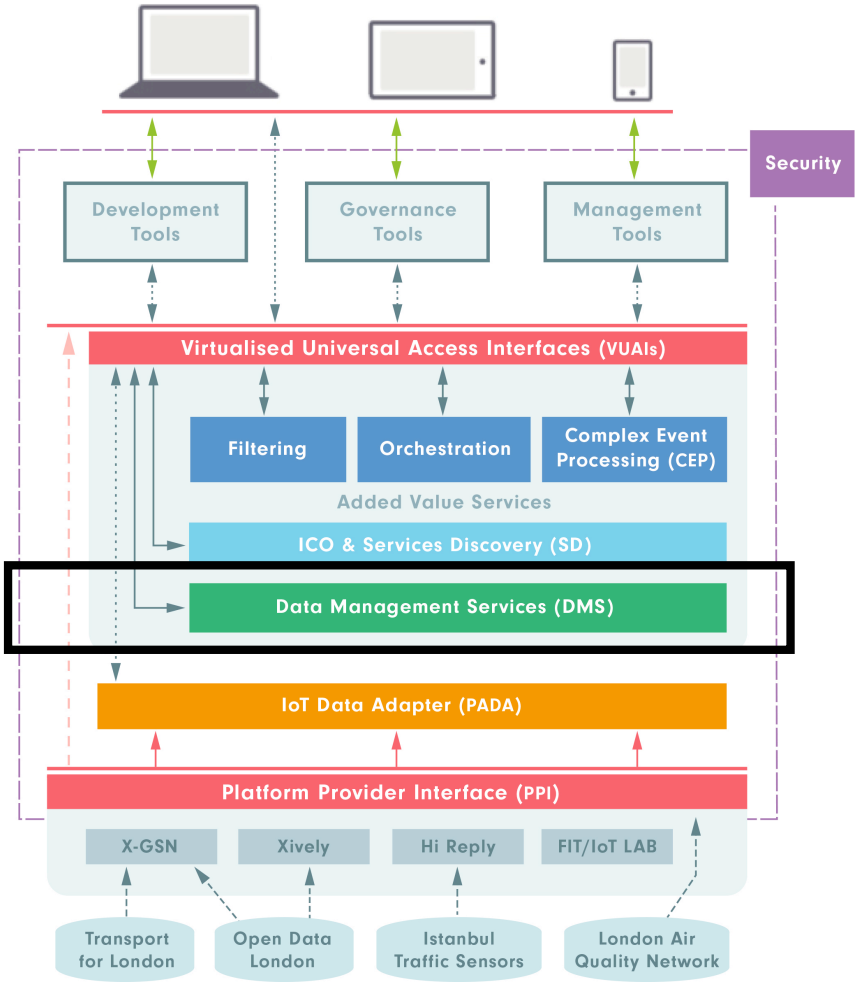

Figure 9: DMS position in the VITAL architecture.

Data Management Service (**DMS**) is responsible for storing and allowing access to all kinds of metadata (i.e. metadata about IoT systems, IoT services and ICOs), as well as to historical measurement data. The data is modelled using the VITAL ontology and access to metadata and observations is provided through RESTful interfaces. DMS provides eight RESTful interfaces that can be used to:

- Store metadata about systems, services, and sensors
- Store sensor observations
- Query metadata about systems, services, and sensors
- Query sensor observations

As noted in the previous version of the deliverable (D3.2.2), the data format (JSON-LD) returned by DMS was not accurate. This problem has been under investigation, however, there is no solution available as yet, which can support SPARQL queries over JSON-LD data. Until now JSON-LD is not natively supported by Virtuoso [JSONLDSupport]. Most of the approaches adapt the conversion technique from

RDF/XML to JSON-LD (and reverse) which leads to this format problem. Due to project time constraints, DMS has been implemented using MongoDB. More details on DMS implementation can be found in deliverable D6.1.2, which is running in parallel to this document. The data conversion problem will be under investigation, and once resolved, an extra service will be provided to enable a SPARQL endpoint over DMS.

As mentioned above, DMS provides eight types of interfaces. Four of these interfaces are related to pushing metadata and data into DMS, whereas the other four related to their retrieval.

4.1.1 Store Metadata and Observations

There are four entities modelled in VITAL: systems, services, sensors, and observations. In order to enable the storage of information about those entities, DMS provides one interface for each one of them:

- insertSystem
- insertService
- insertSensor
- insertObservation

Figures 6, 7, and 8 have already depicted the context and scenario (via sequence of calls) of utilizing these interfaces for storing metadata and data into the DMS.

Table 21: Insert system.

	Insert System metadata	
Description	This interface is used to insert System metadata into DMS.	
Method	POST	
URL	DMS_BASE_URL/insertSystem	
Request headers	Content-Type	application/ld+json or application/json

Request body	**Example**
	{
	"@context": "http://vital-iot.eu/contexts/system.jsonld",
	"id": "http://vital-iot.eu/istanbul-traffic",
	"type": "vital:IotSystem",
	"name": "Istanbul Live Traffic Data",
	"description": "This is a VITAL-compliant IoT system that provides live traffic data for Istanbul.",
	"operator": "email@domain.com",
	"serviceArea": "http://dbpedia.org/page/Istanbul",
	"status": "vital:Unavailable",
	"services": [
	"http://vital-iot.eu/istanbul-traffic/service/configuration",
	"http://vital-iot.eu/istanbul-traffic/service/monitoring",
	"http://vital-iot.eu/istanbul-traffic/service/observation"
	],
	"sensors": [
	"http://vital-iot.eu/istanbul-traffic/sensor/2-F",
	"http://vital-iot.eu/istanbul-traffic/sensor/3-F",
	"http://vital-iot.eu/istanbul-traffic/sensor/monitoring"
	]
	}	

Response headers	Content-Type	application/json

Response body	**Example** `{ "message" : "Data pushed." }`
Notes	• The context of the request body is the JSON-LD context for systems described in Section 5.1 of D3.1.2. • The response body contains a JSON object that contains message indicating the status of the request.

Table 22: Insert service.

	Insert Service metadata	
Description	This interface is used to insert Service metadata into DMS.	
Method	POST	
URL	DMS_BASE_URL/insertService	
Request headers	Content-Type	application/ld+json or application/json

Request body	**Example**
	`[{`
	`  "@context": "http://vital-iot.eu/contexts/service.jsonld",`
	`  "id": "http://vital-iot.eu/istanbul-traffic/service/configuration",`
	`  "type": "vital:ConfigurationService",`
	`  "operations": [{`
	`  "type": "vital:GetConfiguration",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/service/configuration",`
	`  "hrest:hasMethod": "hrest:GET"`
	`  }, {`
	`  "type": "vital:SetConfiguration",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/service/configuration",`
	`  "hrest:hasMethod": "hrest:POST"`
	`  }]`
	`}, {`
	`  "@context": "http://vital-iot.eu/contexts/service.jsonld",`
	`  "id": "http://vital-iot.eu/istanbul-traffic/service/monitoring",`
	`  "type": "vital:MonitoringService",`
	`  "operations": [{`
	`  "type": "vital:GetSystemStatus",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/status",`
	`  "hrest:hasMethod": "hrest:POST"`
	`  }, {`
	`  "type": "vital:GetSensorStatus",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/sensor/status",`
	`  "hrest:hasMethod": "hrest:POST"`
	`  }, {`
	`  "type": "vital:GetSupportedPerformanceMetrics",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/performance",`
	`  "hrest:hasMethod": "hrest:GET"`
	`  }, {`
	`  "type": "vital:GetPerformanceMetrics",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/performance",`
	`  "hrest:hasMethod": "hrest:POST"`
	`  }, {`
	`  "type": "vital:GetSupportedSLAParameters",`
	`  "hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/sla",`
	`  "hrest:hasMethod": "hrest:GET"`
	` }, {	

	<pre>"type": "vital:GetSLAParameters",
	"hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/sla",
	"hrest:hasMethod": "hrest:POST"
	}]
	}]</pre>

Response headers	Content-Type: application/json
Response body	**Example** <pre>{ "message" : "Data pushed." }</pre>
Notes	• The context of the request body is the JSON-LD context for services described in Section 5.2.2 of D3.1.2. • The response body contains a JSON object that contains message indicating the status of the request.

Table 23: Insert sensor.

	Insert Sensor metadata
Description	This interface is used to insert Sensor metadata into DMS.
Method	POST
URL	DMS_BASE_URL/insertSensor
Request headers	Content-Type: application/ld+json or application/json

Request body	**Example**
	<pre>[{
	"@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"type": "vital:VitalSensor",
	"description": "A traffic sensor in Istanbul.",
	"status": "vital:Unavailable",
	"hasLastKnownLocation": {
	"type": "geo:Point",
	"geo:lat": 41.09301817,
	"geo:lon": 29.0270595
	},
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/2-F",
	"name": "TEM Karanfilköy (Forward Direction)",
	"ssn:observes": [{
	"type": "vital:Speed",
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/2-F/speed"
	}]
	}, {
	"@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"type": "vital:VitalSensor",
	"description": "A traffic sensor in Istanbul.",
	"status": "vital:Unavailable",
	"hasLastKnownLocation": {
	"type": "geo:Point",
	"geo:lat": 41.09226678,
	"geo:lon": 29.08480696
	},
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/3-F",
	"name": "Kavacık FSM (Forward Direction)",
	"ssn:observes": [{
	"type": "vital:Speed",
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/3-F/speed"
	}]
	}]</pre>

Response headers	Content-Type	application/json
Response body	**Example** `{ "message" : "Data pushed." }`	
Notes	• The context of the request body is the JSON-LD context for sensors described in Section 3.3 of D3.1.2. • The response body contains a JSON object that contains message indicating the status of the request.	

Table 24: Insert observation.

	Insert Observation(s)	
Description	This interface is used to insert sensor Observations into DMS.	
Method	POST	
URL	DMS_BASE_URL/insertObservation	
Request headers	Content-Type	application/ld+json or application/json

Request body	**Example**
	`{`
	`  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",`
	`  "id": "http://vital-iot.eu/istanbul-traffic/sensor/monitoring/observation/status/1435743568541",`
	`  "type": "ssn:Observation",`
	`  "ssn:observedBy": "http://vital-iot.eu/istanbul-traffic/sensor/monitoring",`
	`  "ssn:observationProperty": {`
	`  "type": "vital:OperationalState"`
	`  },`
	`  "ssn:observationResultTime": {`
	`  "time:inXSDDateTime": "2015-07-01T09:39:28Z"`
	`  },`
	`  "ssn:featureOfInterest": "http://vital-iot.eu/istanbul-traffic",`
	`  "ssn:observationResult": {`
	`  "type": "ssn:SensorOutput",`
	`  "ssn:hasValue": {`
	`  "type": "ssn:ObservationValue",`
	`  "value": "vital:Unavailable"`
	`  }`
	`  }`
	`}`	

Response headers	Content-Type	application/json
Response body	**Example** `{ "message" : "Data pushed." }`	
Notes	• The context of the request body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2. • The response body contains a JSON object that contains message indicating the status of the request.	

4.1.2 Query Metadata and Observations

In order to allow access to metadata and observations stored in DMS, four interfaces are provided. The body of the requests to all these interfaces is a JSON object that contains the query parameters based on the MongoDB query language. Upon a valid request, the interfaces return data in JSON format.

In order to allow VITAL modules to query metadata and data stored in DMS, the following interfaces are provided:

- querySystem
- queryService
- querySensor
- queryObservation

Figures 12 and 14 depict the context and scenario (via sequence of calls) of utilizing these interfaces for querying metadata and data from the DMS.

Table 25: Query system.

<table>
<tr><td></td><td colspan="2">Query System metadata</td></tr>
<tr><td>Description</td><td colspan="2">This interface is used to query System metadata stored in DMS.</td></tr>
<tr><td>Method</td><td colspan="2">POST</td></tr>
<tr><td>URL</td><td colspan="2">DMS_BASE_URL/querySystem</td></tr>
<tr><td>Request headers</td><td>Content-Type</td><td>application/json</td></tr>
<tr><td></td><td>Cookie</td><td>vitalAccessToken</td></tr>
<tr><td>Request body</td><td colspan="2">Example
<pre>{
   "@type": "http://vital-iot.eu/ontology/ns/IotSystem"
}</pre></td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example
<pre>[
 {
   "@context": "http://vital-iot.eu/contexts/system.jsonld",
   "id": "http://vital-iot.eu/istanbul-traffic",
   "type": "vital:IotSystem",
   "name": "Istanbul Live Traffic Data",
   "description": "This is a VITAL-compliant IoT system that
provides live traffic data for Istanbul.",
   "operator": "arou@ait.edu.gr",
   "serviceArea": "http://dbpedia.org/page/Istanbul",
   "status": "vital:Unavailable",
   "services": [
     "http://vital-iot.eu/istanbul-traffic/service/configuration",
     "http://vital-iot.eu/istanbul-traffic/service/monitoring",
     "http://vital-iot.eu/istanbul-traffic/service/observation"
   ],
   "sensors": [
     "http://vital-iot.eu/istanbul-traffic/sensor/2-F",
     "http://vital-iot.eu/istanbul-traffic/sensor/3-F",
     "http://vital-iot.eu/istanbul-traffic/sensor/monitoring"
   ]
 }
]</pre></td></tr>
<tr><td>Notes</td><td colspan="2">• The context of the response body is the JSON-LD context for systems</td></tr>
</table>

	described in Section 5.1 of D3.1.2. • Request must also include vitalAccessToken Cookie.

Table 26: Query service.

	Query Service metadata	
Description	This interface is used to query Service metadata stored in DMS.	
Method	POST	
URL	DMS_BASE_URL/queryService	
Request headers	Content-Type	application/json
	Cookie	vitalAccessToken

Request body	**Example**
	`{`
	`"@type": "http://vital-iot.eu/ontology/ns/ConfigurationService"`
	`}`	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	`[`
	`{`
	`"@context": "http://vital-iot.eu/contexts/service.jsonld",`
	`"id": "http://vital-iot.eu/istanbul-traffic/service/configuration",`
	`"type": "vital:ConfigurationService",`
	`"operations": [`
	`{`
	`"type": "vital:GetConfiguration",`
	`"hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/service/configuration",`
	`"hrest:hasMethod": "hrest:GET"`
	`},`
	`{`
	`"type": "vital:SetConfiguration",`
	`"hrest:hasAddress": "http://vital-iot.eu/istanbul-traffic/ppi/service/configuration",`
	`"hrest:hasMethod": "hrest:POST"`
	`}`
	`]`
	`}`
	`]`	

Notes	• The context of the response body is the JSON-LD context for services described in Section 5.2.2 of D3.1.2. • Request must also include vitalAccessToken Cookie.	

Table 27: Query sensor.

	Query Sensor metadata from DMS	
Description	This interface is used to query Sensor metadata stored in DMS.	
Method	POST	
URL	DMS_BASE_URL/querySensor	
Request headers	Content-Type	application/json
	Cookie	vitalAccessToken

Request body	**Example**
	`{`
	`"http://vital-iot.eu/ontology/ns/hasLastKnownLocation":`
	`{`
	`$elemMatch:`
	`{`
	`"http://www.w3.org/2003/01/geo/wgs84_pos#lat" :`	

	<pre>{
	$elemMatch: { "@value" : {$gt: 51} }
	}
	}
	}
	}</pre>

Response headers	Content-Type: application/ld+json or application/json

Response body	**Example**
	<pre>[
	{
	"@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"type": "vital:VitalSensor",
	"description": "A traffic sensor in Istanbul.",
	"status": "vital:Unavailable",
	"hasLastKnownLocation": {
	"type": "geo:Point",
	"geo:lat": 41.09301817,
	"geo:lon": 29.0270595
	},
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/2-F",
	"name": "TEM Karanfilköy (Forward Direction)",
	"ssn:observes": [
	{
	"type": "vital:Speed",
	"id": "http://vital-iot.eu/istanbul-traffic/sensor/2-F/speed"
	}
	]
	}
	]</pre>

Notes	• The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.2. • Request must also include vitalAccessToken Cookie.

Table 28: Query observation.

	Query Observations from DMS	
Description	This interface is used to query Observations stored in DMS.	
Method	POST	
URL	DMS_BASE_URL/queryObservation	
Request headers	Content-Type	application/json
	Cookie	vitalAccessToken

Request body	**Example**
	<pre>{
	"http://purl.oclc.org/NET/ssnx/ssn#observationProperty":
	{
	$elemMatch:
	{
	"@type" : "http://vital-iot.eu/ontology/ns/OperationalState"
	}
	}
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	<pre>[
	{
	"@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	"id": "http://104.131.128.70:8080/istanbul-traffic/sensor/monitoring/observation/status/1435743568541",</pre>	

	"type": "ssn:Observation",
	"ssn:observedBy": "http://104.131.128.70:8080/istanbul-traffic/sensor/monitoring",
	"ssn:observationProperty": {
	"type": "vital:OperationalState"
	},
	"ssn:observationResultTime": {
	"time:inXSDDateTime": "2015-07-01T09:39:28Z"
	},
	"ssn:featureOfInterest": "http://104.131.128.70:8080/istanbul-traffic",
	"ssn:observationResult": {
	"type": "ssn:SensorOutput",
	"ssn:hasValue": {
	"type": "ssn:ObservationValue",
	"value": "vital:Unavailable"
	}
	}
	}
	]

Notes	• The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.2. • Request must also include vitalAccessToken Cookie.

4.2 Interfaces to Service Discovery

Service Discovery is in charge of discovering ICOs, systems, and services horizontally integrated in the VITAL platform. It operates on the IoT resources that are stored by the Data Management Service (DMS) and are requested by other modules in the VITAL Architecture (e.g. CEP and Orchestration).

The service discovery module supports the following operations:

- the **getICOs** operation that allows the discovery of ICOs that satisfy certain criteria
- the **getServices** operation that allows the discovery of IoT services based on specific criteria
- the **getSystems** operation that allows the discovery of IoT systems based on specific criteria

The above operations are designed and implemented as RESTful web services in WP4.

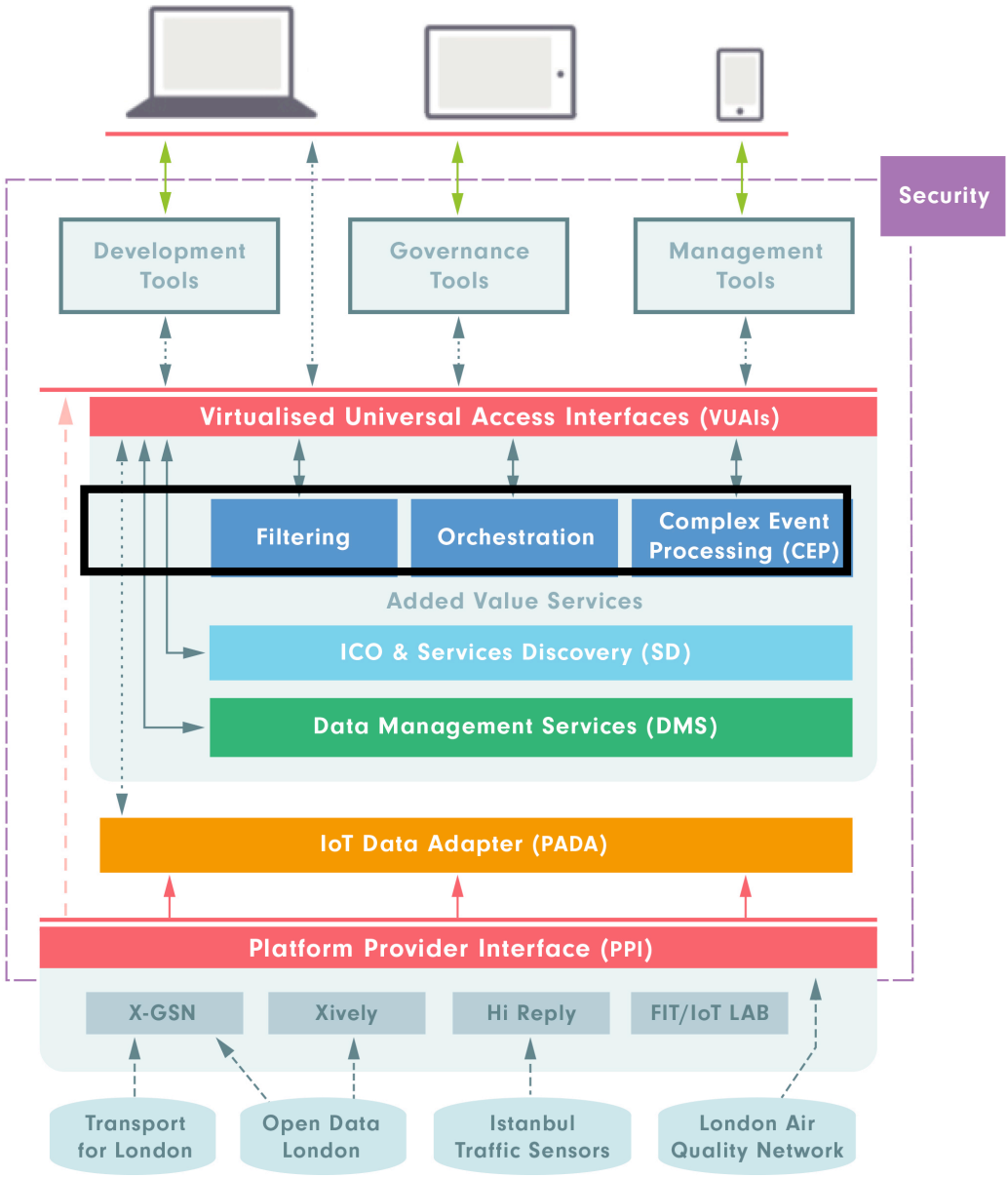

Figure 10: Service discovery position in the VITAL architecture.

Table 29: Get systems.

	Get IoT systems	
Description	Returns the IoT systems that match the specified criteria.	
URL	DISCOVERY_BASE_URL/system	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	{
	"type": "http://vital-iot.eu/ontology/ns/VitalSystem"
	}	

Response headers	Content-Type	application/ld+json or application/json
Response body	**Example**	

	[
	  {
	  "@context": "http://vital-iot.eu/contexts/system.jsonld",
	  "id": "http://example.com",
	  "type": "vital:VitalSystem",
	  "name": "Sample IoT system",
	  "description": "This is a VITAL compliant IoT system.",
	  "operator": "http://example.com/people#john_doe",
	  "serviceArea": "http://dbpedia.org/page/Camden_Town",
	  "sensors":
	  [
	  "http://example.com/sensor/1",
	  "http://example.com/sensor/2"
	  ],
	  "services":
	  [
	  "http://example.com/service/1",
	  "http://example.com/service/2",
	  "http://example.com/service/3"
	  ],
	  "status": "vital:Running"
	  }
	]

Notes	• More details about the request body can be found in D4.1. • The context of the response body is the JSON-LD context for systems described in Section 5.1 of D3.1.1.

Table 30: Get ICOs.

	Get ICOs	
Description	Returns the ICOs that match the specified criteria.	
URL	DISCOVERY_BASE_URL/ico	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	{
	  "observes": "http://lsm.deri.ie/OpenIot/Temperature",
	  "type": "http://vital-iot.eu/ontology/ns/VitalSensor",
	}	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	[
	  {
	  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	  "id": "http://example.com/sensor/2",
	  "name": "A sensor.",
	  "type": "vital:VitalSensor",
	  "description": "A sensor.",
	  "hasLastKnownLocation":
	  {
	  "type": "geo:Point",
	  "geo:lat": 53.2719,
	  "geo:long": -9.0849
	  },
	  "ssn:observes":
	  [
	  {
	  "type": "openiot:Temperature",
	  "id": "http://example.com/sensor/2/temperature"	

	<pre> }
	    ]
	  }
	]</pre>

Notes	• More details about the request body can be found in D4.1. • The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1.

Table 31: Get services.

	Get IoT services	
Description	Returns the IoT services that match the specified criteria.	
URL	DISCOVERY_BASE_URL/service	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	<pre>{
	  "type": "http://vital-iot.eu/ontology/ns/ObservationService"
	  "system": "http://example.com"
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	<pre>[
	  {
	    "@context": "http://vital-iot.eu/contexts/service.jsonld",
	    "id": "http://example.com/service/3",
	    "type": "vital:ObservationService",
	    "operations":
	    [
	      {
	        "type": "vital:GetObservations",
	        "hrest:hasAddress": "http://example.com/sensor/observation",
	        "hrest:hasMethod": "hrest:POST"
	      }
	    ]
	  }
	]</pre>	

Notes	• More details about the request body can be found in D4.1. • The context of the response body is the JSON-LD context for services described in Section 5.2.2 of D3.1.1.	

4.3 Interface to Filtering

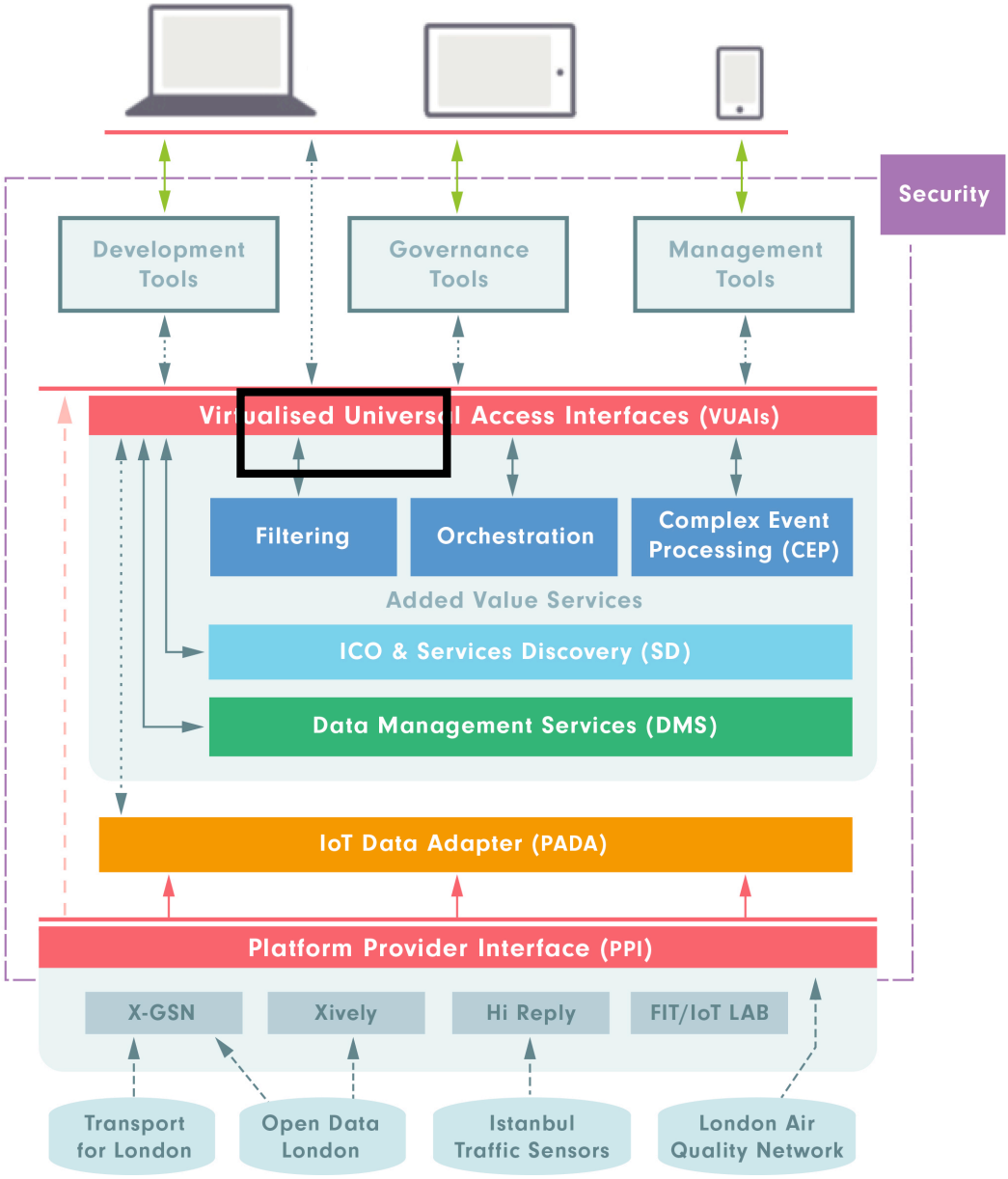

Figure 11: Filtering position in the VITAL architecture.

Filtering offers mechanisms that enable the filtering of data stemming from different sources. Figure 12 shows an example of the interactions between the Orchestrator, the Service Discovery and the Filtering modules. In this case, the Orchestrator directly uses the Filtering module, in order to resample the values collected by a sensor in a specific period in the past. Filtering will apply its filtering functionalities and will send back the results to the Orchestrator.

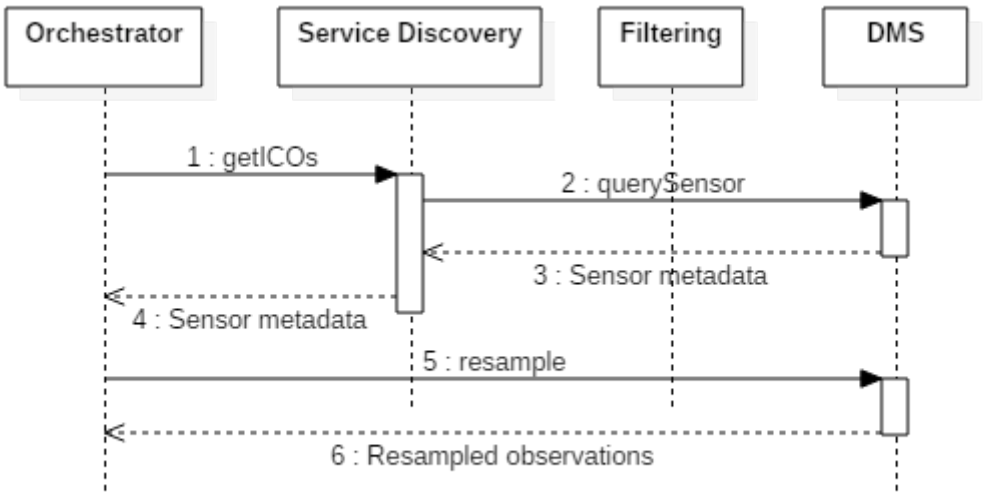

Figure 12: Sample interaction with filtering.

The following tables describe the interfaces that the filtering module provides. More specifically, filtering implements two interfaces: one that enables filtering of numeric observations based on a specific threshold, and one that enables re-sampling.

Table 32: Threshold.

	Threshold	
Description	Returns observations that satisfy the specific criteria.	
URL	FILTERING_BASE_URL/threshold	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example:**
	
	`{`
	`  "ico": "http://example.com/sensor/2",`
	`  "observationProperty": "http://lsm.deri.ie/OpenIoT/Temperature",`
	`  "inequality": "gt",`
	`  "value": 80`
	`}`	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	`[`
	`  {`
	`  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",`
	`  "id": "http://example.com/sensor/2/observation/1",`
	`  "type": "ssn:Observation",`
	`  "ssn:observedBy": "http://example.com/sensor/2",`
	`  "ssn:observationProperty":`
	`  {`
	`  "type": "openiot:Temperature"`
	`  },`
	`  "ssn:observationResultTime":`
	`  {`
	`  "time:inXSDDateTime": "2015-11-09T18:47:32+01:00"`
	`  },`
	`  "dul:hasLocation":`
	`  {`
	`  "type": "geo:Point",`
	`  "geo:lat": "55.701",`	

	<pre>"geo:long": "12.552",
	"geo:alt": "4.33"
	},
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "21.0",
	"qudt:unit": "qudt:DegreeCelsius"
	}
	}
	}
	]</pre>

Notes	• More information about the request body can be found in D4.2. • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.

Table 33: Resample.

	Resample	
Description	Resamples observations from an ICO.	
URL	FILTERING_BASE_URL/resample	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example:**
	<pre>{
	"ico": "http://example.com/sensor/2",
	"observationProperty": "http://vital-iot.eu/ontology/ns/Speed",
	"timeValue": 15,
	"timeUnit": "minute",
	"from": "2015-11-09T17:21:03+01:00",
	"to": "2015-11-09T23:25:03+01:00"
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	<pre>[
	{
	"@context": "http://vital-iot.org/contexts/measurement.jsonld",
	"id": "http://example.com/sensor/2/observation/1",
	"type": "ssn:Observation",
	"ssn:observedBy": "http://example.com/sensor/2",
	"ssn:observationProperty":
	{
	"type": "openiot:Temperature"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2015-11-09T18:47:32+01:00"
	},
	"dul:hasLocation":
	{
	"type": "geo:Point",
	"geo:lat": "55.701",
	"geo:long": "12.552",
	"geo:alt": "4.33"
	},
	"ssn:observationResult":
	{</pre>	

	<pre>"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "21.0",
	"qudt:unit": "qudt:DegreeCelsius"
	}
	}
	}
	]</pre>

Notes	• More information about the request body can be found in D4.2. • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.

4.4 Interfaces to CEP Module

Complex Event Processing (CEP) is an added-value mechanism in the VITAL project. It is responsible for managing event processing over observation streams. A **CEPICO** is an internally generated virtual ICO that represents a CEP instance that is running user-defined DOLCE rules over a specified observation stream. CEPICO generates results of detected complex events and serves them as observations.

CEP provides four interfaces to manage CEPICOs:

- **getcepicos**: returns the metadata of all CEPICOs
- **getcepico**: returns the metadata of the CEPICO with a specific ID
- **createcepico**: receives data source URI and DOLCE specifications as input, and generates a new CEPICO with these specifications. If an ID provided, this operation will update the existing CEPICO with this ID
- **deletecepico**: deletes the CEPICO with a specific ID
- The **getcepicos** interface returns information or metadata of all instantiated CEPICOs. CEP can manage a big number of CEPICOs processing as many DOLCE specifications as the number of CEPICOs. The DOLCE specifications of each CEPICO is not provided in order to not overload the network or the user system.
- In order to get the detailed information of a CEPICO the **getcepico** interface is provided and it requires the ID of the CEPICO.

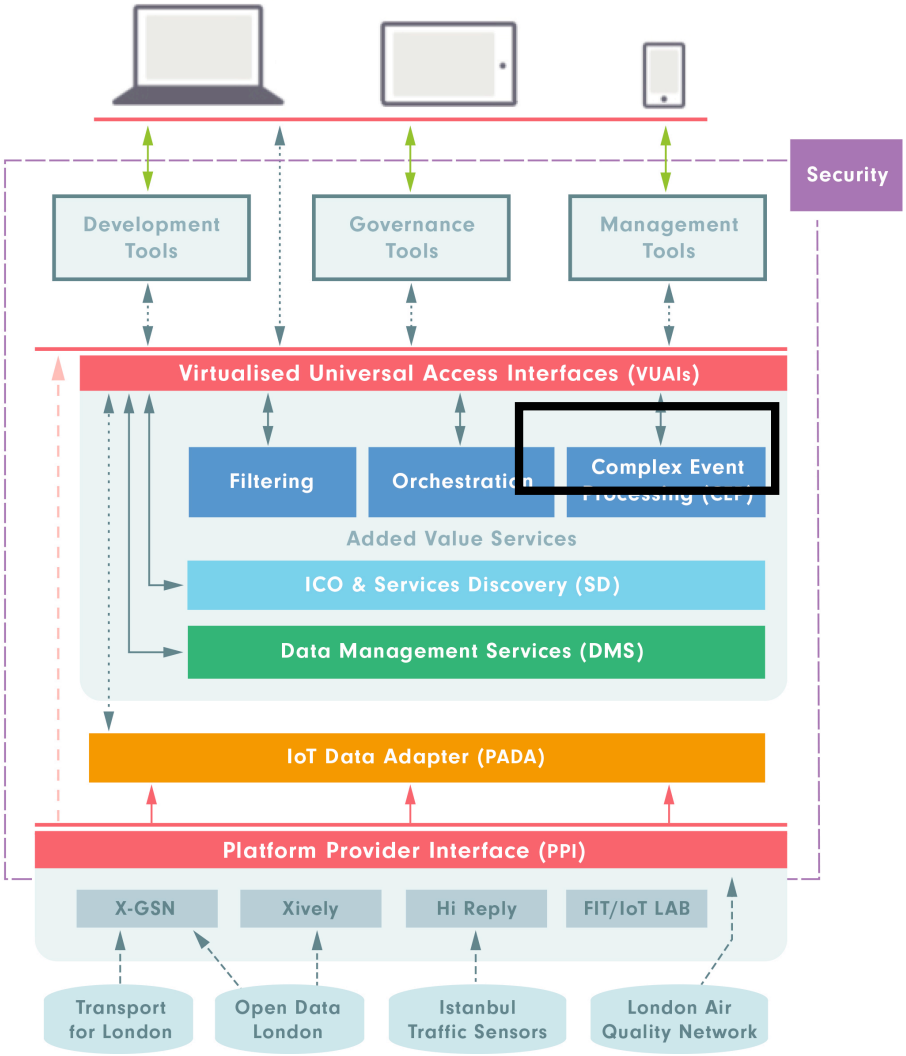

Figure 13: CEP position in the VITAL architecture.

In order to create a CEPICO the **createcepico** interface is provided and the mandatory input fields for this interface are described in Table 34.

Table 34: Fields in the createcepico interface.

Field	Description
source	This field specifies an array of sensor or virtual sensor IDs in order to use their observations as input data for the complex event processing.
dolceSpecification	This field is a JSON object that specifies the DOCE rules that define the filter behaviour of the sensor to be created. The mandatory fields of that object are complex, event, and id. The structure of the DOLCE rules are specified in detail in the D4.3.1.

complex	A JSON array of objects that describe the complex events in DOLCE. Each object is composed by the definition of the complex rule and the id.
event	A JSON array of objects that describe the events in DOLCE. Each object is composed by the definition of the event and the id.

	external	A JSON array of objects that describe the external in DOLCE. Each object is composed by the definition of the external and the id.
	id	String Identifier for the DOLCE specification.

Table 35: Get CEPICOs.

	Get CEPICOs	
Description	Gets metadata about all CEPICOs.	
URL	CEP_BASE_URL/getcepicos	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/ld+json or application/json
Response body	**Example** `[{ "@context": "http://vital-iot.eu/contexts/sensor.jsonld", "ssn:observes": [{ "id": "http://example.com/cep/sensor/1/trafficJam", "type": "vital:ComplexEvent" }, { "id": "http://example.com/cep/sensor/1/slowSpeedAvg75", "type": "vital:ComplexEvent" }], "name": "Traffic Incident CEPICO", "description": "CEPICO for Traffic Incidents.", "source": ["http://example.com/sensor/1", "http://example.com/sensor/2", "http://example.com/sensor/3", "http://example.cm/sensor/4"], "id": "http://example.com/cep/sensor/1", "type": "vital:CEPSensor", "status": "vital:Running" }]`	
Notes	• The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1.	

Table 36: Get CEPICO.

	Get CEPICO	
Description	Gets metadata about a CEPICO with a specific ID.	
URL	CEP_BASE_URL/getcepico	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example**	

<table>
<tr><td></td><td colspan="2"><pre>{
  "id": "http://example.com/cep/sensor/1"
}</pre></td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example<pre>{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
  "ssn:observes":
  [
    {
      "id": "http://example.com/cep/sensor/1/trafficJam",
      "type": "vital:ComplexEvent"
    },
    {
      "id": "http://example.com/cep/sensor/1/slowSpeedAvg75",
      "type": "vital:ComplexEvent"
    }
  ],
  "name": "Traffic Incident CEPICO",
  "description": "CEPICO for Traffic Incidents.",
  "source":
  [
    "http://example.com/sensor/1",
    "http://example.com/sensor/2",
    "http://example.com/sensor/3",
    "http://example.cm/sensor/4"
  ],
  "dolceSpecification":
  {
    "id": "ppDoce",
    "complex":
    [
      {
        "definition": "group id;\n\n   payload{\n\n   string id
= id,\n   float value = value,\n\tpos location = location\n\n
};\n\n   detect Speed where (avg(value)<80 && count(Speed) > 3) in
[40 seconds];\n",
        "id": "trafficJam"
      },
      {
        "definition": "group id;\n payload{\n   string id = id,\n
float value = value,\n   pos location = location\n };\n detect
Speed where (avg(value) < 90) in [30 seconds];",
        "id": "slowSpeedAvg75"
      }
    ],
    "event":
    [
      {
        "definition": " use\n\n   {\n\n   string id,\n\n
pos location,\n\n   float value\n\n   };",
        "id": "Speed"
      }
    ]
  },
  "id": "http://example.com/cep/sensor/1",
  "type": "vital:CEPSensor",
  "status": "vital:Running"
}</pre></td></tr>
<tr><td>Notes</td><td colspan="2">• The request body is a JSON object with the id field, whose value is a CEPICO ID (e.g. http://example.com/sensor/1). The id field is mandatory.
• The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1.</td></tr>
</table>

Table 37: Create CEPICO.

	Create / Update CEPICO	
Description	Creates or updates a CEPICO.	
URL	CEP_BASE_URL/createcepico	
Method	PUT	
Request headers	Content-Type	application/ld+json or application/json

Request body	**Example**
	
	`{`
	`  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",`
	`  "description": "CEPICO for Traffic Incidents.",`
	`  "name": "Traffic Incident CEPICO",`
	`  "source":`
	`  [`
	`  "http://example.com/sensor/1",`
	`  "http://example.com/sensor/2",`
	`  "http://example.com/sensor/3",`
	`  "http://example.cm/sensor/4"`
	`  ],`
	`  "dolceSpecification":`
	`  {`
	`  "id": "ppDoce",`
	`  "complex":`
	`  [`
	`  {`
	`  "definition": "group id;\n\n  payload{\n\n  string id = id,\n  float value = value,\n\tpos location = location\n\n};\n\n  detect Speed where (avg(value)<80 && count(Speed) > 3) in [40 seconds];\n",`
	`  "id": "trafficJam"`
	`  },`
	`  {`
	`  "definition": "group id;\n  payload{\n  string id = id,\n  float value = value,\n  pos location = location\n  };\n  detect Speed where (avg(value) < 90) in [30 seconds];",`
	`  "id": "slowSpeedAvg75"`
	`  }`
	`  ],`
	`  "event":`
	`  [`
	`  {`
	`  "definition": " use\n\n  {\n\n  string id,\n\n  pos location,\n\n  float value\n\n  };",`
	`  "id": "Speed"`
	`  }`
	`  ]`
	`  }`
	`}`	

Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "id": "http://example.com/cep/sensor/1"`
	`}`	

Notes	• The context of the request body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1. • The response body is a JSON object with an **id** field, whose value is the created or updated sensor ID (e.g. http://example.com/sensor/1).	

Table 38: Delete CEPICO.

	Delete CEPICO	
Description	Deletes a CEPICO with a specific ID.	
URL	CEP_BASE_URL/deletecepico	
Method	DELETE	
Request headers	Content-Type	application/json

Request body	**Example**
	`{`
	`  "id": "http://example.com/cep/sensor/1"`
	`}`	

Response headers	-	
Response body	**-**	
Notes	• The request body is a JSON object with the **id** field, whose value is the ID (e.g. http://example.com/sensor/1) of the CEPICO to be deleted. The id field is mandatory.	

Figure 14 shows an example of an interaction between Orchestrator, CEP and DMS. In this case, the Orchestrator creates a CEPICO. As a result, CEP sets up a CEP instance for it, and stores the metadata for the new ICO in DMS. From then on, CEP periodically queries observations stemming from the sources of the CEPICO from DMS, applies the corresponding DOLCE rules, and stores any generated events as observations of the CEPICO in DMS. At any moment, the Orchestrator can query DMS to get observations from the CEPICO, as it would do for observations of an ICO of any other type.

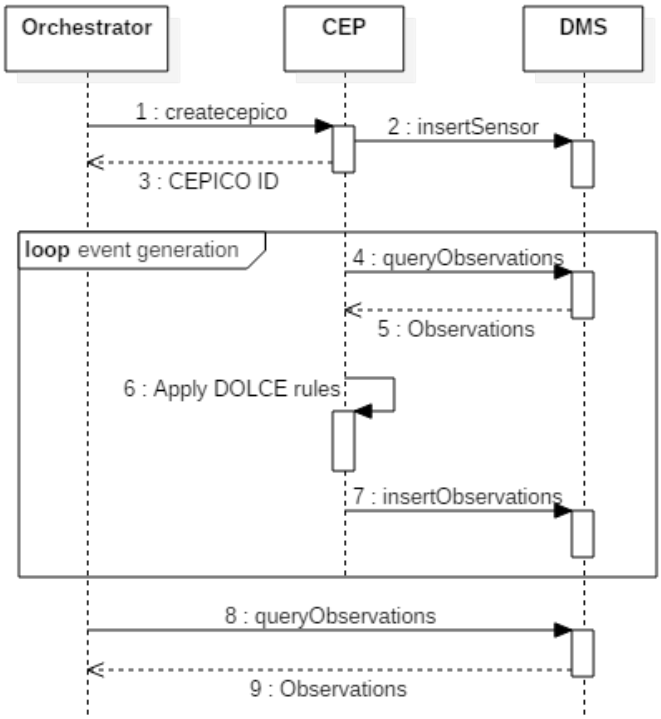

Figure 14: Sample interaction with CEP.

4.5 Interfaces to Workflow Management

Workflow Management (a.k.a. **Orchestrator**) is a higher-level module that offers the functionality to combine systems and services in the VITAL architecture to create more complex meta-services. To support this, it utilizes workflow descriptions generated by users and modeled in a scripting language. Workflows are then deployed, enabled/disabled, removed and executed as new services. This module is described with more details in deliverable D4.4.

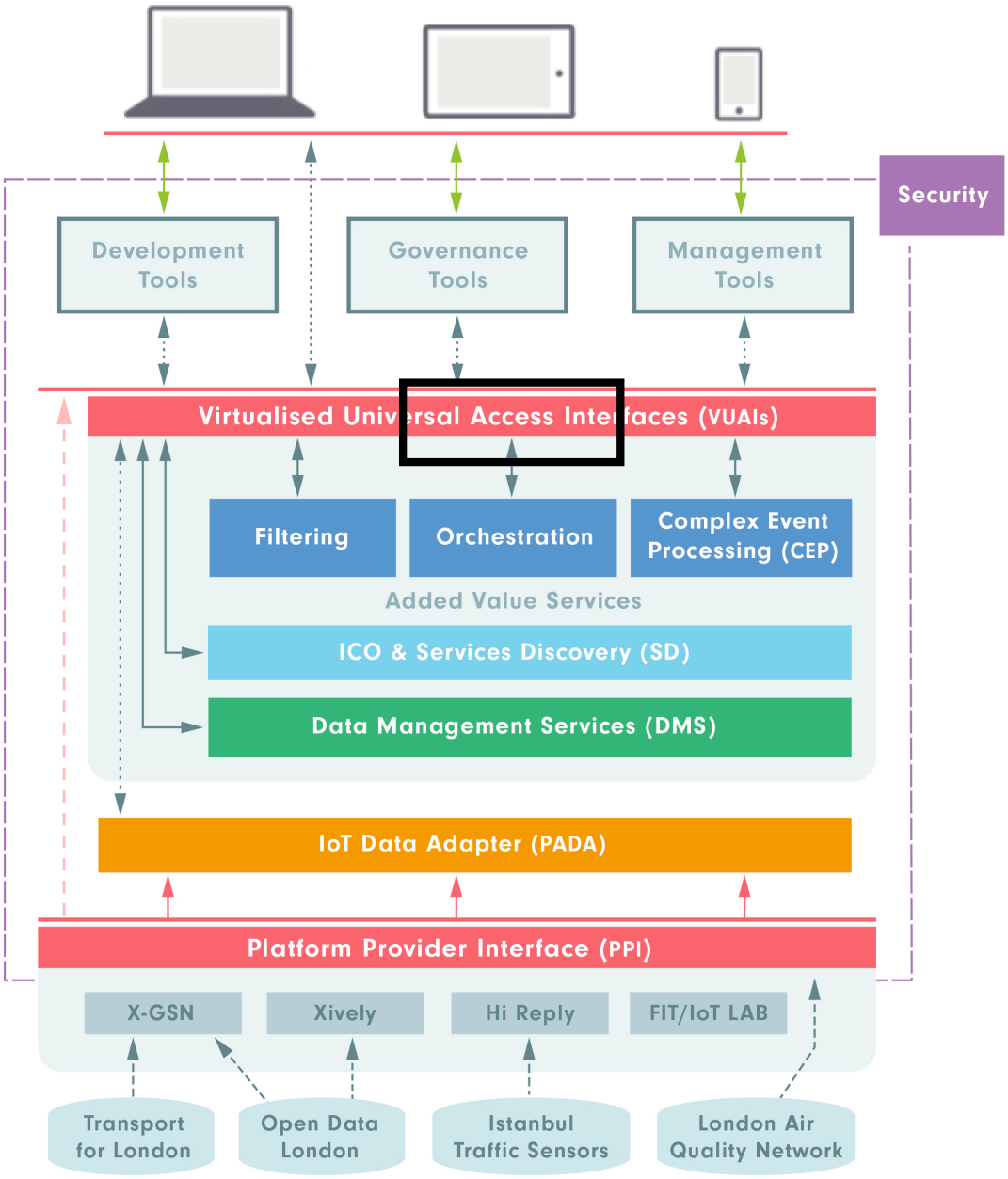

Figure 15: Orchestrator position in the VITAL architecture.

In the rest of the section, the set of APIs exposed for workflow management are outlined.

Table 39: Get operations.

	Get operations	
Description	A method to obtain all operations.	
URL	WM_BASE_URL/operation	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`[`
	`  {`
	`  "id": "AU3DqxXNmcpmpnqLJEl8",`
	`  "name": "GetSystemMetaData",`
	`  "description": "Returns a JSON-LD representation of a system",`
	`  "script": "function execute(input) { <see below> }",`
	`  "inputData": "{n\t\"system\" :`
	`\"http://104.131.128.70:8080/istanbul-traffic\"\n}",`
	`  "dateCreated": 1433506878924`
	`  },`
	`...`
	`]`	

Table 40: Get operation.

	Get operation	
Description	A method to obtain an operation.	
URL	WM_BASE_URL/operation/{id}	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "id": "AU3DqxXNmcpmpnqLJEl8",`
	`  "name": "GetSystemMetaData",`
	`  "description": "Returns a JSON-LD representation of a system",`
	`  "script": "function execute(input) { <see below> }",`
	`  "inputData": "{n\t\"system\" :`
	`\"http://104.131.128.70:8080/istanbul-traffic\"\n}",`
	`  "dateCreated": 1433506878924`
	`}`	

Table 41: Create operation.

	Create operation	
Description	A method to create a new operation.	
URL	WM_BASE_URL/operation	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "name": "GetSystemMetaData",`
	`  "description": "Returns a JSON-LD representation of a system",`
	`  "script": "function execute(input) { <see below> }",`
	`  "inputData": "{n\t\"system\" : \"http://104.131.128.70:8080/istanbul-traffic\"\n}"`
	`}`	

Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "id": "AU3DqxXNmcpmpnqLJE18",`
	`  "name": "GetSystemMetaData",`
	`  "description": "Returns a JSON-LD representation of a system",`
	`  "script": "function execute(input) { <see below> }",`
	`  "inputData": "{n\t\"system\" : \"http://104.131.128.70:8080/istanbul-traffic\"\n}",`
	`  "dateCreated": 1433506878924`
	`}`	

Table 42: Update operation.

	Update operation	
Description	A method to update an operation.	
URL	WM_BASE_URL/operation/{id}	
Method	PUT	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "id": "AU3DqxXNmcpmpnqLJE18",`
	`  "name": "GetSystemMetaData",`
	`  "description": "Returns a JSON-LD representation of a system",`
	`  "script": "function execute(input) { <see below> }",`
	`  "inputData": "{n\t\"system\" : \"http://104.131.128.70:8080/istanbul-traffic\"\n}"`
	`}`	

Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "id": "AU3DqxXNmcpmpnqLJE18",`	

	"name": "GetSystemMetaData", "description": "Returns a JSON-LD representation of a system", "script": "function execute(input) { <see below> }", "inputData": "{n\t\"system\" : \"http://104.131.128.70:8080/istanbul-traffic\"\n}", "dateCreated": 1433506878924 }

Table 43: Delete operation.

	Delete operation
Description	A method to delete an operation.
URL	WM_BASE_URL/operation/{id}
Method	DELETE
Request headers	-
Request body	-
Response headers	-
Response body	-

Table 44: Test operation execution.

	Test operation execution	
Description	A method to test the execution of an operation.	
URL	WM_BASE_URL/execute/operation	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	{
	"name": "GetSystemMetaData",
	"script": "function execute(input) { ... }",
	"inputData": "{ ... }"
	}	

Response headers	Content-Type	application/json

Response body	**Example**
	{
	"name": "GetSystemMetaData",
	"log": [
	...
	],
	"outputData": {...}
	}	

Table 45: Get workflows.

	Get workflows	
Description	A method to obtain all workflows.	
URL	WM_BASE_URL/workflow	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/json

Response body	**Example**
	[
	{
	"id": "AU3D75_-mcpmpnqLJEmC",
	"name": "Speed from filtered sensor list",
	"operationList":
	[
	{
	"name": "GetSensorList",
	"description": "Returns a list of sensors in the system",
	"script": "function execute(input) { ... }"
	},
	{
	"name": "FilterSensors",
	"description": "Filters the list of sensors according to the query",
	"script": "function execute(input) { ... }"
	},
	{
	"id": "AU3DxhgUmcpmpnqLJEl_",
	"name": "GetSpeedFromSensors",
	"description": "Queries each Sensor and returns observations with the specified type",
	"script": "function execute(input) { ... }"
	}
	],
	"dateCreated": 1433511370750
	}
	]	

Table 46: Get workflow.

	Get workflow	
Description	A method to obtain a workflow.	
URL	WM_BASE_URL/workflow/{id}	
Method	GET	
Request headers		
Request body		
Response headers	Content-Type	application/json
Response body	**Example**	


```
{
  "id": "AU3D75_-mcpmpnqLJEmC",
  "name": "Speed from filtered sensor list",
  "status": "ENABLED",
  "operationList":
  [
    {
      "name": "GetSensorList",
      "description": "Returns a list of sensors in the system",
      "script": "function execute(input) { ... }"
    },
    {
      "name": "FilterSensors",
      "description": "Filters the list of sensors according to the
query",
      "script": "function execute(input) { ... }",
    },
    {
      "id": "AU3DxhgUmcpmpnqLJEl_",
      "name": "GetSpeedFromSensors",
      "description": "Queries each Sensor and returns observations
with the specified type",
      "script": "function execute(input) { ... }"
    }
  ],
  "dateCreated": 1433511370750
}
```


Table 47: Create workflow.

	Create workflow	
Description	A method to create a new workflow.	
URL	WM_BASE_URL/workflow	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "name": "Speed from filtered sensor list",`
	`  "operationList":`
	`  [`
	`  {`
	`  "name": "GetSensorList",`
	`  "description": "Returns a list of sensors in the system",`
	`  "script": "function execute(input) { ... }"`
	`  },`
	`  {`
	`  "name": "FilterSensors",`
	`  "description": "Filters the list of sensors according to the query",`
	`  "script": "function execute(input) { ... }",`
	`  },`
	`  {`
	`  "id": "AU3DxhgUmcpmpnqLJEl_",`
	`  "name": "GetSpeedFromSensors",`
	`  "description": "Queries each Sensor and returns observations with the specified type",`
	`  "script": "function execute(input) { ... }"`	

	<pre> }] }</pre>
Response headers	Content-Type: application/json

Response body	**Example**
	<pre>{
	  "id": "AU3D75_-mcpmpnqLJEmC",
	  "name": "Speed from filtered sensor list",
	  "status": "DISABLED",
	  "operationList":
	  [
	    {
	      "name": "GetSensorList",
	      "description": "Returns a list of sensors in the system",
	      "script": "function execute(input) { ... }"
	    },
	    {
	      "name": "FilterSensors",
	      "description": "Filters the list of sensors according to the query",
	      "script": "function execute(input) { ... }",
	    },
	    {
	      "id": "AU3DxhgUmcpmpnqLJEl_",
	      "name": "GetSpeedFromSensors",
	      "description": "Queries each Sensor and returns observations with the specified type",
	      "script": "function execute(input) { ... }"
	    }
	  ],
	  "dateCreated": 1433511370750
	}</pre>

Table 48: Update workflow.

	Update workflow
Description	A method to update a workflow.
URL	WM_BASE_URL/workflow/{:id}
Method	PUT
Request headers	Content-Type: application/json

Request body	**Example**
	<pre>{
	  "id": "AU3D75_-mcpmpnqLJEmC",
	  "name": "Speed from filtered sensor list",
	  "status": "ENABLED",
	  "operationList":
	  [
	    {
	      "name": "GetSensorList",
	      "description": "Returns a list of sensors in the system",
	      "script": "function execute(input) { ... }"
	    },
	    {
	      "name": "FilterSensors",
	      "description": "Filters the list of sensors according to the</pre>

	query",
	"script": "function execute(input) { ... }",
	},
	{
	"id": "AU3DxhgUmcpmpnqLJEl_",
	"name": "GetSpeedFromSensors",
	"description": "Queries each Sensor and returns observations with the specified type",
	"script": "function execute(input) { ... }"
	}
	],
	"dateCreated": 1433511370750
	}

Response headers	Content-Type: application/json

Response body	**Example**
	{
	"id": "AU3DqxXNmcpmpnqLJEl8",
	"name": "GetSystemMetaData",
	"description": "Returns a JSON-LD representation of a system",
	"script": "function execute(input) { <see below> }",
	"inputData": "{n\t\"system\" : \"http://104.131.128.70:8080/istanbul-traffic\"\n}",
	"dateCreated": 1433506878924
	}

Table 49: Delete workflow.

	Delete workflow	
Description	A method to delete a workflow.	
URL	WM_BASE_URL/workflow/{:id}	
Method	DELETE	
Request headers		
Request body		
Response headers	Content-Type	application/json
Response body	No-Content	

Table 50: Test workflow execution.

	Test workflow execution	
Description	A method to test the execution of a workflow.	
URL	WM_BASE_URL/execute/workflow	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example** { "name": "Speed from filtered sensor list",	

	<pre>{
	  "operationList":
	  [
	    {
	      "name": "GetSensorList",
	      "description": "Returns a list of sensors in the system",
	      "script": "function execute(input) { ... }"
	    },
	    {
	      "name": "FilterSensors",
	      "description": "Filters the list of sensors according to the query",
	      "script": "function execute(input) { ... }",
	    },
	    {
	      "id": "AU3DxhgUmcpmpnqLJEl_",
	      "name": "GetSpeedFromSensors",
	      "description": "Queries each Sensor and returns observations with the specified type",
	      "script": "function execute(input) { ... }"
	    }
	  ]
	}</pre>

Response headers	Content-Type: application/json

Response body	**Example**
	<pre>[
	  {
	    "name": "GetSensorObservation",
	    "log":
	    [
	      "Tue Jun 09 16:10:54 EEST 2015: GetSensorObservation",
	      "Tue Jun 09 16:10:55 EEST 2015: End: GetSensorObservation"
	    ],
	    "outputData": { ... }
	  },
	  {
	    "name": "ConvertObservation",
	    "log":
	    [
	      "Tue Jun 09 16:10:55 EEST 2015: ConvertObservation",
	      "Tue Jun 09 16:10:55 EEST 2015: End: ConvertObservation"
	    ]
	    "outputData": { ... }
	  }
	]</pre>

Table 51: Get meta-services.

	Get meta-services	
Description	A method to obtain all meta-services.	
URL	WM_BASE_URL/metaservice	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/json

Response body	**Example**
	
	[
	  {
	  "id": "AU3D75_-mcpmpnqLJEmC",
	  "workflow": {}, // A workflow as described in the previous chapter
	  "dateCreated": 1433511370750
	  }
	]

Table 52: Get meta-service.

	Get meta-service	
Description	A method to obtain a meta-service.	
URL	WM_BASE_URL/metaservice/{:id}	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	  "id": "AU3D75_-mcpmpnqLJEmC",
	  "workflow": {}, // A workflow as described in the previous section
	  "dateCreated": 1433511370750
	}	

Table 53: Deploy meta-service.

	Deploy meta-service	
Description	A method to deploy a meta-service.	
URL	WM_BASE_URL/metaservice	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	{
	  "workflow": {} // A workflow as described in the previous section
	}	

Response headers	Content-Type	application/json
Response body	**Example** { "id": "AU3D75_-mcpmpnqLJEmC",	

	<pre>{
	  "workflow": {}, // A workflow as described in the previous section
	  "dateCreated": 1433511370750
	}</pre>

Table 54: Un-deploy meta-service.

	Un-deploy meta-service
Description	A method to un-deploy a meta-service.
URL	WM_BASE_URL/metaservice/{id}
Method	DELETE
Request headers	-
Request body	**-**
Response headers	-
Response body	-

Table 55: Execute service.

	Execute service.	
Description	A method to execute a service.	
URL	WM_BASE_URL/execute/service/{id}	
Method	PUT	
Request headers	Content-Type	application/json
Request body	The input of the first operation in the flow.	
Response headers	Content-Type	application/json
Response body	The output of the last operation in the flow.	

5 SECURITY OF VIRTUALIZED UNIFIED ACCESS INTERFACES

A solution based on an open-source identity and access management system, supporting Single Sign-On (SSO) and various protocols and authentication methods, has been devised for the VITAL security layer.

A common terminology related to security defines the following roles:

- **Identity Provider (IdP)**: Holds all information about the users (for example information stored in an LDAP directory) and its main duty is to authenticate users and decide what kind of information it shares about them with other parties.
- **Service Provider (SP)**: Is an extra layer in front of the application. Its job is to authorize resource requests, and, if there is no authenticated session at the service provider, to initiate an authentication request to the identity provider.

Identity providers supply user information, while service providers consume this information and give access to secure content.

SAML (Security Assertion Markup Language) is an XML-based, open-standard data format for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. **SAML 2.0** enables web-based authentication and authorization including cross-domain SSO; it is a widely used standard, cross-platform, with many different open-source implementations.

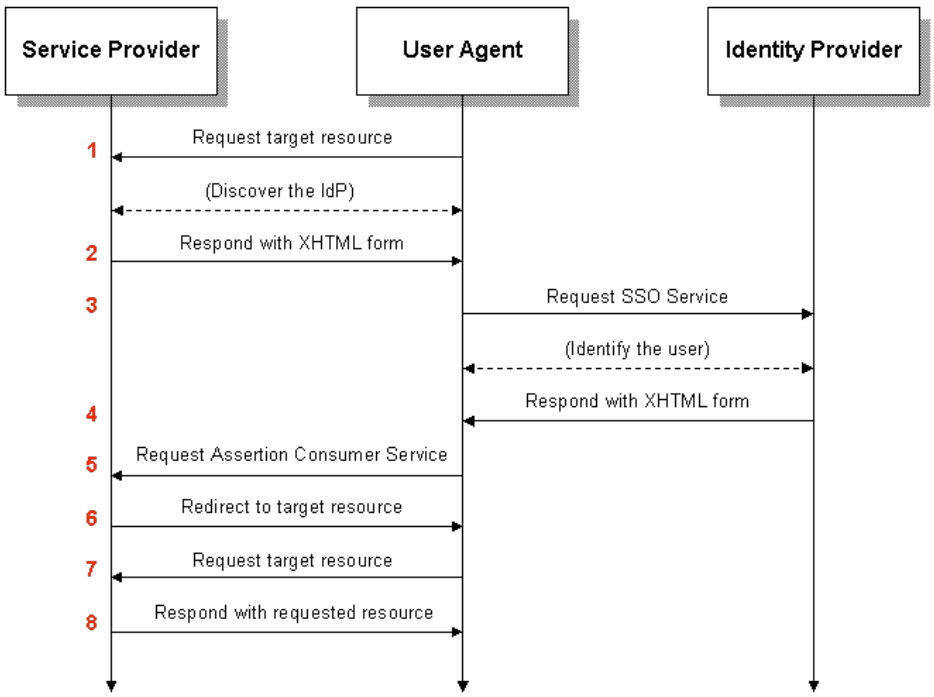

Figure 16: Basic flow of an SAML 2.0 SSO through a web browser using HTTP.

A technology that has been evaluated for use in VITAL is **Shibboleth**. Shibboleth is a free, open-source project that provides “Federated Single Sign-on”, and that implements the HTTP/POST, artifact, and attribute push profiles of SAML, including both identity provider and service provider components. Standards adoption allows the integration between these Shibboleth components and other solutions.

OpenAM is an open-source product by ForgeRock, started as continuation of the Oracle OpenSSO project, designed to provide services for the Web, the Cloud, mobile devices and things. The code of OpenAM is licensed under the CDDL license. Every release of the OpenAM platform, except the most recent one, is free to download. The latest version is available only after a paid subscription, in addition to a more in-depth customer service. OpenAM has a highly scalable, modular, architecture that supports:

- Authentication
- Single Sign-On (SSO)
- Authorization
- Federation
- Entitlements
- Adaptive authentication
- Strong authentication
- Web service security

It also provides a useful REST API to make these features accessible.

OpenAM can exploit a number of policy agents (e.g. web policy agents, J2EE policy agents), provided by ForgeRock, which are software components enforcing policies for OpenAM. In particular, a **Web Policy Agent** installed in a web server can intercept requests from users trying to access a protected web resource, and denies access until the user has authorization from OpenAM to access the resource.

OpenAM and Shibboleth can be combined in different ways (e.g. OpenAM can be used as both the identity and the service provider, OpenAM identity provider can be combined with Shibboleth service provider) to obtain different setups, each with different characteristics. A setup combining OpenAM identity provider federation with Shibboleth service provider solution has been configured for evaluation purposes.

The technologies selected for the VITAL security layer, based on ForgeRock's open-source components, are described in the following sections.

5.1 Overview of VUAI Security Architecture and Implementation

5.1.1 Security Modules and Libraries

OpenAM can manage access to resources available over the network thanks to a centralized access management service that controls:

- who can access what resource
- when a resource can be accessed
- the conditions under which a resource can be accessed

OpenAM handles both authentication (the process that confirms an identity) and authorization (the process that, using policies defined in OpenAM, decides whether to grant access to someone who has been authenticated).

OpenAM can protect a generic web page, an application, a web service or anything accessible over HTTP. Thanks to decoupling policies from applications, if a policy changes or an issue is found after an application is deployed, the only change to do is to update the policy definition in OpenAM, and the application can be left untouched.

OpenAM supports also the following authentication/authorization protocols:

- OAuth 2.0
- OpenID Connect
- SAML 2.0 SSO & Federation

In the **OAuth 2.0 flow**, OpenAM can function as:

- **OAuth 2.0 authorization server**: in this role, OpenAM authenticates resource owners and obtains their authorization in order to return access tokens to clients. OpenAM supports the four main grants for obtaining authorization described in the OAuth 2.0 RFC: the authorization code grant, the implicit grant, the resource owner password credentials grant and the client credentials grant.
- **OAuth 2.0 client**: OpenAM can function as an OAuth 2.0 client for installations where OpenAM protects the web resources. are protected by OpenAM. When OpenAM functions as an OAuth 2.0 client, OpenAM provides an OpenAM SSO session after successfully authenticating the resource owner and obtaining authorization. This means the client can then access resources protected by policy agents. In this respect, the OpenAM OAuth 2.0 client is just like any other authentication module, one that relies on an OAuth 2.0 authorization server to authenticate the resource owner and obtain authorization.

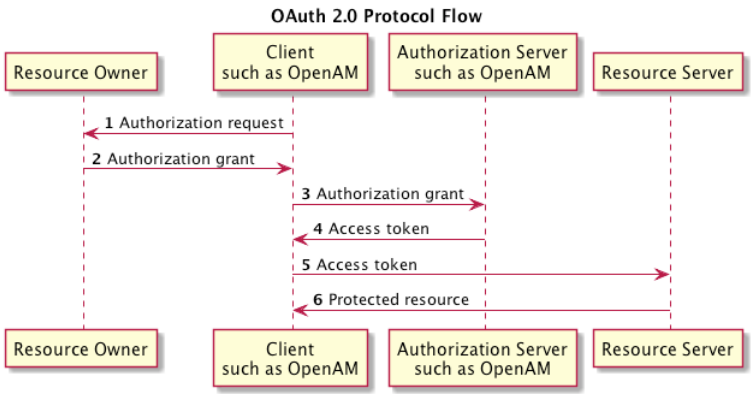

Figure 17: OpenAM in the OAuth 2.0 flow.

In the **OpenID Connect flow**, OpenAM can function as **OpenID Provider**; OpenID Provider holds the user information and grants access. In its role as OpenID Provider, OpenAM:

- allows OpenID Connect relying parties (clients) discover its capabilities
- handles both dynamic and static registration of OpenID Connect relying parties

- responds to relying party requests with authorization codes, access tokens, and user information according to the Authorization Code and Implicit flows of OpenID Connect
- manages sessions

SAML 2.0 SSO is part of federated access management. Federation enables access management cross- organizational boundaries. Federation helps organizations share identities and services without giving away their identity information, or the services they provide.

To achieve SAML 2.0 SSO, OpenAM separates identity providers from service providers. These two components are included in a circle of trust:

- An identity provider stores and serves identity profiles, and handles authentication.
- A service provider offers services that access protected resources, and handles authorization.
- A circle of trust groups at least one identity provider and at least one service provider who agree to share authentication information, with assertions about authenticated users that let service providers make authorization decisions. Providers in a circle of trust share *metadata*, configuration information that federation partners require to access each other's services.
- SAML 2.0 SSO maps attributes from accounts at the identity provider to attributes on accounts at the service provider. The identity provider makes assertions to the service provider, for example to attest that a user has authenticated with the identity provider. The service provider then consumes assertions from the identity provider to make authorization decisions.

Using this feature, OpenAM could be combined with other SAML compliant components, such as Shibboleth ones.

In fact, an alternative to the **OpenAM plus Policy Agent** solution is combining OpenAM and Shibboleth in a solution where OpenAM plays the role of the hosted identity provider and Shibboleth plays the role of remote service provider instead of the Web Policy Agent. OpenAM shares information about users with the service provider and handles the authentication flow. Shibboleth offers services that access protected resources and handles authorization. Shibboleth maps attributes from accounts at the identity provider to attributes on accounts at the service provider. The identity provider makes assertions to the service provider. These attributes are sent along with every assertion and can be used to implement some authorization rules similar to the policy agent ones (e.g. which users or users with what attributes can access a resource).

The solution using ForgeRock's Policy agent has been preferred to the one based on Shibboleth service provider due to the richest REST support. More importantly, the authorization rules that can be defined in the latter solution are very few and less specific than the rules managed by the Web Policy Agent and related to the web server (e.g. Apache HTTP) on which the service provider is running.

Another solution that has been considered is to use, as a service provider, ForgeRock **OpenIG** (**Open Identity Gateway**), based on a reverse proxy architecture. All HTTP traffic to each protected application is routed through OpenIG,

enabling inspection, transformation and filtering of each request. By inspecting the traffic, OpenIG is able to intercept requests that would normally require the user to authenticate, obtain the user's login credentials and send the necessary HTTP request to the target application, thereby logging in the user without modifying or installing anything on the application.

To allow protection of resources retrieved using a PUSH-based mechanism access control can be performed on the subscribe and unsubscribe services (**subscribeToObservationStream** and **unsubscribeFromObservationStream** in the case of PPIs); for these services the authorization process is just as in the "normal" PULL-based scenario.

We should also take into account the case where access to resources would not be triggered by HTTP requests (that can be intercepted by the Policy Agent); it would thus be needed to evaluate on demand whether the user is allowed to access the resource(s) or not. This might be useful also for requests where the resource URI is contained in the request body (it would be necessary to intercept the requests and extract the URIs).

The possibility to perform authorization without the help of a Policy Agent is offered by the VITAL Security Module described in document D5.1.2. This software module exposes a RESTful interface which allows to specify the resource(s) and the user Token ID (obtained at the time of authentication) and returns a decision about which operations are allowed on the specified resource(s) for that user; this software module contacts the OpenAM server to obtain the needed information.

In our scenario, we also want to perform fine-grained data access control, that is when a user requests data to the DMS or directly to PPIs, we wish to return only a set of data that the user is entitled to retrieve. That is why the Security Module exposes an additional endpoint which returns a set of conditions about the documents the user has permissions to access. This information is used by the DMS interfaces and by a PPI gateway to answer to the requester with only the documents the user has a right to retrieve.

5.1.2 Authentication

The authentication can be provided by using a variety of authentication modules connected to identity repositories that store identities and provide authentication services. These identity repositories can be implemented in various technologies, such as LDAP directories, relational databases, one-time password services, other standards-based access management systems and more.

OpenAM allows to chain together the authentication services used to configure stronger authentication for more sensitive resources for example. In the proposed architecture, authentication is provided by using OpenDJ as LDAP directory where users and groups are stored.

Clients request authentication to the identity provider. If the user is authenticated to the identity provider, an authentication token is returned to the user. This is an SSO Token value, an encrypted reference to the session stored by OpenAM. In VITAL the authentication is performed through an endpoint of the Security Module, which takes as form parameters the user credentials and returns some information about the authenticated user while setting the SSO token in a secure cookie.

The token has by default a duration of 120 minutes. This duration is configurable.

5.1.3 Authorization

In the proposed architecture, the authorization can be provided by using OpenAM to manage access policies, thanks to a **Web Policy Agent**. It can be installed as a plugin in the most common web and application servers and it requests policy decision from OpenAM.

The Policy Agent checks the token's validity by querying OpenAM. This allows OpenAM to make policy decisions based on who is authenticated.

Every policy is defined based on:

- **Resources**: The resource definitions constrain which resources, such as web pages or access to the boarding area, the policy applies to.
- **Action**: The actions are verbs that describe what the policy allows or denies users to do to the resources (e.g. GET, POST).
- **Subject conditions**: The subject conditions constrain who the policy applies to (e.g. to all authenticated users, only to administrators or to specific groups of users).
- **Environment conditions**: The environment conditions set the circumstances under which the policy applies.
- **Response attributes**: The response attributes define information that OpenAM attaches to a response following a policy decision (e.g. a name, an email address, etc.)..).

When queried about whether to let a user through to a protected resource OpenAM decides to authorize access or not based on applicable policies. OpenAM communicates its decision to the application using OpenAM for access management. The policy agent installed on the server where the application runs enforces the authorization decision from OpenAM.

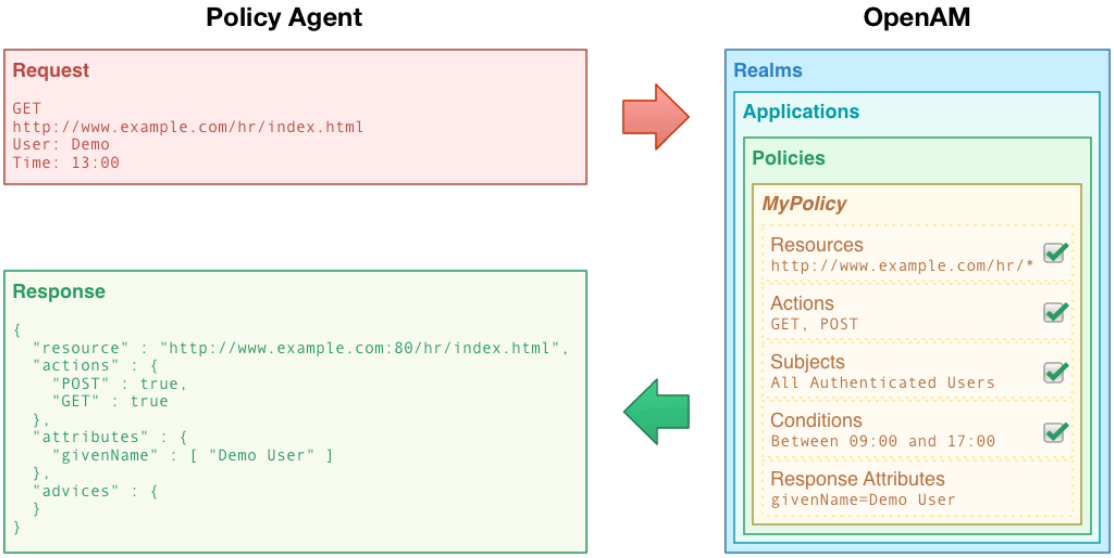

Figure 18: Policy Agent & OpenAM integration

OpenAM relies on policies to reach authorization decisions, such as whether to grant or deny access to a resource. OpenAM acts as the **Policy Decision Point** (**PDP**), whereas OpenAM Policy Agent acts as the **Policy Enforcement Point** (**PEP**). The policy agent takes responsibility only for enforcing a policy decision made by OpenAM. When applications and their policies are configured in OpenAM, OpenAM is also used as the **Policy Administration Point** (**PAP**); to access these features the Security section of the VITAL Management Application, a web interface allowing system administration, can be used; the web application makes requests to the Security Module, which in turn communicates as needed with OpenAM.

When a PEP requests a policy decision from OpenAM, it specifies the target resource(s), the application and information about the subject and the environment. OpenAM as the PDP retrieves policies within the specified application that apply to the target resource(s). OpenAM then evaluates those policies to make a decision based on the conditions matching those of the subject and environment. When multiple policies apply to a particular resource, the default logic for combining decisions is that the first evaluation resulting in a decision to deny access takes precedence over all other evaluations**.** OpenAM only allows access if all applicable policies evaluate to a decision to allow access.

OpenAM communicates the policy decision to the PEP. The concrete decision, applying policy for a subject under the specified conditions, is called an **entitlement**. The entitlement indicates the resource(s) it applies to, the actions permitted and denied for each resource, and optionally response attributes and advice. When OpenAM denies a request due to a failed condition, it can send advice to the PEP, which can then take remedial action.

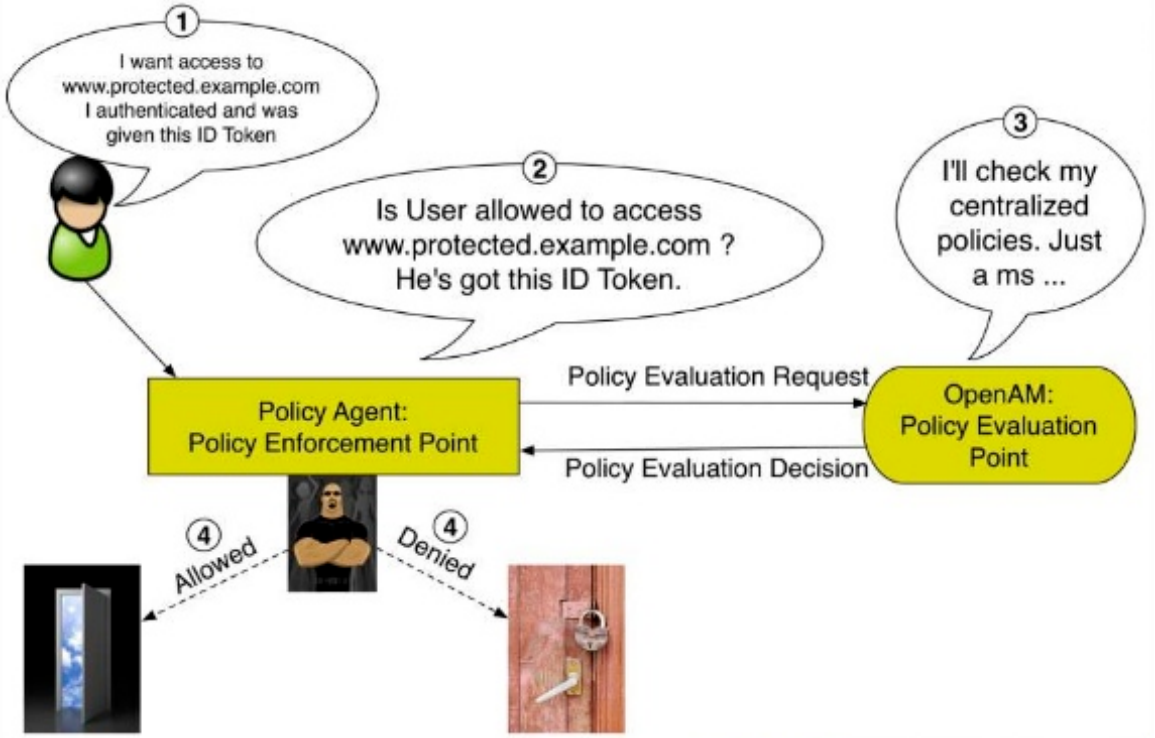

Figure 19: Policy Agent & OpenAM authentication/authorization flow.

The proposed architecture for VUAI security is summarized in Figure 20.

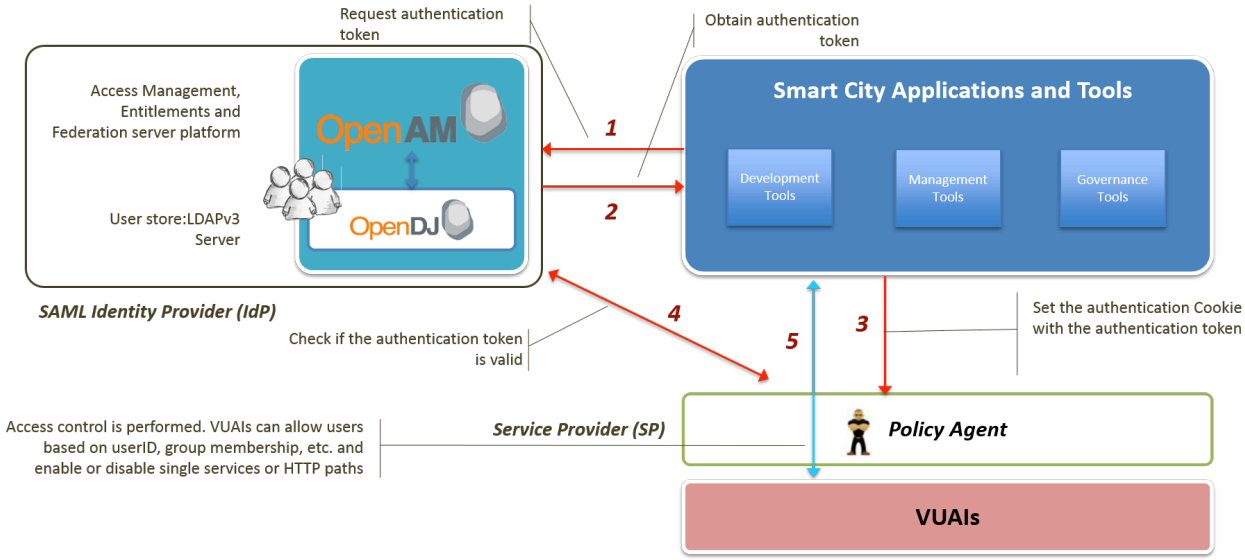

Figure 20: VUAI security architecture overview.

The protection of the VUAIs exposed to the Smart City Applications and Tools follows the same approach as the PPI protection. The policy agent checks the policies defined for a resource, in this case a VUAI endpoint.

Figure 21 shows the proposed architecture for PPI security.

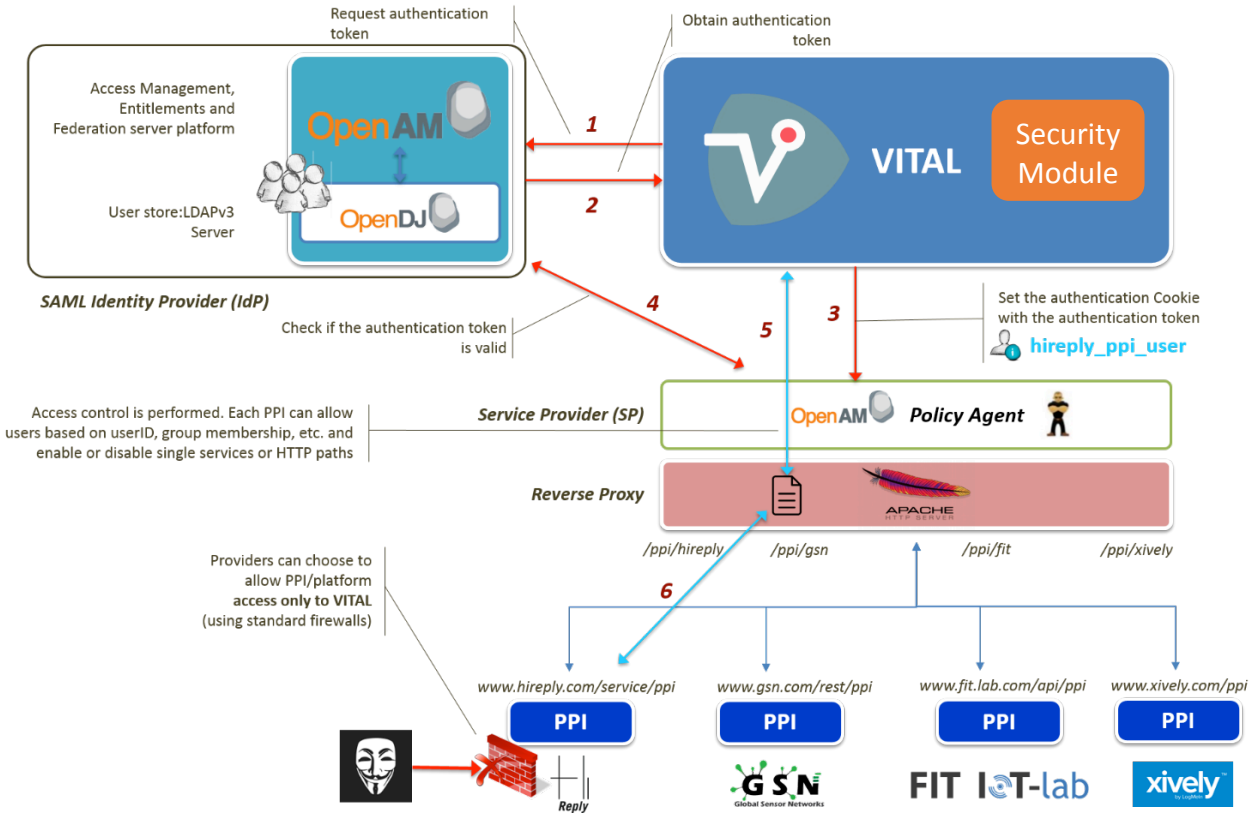

Figure 21: PPI security architecture overview.

The security layer architecture contains four main components:

- **OpenAM identity provider**, described in **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**
- **Policy Agent service provider**, described in **Error! Reference source not found.Error! Reference source not found.**
- **Reverse Proxy** that masks the underlying PPI implementations and VUAIs
- **VITAL Security Module**, the interface to the security services of the system, described more in detail in section 5.7 of D5.1.2
- **PPI gateway**, a module which intercepts the requests to PPIs and returns only the data the user is authorized to get

The OpenAM identity provider can use different data stores, namely Oracle Directory Server, Microsoft Active Directory, IBM Tivoli Directory Server, ForgeRock OpenDJ. For VITAL we are currently using OpenDJ data store; OpenDJ is a high performance and highly scalable open source LDAPv3 compliant server. Moreover, it is fully integrated with OpenAM. OpenDJ comes embedded as a configuration store and it exposes a RESTful API.

The choice of using a reverse proxy provides high flexibility; with this solution, it is possible to attach different services behind the proxy with simple configurations and a firewall configuration for each service. In this way, there is a single point of security enforcement and the overall system is easily extendable and scalable.

Using OpenAM, VITAL services will authenticate to the identity provider using *ad-hoc technical users* (when and where needed) and will obtain the *access token* that the Policy Agent will use to grant or not access to the specific requested location. Otherwise, the token coming from the user authentication (performed by the application) may be used to authorize access to data and services based on the original requester rights.

On the reverse proxy, a location is mapped for every PPI and VUAI.

The main advantage of using OpenAM plus the Web Policy Agent is that it is not needed to modify, redeploy or redesign the applications that need a security layer each time that access rules are changed, as OpenAM handles everything. In addition. Also, OpenAM exposes RESTful APIs that enable JSON or XML over HTTP, allowing users to access authentication, authorization, and identity services from web applications using simple REST clients.

The Web Policy Agent is completely integrated with OpenAM, fully compatible and REST friendly. In addition, the Web Policy Agent makes it possible to define very detailed rules for the authorization flow with a high granularity (e.g. rules on group membership or on user account).

These two components can be used in the same solution and make it possible to integrate and federate different services in a way that is language and platform agnostic and totally REST friendly.

The additional custom Security Module of VITAL then acts as a central component allowing in one place easy access to all the security functions of the system adapting the OpenAM based solution to the needs of VITAL. The modulemodule, in coordination with the other modules of the system, can also perform a more fine-grained control over access to data when the policy agent cannot (because data is not identified by the URI of the HTTP request); this situation is typically that of a query to retrieve data from the DMS or the PPIs. In the case of the former, the DMS retrieves from the Security Module a set of conditions on values that the attributes of the documents to retrieve must or must not have; these define the documents that the user has the right to access. The DMS then simply adds these conditions to the user MongoDB query to retrieve the data from the database. In the case of PPIs, in order to have one point of enforcement and not to ask every stakeholder wishing to add its IoT system to integrate this in its specific PPI, a gateway is used. Requests to PPIs go through this module, which forwards the requests to the right endpoint and properly filters the data based on user permissions obtained from the Security Module.

5.2 Example: PPI Authentication and Authorization

The Security Module exposes a REST API for authentication, in the form of a HTTP POST endpoint. Username and password must be included in the request body; the endpoint returns a set of information about the authenticated user and sets on the requester the **vitalAccessToken** cookie with the SSO token obtained from OpenAM as a value (in order to achieve this the appropriate response header is set), as shown in Table 56. The token needs to be set as a cookie in order to read by the service provider, the Policy Agent.

Table 56: Request for authentication token.

	Get authentication token	
Description	This is the authentication endpoint, where a session token can be requested.	
URL	SECURITY_MODULE_BASE_URL/rest/authenticate	
Method	POST	
Request headers	Content-Type	application/x-www-form-urlencoded
Request body	name=<username> password=<password> testCookie=<false/true>	
Response headers	Content-Type	application/json
	Set-Cookie	vitalAccessToken=AQI[…]yNg..*; Domain=.vital.security.domain; Path=/; secure; HttpOnly

Response body	**Example**
	
	{
	"uid":"userId",
	"name":"FirstName",
	"fullname":"Name Surname",
	"creation":{	

	<pre>"year":"2015",
	"month":"November",
	"day":"03"
	},
	"mailhash":"b18b8f86c170346c0c55da21b2646168"
	}</pre>

The **vitalAccessToken** cookie must be set in all the requests. Using the cookie, all the PPI services exposed under the Hi Reply endpoint, for example, will be accessible as shown in Table 57.

Table 57: Request for a protected resource.

	Get metadata about Hi Reply	
Description	This service can be used to get metadata about Hi Reply.	
URL	https://www.hireply.com/metadata	
Method	POST	
Request headers	Content-Type	application/json
	Cookie	vitalAccessToken=AQICGU4ok.*ANTc3MTM2MjY5Ng..*

Request body	**Example**
	
	{
	}	

Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	...
	}	

At the moment of the request, the Policy Agent checks the SSO token and applies the policies defined at the identity provider (as described in Section 5.1) to grant or not authorization to the requested resource.

5.3 Example: VUAI Authentication and Authorization

As described in Section **Error! Reference source not found.**, VUAIs enable unified access to external IoT systems, and to VITAL components. Authentication and authorization to VUAIs can be provided in the same way as described in Section 5.2. Policies can be defined based on user membership to certain groups, representing roles significant to each VUAI.

6 PPI PROTOTYPE IMPLEMENTATIONS

6.1 PPI Implementation for Open Data Sources

At the moment, we have implemented PPI for the following open data feeds:

- for the footfall data feed, provided by Springboard
- for the bus arrival data feed, provided by Transport for London (TfL)

More details about each implementation are given in the following sections.

6.1.1 PPI Implementation for Footfall Data Feed (Camden)

Springboard has provided us with access to a **footfall data feed** that contains values collected in an hourly basis by small counting devices installed in several locations across **Camden**. Each value in the data feed contains (among others) the following pieces of information:

- when the value was observed
- where the value was observed (i.e. in which location)
- how many people were counted walking up that location
- how many people were counted walking down that location

In order to read values from that data feed within a specific time range, we need to submit an appropriate HTTP GET request to a pre-defined endpoint. The response to that request contains the requested data in **JSON** format.

We decided that, in the case of the footfall data, each location represents a sensor that observes two properties: (1) the number of people walking up that location, and (2) the number of people walking down that location.

We also manually stored information about the locations, where counting devices are mounted, into **MongoDB**.

Then we moved on the actual implementation of PPI, which in essence was (part of) a Java web application deployed in **Wildfly** application server. We first set up a task that pulls values periodically (once every hour) from the data feed, and stores them into MongoDB. We then implemented the following PPI services:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements
- Get supported performance metrics
- Get performance metrics

We exposed the PPI services as RESTful web services using **RESTEasy**, a JAX-RS implementation. For the sensor measurements, only the pull-based mechanism is supported. As for the performance metrics, the current implementation uses **JavaMelody** to monitor the following performance metrics:

- Available memory

- Errors
- Maximum number of requests
- Pending requests
- Served requests
- System load
- System uptime
- Used memory

Table 58 lists the technologies, libraries, and tools used in the implementation of PPI for Springboard's footfall data feed.

Table 58: Technologies used in footfall data feed PPI implementation.

Framework/Library	Version
MongoDB	3.2
Java Platform, Enterprise Edition	8
JavaMelody	1.56.0
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

6.1.2 PPI Implementation for Bus Arrival Data Feed

Transport for London (**TfL**) provides (among others) a live data feed with bus arrival information. Each value in that data feed contains (among others) the following:

- the bus stop where a bus is expected to arrive
- the route (i.e. the line) of that bus
- the destination of that bus
- the direction of that bus
- when that bus is expected to arrive at that bus stop

The data feed is refreshed every 30 seconds. By submitting an HTTP GET request to a specific endpoint, we can retrieve bus arrival information for the next 30 minutes in CSV format.

In this case, we decided to map each bus stop to a sensor that observes a single composite property: the line, the direction and the arrival time of each bus arrival that involves it.

We implemented PPI as part of a Java web application that we deployed in **Wildfly** application server. In order to feed the application with data, we manually stored information about bus stops (provided also by TfL), and we also set up a task to fetch data from the feed every 10 minutes. We used **MongoDB** to store all this data.

Finally, we implemented and exposed as RESTful web services (using RESTEasy) only the mandatory PPI primitives, namely:

- Get IoT system metadata
- Get sensor metadata
- Get sensor observations

Error! Reference source not found. summarizes the technologies, libraries, and tools used in the implementation of PPI for TfL's live bus arrival data feed.

Table 59: Technologies used in live bus arrival data feed.

Framework/Library	Version
MongoDB	3.2
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

6.2 PPI Implementation for X-GSN

X-GSN (**eXtended Global Sensor Networks**) constitutes an enhanced version of the GSN platform that has been developed as part of the OpenIoT project. The main enhancements introduced by X-GSN are:

- the semantic annotation of both sensors and measurements
- the ability to connect to a data management service, like the **Linked Sensor Middleware** (**LSM**) component of OpenIoT, that will store both data and metadata into the Cloud

Since both X-GSN and VITAL use the **Semantic Sensor Network** (**SSN**) ontology to semantically annotate sensors and the observations collected by them, there was no need for any mapping between X-GSN and VITAL entities.

The PPI implementation for X-GSN is a Java web application that we deployed in **Wildfly** application server. In this first prototype, we implemented only the services mandated by the PPI specification, namely:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements

The implementation of the last two PPI services involved only the execution of the appropriate SPARQL queries to the **Virtuoso** instance where LSM stores sensor data and metadata. In these cases, the X-GSN PPI implementation acts in effect as an LSM client.

Table 60 lists the technologies, libraries, and tools used in the implementation of PPI for the X-GSN platform.

Table 60: Technologies used in X-GSN PPI implementation.

Framework/Library	Version
OpenLink Virtuoso	7.2.0.1
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

As implied above, the PPI implementation we provided for X-GSN assumes that X-GSN is connected to LSM for data management purposes.

6.3 PPI Implementation for Xively

Xively is an IoT **Platform as a Service (PaaS)** that enables its users to store their data, query them, and even receive notifications when certain conditions are met (e.g. when a new value has been added). Xively exposes information about its resources (e.g. feeds, datastreams, devices) through a **REST** API in XML, JSON and/or CSV formats.

We decided to map each feed (which in Xively is connected with a single device that is in turn connected with a single product) to a VITAL sensor that observes one property for each datastream that has been added to that feed. Metadata about a sensor are retrieved from the attributes of the corresponding product, device and feed, whereas metadata about a property are retrieved from the attributes of the corresponding datastream. As it is already obvious, datapoints in datastreams are mapped to sensor measurements in VITAL.

We included the PPI implementation for the Xively platform into a Java web application that we deployed in **Wildfly** application server. As part of this implementation, we implemented and exposed (through REST) the following PPI services:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements
- Subscribe to observation stream
- Unsubscribe from observation stream

For the implementation of all PPI primitives we used the REST API exposed by Xively. Especially for implementing the push-based access to sensor measurements, we used triggers that Xively provides support for.

Table 61 summarizes the technologies, libraries, and tools used in the implementation of PPI for the Xively platform.

Table 61: Technologies used in Xively PPI implementation.

Framework/Library	Version
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

This particular PPI implementation has not been tested yet, since we do not have an account for the Xively platform. Xively provides accounts at will and only after request, and we have not received a reply to our requests.

6.4 PPI Implementation for FIT/IoT-LAB

6.4.1 Implementation Overview

FIT IoT-LAB[1] is a platform that provides a very large scale infrastructure facility suitable for testing small wireless sensor devices and heterogeneous communicating objects. FIT IoT-LAB testbeds are located in six different sites across France, which gives forward access to overall 2728 wireless sensor nodes.

Within the VITAL context, the FIT's PPI is a Java application subdivided in 3 logical layers, REST interface, Core and Sensors Communication.

[1] http://iot-lab.info/

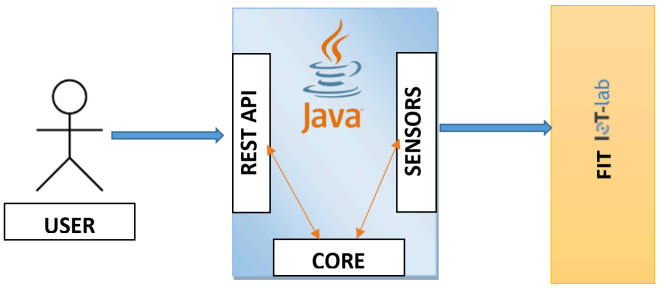

Figure 22: FIT IoT-Lab representation.

REST interface is used in order to allow clients (through a pull-based mechanism) to query the system. We implemented for the PPI the mandatory functions regarding:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements

The results for those requests are returned in JSON-LD format, compliant to the format considered by all the PPI data representations.

To the follow we provide a practical example to clarify step sequences involved.

6.4.2 Examples

One possible example of interaction between the FIT's PPI and a client could be the request of a list of ICOs registered in the platform.

In this case, the client sends a request :

1. Wildfly intercepts the request.
2. The method associated to the request is invoked and invokes the *Core.*
3. The *Core* component analyzes the request and forwards a command to the *Sensor Communication Layer* (SCL).
4. The SCL communicates with FIT platform in order to retrieve the requested parameters.
5. The result is returned back to the *Core* Layer which applies the conversion to JSON-LD format.
6. The result is then returned back to the client that performed the request.

6.5 PPI Implementation for Hi Reply

6.5.1 Implementation Overview

The PPI implementation for the Hi Reply platform is a Java application, whose architecture is shown in Figure 23.

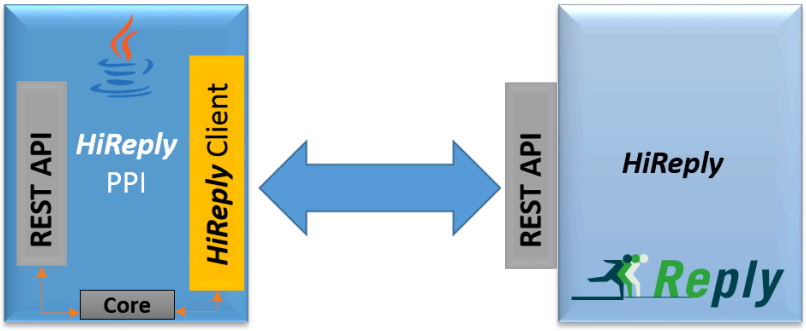

Figure 23: PPI implementation for the Hi Reply platform.

Hi Reply exposes REST endpoints that allow clients to query information about:

- data (e.g. sensor measurements)
- the Hi Reply system (e.g. metrics)

The responses returned by Hi Reply are in XML format. All data is retrieved by a pull-based mechanism.

Table 62 lists the libraries and technologies used for the Hi Reply PPI implementation.

Table 62: Technologies used in Hi Reply PPI implementation.

Framework/Library	Version
WildFly	Any
Jackson	2.6.3
Java (OpenJDK)	8
JAXB2	2.2
JAX-RS	3.0.10
jsonschema2pojo	0.4.15
Log4j	2.4.1
Maven	3.3.9
Apache Commons Lang	2.6
HttpClient	4.5.1
javax	1.2

Hereafter, Hi Reply PPI components are described.

6.5.1.1 Hi Reply Client

The **HiReply Client** component implements methods to query the Hi Reply platform by issuing HTTP GET requests to the defined endpoint where Hi Reply is running.

Every XML response is mapped to a Java POJO class thanks to JAXB and Jaxen libraries. POJO classes are generated with the JAXB Maven plugin; this plugin, starting from an XSD file, that defines the schema of Hi Reply XML models, generates a Java class that maps the XML structure defined in the XSD file.

Thus, the Hi Reply Client wraps all the requests and responses to/from Hi Reply. The implemented methods in this client are used by the upper level Core component to prepare data to be exposed via PPI.

6.5.1.2 Core & Rest API

The **Core** component implements all the methods mandated by the PPI specification. Here, all received requests are analyzed, and then the Hi Reply Client is in charge of retrieving the requested information.

The Core component uses JAX-RS, a Java API that provides support for creating web services according to the **REpresentational State Transfer** (**REST**) architectural pattern. Using annotations, methods defined in a Java class can be exposed at the defined resource path.

PPI uses JSON-LD to represent data. In order to map information retrieved from Hi Reply in JSON-LD format, two libraries have been used:

- **Jackson**: Serializes a POJO to a JSON string or deserializes a JSON string to the POJO that this JSON string represents.
- **jsonschema2pojo**: A Maven plugin that generates Java classes starting from JSON schemas.

The Core component uses these libraries to parse each request, execute the proper Hi Reply Client operation, and return the correct response.

WildFly has been used to deploy the application and expose the JAX-RS resources on the specified host and port. WildFly associates the PPI context to the module and forwards the requests to the right endpoints.

In order to retrieve information about performance metrics, two classes implementing ContainerRequestFilter and ContainerResponseFilter interfaces have been created in the Core component. These classes are called whenever a request is performed on an endpoint and when the request is served (with or without errors) and thus allow tracking the entire lifecycle of requests.

Using these classes and the information they provide, performance metrics can be calculated at the PPI level. In particular the following metrics are computed:

- Active requests (ContainerRequestFilter method called, but not ContainerRersponseFilter one)
- Errors (ContainerResponseFilter method called with error status)
- Served requests (both classes methods called without errors)

6.5.2 Examples

This section provides an example of the execution flow of a request to the PPI.

Suppose that a client sends a HTTP request to http://*www.hireply.com/iot/hireply/perf/upTime*.

1. WildFly intercepts the request.
2. The method associated to the request is invoked.
    a. The Core component invokes the Hi Reply Client that in turn issues to the Hi Reply platform the request to retrieve the timestamp when the platform started running.
    b. The Hi Reply Client returns an object that maps the XML response.
    c. The timestamp field is retrieved from the object returned by the client.
    d. Timespan (in ms) between the start time and the request time is calculated.
    e. A POJO that will map to the expected JSON-LD response (defined in 3.2.6.1) is then instantiated and filled with the correct data.
    f. Using the Jackson library, the created POJO is serialized to a JSON-LD string.
    g. The string is returned by the method.

In the meantime, the event handler registers the event lifecycle. After the ContainerRequestFilter has been called, the event is considered as a "pending request" as long as ContainerResponseFilter is not invoked too. After that, the event is pulled from the "pending request" queue and the counter of the served requests is incremented. If the event throws an exception during his execution, the total error number is incremented.

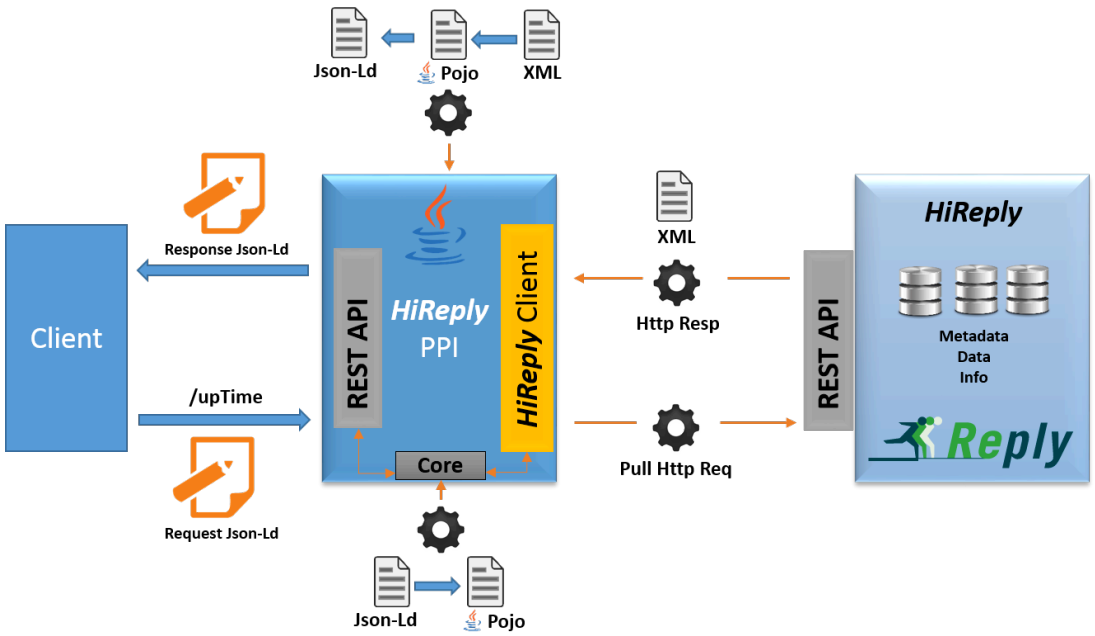

Figure 24: Example flow.

7 CONCLUSIONS

VITAL relies on a number of abstract virtualized interfaces to access (IoT-based) data streams and services in smart cities regardless of the IoT system (platform, data sources) that enables these functionalities. Earlier sections have provided an initial specification of VITAL's abstract interfaces for accessing IoT data and services in a platform-agnostic way. These interfaces have been classified in three categories: namely 1) interfaces to platforms, 2) interfaces to added-value functionalities of the VITAL platform and 3) interfaces to data processing functions. For each of the categories we have presented the interfaces. In the case of platform and added-value functionality access, the interfaces have been described at a very fine level of detail, i.e. down-to-implementation detail. This is not the case with the specification of the data processing functions, which is still in less mature state, both in terms of completeness and in terms of detail.

In parallel with the specification of the above-listed abstract interfaces, the partners have undertaken a significant step towards the realization of the interfaces. This is for example the case with the implementation of the PPIs over the four IoT platforms selected (in WP2): PPI implementations have been already realized (over X-GSN, FIT, Hi REPLY and Xively.com) and they are under testing and validation. Moreover, the implementation of the interfaces for the added-value functionalities is also in progress, as part of the implementation of the respective modules in WP4 and WP5 of the project. As expected, the implementation of the data processing interfaces is still in its infancy, given the need to finalize the selection of the open-source tools that will empower the realization of the VITAL statistical and BigData processing functionalities.

8 REFERENCES

[Di Ciaccio12] Agostino Di Ciaccio, Mauro Coli, Jose Miguel Angulo Ibanez Eds.) «Advanced Statistical Methods for the Analysis of Large Data-Sets», Series: Studies in Theoretical and Applied Statistics, Subseries: Selected Papers of the Statistical Societies 2012, XIII, 484p.

[Marz14] Nathan Marz, James Warren, «Big Data: Principles and Best Practices of Scalable Real-time Data Systems», Feb 2014, Paperback ISBN13: 9781617290343, ISBN10: 1617290343

[Sioshansi11] Fereidoon P. Sioshansi, «Smart Grid: Integrating Renewable, Distributed & Efficient Energy», November 2011, ISBN-10: 0123864526 | ISBN-13: 978-0123864529.

[JSONLDSupport] https://github.com/openlink/virtuoso-opensource/issues/478, Accessed at: December 22, 2015

Vital 2016

