This project has received funding from the European's Union Horizon 2020 research innovation programme under Grant Agreement No. 957258

Architecture for Scalable, Self-human-centric, Intelligent, Secure, and Tactile next generation IoT

D4.2 Core Enablers Specification and Implementation

Deliverable No.	D4.2	**Due Date**	30-APR-2022
Type	Report	**Dissemination Level**	Public
Version	1.0	**WP**	WP4
Description	Core specification and implementation status of Smart IoT Devices, Edge Nodes and enablers of the horizontal planes of ASSIST-IoT. Intermediate specification and first version of implementation of components of horizontal planes.		

Copyright

The ASSIST-IoT consortium consists of the following 15 partners:

UNIVERSITAT POLITÈCNICA DE VALÈNCIA	Spain
PRODEVELOP S.L.	Spain
SYSTEMS RESEARCH INSTITUTE POLISH ACADEMY OF SCIENCES IBS PAN	Poland
ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS	Greece
TERMINAL LINK SAS	France
INFOLYSIS P.C.	Greece
CENTRALNY INSTYUT OCHRONY PRACY	Poland
MOSTOSTAL WARSZAWA S.A.	Poland
NEWAYS TECHNOLOGIES BV	Netherlands
INSTITUTE OF COMMUNICATION AND COMPUTER SYSTEMS	Greece
KONECRANES FINLAND OY	Finland
FORD-WERKE GMBH	Germany
GRUPO S 21SEC GESTION SA	Spain
TWOTRONIC GMBH	Germany
ORANGE POLSKA SPOLKA AKCYJNA	Poland

Disclaimer

Authors

Name	Partner	e-mail
Alejandro Fornés	P01 UPV	alforlea@upv.es
Francisco Mahedero	P01 UPV	framabio@upv.es
Rafael Vañó	P01 UPV	ravagar2@upv.es
Ignacio Lacalle	P01 UPV	iglaub@upv.es
Eduardo Garro	P02 PRO	egarro@prodevelop.es
Ernesto Calas	P02 PRO	ecalas@prodevelop.es
Miguel Llacer	P02 PRO	mllacer@prodevelop.es
Paweł Szmeja	P03 IBSPAN	pawel.szmeja@ibspan.waw.pl
Piotr Sowiński	P03 IBSPAN	piotr.sowinski@ibspan.waw.pl
Evripidis Tzionas	P04 CERTH	tzionasev@iti.gr
Georgios Stavropoulos	P04 CERTH	stavrop@iti.gr
Nikolaos Vrionis	P06 INFOLYSIS	nvrionis@infolysis.gr
Theoni Dounia	P06 INFOLYSIS	tdounia@infolysis.gr
Aggeliki Papaioannou	P06 INFOLYSIS	apapaioannou@infolysis.gr
Alex van den Heuvel	P09 NEWAYS	alex.van.den.heuvel@newayselectronics.com
Ron Schram	P09 NEWAYS	Ron.Schram@newayselectronics.com
Fotios Konstantinidis	P10 ICCS	fotios.konstantinidis@iccs.gr
Tina Katika	P10 ICCS	tina.katika@iccs.gr
Thomas Papaioannou	P10 ICCS	thomas.papaioannou@iccs.gr
Aristeidis Dadoukis	P10 ICCS	aristeidis.dadoukis@iccs.gr
Konstantinos Routsis	P10 ICCS	konstantinos.routsis@iccs.gr
Zbigniew Kopertowski	P15 OPL	Zbigniew.Kopertowski@orange.com

History

Date	Version	Change
21-Dec-2021	0.1	ToC presented
7-Apr-2022	0.2	First round of contributions completed
14-Apr-2022	0.3	Integration and minor modifications
19-Apr-2022	0.4	Second round of contributions completed. Ready for internal review
28-Apr-2022	0.9	Integration of changes from IR
29-Apr-2022	1.0	Final version submitted to EC

Key Data

Keywords	Enablers, Edge nodes, Smart IoT devices, Implementation
Lead Editor	Eduardo Garro (P02 - PRO), Alejandro Fornés (P01 - UPV)
Internal Reviewer(s)	Ignacio Lacalle (P01 - UPV), Francisco Blanquer (P05 - TL)

Executive Summary

This deliverable is written in the framework of WP4 – Core enablers design and development of **ASSIST-IoT** project under Grant Agreement No. 957258. The document gathers the work and outcomes of the four tasks of the work package in the period M9-M18, which are devoted to the design and implementation of enablers and hardware elements required to implement the different planes of the ASSIST-IoT architecture.

The realisation of the ASSIST-IoT architecture requires the design and development of specific elements, both software and hardware, that support the exposed functionalities and in the way they have been conceived: these are the Smart IoT Devices, the Edge Nodes and the enablers of the architecture. Being the second version of a series of three deliverables, this document updates and extends the specifications presented in the first deliverable (D4.1), accompanied by the software artifacts (i.e., enablers) developed so far.

Regarding the **hardware specifications** of the equipment designed and (to be) produced within the project, the schematic design, the PCB layout of ASSIST-IoT's Gateway/Edge Node (GWEN) and the **updates** with respect to the inputs of the first iteration are **presented**. The final design of the Smart IoT devices, namely the localisation tag and the fall arrest device, are also depicted. First products are expected to be ready for use in the pilots from M23.

Enablers will be the main output of the tasks related to the upper planes of ASSIST-IoT (T4.2, T4.3 and T4.4). An enabler represents a collection of components, running on hardware nodes, that communicate with each other for delivering a particular functionality to the system. Enablers can only be interacted with via their exposed interfaces. A total of 19 enablers were introduced in the first iteration of the deliverable. In this iteration, apart from **design updates**, a **first functional version** of most (15) of the enablers is **delivered**, specifically:

- All the WP4 essential enablers (i.e., those that are vital to be present in any deployment for the ASSIST-IoT system to work): (i) the Smart Orchestrator enabler, (ii) the VPN enabler, (iii) the Edge Data Broker enabler, (iv) the Long-Term Storage Enabler (i.e., LTSE), (v) the Tactile Dashboard and (vi) the OpenAPI management enabler will have a first version.
- Most of the non-essential enablers: (i) the SDN Controller, (ii) the Traffic Classification enabler, (iii) the Semantic Repository enabler, (iv) the Semantic Translator enabler, (v) the Semantic Annotator enabler, the (vi) business KPI reporting enabler, the (vii) Performance and Usage Diagnosis (PUD) enabler, (viii) the Video Augmentation enabler and (ix) the MR enabler.

Apart from the software outcomes, additional information is presented (diagram of use cases, endpoints, implementation status) jointly with an update of the features and internal structure, when needed. Being the second of a series of three iterations, the software products and the information provided in this deliverable are still subject to change, due to potential addition of new (or yet not implemented) features.

Table of contents

List of figures

List of tables

List of acronyms

Acronym	Explanation
AI	Artificial Intelligence
AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
B2B	Board-to-Board
CDR	Custom Definition Resource
CLI	Command Line Interface
CNI	Container Network Interface
CNF	Cloud-native Network Function / Containerised Network function
CSV	Comma Separated Value
DB	Database
DHCP	Dynamic Host Configuration Protocol
DNN	Deep Neural Network
DNS	Domain Name System
DT	Decision Tree
EDB	Edge Data Broker
FL	Federated Learning
GWEN	(ASSIST-IoT's) Gateway/Edge Node
gRPC	gRPC Remote Procedure Calls
GUI	Graphical User Interface
HA	High Availability
HMD	Head-Mounted Device
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IdM	Identity Manager
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
IPSM	Inter Platform Semantic Mediator
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbours
KPI	Key Performance Indicator
LED	Light Emitting Diode

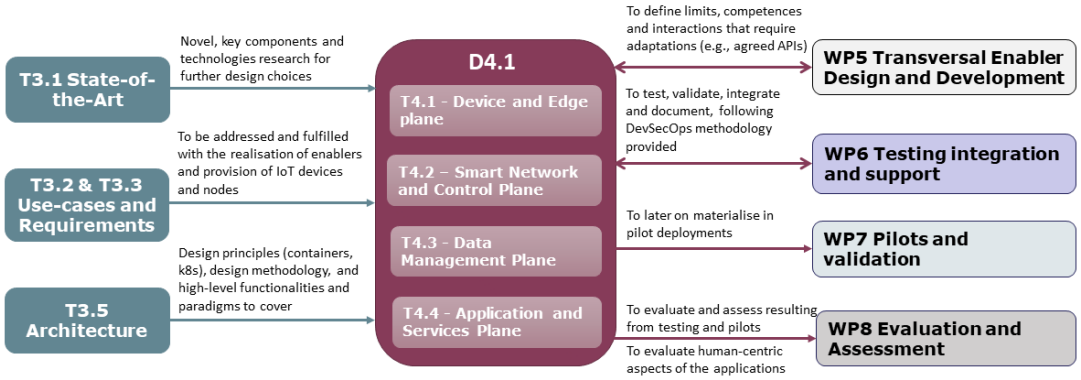
LTSE	Long-Term Storage Enabler
MANO	Management and Orchestration
ML	Machine Learning
MQTT	MQ Telemetry Transport
MR	Mixed Reality
NAT	Network Address Translation
NB	Northbound
NFV	Network Function Virtualisation
NFVO	Network Function Virtualisation Orchestrator
NoSQL	Not Only Structured Query Language
ONOS	Open Network Operating System
OSM	Open Source MANO
OVN	Open Virtual Network
OVS	Open vSwitch
PCB	Printed Circuit Board
PUD	Performance and Usage Diagnosis
PUI9	Prodevelop User Interface
PromQL	Prometheus Query Language
PV	Persistent Volume
PVC	Persistent Volume Claim
RDF	Resource Description Framework
REST	REpresentational State Transfer
RF	Random Forest
RML	RDF Mapping Language
RTT	Round-Trip Time
SB	Southbound
SDN	Software-Defined Networking
SD-WAN	Software-Defined Wide Area Network
SPA	Single Page Applications
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
VIM	Virtualised Infrastructure Manager
VPN	Virtual Private Network
WAN	Wide Area Network
XML	Extensible Markup Language

1. About this document

The objective of this deliverable is two-fold: (i) to **update** the **specifications** and **attach additional information** regarding the horizontal **enablers** designed, and (ii) to **provide a first functional version** of the enablers developed so far. These enablers are the cornerstone of the project, since they will enable the deployment of an ASSIST-IoT architecture in a particular environment, allowing its further evaluation within the pilots involved in the project. Being the only document that gathers the outcomes of WP4, D4.2 will also include the updated specifications and schematics of the ASSIST-IoT's Gateway/Edge Node (GWEN) and the Smart IoT devices that are to be developed specifically for the project, despite not being enablers per se.

It should be highlighted that this deliverable corresponds to the second document of a series of three iterations, and therefore its content will be expanded and adapted as the project evolves. The rationale behind the iterative nature of its development is based on the fact that both the requirements and the architecture, produced by the developments in WP3, are still evolving (and therefore new enablers or modifications in the current ones may be needed), and as a result the interactions between enablers from WP4 and WP5 may require adapting them (in the form of new interfaces, methods, components, etc.).

1.1. Deliverable context

Keywords	Lead Editor
Objectives	**O2:** D4.1 presents the updated specifications of the enablers of the Network's plane (some of them), as well as a first functional version of some of them. **O3:** Specifications of enablers focused on data (semantics, broker, storage) are provided, as well as a first functional version of them. **O5:** Human-centric interfaces for the use cases are presented.
Work plan	
Milestones	This deliverable does not mark any specific milestone; still, it contributes to an update of *MS3 – Enablers defined*, achieved in M12. Although far in time, it is also central part of *MS6 – Software structure finished.*
Deliverables	This deliverable receives inputs from D3.3 (requirements and use cases – second iteration) and D3.6 (architecture definition – second iteration). Outcomes will feed WP6 deliverables related to testing, integration, distribution and documentation, they will be the cornerstone of pilots' implementations of WP7, and they will be a key part in the technical evaluation to be performed under the scope of WP8.

1.2. The rationale behind the structure

This deliverable consists of four main sections, following a different approach with respect to D4.1, where most of the content was moved to annexes. It starts with an introduction (Section 2), followed by a section dedicated to the design of the GWEN and the Smart IoT devices (Section 3). Afterwards, Section 4 presents an update of the features provided (or to be provided) by the enablers of each horizontal plane, as well as new information related to endpoints, use-case diagrams and implementation status. Finally, the remaining work towards the last year of activity in the Work Package is summarised in Section 5 and will be documented in D4.3.

1.3. Outcomes of the deliverable

The deliverable consists not only of the present document, but also of the software artifacts developed and implemented so far. Apart from providing updated information with respect to the previous deliverable iteration (D4.1), this document formalises primarily the technological choices and the interaction between the internal components of the enablers.

Software artifacts are being developed at varying paces, having focused on the enablers catalogued as essential (this does not entail being more important or interesting, but rather that they need to be in place before other for facilitating their use, configuration and other common features – see deliverable D3.6 for more information about "essential" enablers). Also, functional versions of these enablers might be ready for Kubernetes or just for Docker (to be deployed via Docker Compose). Although documented more extensively in the Readthedocs documentation related to T6.4, a summary of each enabler implementation status is given.

In summary, the following enablers have a first functional version: The Smart Orchestrator enabler, the SDN Controller, the Traffic Classification enabler, the VPN enabler, the Semantic Repository enabler, the Semantic Translator enabler, the Semantic Annotator enabler the Edge Data Broker enabler, the Long-Term Storage Enabler (i.e., LTSE), the Tactile Dashboard, the business KPI reporting enabler, the Performance and Usage Diagnosis (PUD) enabler, the OpenAPI Management enabler, the Video Augmentation enabler and the MR enabler and will have a first version. However, there is pending effort to make them functional to be managed by K8s masters, with dedicated manifests, as many of them are still just valid for Docker (with Docker Compose).

During the next months, the missing features of the enablers will be implemented, and the needed adaptations to work in a Kubernetes environment and to interact among them will be assessed and executed. These outcomes will feed the Work Packages related to integration, deployment and assessment (i.e., WP6, WP7 & WP8).

1.4. Lessons learnt

During the past months, partners of the Consortium have focused their effort in developing and implementing the components that will compose the enablers of the project. In addition, the design of the hardware elements of the Edge and Device plane has been completed, being under production. The following insights have been extracted:

- Modularity in the GWEN is of great interest, as it redounds in low base price and in case extra features are required, it can be easily adapted.
- Designing enablers for IoT deployed in K8s clusters requires taking into account how K8s works. Latency requirements might be severely affected if workloads are not deployed in nodes close to actions or to the source of data.
- Although containers are portable, they are usually tight to a specific CPU architecture (e.g., ARMv7, ARM64, x86, x64, etc.). Enablers that will be deployed in edge, low-resource nodes should have components that consider ARM architectures, like that of RaspberryPis (either ready for that or attaching guidelines to build them).
- SDN strategies based on typical controllers (e.g., OpenDaylight, ONOS, etc.) cannot be easily integrated with K8s. In Kubernetes world, network-related rules are implemented through Container Network Interface (CNI) technologies and plugins that, despite following very similar principles (also SDN), do not follow the same architecture, technologies and protocols. Hence, SDN will be tested under the hood, controlling the underlying network but not integrating it with K8s, as this kind of projects have been discarded.
- The implementation of some components was relying in technologies that require too many resources, which might be problematic in constrained environments. For this reason, some technological choices have changed during the execution of the developments in M9-M18.

1.5. Deviation and corrective actions

The Consortium is exerting efforts to formalise (in D4.1) and materialise (in D4.2 and later on finished in D4.3) the envisioned enablers. However, there are some deviations or delays that have slightly altered the initial plan:

- SDN concepts applied in K8s will be realised considering Cilium as main technology. This was not the original plan, becoming a feature that will be added to the Smart Orchestrator as it is the enabler in charge of deploying resources over K8s infrastructure (in this case, CNI-related resources).
- Building the components into containers considering processor architectures, and realising the K8s manifests (and later on, the Helm charts for packaging them into enablers) is not a straightforward task. As some partners do not have the same degree of experience on these technologies, the implementation time varies, and therefore a "Kubernetes task force" has been created in the project, with internal workshops taking place to aid in the implementation and encapsulation of enablers.
- Implementing networking components in the form of containers managed by Kubernetes is far from simple. There are several alternatives to manage underlying interfaces and to force that traffic travels through specific components, and their configuration is complicated. Assessing the available options has taken more time than expected, however, the implementation choices regarding specifically SD-WAN and network function chaining are now clear. Expecting this deviation, the Consortium decided to shift the beginning of the development of the related enablers/features until the next phase (i.e., beyond M18).
- During the execution of the tasks, it became clear that some enables might not be able to follow the encapsulation principles of the ASSIST-IoT architecture. An example of this is the MR enabler, which is composed of very specific software that cannot be installed as a set of containers in a Head-Mounted Device (HMD). This entailed the introduction of "encapsulation exceptions" in the architecture.

2. Introduction

In the ASSIST-IoT architecture, the horizontal planes represent a classification of logical functions that fall under the scope of a particular domain (similar to classic layered stack classification), namely: edge/device, network, data, and applications/services. Each plane includes a set of functional blocks, providing specific functionalities in the architecture. These blocks have been identified, designed, and implemented considering different real-life use case requirements and concerns (particularly, those addressed in the action) as well as technological limitations, aiming at being agnostic-enough to be deployable in multiple business scenarios.

Functionalities are delivered by means of enablers, which is an abstraction term that represents a collection of components that work together to provide a specific functionality to the system. Some common conventions are common to all the enablers (e.g., metrics and logs gathering, common endpoints), (almost) all of them following an ***encapsulation*** **paradigm** (i.e., enablers can only communicate via exposed interfaces, denying interactions between internal components of different enablers).

In the initial iteration of the deliverable, a template was provided with information with respect to each enabler. In this second document, some particular specifications such as features, endpoints and internal structure are updated, and novel data regarding use cases and implementation status are provided. Apart from this documentation, **a first functional version of most of the enablers** (15 out of 19), and specifically all those identified as essential, are described. It is worth mentioning that an updated and extended set of specifications for the Smart IoT Devices and the GWEN are also included, as key part of the outcomes from T4.1 – Device and Edge plane. As a reminder, the artifacts that are responsibility of WP4 are depicted in Figure 1.

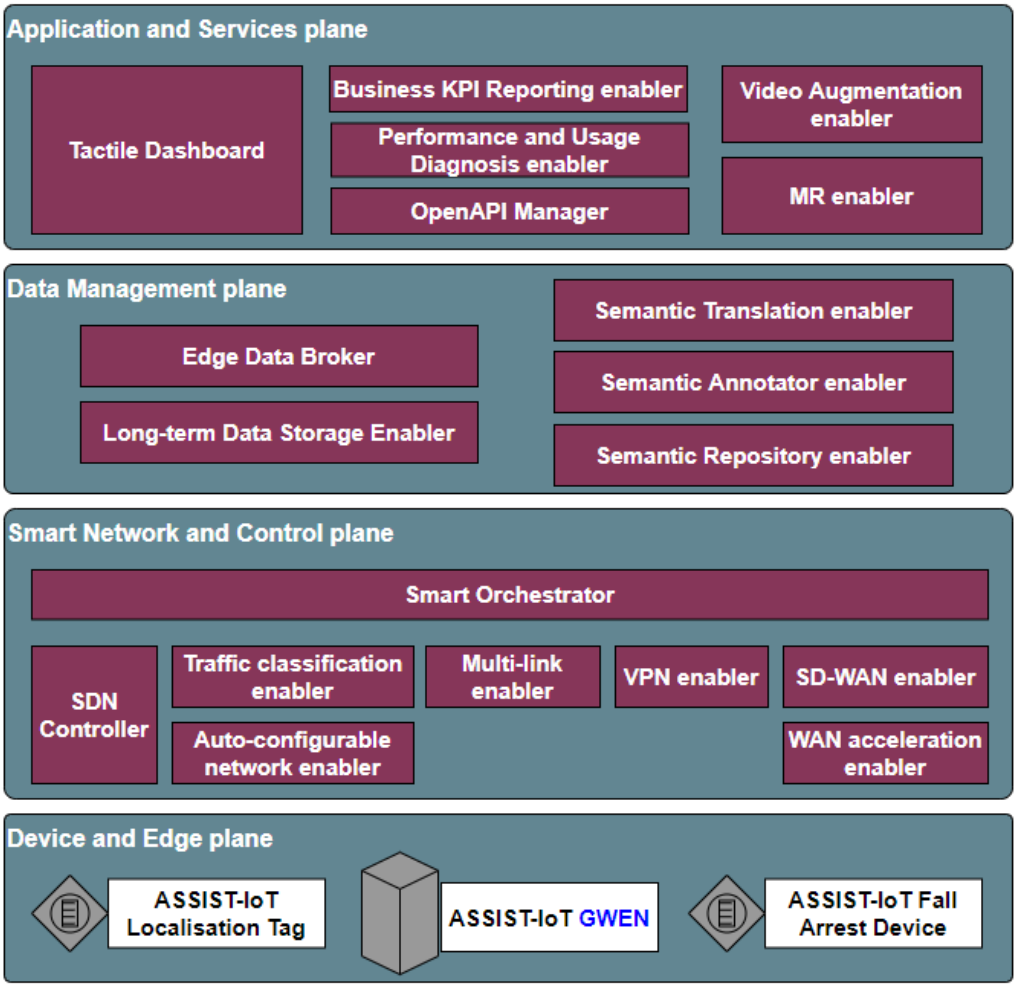


Figure 1. ASSIST-IoT enablers and hardware elements formalised

3. Devices specifications

Two Smart IoT devices are being developed under the scope of the project: the localisation tag and the fall arrest device. They have been designed primarily to solve specific requirements of the pilots of the project, hence not being intrinsic blocks of the ASSIST-IoT architecture. Evidently, they will belong to the Device and Edge plane of the architecture when they are part of a specific deployment, but in the same way that any device (intelligent or not) would be. In any case, these devices can be of utility in different verticals, and therefore they can address several use cases (alongside the associated software). The same occurs with the ASSIST-IoT Edge Node (branded as "GWEN" in the most recent architecture document – D3.6). In this case, GWENs might be part of virtually any architecture deployment, however, it cannot be considered as an inherent part of the architecture as its functionalities could be provided by another device (or group of devices) of similar nature.

This section presents an update of the status of these devices, developed under the scope of the Edge and Device task, and which specifications were depicted in D4.1. Minor features have been added to the Smart IoT devices, while the GWEN has suffered the larger changes, as a modular approach has guided its recent design and development activities.

3.1. Specifications update

3.1.1. ASSIST-IoT localisation tag

The localization tag has two additional features compared to what has been described in D4.1. These correspond to the implementation of a push button and an IMU (Inertia Measurement Unit) sensor. When a user detects a dangerous situation, they can push the button to alert the system that it is happening (this modification is motivated based on inputs from the second pilot of the project). The IMU is added for future use, as an additional feature. It can be used to detect if there is movement or not, so when there is no movement the position update rate can be decreased in the future to reduce the energy consumption and thus increase battery life time. The updated block schematic diagram of the localization tag is given in Figure 2.

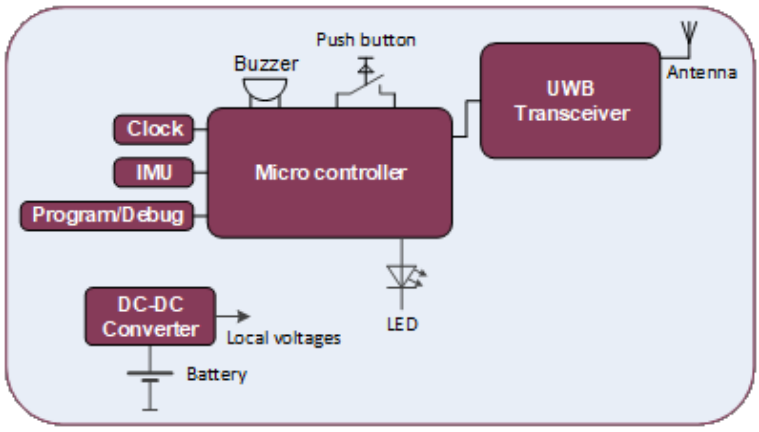


Figure 2. Block schematic diagram of the localisation tag.

3.1.2. ASSIST-IoT fall arrest device

Based on a refinement of its related requirements, the fall arrest device will be used in a different way than initially expected, so the hardware has been modified accordingly as one can see in Figure 3. Initially, a fall arrest sensor interface for a separate fall arrest detection device was planned to be used in order to detect if a person is falling. Since an IMU is implemented, a fall can be detected by it. This means that when a worker falls on the floor, this can be detected too and the fall arrest device can send an alert to the system. Although very similar in underlying hardware, major differences between both devices are found in firmware and enclosure.

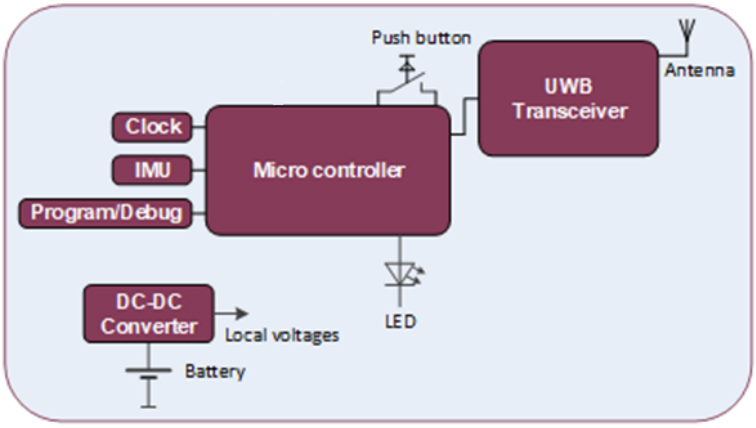


Figure 3. Block schematic diagram of the fall arrest device.

3.1.3. GWEN

The updated block schematic diagram of the GWEN is given in Figure 4. The changes that have been made correspond to the embedding of two M.2 interfaces and two SD card connectors instead of one. Having two M.2 interfaces enables the use of WiFi6 and 5G in parallel. Two SD cards are used to separate firmware from data storage. This means that the boot software and data are separated to prevent unintended boot software corruption when manipulating data, minimising risks. At the same time, additional storage for data is available.

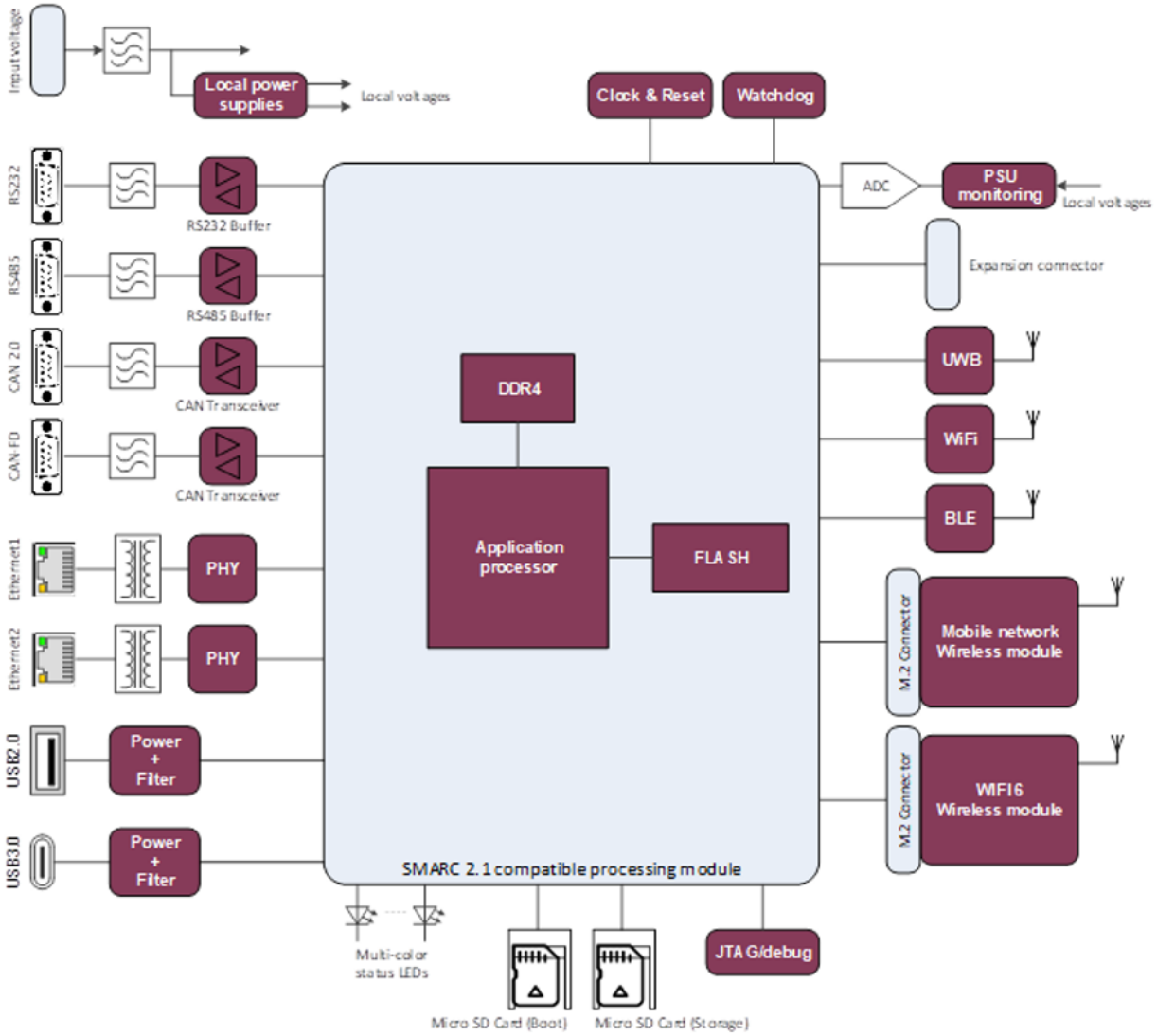


Figure 4. Gateway/Edge node block schematic diagram.

Several interfaces and parts of Figure 4 will always be present, being thus implemented at a carrier board. However, other interfaces will be implemented as expansion modules. The top view of the carrier board is given in Figure 5.

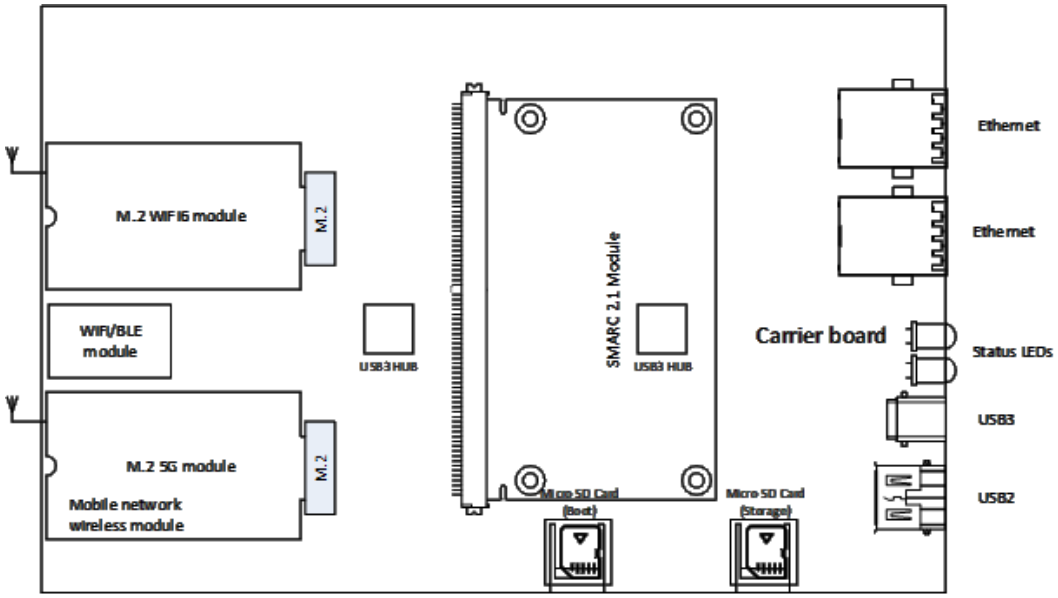


Figure 5. Top view of the carrier board.

To implement the remaining functionality needed, proprietary add-on expansion modules will be used and mounted at the bottom of the carrier board. Size, Board-to-Board (B2B) connector, connector pinning and mounting hole positions are fixed and positioned in a way that two sizes of modules can be inserted. In Figure 6 an example configuration of mounted add-on expansion modules is given.

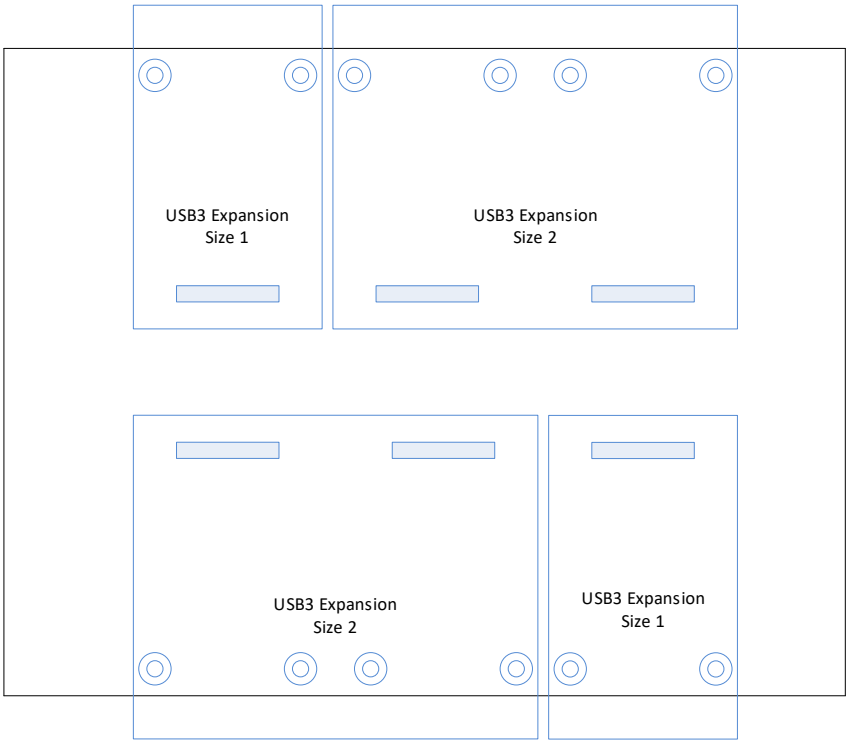


Figure 6. Bottom view of the carrier board.

The input voltage range for the GWEN was initially set to 12V ± 5%. To support the project pilot where the GWEN is used in a car and connected to its board supply, this voltage range has been changed to 12V ± 20%.

3.2. Development status

Table 1. Development status of GWEN and Smart IoT devices

Category	*Status*
Localization tag and fall arrest device hardware	The main part of the electronics is available ➔ 90% The push button, the LED and the buzzer need to be added ➔ planned, 0% An enclosure needs to be created ➔ planned, 0%
Localization tag and fall arrest device firmware	The main part of the firmware is available, this needs to be changed to meet the ASSIST-IoT functionality. Inventorying of changes ➔ 90%
GWEN hardware implementation	Electronics design ➔ finished Schematic design ➔ finished PCB layout design ➔ under development, 20% A customer of the shelf enclosure has been selected. This enclosure will be changed depending on connector placement. PCB layout design and enclosure changes depend on each other ➔ 10%
GWEN firmware implementation	Not started yet, resources have been planned
RS232/RS485 interface module	Electronics design ➔ finished Schematic design ➔ finished PCB layout design ➔ finished Manufacturing ➔ started ordering components, 5%
CAN interface module	Electronics design ➔ finished Schematic design ➔ finished PCB layout design ➔ finished Manufacturing ➔ started ordering components, 5%

4. Horizontal enablers update and implementation status

This section has two purposes. On the one hand, it aims at updating all the information related to the features, the internal structure and the endpoints of the enablers designed in the early stages of the project. Although high-level functionalities may not have changed significantly, additional or supporting features might have been added during the development process. Also, the initial design of internal components is susceptible to change during development: components can be combined for performance reasons (i.e., deploying two containers performing two different functionalities might be good for conceptual understanding/logic and further maintenance, but not as good in terms of performance), separated, and/or unexpected requirements may have required the addition of a component or even a re-design of the entire solution (which in turn, can affect the expected endpoints).

On the other hand, this section also provides new information regarding the enablers, related to use cases – in this case, understood as the interaction between the components under a specific event/call, and implementation status.

4.1. Smart Network and Control enablers

4.1.1. Smart Orchestrator

4.1.1.1. Structure and functionalities

Currently, in the edge cloud continuum arises the need to distribute the workloads between the available edge nodes or at cloud premises, which has more computing capacity. The selection of the optimal location depends on multiple factors, like latency or computation needs. The Smart Orchestrator is in charge of deploying these workloads, developed by combining three main technologies: Kubernetes. Mck8s and MANO. These technologies provide the following features:

- Selection of the cluster where the workloads will be deployed, either manually (by an administrator) or automatically by a dedicated component (scheduler).
- Instantiation of workloads arranged in a package repository and installed using Helm.
- Orchestration of computing, network, and storage infrastructure, (container-centred) of the workloads and services.
- Automatic application of network-related rules, to enable communication between enablers via exposed interfaces and to block communication between internal components of different enablers.

The updated diagram is shown in the following figure. It is composed of 4 components and a supporting database:

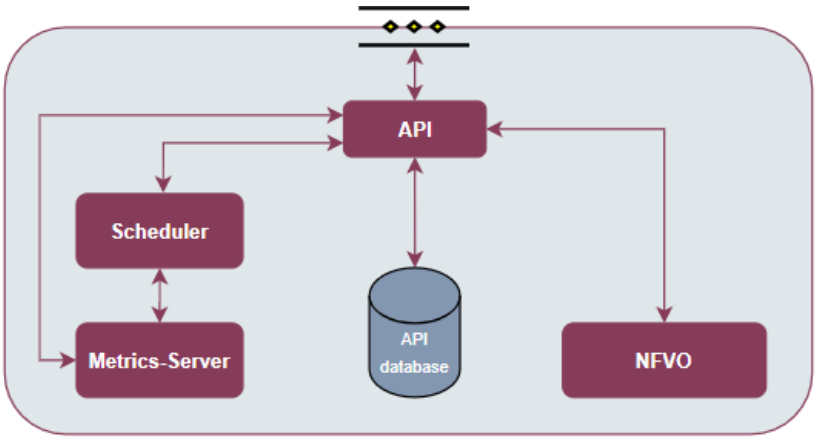


Figure 7. High-level diagram of the Smart Orchestrator enabler

Implementation technologies

Table 2. Implementation technologies for the Smart Orchestrator enabler

Technology	Justification	Component(s)
Python	Programming language selected due to its facility on creating its own functions and the excellent performance with Kubernetes API.	Scheduler
Node.js	It is a cross-platform execution environment. It is written in JavaScript and allows the realization of highly scalable web servers. The enabler API uses node.js as the leading technology and makes it easy to connect to the database.	API
Mongo DB	NoSQL database provides a quick and easy integration of data in the enabler API.	API database
OSM	It allows the instantiation of services through Helm and the orchestration of these. OSM also supports the orchestration requirements of NFV networks.	NFVO
Mck8s	Allows choosing between different policies for the automatic allocation of resources between clusters. It facilitates the implementation of new policies	Scheduler
Cilium	Kubernetes plugin to manage service mesh and network-related aspects (Layers 3, 4 and 7)	Scheduler
Prometheus	Provides network traffic information to the Scheduler when the traffic-most policy is selected.	Metrics-server

NOTE1: The instantiation of the enablers on a cluster follows a process that has not been monitored yet. In this way, if the instantiation process fails, it is not possible to control the error, so the enabler will not be eliminated automatically and at this moment; it must be removed manually. Moreover, OSM cannot delete some resources created directly by Kubernetes, such as Persistent Volumes (PVs) and Claims (PVCs), and hence the Smart Orchestrator does not have this functionality implemented yet.

NOTE2: Kubernetes and OSM are in constant development, which could lead to an incompatibility between versions or changes in their performance since the old versions are no longer maintained. Hence, it is recommendable to always deploy this enabler considering tested versions.

4.1.1.2. Communication interfaces

Table 3. Communication interfaces (API) of the Smart Orchestrator enabler

Method	Endpoint	Description
GET	**/api/k8sclusters/**	Get all the clusters added to OSM.
POST	**/api/k8sclusters/**	Post a cluster in the Smart Orchestrator enabler.
DELETE	**/api/k8sclusters/{id}**	Delete a cluster by Id.
GET	**/api/chartrepo**	Get all the Helm repositories added to OSM.
POST	**/api/chartrepo**	Post a Helm repository to OSM.
DELETE	**/api/chartrepo/{id}**	Delete a repository in OSM by id.
GET	**/api/enabler/instanced**	Get all the enablers deployed by the moment.
POST	**/api/enabler/**	Post an enabler to the Smart Orchestrator.
POST	**/api/enabler/{id}/terminate**	Terminate a deployed enabler by id.
DELETE	**/api/enabler/{id}**	Delete an enabler by id.
POST	**/api/login/tokens**	Returns an access token.

4.1.1.3. Use cases

There are 8 use cases identified applying to this enabler. The **first case** is related to an administrator who wants to **add a cluster** to the Smart Orchestrator. Its diagram for the use case and the included steps are the following:

STEPS 1-2: The user interacts through the API of the enabler indicating the configuration file of the cluster of Kubernetes that will be added. The API receives the configuration and sends the request to create a VIM in the NFVO.

STEPS 3-4: The NFVO creates the VIM for the selected cluster, and returns an answer to the API.

STEP 5: The API receives the configuration and sends the request to create a cluster in the NFVO.

STEPS 6-7: The NFVO adds the cluster, and returns an answer to the API.

STEP 8: Once the answer is received, a request is sent to the Metrics-server to add the cluster to its configuration files, allowing the Scheduler to obtain the added clusters metrics.

STEPS 9-10: The metric-server adds the configuration to its target file, and returns the answer to the API.

STEP 11: Once the answer is received, a request is sent to write the configuration of the added cluster in the database, which will be used to create the Kubernetes configuration file for the Scheduler.

STEPS 12-13: The configuration file of the added cluster is stored in the database, which returns an answer to the API.

STEP 14: Once the process is finished, the API returns a confirmation message. The whole process still has some errors to handle properly.

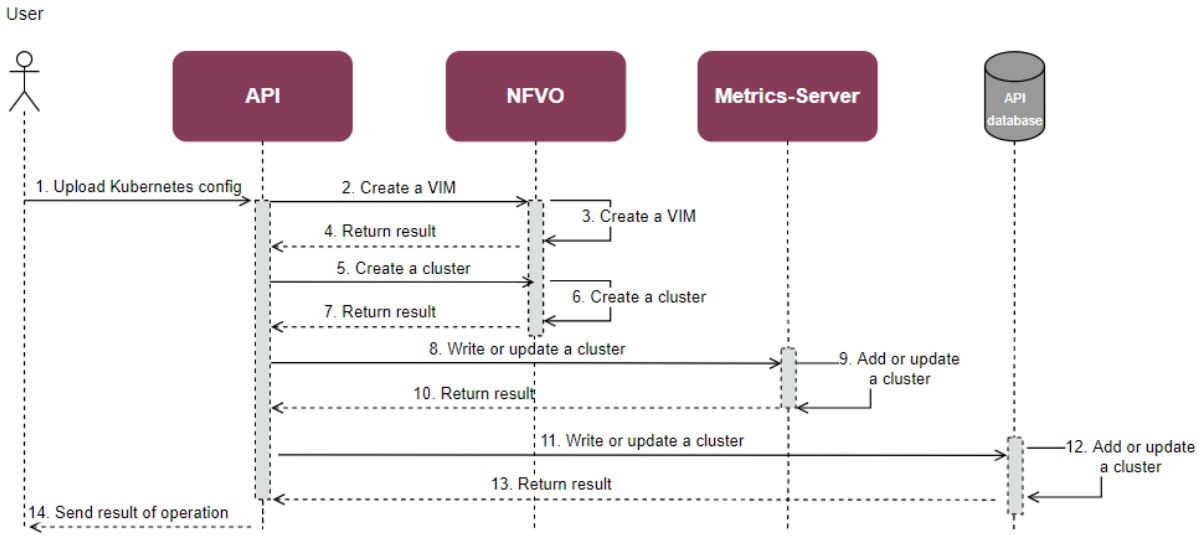


Figure 8. Smart Orchestrator enabler UC1 (add cluster)

The **second use case** is previous to the instantiation of an enabler, where the user administrator **adds a Helm repository** where all the charts are saved. Diagram and related steps are the following

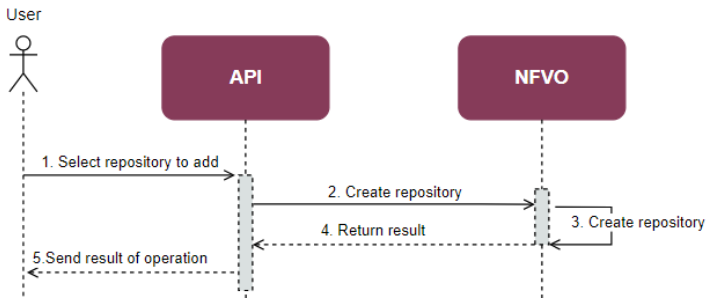


Figure 9. Smart Orchestrator enabler UC2 (add Helm repository)

STEP 1: The user interacts through the API indicating the Helm Chart repository to add.

STEPS 2-3: The API receives the information and send a request to create the repository to the NFVO.

STEP 4: The NFVO creates the repository, and confirms its creation to the API.

STEP 5: Once the process has finished, the API returns a confirmation message.

The **third use case** is related to a user who wants to **deploy an enabler** using the Smart Orchestrator. This operation can be done through the scheduler (**automatically** deploying the enabler in any of the added clusters) or manually instantiate them by choosing which is the cluster to deploy the enabler. The diagram to instantiate an enabler automatically and the related steps are the following:

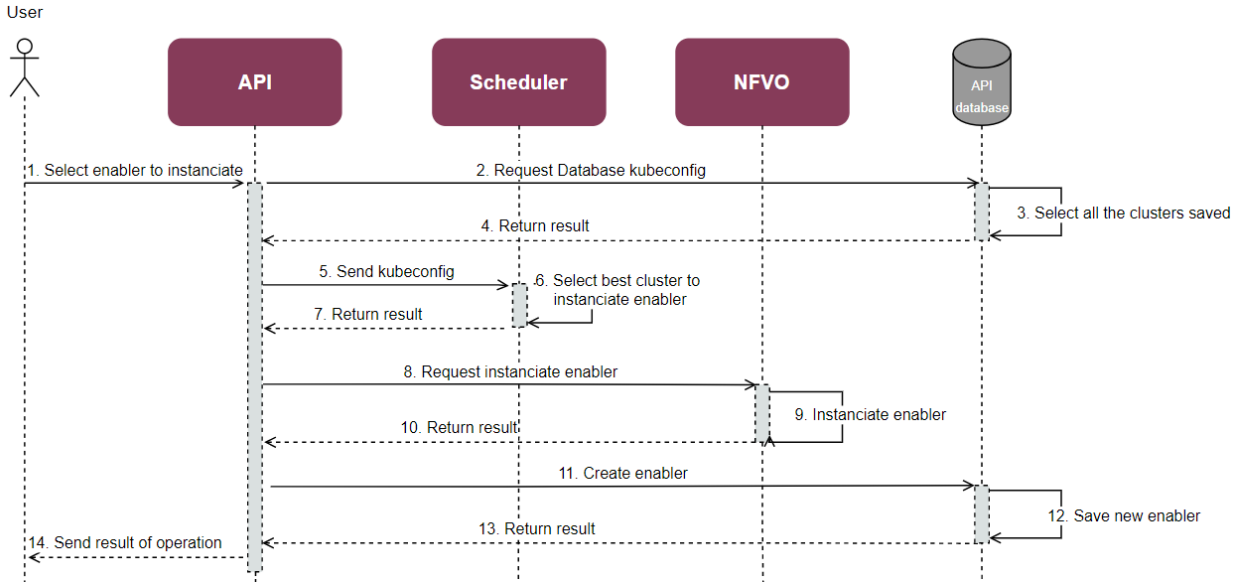


Figure 10. Smart Orchestrator enabler UC3 (deploy enabler, automatic case)

STEP 1: The user interacts through the API to instantiate an enabler with the automatic option to let the Smart Orchestrator deploy it in one of the added clusters.

STEPS 2-3: The API receives the configuration and sends the request to the database to receive the added clusters and create a common kubeconfig for the scheduler.

STEP 4: The database returns the results to the API.

STEP 5: Once the answer is received, the kubeconfig is sent to the scheduler.

STEPS 6-7: Depending on the placement policy, the Scheduler selects which is the best cluster to deploy the enabler, returning the answer.

STEP 8: With the scheduler answer, a request is created to the NFVO component to deploy the enabler in the selected cluster.

STEPS 9-10: The corresponding chart is installed in the selected cluster, and the NFVO returns the answer of the operation.

STEPS 11-13: Once the deployed enabler has been instantiated, a registry of the operation is saved in the database, which confirms the operation.

STEP 14: Once the process has finished, the API returns a confirmation message.

In case the cluster is selected **manually**, the diagram is considerably simplified (essentially, steps 2 to 7 are not needed, as the user selects the target cluster), being the following:

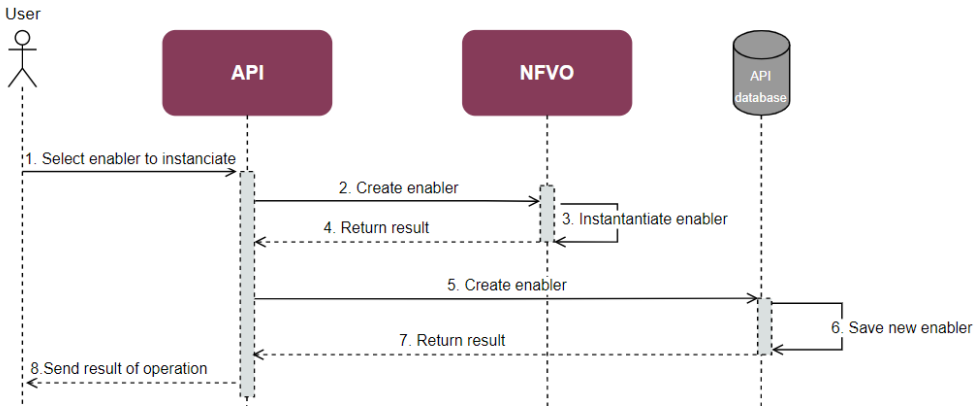


Figure 11. Smart Orchestrator enabler UC3 (deploy enabler, manual case)

The **fourth use case** is related to the user who wants to **terminate an instance of an enabler**, which is a step needed before removing it completely. The diagram is the following:

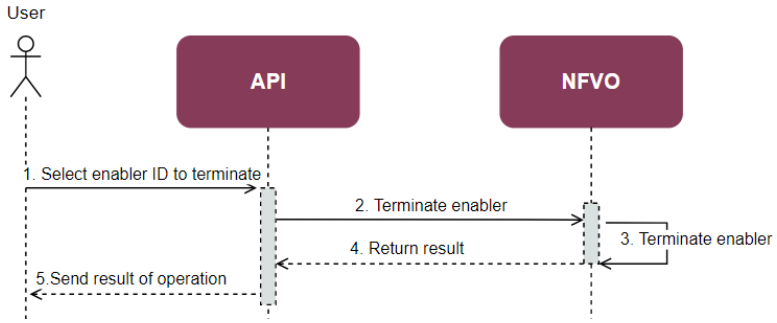


Figure 12. Smart Orchestrator enabler UC4 (terminate enabler)

STEP 1: The user interacts through the API indicating the enabler id to be terminated.

STEP 2: A request is created by the API to the NFVO to terminate the enabler.

STEPS 4-5: The enabler is terminated in the cluster where is deployed, and the confirmation from the NFVO is sent back to the API.

STEP 5: Once the process has finished, the API returns a confirmation message.

The **fifth use case**, comes after terminating an enabler, **deleting it completely** (including registries and assigned resources). Its diagram and involved steps are the following:

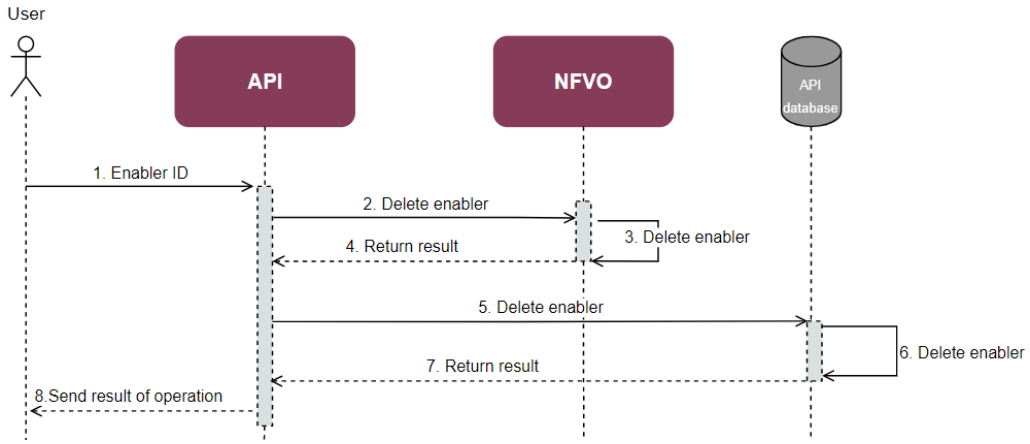


Figure 13. Smart Orchestrator enabler UC5 (delete enabler)

STEP 1: The user interacts through the API indicating the enabler id to be removed

STEP 2: The API receives the configuration and sends the request to the NFVO to delete the enabler in the corresponding cluster.

STEPS 3-4: The enabler is deleted by the NFVO component, which confirms the operation.

STEPS 5-7: The API creates a request to delete the cluster in the database, which confirms the operation.

STEP 8: Once the process has finished, the API returns a confirmation message.

For the **sixth use case**, the user wants to **get all the enablers instantiated, the added clusters or the added repositories** (the flow is identical for the three operations)**.** The diagram and involved steps are the following:

STEP 1: The user interacts through the API enabler indicating the resource to get.

STEP 2: A request is created by the API to the NFVO component to get it.

STEPS 3-4: The NFVO finds the resources and sends the answer back to the API.

STEP 5: Once the process has finished, the API returns the resources to the requester.

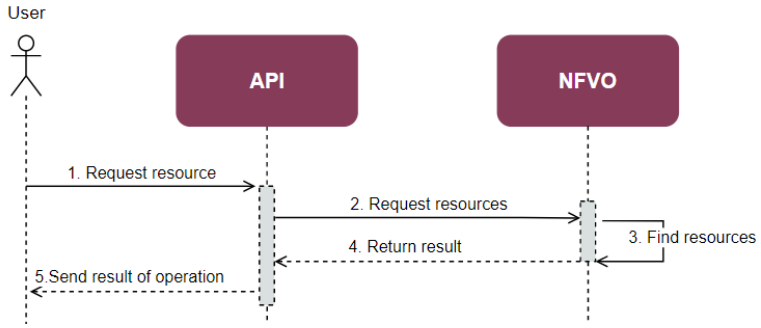


Figure 14. Smart Orchestrator enabler UC6 (get enablers)

The **next (seventh) use case** is related to the administrator user who intends to **remove a cluster**. The interaction diagram and the involved steps are the following:

STEP 1: The user interacts through the API indicating the cluster id to be removed.

STEP 2: The API receives the configuration and sends the request to delete the cluster in the NFVO.

STEPS 3-4: The NFVO deletes the cluster and confirms that the operation is successful.

STEP 5: Once the answer is received, a request is sent to the NFVO again to delete the VIM related to the deleted cluster.

STEPS 6-7: The NFVO deletes the VIM and confirms the operation.

STEP 8: Once the answer is received, a request is sent to the metrics-server to delete the configuration of the removed cluster in the targets file.

STEPS 9-10: The metrics-server deletes the cluster and confirms the operation.

STEPS 11-13: Finally, the API creates a request to delete the cluster in the database, which deletes it and confirms the operation.

STEP 14: Once the process has finished, the API returns a confirmation message.

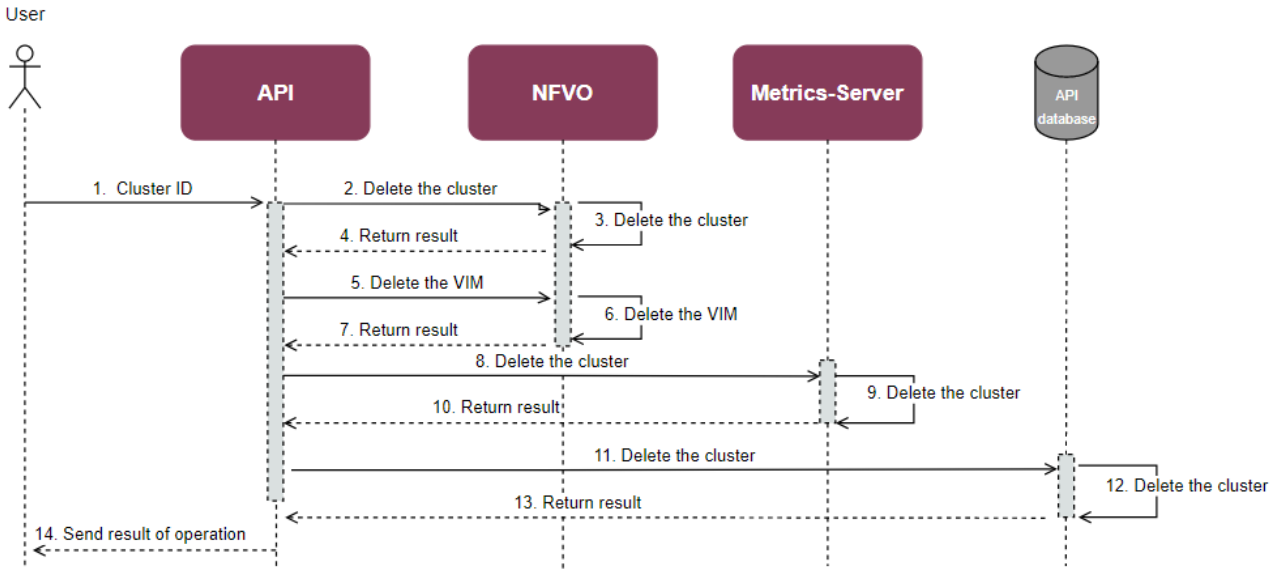


Figure 15. Smart Orchestrator enabler UC7 (delete cluster)

The **eight (and last) use case** is related to the administrator user who decides to **delete a repository**. The diagram and related steps are the following:

STEP 1: The user interacts through the API indicating the repository id to be removed.

STEP 2: The API receives the information and sends the request to the NFVO to delete the repository.

STEPS 3-4: The NFVO deletes the cluster and sends a confirmation to the API.

STEP 5: Once the process has finished, the API returns a confirmation message.

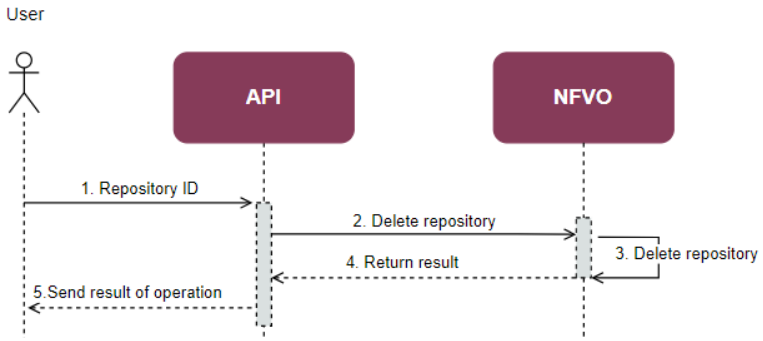


Figure 16. Smart Orchestrator enabler UC8 (remove Helm repository)

4.1.1.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/smart_orchestrator.html

Table 4. Implementation status of the Smart Orchestrator enabler

Category	Status
Components implementation	A first version of all the components is implemented, but not finalised yet.
Feature implementation status	The enabler is capable to execute its main objective, with is the deployment workloads in K8s clusters, either manually or automatically based on policies. It can also register Helm repositories and K8s clusters to be managed. Additional features are pending, such as: • Automatic generation and application of network-related and service mesh rules (via Cilium). • Additional placement features, at least one considering AI. • Error handling, including enablers and K8s resources elimination.
Encapsulation readiness	All components are encapsulated in Docker images and work in a Kubernetes cluster. K8s manifests nor Helm charts have been prepared yet.
Deployed with the Orchestrator in a laboratory environment	N/A (as it is the orchestrator itself)

4.1.2. SDN Controller

4.1.2.1. Structure and functionalities

The SDN Controller is the key element of an SDN network, implementing control plane functionalities related to network management, traffic management and monitoring. In a typical controller architecture (see high-level architecture in Figure 17), one can distinguish core functional modules like Configuration, Control, Topology, and Northbound (NB) and Southbound (SB) APIs. Core subsystems are related to device, link, host, topology, etc. On the one hand, the usage of the SB API on the network level facilitates the integration of different vendors' devices. On the other hand, the NB API is available for application developers. The main functions envisioned in the project to be useful are the following: Device, Link, Host, Topology, Path, Flow, Flow Objectives, Group, Meter, Intent, Application, Component Configuration.

Two open source controllers have been selected as suitable solution for the ASSIST-IoT architecture: **Tungsten** (v.5.1) and **ONOS** (v2.7.0, µONOS version with Kubernetes containerization). Both of them, composed by a set of components, provide the same capabilities in control plane, exposing APIs (NB and SB) to use them. Once tested, their performance and usability and will be evaluated to providing recommendation for best practices, aiming at incorporating them in the network fabric below the K8s clusters.

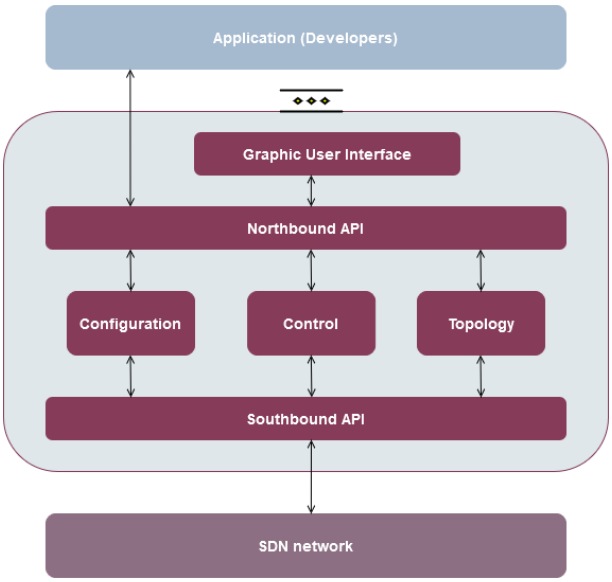


Figure 17. High-level diagram of the SDN Controller.

4.1.2.2. Communication interfaces

For developers, the NB interface of SDN controller (API) is required to communicate with the controller. The list of the basic interactions with SDN controller required to manage the network configuration and flows is depicted below.

Table 5. Communication interfaces (API) of the SDN Controller enabler

Method	Endpoint	Description
GET/POST/PUT/DELETE	**/link/?{device=deviceId} {port=portNumber} {direction= [ALL,INGRESS,EGRESS]}**	Lists, creates, updates and deletes infrastructure links (e.g., connection between switches and/or between them and hosts).
GET/POST/PUT/DELETE	**/devices/{deviceid}/ports**	Lists, creates, updates and deletes infrastructure devices (e.g., SDN switches).
GET/POST/PUT/DELETE	**/hosts/{hostId}**	Lists all end-stations hosts.
GET	**/topology/clusters/{clusterId}**	Gets list of topology cluster overviews.
GET/POST/DELETE	**/paths/{elementId}/{elementId}**	Gets set of pre-computed shortest paths between the specified source and destination network elements.
GET/POST/DELETE	**/flows/{deviceId}/{flowId}**	Creates, lists, deletes a single flow rule applied to the specified infrastructure device.
GET/POST/DELETE	**/meters/{deviceId}**	Creates, lists, deletes a single meter entry applied to the specified infrastructure device.
GET/POST/DELETE	**/intents/{app-id}/{intentId}**	Gets the details for the given Intent object. Creates and deletes a new intent object.
GET/POST/PUT/DELETE	**/applications/{app-name}**	Gets a list of all installed applications. Activates and deactivates the named application.
GET/POST/DELETE	**/configuration/{component}**	Gets the configuration values for a single component. Adds and removes a set of configuration values to a component.

4.1.2.3. Use cases

There are many use cases in which an SDN controller could be of use, as many as applications might leverage its exposed features. Three exemplary diagrams are presented below, being the flow almost identical (the major change is on the SDN controller internal function consumed).

The **first use case** shown is related to the **configuration of a device (SDN switch)**, following the next sequence and related steps:

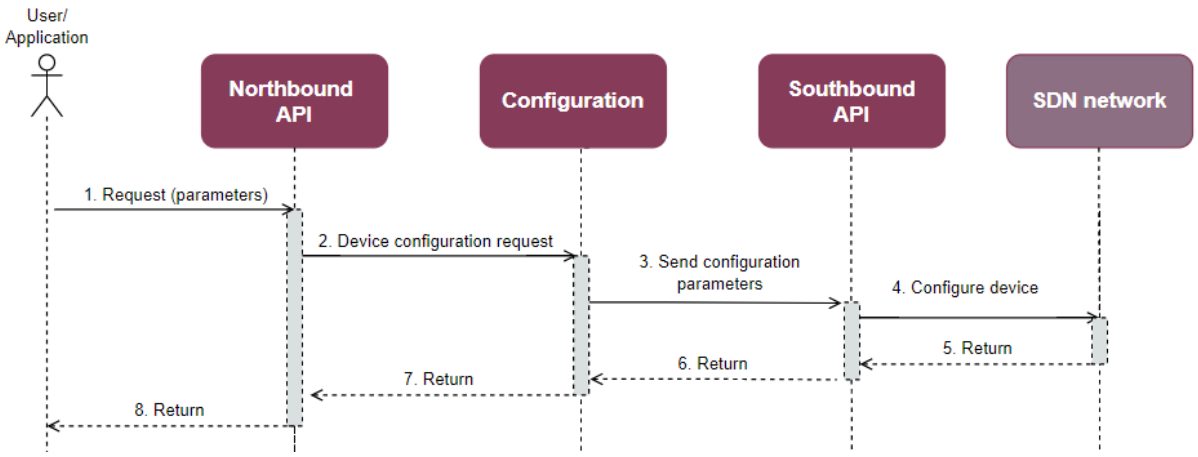


Figure 18. SDN Controller UC1 (device configuration)

STEP 1: The user/application/enabler interacts through the NB API of SDN controller enabler requesting the configuration of the given device with specified parameters.

STEP 2: The NB API receives the configuration and sends the request to the configuration module for processing and formatting.

STEP 3: Configuration module sends the configuration request to SB API in the required format.

STEP 4: SB API sends in a given format the configuration request to the selected device.

STEPS 5-7: A message of the result of the operation is returned to back to the NB API.

STEP 8: Once the process has finished, the API returns a confirmation message.

The **second use case** shown is related to the **deployment of an intent**, which essentially specifies how the network should behave in terms of policies or directives rather than specific actions. The flow and steps are the following:

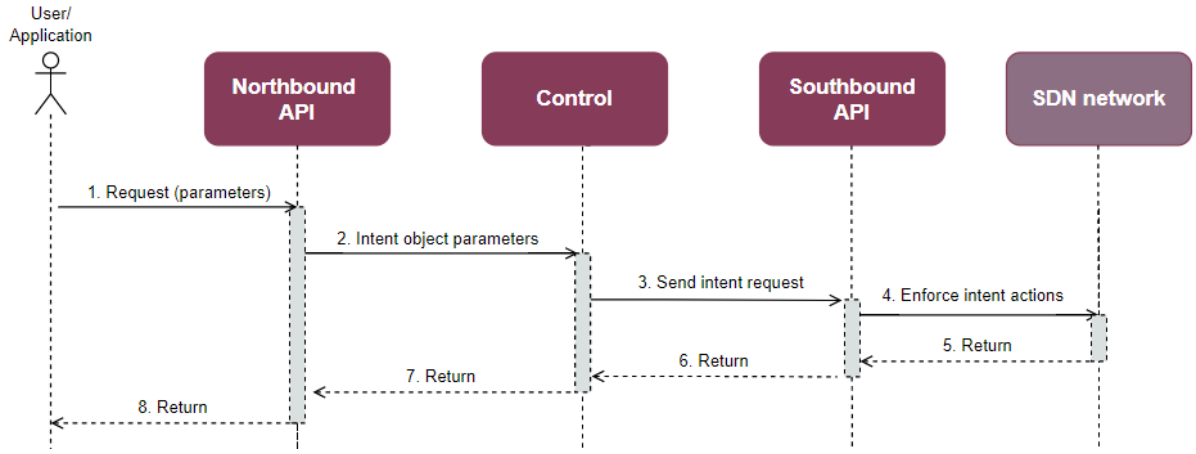


Figure 19. SDN Controller UC2 (intent deployment)

STEP 1: The user/application/enabler interacts through the NB API of SDN controller enabler requesting the intent object action with specified parameters.

STEP 2: The NB API receives the request and sends it to control module for processing.

STEP 3: Control module sends request to deploy intent in the network using SB API.

STEP 4: SB API enforce the intent action in the SDN network.

STEPS 5-7: A message of the result of the operation is returned to back to the NB API.

STEP 8: Once the process has finished, the API returns a confirmation message.

The **last use case** depicted is related to **topology discovery**. In this case, the diagram and steps are the ones described below:

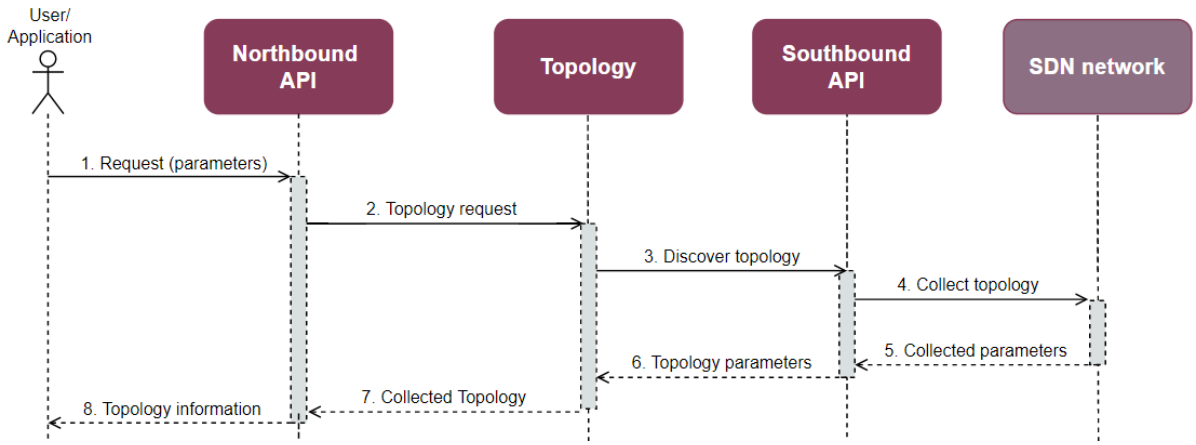


Figure 20. SDN Controller UC3 (topology discovery)

STEP 1: The user/application/enabler interacts through the NB API of SDN controller enabler requesting the topology discovery.

STEPS 2-3: The NB API receives the request and forwards it to the topology module for processing, which then sends a request to deploy a specific action in the network using SB API.

STEP 4: The SB API asks for the needed information in the SDN network.

STEPS 5-6: Information about topology is collected by the SB API module, which sends the collected information to the topology module.

STEPS 7-8: Once processed, the topology module sends the answer with the information to NB API module, which returns it to the user/application/enabler.

4.1.2.4. Implementation status

Link to Readthedocs:
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/sdn_controller.html

Table 6. Implementation status of the SDN Controller enabler

Category	Status
Components implementation	The components are in place, although validation is needed.
Feature implementation status	It is still pending to integrate with other enablers, especially with traffic classification and auto-configurable enablers.
Encapsulation readiness	Controller is encapsulated in Docker images and work in a Kubernetes cluster. Any Helm chart has been prepared yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.3. Auto-configurable Network enabler

4.1.3.1. Structure and functionalities

This enabler provides functionalities for optimising network configuration leveraging the SDN Controller of an ASSIST-IoT ecosystem. It assumes generation of the policies and enforces them using the northbound APIs of the SDN Controllers. Polices can be set manually or automatically (using different algorithms like AI solutions) to improve the performance and quality of selected KPIs of the network (e.g., network load distribution, data transfer losses and latency). The selection of the most suitable model depends on the use cases, and the objective of this enabler is to provide three different strategies to be compatible with the selected Controller.

This enabler considers two components: (i) a **policy engine**, in charge of the creation of polices and their execution in the SDN network for optimising the network traffic and the creation of routing paths. It obtains network information through the SDN controller, and data traffic via (ii) a **monitoring module**, responsible for collecting network traffic statistics. The internal structure is presented in Figure 21:

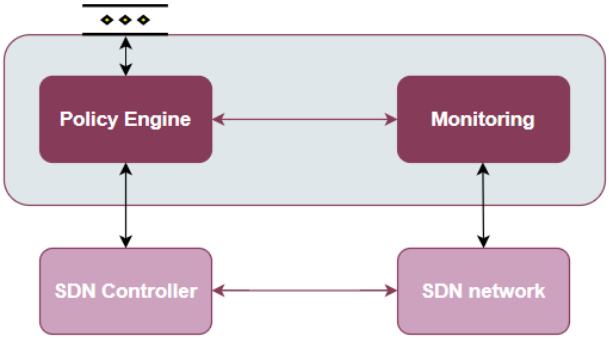


Figure 21. High-level diagram of the Auto-configurable Network enabler.

Implementation technologies

Table 7. Implementation technologies for the Auto-configurable Network enabler

Technology	Justification	Component(s)
sFlow	Agent and collector technology for gathering data from OVS switches.	Monitoring module
REST API	Users and other enablers will interact with it following REST API specifications.	Policy engine

4.1.3.2. Communication interfaces

Currently, this enabler is envisioned to work automatically, without interacting with users. Any configuration parameter needed (e.g., SDN Controller address) will be passed to the enabler at instantiation time. In further releases, a feature related to enable manual activation/deactivation of the policies will be assessed and, if needed, implemented following the endpoint indicated below.

Table 8. Communication interfaces (API) of the Auto-configurable Network enabler

Method	Endpoint	Description
POST	**/enabled/{true/false}**	Enables/Disables the enabler

4.1.3.3. Use cases

The usage of the enabler is related to the strategy of the performance/quality parameters goal optimisation. Three strategies (currently under development) are intended to be implemented, aiming at optimising traffic load optimisation, data transfer losses and latency in the network (RTT).

A flow diagram and related steps of the **main use case** is presented below, consisting in **the policy-based adaptation of the network**, also considering the gathering of needed information:

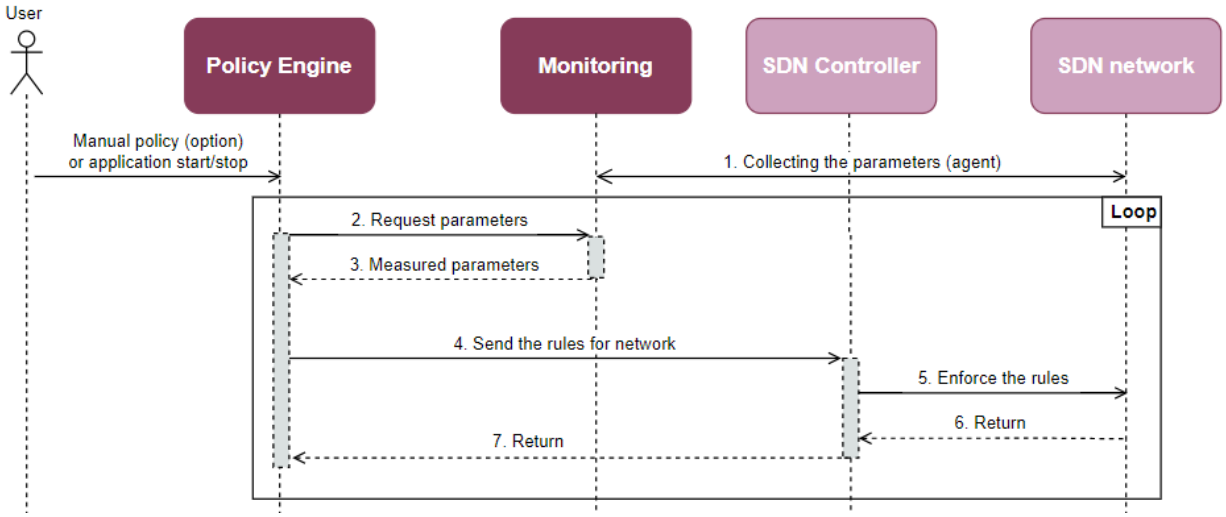


Figure 22. Auto-configurable Network enabler UC (policy-based network adaptation)

STEP 1: The policy engine requires data from the network. The monitoring module has to collect them previously, communicating with agents present in network nodes. This will be a continuous operation once the enabler is on.

STEP 2: The policy engine requests the selected parameters for a given purpose (optimise the load traffic, data losses or latency) from the monitoring module.

STEPS 3-4: After data reception, the policy module generates the rules and sends them to the SDN controller.

STEP 5: SDN controller deploys the rules in the SDN network.

STEPS 6-7: Confirmation messages are sent back to the policy engine.

The policy engine will work in a standalone fashion, triggering itself regularly or based on the threshold over defined KPIs. In the future, the addition of an endpoint to manually enabling and disabling it will be evaluated.

4.1.3.4. Implementation status

Link to Readthedocs:
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/auto_configurable_network_enabler.html

Table 9. Implementation status of the Auto-configurable Network enabler

Category	Status
Components implementation	The policy engine is under development (50%), and the monitoring module technology is under testing.
Feature implementation status	This enabler does not offer the expected features yet. The monitoring module still needs to be adapted to the needs of the project and the strategies.
Encapsulation readiness	This enabler has not been encapsulated yet(?) (or will be an exception?).
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.4. Traffic Classification enabler

4.1.4.1. Structure and functionalities

In SDN-enabled networks, a controller is responsible for controlling the underlying switches that distribute traffic according to different rules, including sources/sinks, ports and type of traffic. Regarding the latter, it is possible that the controller is not able to acknowledge the type of traffic of a specific packet, needing a specific SDN application to identify it on its behalf. This enabler will be in charge of this functionality, allowing:

- Training a machine learning model to classify traffic packets, based on the combination of different algorithms.
- To infer the type of traffic of a specific packet based on different packet parameters.

As one can see in the figure below, the diagram has changed a little from the initial design. It is composed of an API, the classifier itself, and the training module. The knowledge database has been removed since the internal components will work with K8s volumes in order to store/retrieve the trained model and to get the data required for training.

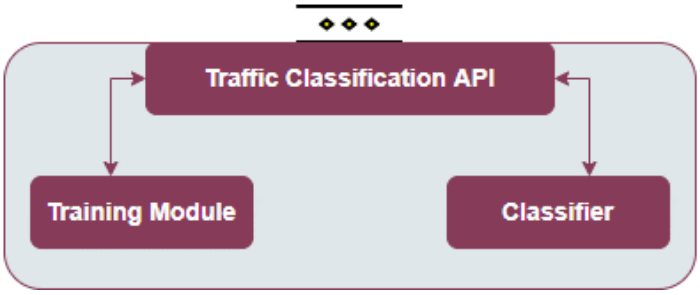


Figure 23. High-level diagram of the Traffic Classification enabler

Implementation technologies

Table 10. Implementation technologies for the Traffic Classification enabler

Technology	Justification	Component(s)
Flask	Widely used technology to developed REST APIs in an easy way	API
scikit-learn, keras and tensorflow	Scikit-learn will use for obtaining predictive models (training with K-Nearest Neighbours - KNN, Random Forest – RF, and Decision Tree - DT algorithms). Tensorflow and keras will be used for combining their results using a Deep Neural Network (DNN), getting a meta-classifier	Training module
keras	It allows getting an inference from the trained DNN	Classifier

4.1.4.2. Communication interfaces

Table 11. Communication interfaces (API) of the Traffic Classification enabler

Method	Endpoint	Description
POST	**/training**	Starts a training session, with the model and data (currently) stored in host's folders, passed via volumes.
POST	**/inference**	Returns the class of a specific packet, based on the inputs received and the application of the DNN model.

4.1.4.3. Use cases

The two main use cases of this enabler are related to the **training of the DNN model** and the inference process.

The **first use case** will be instantiated by a user, once the repository with labelled data is updated with new samples or substituted. The repository, which will be a .csv file with a row per labelled packet, each of them with seven different features of a .pcap file), will be placed in a dedicated volume accessible by the two internal components. The steps related to the first use case are:

STEP 1: The user starts a new training process via API command, once a new database is present in a folder accessible by the training module (a method might be added in the future to allow upload a new database from the API remotely).

STEP 2: The API communicates with the training module to start a new process, which will end up with the substitution of the previous model.

STEP 3: When the training process is finished, a message is sent back to the API.

STEP 4: Once the process has finished, the API returns a confirmation message.

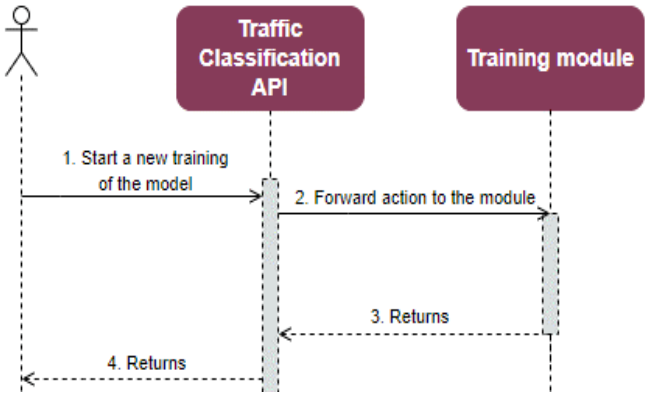


Figure 24. Traffic Classification enabler UC1 (model training)

The **second use** case is related to the **classification of a packet**. This action will be started (primarily) by the SDN Controller, when it is not able to acknowledge the type of a packet. In this case, the next steps are followed:

STEP 1: A external entity (SDN Controller) starts an inference process via API command, making use of model trained previously by the dedicated module. A set of packet features are attached in the command body.

STEP 2: The API communicates with the classifier to start a new process, forwarding the data received.

STEP 3: When the process is finished, a message with the inferred class is sent back to the API (much faster than the training time, sub-second).

STEP 4: Once the process has finished, the API returns the class of the packet to the requester.

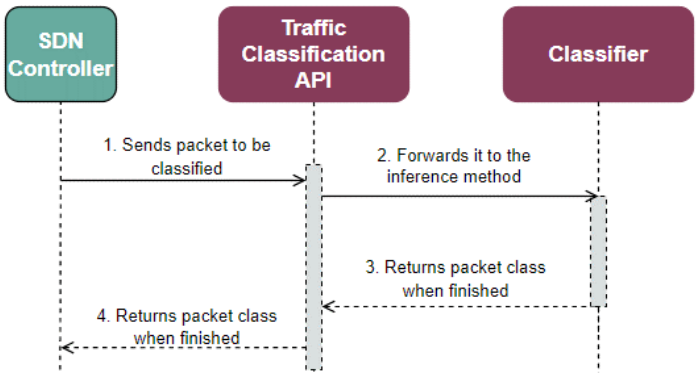


Figure 25. Traffic Classification enabler UC2 (packet classification)

4.1.4.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/traffic_classification_enabler.html

Table 12. Implementation status of the Traffic Classification enabler

Category	Status
Components implementation	In this moment, the classifier is the only component developed. The implementation of the API and the training module has started.
Feature implementation status	Currently, this enabler can classify a specific packet if the inputs are provided. However, there are some missing features: • Training/updating the current model based on a tailored, labelled data-traffic collection. • API not ready for interaction with the SDN controller.
Encapsulation readiness	The developed component has been encapsulated in a Docker image. Kubernetes manifest and Helm chart have not been prepared yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.5. Multi-link enabler

4.1.5.1. Structure and functionalities

The requirements of the multi-link enabler have experienced several changes with respect to the original ones, and as a result the enabler needed a re-design of its structure. The main goal of this enabler is to manage different wireless access networks, so in case the primary link is down, a second communication link goes up without noticing (at least, not by the user) any kind of service disruption (i.e., seamlessly). Originally, this enabler was targeting video traffic (layer 7), but now it will target any traffic (layer 2-3); also, it intended to implement redundancy, but as transmitting over many radio access technologies increases business costs, it will be finally implemented as a reliability mechanism (not simultaneous transmissions but rather a backup mechanism – see recommendation from AB member Prof. Joydeep Mitra in D2.9). Other features that need to be in place are:

- It should work at least for WiFi, fluidmesh and 5G networks, allowing establishing prioritisation of channels.
- It must set up automatically the necessary tunnels.
- In case the primary link is restored, it should go back to the initial wireless technology.

The enabler has been completely re-designed without reducing its scope or diminishing the ambition of the task (rather on the contrary), as when redundancy mechanisms were considered there was no need to set up the wireless channels, just to check the one with better performance among the available ones (in terms of latency and bandwidth). The current design is the one shown in Figure 26. It is an experimental approach, yet to be implemented and therefore, it is still subject to modifications.

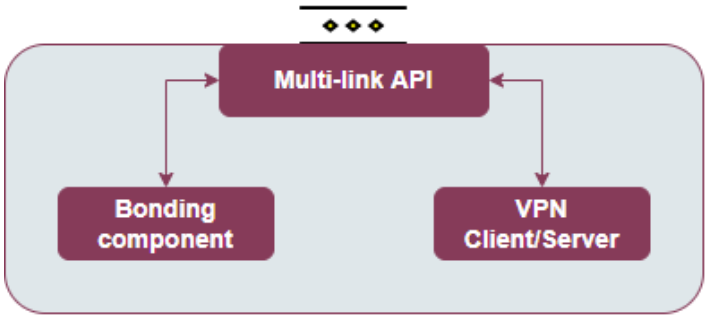


Figure 26. High-level diagram of the Multi-link enabler

NOTE1: This enabler will have two instances, one with the VPN client active and the other with the associated VPN server running, as these tunnels require implementing a server-client mode (hence, connected). For easing its management, they will be configured via a unified user interface.

NOTE2: The "internal" VPN will have a different implementation than the VPN enabler. The internal VPN will require a "TAP" implementation, being the homonymous enabler implemented in "TUN" (TAP and TUN are VPN modes, that work at layers 2 and 3, respectively). The dedicated enabler supports many more connections without reducing performance, whereas in this case this is not as important as large number of backup wireless technologies are not expected (tests will be done with three).

Implementation technologies

Table 13. Implementation technologies for the Multi-link enabler

Technology	Justification	Component(s)
Flask	Widely used technology to easily develop REST APIs	API
OpenVPN	Main technology considered for provisioning TAP tunnels between the different access points of the network	VPN (Client and Server)
Python	Custom component that will be used for establishing the necessary bridges, bonding and prioritisation rules in a GWEN (or similar device). It will bond the tunnels that will travel over different radio technologies, establishing a primary one and backups.	Bonding component

4.1.5.2. Communication interfaces

Table 14. Communication interfaces (API) of the Multi-link enabler

Method	Endpoint	Description
POST	**/interfaces/add**	Adds an interface to be bonded
GET	**/interfaces**	Gets a list of managed interfaces
PUT	**/interfaces**	Modifies the order of priority among the managed interfaces
POST	**/tunnel/client/add**	Provisions a new client in the server, generating a set of keys (returned)
GET	**/tunnel/client/**	Returns the list of clients registered in the server
PUT	**/tunnel/client/enable**	Enables a client with the specified the public key
PUT	**/tunnel/client/disable**	Disables a client with the specified the public key
DELETE	**/tunnel/client/delete**	Deletes a client with the specified the public key

Method	Endpoint	Description
POST	**/interfaces/add**	Adds an interface to be bonded
GET	**/interfaces**	Gets a list of managed interfaces
PUT	**/interfaces**	Modifies the order of priority among the managed interfaces
POST	**/tunnel/attach**	(In the client side) Connects to a VPN server making use of the keys generated by the server when provisioned.

4.1.5.3. Use cases

The **main use case** of the enabler is related to the **addition/elimination of tunnel interfaces** to be managed (or not) by this enabler. Hence, tunnels must have been created previously, making use of the dedicated endpoints (the VPN-related use cases are very similar to those presented for the VPN enabler – Section 0, and for the sake of avoiding presenting similar content, they are not indicated here). This includes bonding the selected tunnels together, as well as bridging all of them.

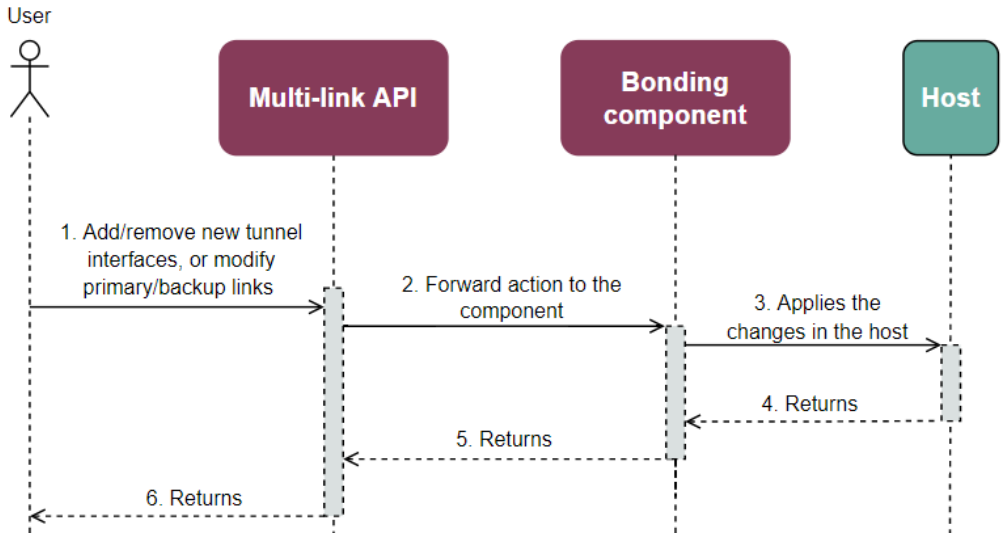


Figure 27. Multi-link enabler UC (addition/elimination of interfaces)

STEP 1: The user adds a tunnel interface (previously provisioned) to be managed by the multi-link enabler.

STEP 2: The bonding component receives information related to this tunnel, and implements a manifest (or a set of them) to be applied at host level.

STEP 3: The host applies the new network-related manifest to modify the involved rules. This implies a restart of the service.

STEPS 4-6: After receiving the corresponding return messages, the API sends a message confirming (or rejecting) the addition of the interface to the pool of those managed by the enabler.

NOTE: This flow is identical in the case of removal of interfaces and the arrangement of links into primary and backup ones.

4.1.5.4. Implementation status

Due to the huge changes that this enabler has suffered, its implementation has been re-started, hence resulting in not having its components ready by M18 (and hence, being features not available yet). The Readthedocs is still in a very immature status, which is expected to be improved during the next period. Link to Readthedocs (structure defined in WP6 documentation task):

https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/multi_link_enabler.html

Table 15. Implementation status of the Multi-link enabler

Category	Status
Components implementation	Due to the necessity of re-designing the enabler, the effort has been focused on assessing the overall solution with dedicated hardware equipment, to then move to a (more) virtualised approach and developing the necessary components.
Feature implementation status	Since the components have not been implemented, any feature is available at this moment.
Encapsulation readiness	Any container has been developed yet (hence, K8s manifests and Helm charts are not ready)
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.6. SD-WAN enabler

4.1.6.1. Structure and functionalities

The objective of this enabler is to provide access between nodes from different sites based on SD-WAN technology. In particular, this enabler will implement mechanisms to connect K8s clusters via private tunnels, facilitating (i) the deployment and chaining of virtual functions to secure connections between them and/or towards the Internet and (ii) the implementation of functions to optimise WAN traffic (via WAN Acceleration enabler – Section 4.1.7).

The SD-WAN enabler was initially designed with a central and (some) edge components, however, they will be finally realised as independent enablers. This change is motivated mostly for deployment reasons, as an SD-WAN edge has to be deployed independently on each cluster that will be included within the SD-WAN-managed architecture. The functionalities of the WAN optimisation enabler will be combined with the original SD-WAN edge component. Hence, the present enabler will comprise the central elements, which will be in charge of controlling automatically the SD-WAN edges and hubs, enabling and securing the connections. Its structure is presented Figure 28, consisting of the following elements:

- SD-WAN controller. Component in charge of managing the aspects related to SD-WAN communication, including overlays, IP provisioning, tunnels, hub registration, connection and observation, and cluster addition to be managed by it. Provides a REST API to interact with it.
- Rsync: Service that receives requests from the controller and dispatch K8s resources to the WAN acceleration enablers and K8s resources of the involved clusters to setup the dedicated tunnels.
- Database: Stores key information regarding managed clusters, hubs, overlays, IP ranges, etc.
- Etcd: Internal metadata database used to exchange configuration between the controller and rsync.

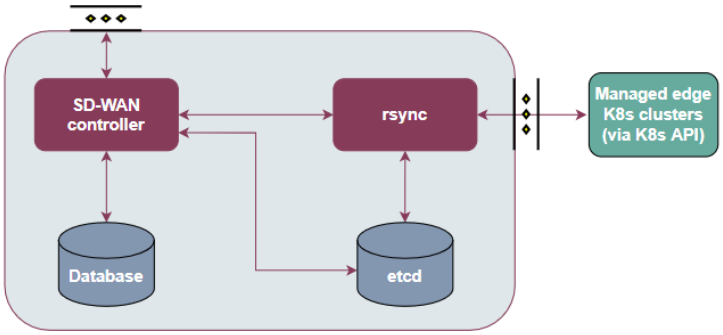


Figure 28. High-level diagram of the SD-WAN enabler

An instance of the WAN acceleration enabler will be deployed in every cluster belonging to this architecture, managed by the SD-WAN enabler. Any traffic that is sent to other clusters will go through it, over tunnels provisioned by the SD-WAN enabler. It should be highlighted that, because of requiring configuring multiple aspects related to K8s to be functional, the SD-WAN controller acts over K8s resources specifically designed for the solution, therefore it interacts with the K8s API of the target cluster instead of the API of the WAN acceleration enabler. In the edge part, its K8s API interacts with the custom K8s controller of the WAN acceleration enabler, which configures its CNF (see WAN acceleration, Section 4.1.7).

Implementation technologies

Table 16. Implementation technologies for the SD-WAN enabler

Technology	Justification	Component(s)
MongoDB	Technology used to store information related to the objects managed by the controller.	Database
gRPC	Internal communication interface, having greater performance than API REST technologies.	rsync
Go	Main language of the components.	All
K8s custom resources	A set of K8s manifests must be send to the involved clusters to be able to provision tunnels and virtualised service chaining functionalities.	rsync

4.1.6.2. Communication interfaces

Table 17. Communication interfaces (API) of the SD-WAN enabler

Method	Endpoint	Description
GET/POST/PUT/DELETE	**/overlays**	Endpoint in charge of creating, modifying, deleting and getting information regarding a set of edge clusters (and hubs) managed by the enabler.
GET/POST/PUT/DELETE	**/overlays/{id}/proposal**	Endpoint in charge of defining IPSec proposals that can be used for tunnels in an overlay.
GET/POST/PUT/DELETE	**/overlays/{id}/hubs**	Defines a traffic hub in an overlay. Requires certificate and kubeconfig file to be able to manage it. See Section 4.1.7 to get information about the role of hubs in the architecture.
GET/POST/PUT/DELETE	**/overlays/{id}/ipranges**	Defines the overlay IP range used for the edge clusters
GET/POST/PUT/DELETE	**/overlays/{id}/devices**	Defines an edge cluster location (with SD-WAN acceleration enabler). Among other input, it required kubeconfig file and certificate information.
GET/POST/PUT/DELETE	**/overlays/{id}/hubs/{id}/devices/{id}**	Defines a connection between a hub and an edge cluster.

4.1.6.3. Use cases

Although there are many operations, some of them follow the same communication schema, so they will be grouped.

The **first use case** is related to the **management of an overlay**, which defines the clusters managed by the enabler. The diagram and related steps are the following:

STEP 1: The user consumes the API of the SD-WAN controller to create, modify or delete an edge cluster part of an overlay.

STEP 2: The information is stored or updated in the database.

STEP 3: The database confirms that the operation has been completed successfully.

STEP 4: Once the process has finished, the API returns a confirmation message.

NOTE: The flow is identical for the **use cases** related **to definition of IP ranges** to be used for the connections, and **IPSec configuration proposals** for an overlay.

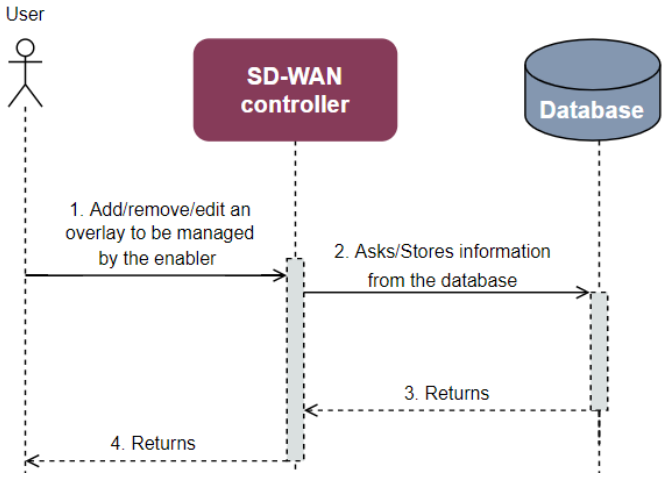


Figure 29. SD-WAN enabler UC1 (overlay management)

The **next use case** is related to the provisioning and establishment of **SD-WAN tunnels for edge nodes** (and hubs) belonging to an overlay. The diagram and involved steps are the following:

STEP 1: The user consumes the API of the SD-WAN controller to create, modify or delete a SD-WAN connection (establish a tunnel).

STEPS 2-3 The SD-WAN Controller gathers the needed information about the overlay, the IP addresses and IPSec proposals available from the database,

STEP 4: The Controller sends the required data to the rsync component.

STEP 5: The rsync component provisions the needed manifests and interacts with the API of the target cluster.

STEPS 6-7: If the operation is performed successfully (connection established, modified or deleted, accordingly), a confirmation message is sent back from the API of the target edge cluster to the SD-WAN controller.

STEP 8: Once the process has finished, the Controller returns a confirmation message.

NOTE: Although not shown in the diagram, some metadata information shared between the components is stored in the etcd database.

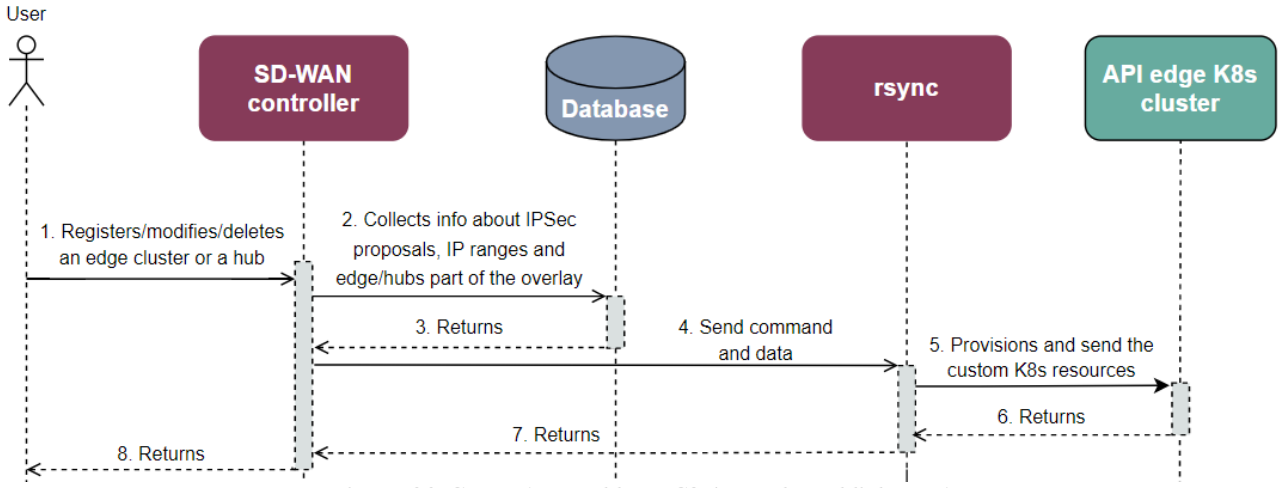


Figure 30. SD-WAN enabler UC2 (tunnel establishment)

The **last use case** is related to the **connection of hubs with edge cluster.** The diagram and related steps are depicted below. It should be mentioned that the flow may be activated in alternative ways (for instance, in the previous use case, when a tunnel with the edge cluster is established, the connection with a hub can be indicated and be part of the flow as well).

STEP 1: The user consumes the API of the SD-WAN Controller to create, modify or delete a connection (establish a tunnel) between a hub and an edge cluster.

STEPS 2-3: The SD-WAN Controller gathers needed information about the overlay, the IP addresses and IPSec proposals available from the database, and sends the required data to the rsync component.

STEP 4: The Controller sends the required data to the rsync component to setup the hub.

STEP 5: The rsync provisions the needed manifests and interacts with the API of the target hub cluster.

STEPS 6-7: If the operation is performed successfully (connection established, modified or deleted, accordingly), a confirmation message is sent from the API of the target hub cluster to the SD-WAN Controller.

SETP 8: Then, the controller mandates the rsync to prepare the required K8s resources so the hub provisions (modifies or deletes) the tunnel with the edge node.

STEPS 9-10: By means of custom K8s resources, the hub cluster sends in turn a set of K8s resources to the edge cluster to set up (modify or delete) the secured connection between them.

STEPS 11-13: If the operation is performed successfully, a confirmation message is sent back from the API of the target hub cluster to the SD-WAN controller.

STEP 14: Once the process has finished, the Controller returns a confirmation message.

NOTE: Although not shown in the diagram, some metadata information shared between the components is stored in the etcd database.

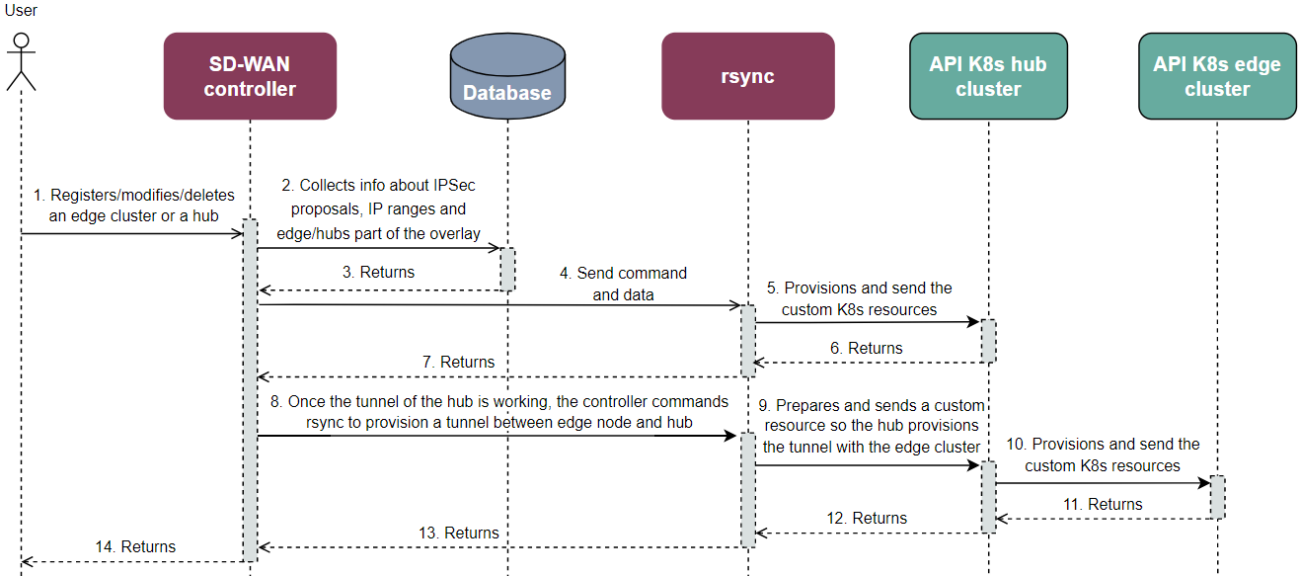


Figure 31. SD-WAN enabler UC3 (connection of hubs with edge cluster)

4.1.6.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/sd_wan_enabler.html

Table 18. Implementation status of the SD-WAN enabler

Category	Status
Components implementation	Because of internal planning, the implementation of this enabler will start at M18. Therefore, components are still immature to be delivered.
Feature implementation status	Since the components have not been implemented, any feature is available at this moment.
Encapsulation readiness	Any container has been developed yet (hence, K8s manifests and Helm charts are not ready)
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.7. WAN Acceleration enabler

4.1.7.1. Structure and functionalities

The WAN acceleration enabler will incorporate features that will improve the connections among the clusters and/or sites managed by ASSIST-IoT, and towards the Internet. It will be controlled by the SD-WAN enabler for establishing tunnels, and will be in charge of implementing features to support multiple WAN links, firewalling, tunnelling setups and traffic control, including traffic shaping. Depending on its configuration (via the SD-WAN enabler), it could act as:

- An SD-WAN Edge component, present in each K8s cluster, with a dedicated K8s controller and a Containerised Network function (CNF) through which traffic goes through it. The CNF will embed functions to setup aspects such related to IPSec, firewalling, DNS, DHCP and WAN link management, whereas a Custom Definition Resource (CRD) controller contains all the sub-controllers to create, query and configure these features.
- A SD-WAN hub, which will act as a middleware among clusters and/or between them and the Internet, enabling the introduction of additional CNFs related to security, filtering, traffic shaping, etc. Once the basic features are implemented, the incorporation of additional ones (as CNFs) will be evaluated.

The structure diagram of the enabler is presented in the Figure 32. Although the CNF exposes an API, this will be only consumed by the enabler's dedicated K8s controller, which will be triggered via the host's K8s API as a response to a user command, or after a call from the SD-WAN enabler.

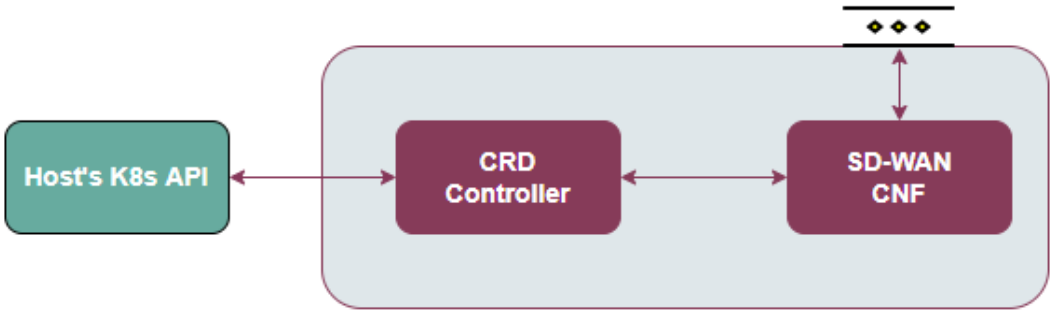


Figure 32. High-level diagram of the WAN Acceleration enabler

Implementation technologies

Table 19. Implementation technologies for the WAN acceleration enabler

Technology	Justification	Component(s)
openWRT	Linux operating system targeting embedded devices to route network traffic. It enables acting over network-related aspects.	SD-WAN CNF
ovn4nfv-k8s-plugin	CNI plugin based on OVN to create virtual networks in run time over a K8s cluster.	Deployed on target cluster, CRD controller
K8s custom resource definition controller	Controller created to act upon the K8s manifest sent by the SD-WAN enabler, to configure CNF rules related to firewalling, IPSec and access networks.	CRD Controller
IPSec	Technology to establishing tunnels between clusters managed by SD-WAN.	SD-WAN CNF
Multus CNI	CNI plugin to allow containers to have multiple network interfaces (in this case, Cilium as the main one from the project, and the aforementioned ovn4nfv-k8s-plugin)	Deployed on target cluster

4.1.7.2. Communication interfaces

Table 20. Communication interfaces (API) of the WAN Acceleration enabler

Method	Endpoint	Description
GET/PUT	**/services/{id}**	To list all the services supported by the CNF, and execute an operation for one (e.g., mwan3, firewall, IPsec-related).
GET/PUT	**/interfaces/{id}**	To list all the available interfaces and their specific information, allowing enabling or disabling them.
GET/POST/PUT/DELETE	**/mwan3/policies/{id}**	Policies define how traffic will be routed through the WAN managed interfaces.
GET/POST/PUT/DELETE	**/mwan3/rules/{id}**	Rules apply policies over specific source/destiny IP addresses, ports, IP type, protocol, etc.
GET/POST/PUT/DELETE	**/firewall/zones/{id}**	Groups one or many interfaces to be source or destination for forwarding, rules and redirects.
GET/POST/PUT/DELETE	**/firewall/redirects/{id}**	To define NAT rules.
GET/POST/PUT/DELETE	**/firewall/rules/{id}**	To specify accept, drop and reject rules to restrict access to specific ports or hosts.
GET/POST/PUT/DELETE	**/firewall/forwardings/{id}**	To control traffic between zones.

4.1.7.3. Use cases

The **use cases** related to the depicted endpoints will always follow the same flow, either for **configuring or querying** the CNF components. These use cases are related to WAN interfaces, policies, firewall, as explained above, and the diagram and involved steps are the following:

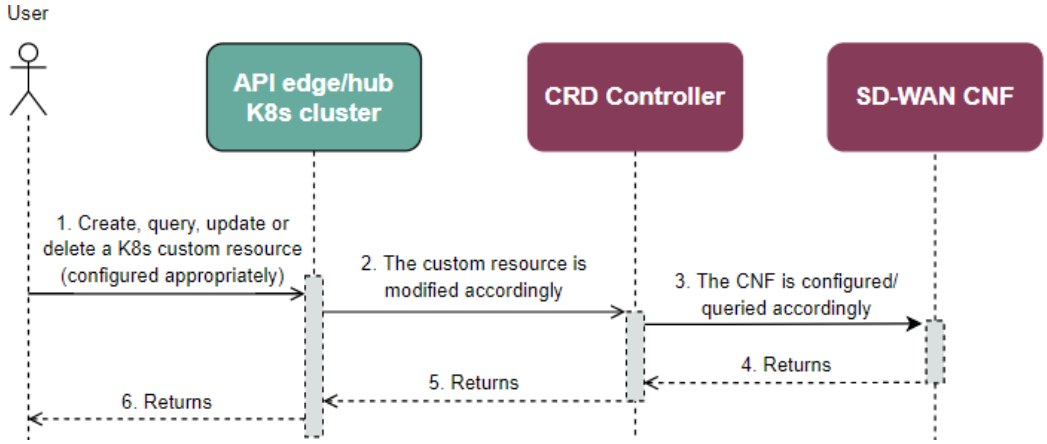


Figure 33. WAN Acceleration enabler UC (configuring/querying the CNF)

STEP 1: The user consumes the Kubernetes API to interact with the K8s' custom resource developed for the enabler. In the end, it will entail configuring or querying the CNF.

STEP 2: The information is sent from the K8s API to the involved controller.

STEP 3: The controller performs the required action, interacting with the API exposed by the CNF.

STEPS 4-6: Once the process has finished, the K8s API will return a confirmation message, based on the response from the CNF.

Another **use-case** is triggered by the SD-WAN enabler, when it **establishes, configures and deletes tunnels** on the edge cluster or hub that it manages. The flow is identical, but instead of initiated by a user it is done by the SD-WAN enabler.

4.1.7.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/wan_acceleration_enabler.html

Table 21. Implementation status of the WAN Acceleration enabler

Category	Status
Components implementation	Because of internal planning, the implementation of this enabler will start at M20. Therefore, components are still immature to be delivered.
Feature implementation status	Since the components have not been implemented, no feature is available at this moment.
Encapsulation readiness	No container has been developed yet (hence, K8s manifests and Helm charts are not ready)
Deployed with the Orchestrator in a laboratory environment	Not yet

4.1.8. VPN enabler

4.1.8.1. Structure and functionalities

This enabler facilitates access to a node or device from a different network to the site's private network using a public network (e.g., the Internet) or a non-trusted private network. VPN enabler should allow High Availability (HA) strategies, with servers distributed in different nodes, and failover mechanisms. As a first step, the site's network will be considered trusted, so VPNs will not be needed to connect nodes or devices that belong to it. The VPN Enabler will expose two endpoints: (i) an HTTP REST API for managing the VPN and its clients and (ii) the VPN endpoint to which the clients will connect, as one can observe from the general diagram of the enabler (Figure 34).

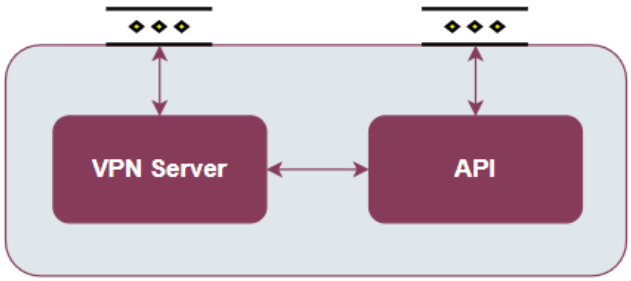


Figure 34. High-level diagram of the VPN enabler

NOTE: At this moment, to connect two host machines directly using a VPN (for instance, to add it as a remote K8s cluster/node via VPN), it is recommended to use a VPN without using the containerised version. Instructions for this use case will be provided (encapsulation exception that will be documented in D3.7).

Implementation technologies

Table 22. Implementation technologies for the VPN enabler

Technology	Justification	Component(s)
Node.js	It is a Cross-platform execution environment. It is written in JS and allows the realization of highly scalable web servers. The enabler API uses Node JS as the leading technology.	API
Express	A fast and minimalist web framework for used to create the API endpoints.	API
WireGuard	WireGuard is a fast, modern, secure and easy to configure VPN tunnel. WireGuard is faster, simpler, leaner, and more useful than IPsec, so it intends to be considerably more performant than OpenVPN.	VPN server

4.1.8.2. Communication interfaces

Table 23. Communication interfaces (API) of the VPN enabler

Method	Endpoint	Description
GET	**/info**	Adds an interface to be bonded
GET	**/info/conf**	Gets a list of managed interfaces
GET	**/keys**	Modifies the order of priority among the managed interfaces
GET/POST/DELETE	**/client**	Endpoint to get information about a client, eliminating it, or activating it.
DELETE	**/client**	Returns the list of clients registered in the server
PUT	**/client/enable**	Enables a client with the specified the public key
PUT	**/client/disable**	Disables a client with the specified the public key

Table 24. Communication interface (UDP) of the VPN enabler – VPN server

VPN Tunnel	**Dedicated port**	Port to connect VPN clients to the VPN server

4.1.8.3. Use cases

The **first use** case of this enabler appears when a user wants to **obtain information about the network interface of the VPN server**. Its diagram and related steps are the following:

STEP 1: The user makes an HTTP GET request to the API to obtain the information about the VPN server network interface.

STEP 2: The API executes interacts with the VPN server to get the information.

STEPS 3-4: The output returned by the server is sent to the user via the API, finishing the operation.

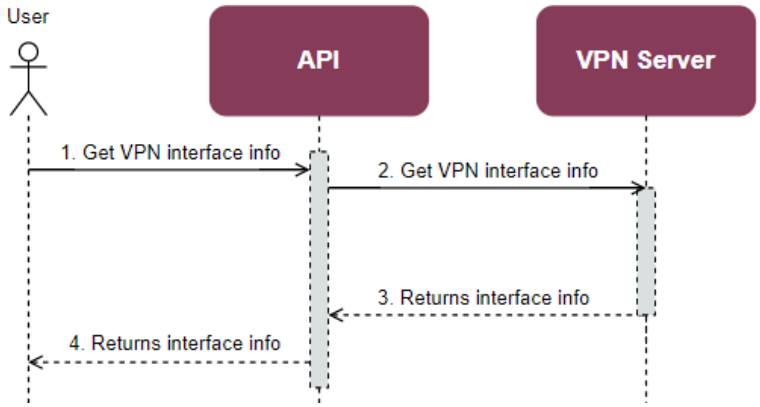


Figure 35. VPN enabler UC1 (get network interface information)

NOTE: The flow is identical for retrieving the configuration file of the network interface (in step 2, considering another command).

The **second use case** is to **generate the needed keys to create a new VPN client**. The diagram and the involved steps are the following:

STEP 1: The user makes an HTTP GET request to the API to generate the needed keys to create a new VPN client.

STEP 2: The API forwards the action to the VPN Server.

STEP 3: The VPN Server generates the needed keys (public, private and pre-shared) and returns them to the API.

STEP 4: The API passes the keys to the user. With these steps, the client keys are provisioned but the client is not enabled yet. To enable it, the following flow applies, initiated by the user:

STEP 5: A user makes an HTTP POST request to the API to create a new client, attaching the pre-shared and the public keys in the request body.

STEP 6: The API assigns an IP address of the VPN server subnet to the new client and communicates with the VPN server to provision the client, using the provided keys and the assigned IP.

STEP 7: The VPN server adds the new client to its configuration and to the network interface.

STEP 8: The VPN server returns the result of the operation to the API.

STEP 9: The API sends an HTTP request to the LTSE API to save the information of the new client.

STEP 10: If it is stored successfully, the LTSE returns a confirmation.

STEP 11: Finally, the API returns the necessary data (server public key, client IP, ...) to configure a client and establish a connection to the VPN server.

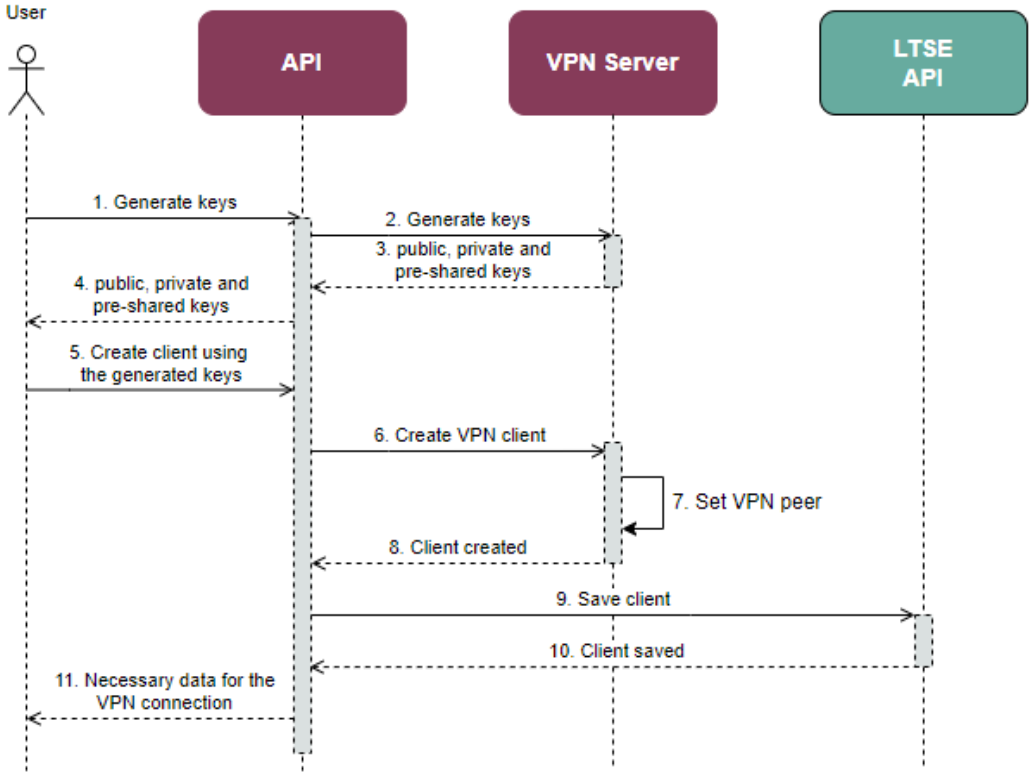


Figure 36. VPN enabler UC2 (generate client keys and create client)

The **third use case** is to **delete a VPN client**. The diagram and steps are the following:

STEP 1: The user makes an HTTP DELETE request to the API to delete the client specified by its public key.

STEP 2: The API forwards the action to the VPN Server.

STEPS 3-4: The VPN server removes the client from its configuration and from the network interface, returning the result of the operation.

STEP 5: The API sends an HTTP request to the LTSE API to delete the client.

STEP 6: If it is deleted successfully, the LTSE returns a confirmation.

STEP 7: The API returns the result of the operation.

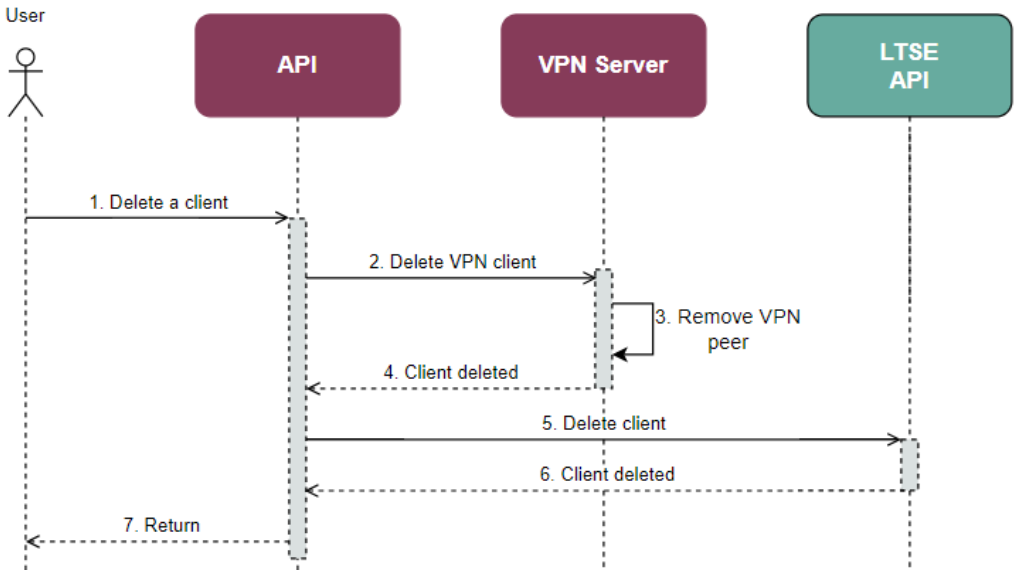


Figure 37. VPN enabler UC3 (delete client)

The **fourth use case** is to **enable/disable a VPN client**. The VPN server does not distinguish between creating and enabling a client, nor deleting and disabling it. However, thanks to the LTSE, the keys and internal IP addresses are kept in case clients are enabled or disabled. The diagram and involved steps are the following:

STEP 1: The user makes an HTTP PUT request to the API to enable the client specified by its public key.

STEPS 2-3: The API sends an HTTP request to the LTSE API to obtain the client information, which returns it.

STEP 4-6: The API communicates with the VPN server to create or delete the user. It also adds/removes the peer to its configuration and to the network interface, returning the result of the operation.

STEP 7-8: The API sends an HTTP request to the LTSE API to update the client (set *enabled* field to *true*). If everything is OK, the LTSE API returns an answer to the API,

STEP 9: The API returns the result of the whole operation.

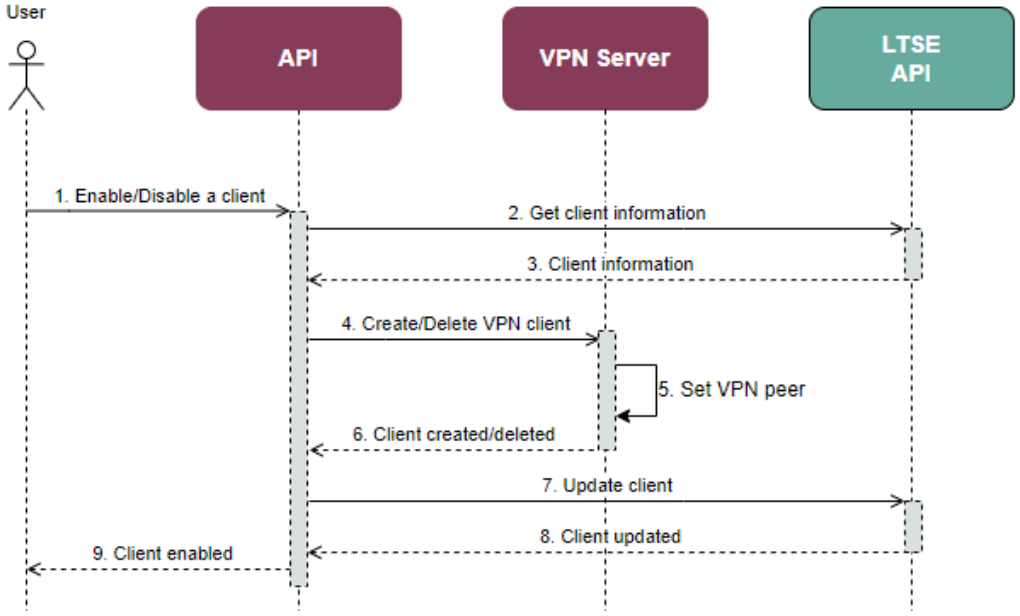


Figure 38. VPN enabler UC4 (enable/disable client)

The **last use case** is to **connect to the VPN using a client**. To that end, a user has to configure an external VPN client. The diagram and involved steps are the following:

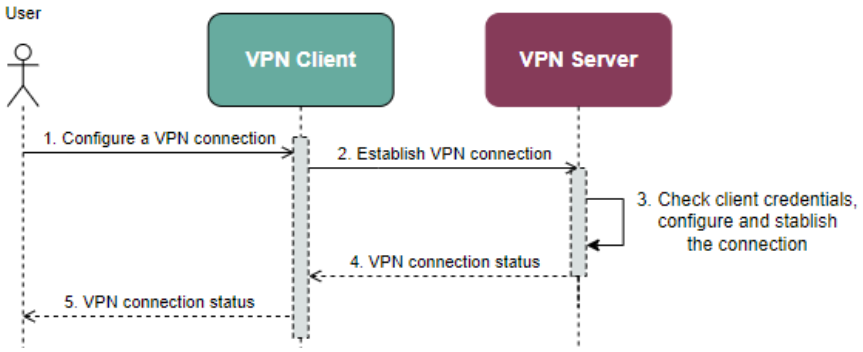


Figure 39. VPN enabler UC5 (connect client)

STEP 1: The user configures a VPN connection and starts the connection process using a client.

STEP 2: The client tries to establish a connection to the server exposed by the VPN enabler.

STEP 3: The server checks the client credentials (the keys) and, if the credentials are valid, establishes the VPN connection.

STEPS 4-5: Information about the connection is sent to the client, which can be seen by the user.

4.1.8.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/vpn_enabler.html

Table 25. Implementation status of the VPN enabler

Category	Status
Components implementation	A first version of the components is already in place.
Feature implementation status	There are some pending features before closing the development of this enabler: • HA strategy has not been implemented yet. • Endpoints for listing and auditing clients in a customized way will be added in future versions. Now, the information about the VPN status and its clients only is provided by the execution of WireGuard commands
Encapsulation readiness	Docker images and Helm Chart created and tested
Deployed with the Orchestrator in a laboratory environment	Not yet

4.2. Data Management enablers

4.2.1. Semantic Repository enabler

4.2.1.1. Structure and functionalities

This enabler offers a "nexus" for data models and ontologies, that can be uploaded in different file formats, and served to users with relevant documentation. This enabler aims at supporting files that describe data models or data transformations, such as ontologies, schema files, semantic alignment files, etc. Additionally, human-readable documentation for the models will be served. Offered files, their metadata and documentation are, in principle, public, so that this enabler could be used as support for a shared semantic ecosystem. The secure access restrictions may be more tightly controlled with an API manager in future versions. In short, the core function of this enabler is to be a database of data models and ontologies, with public read access, supporting:

- Versioning: different versions of data models,
- Ownership: only the data model owner may update a data model,
- Provision & search: data models are public and browseable,
- Documentation: serving that provided by data model owner.

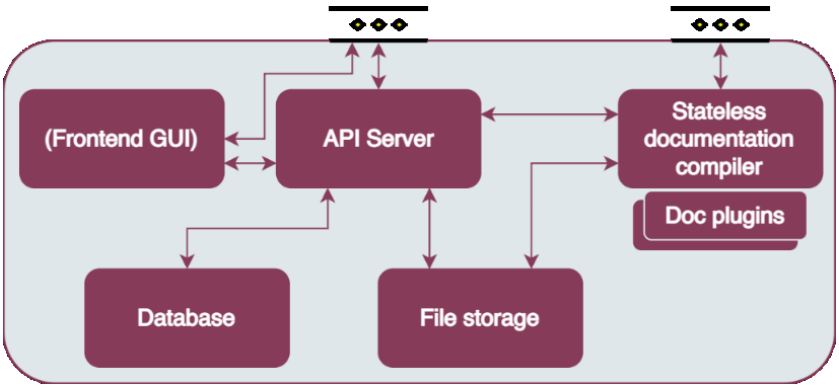


Figure 40. High-level diagram of the Semantic Repository enabler

Implementation technologies

Table 26. Implementation technologies for the Semantic Repository enabler

Technology	Justification	Component(s)
Akka HTTP	HTTP server	API server
MongoDB	Data persistence	Database
MinIO	Store for the data models and documentation pages	File storage
Scala	Custom documentation compiler server and plugins	Documentation compiler

4.2.1.2. Communication interfaces

Most of the endpoint URLs contain the version id fragment, which may be for example numeric, conforming to the Semantic Versioning standard, or almost any other string. To specify the latest available version, "latest" should be used as the version id.

Table 27. Communication interfaces (API) of the Semantic Repository enabler

Method	Endpoint	Description
GET	**/**	Lists available repositories.
POST/PUT/ DELETE	**/{namespace id}**	Creates (POST), updates (PATCH), or removes (DELETE) a specified namespace and its settings.
GET	**/{namespace id}**	Returns the settings of the namespace and lists models in it.
GET	**/{namespace id}/{model id}**	Returns the metadata of the model and lists the available versions of the given model.
POST/PUT/ DELETE	**/{namespace id}/{model id}/{version id}**	Creates (POST), updates (PATCH), or removes (DELETE) metadata of a version of a model (version, creation data, modification date, description, etc.).
GET	**/{namespace id}/{model id}/{version id}**	Returns the metadata of the given model version.
POST/ DELETE	**/{namespace id}/{model id}/{version id}/content? format={data format}**	Sets (POST) or removes (DELETE) a specified file from the server.
GET	**/{namespace id}/{model id}/{version id}/content? format={data format}**	Returns the specified version of a data model in a given format. E.g., /raul/saref/1.1/content?format=rdfxml returns a 'saref' model from repository 'raul' in version 1.1 in file format RDF/XML
POST/ DELETE	**/{namespace id}/{model id}/{version id}/doc/{file name}**	Uploads (POST) or deletes (DELETE) additional documentation source files (images, styles, Markdown) that can be used during documentation compilation.
GET	**/{namespace id}/{model id}/{version id}/doc**	Returns the documentation for a model.
POST	**/dg?[param1=value1 …]**	Requests a compilation of a set of documentation source files in « sandbox » mode. The compiled files are returned to the user.

NOTE: Starting with an empty database, if a user wants to create a repository named e.g., 'tea' and store in it a data model named 'pu-erh' in two formats, the process to do this is as follows:

1. **POST** /tea with JSON specifying the settings for the new repository.
2. **POST** /tea/pu-erh/1.0/ with JSON specifying the metadata of the data model.
3. **POST** /tea/pu-erh/1.0/content?format=rdfxml with the raw XML content.
4. **POST** /tea/pu-erh/1.0/content?format=ttl with the raw Turtle content.

Access to the endpoints in the initial version will be based on simple authentication. This could be later delegated to the authentication and authorisation enablers. Additionally, roles may be defined to control which users can access which functionalities. For example, repository and model creation may be restricted to only selected users.

4.2.1.3. Use cases

The **first use case** of this enabler is related to the modification of metadata, which allows a user to **modify the metadata** of namespaces, models, model versions, and other objects in the Semantic Repository. This is done via an HTTP REST interface, following the sequence diagram and steps specified below:

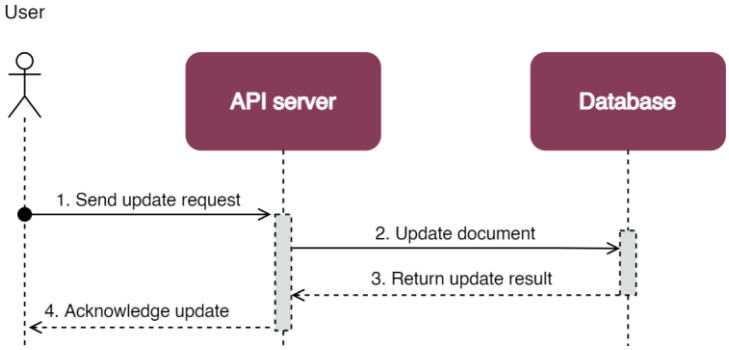


Figure 41. Semantic Repository enabler UC1 (modify metadata)

STEP 1: The user sends an HTTP request to the API server. The server validates the request.

STEPS 2-3: The API server instructs the database to update an appropriate document with the new metadata, which returns the updated result.

STEP 4: The API server reports the update result to the user.

The **second use case** refers to the request of metadata, which allows a user to **retrieve the metadata** of namespaces, models, model versions, and other objects in the Semantic repository.

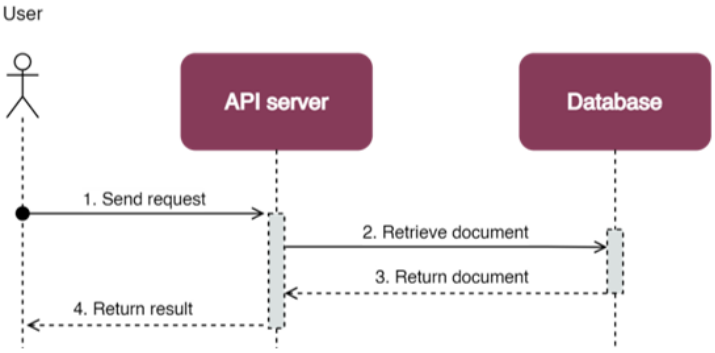


Figure 42. Semantic Repository enabler UC2 (get metadata)

STEP 1: The user sends an HTTP request to the API server. The server validates the request.

STEPS 2-3: The API server requests the needed information from the database, which returns it.

STEP 4: The API server returns the metadata to the user.

For each model, there can be many versions in the Repository, and for each such version there can be multiple available formats.

The **third use case** involves allowing a user to **upload a file representing a given model, with an associated version and format**. The Semantic Repository stores the file, records the upload, and automatically triggers documentation compilation, if there is an appropriate documentation plugin available. Documentation compilation is performed asynchronously.

STEP 1: The user uploads a data model to the API server.

STEPS 2-3: The API server forwards the file stream to file storage, which acknowledges the successful upload of the file.

STEPS 4-5: The API server requests a document update in the database, which returns an updated result.

STEP 6: The API server acknowledges the successful upload to the user and returns additional metadata (e.g., MD5 checksum).

STEP 7: The API server requests to compile the newly uploaded file to the stateless documentation compiler.

STEPS 8-9: The documentation compiler requests the needed files from the file storage, returning them.

STEPS 10-11: The documentation compiler invokes an appropriate documentation compilation plugin, which returns the compiled documentation.

STEPS 12-13: The documentation compiler stores the documentation in the file storage, which acknowledges a successful upload.

STEP 14: The documentation compiler returns the compilation result information to the API server.

STEPS 15-16: The API server updates an appropriate document in the database with the information about the compiled documentation. The database returns an update result.

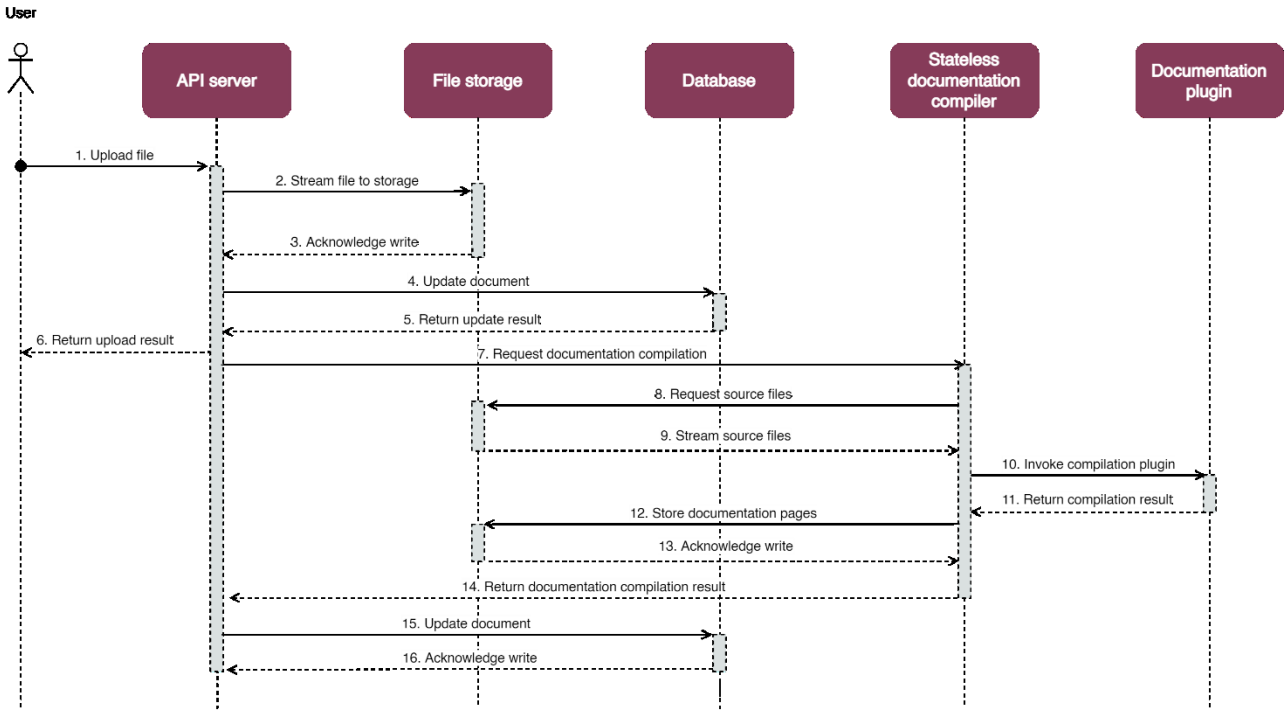


Figure 43. Semantic Repository enabler UC3 (upload file with model)

The **fourth use case** is related to the **downloading of data models and documentation** pages via the API server.

STEP 1: The user sends an HTTP request to the API server. The server validates the request.

STEPS 2-3: The API server requests the needed file from file storage, which returns a stream of the requested file.

STEP 4: The API server forwards the file stream to the user.

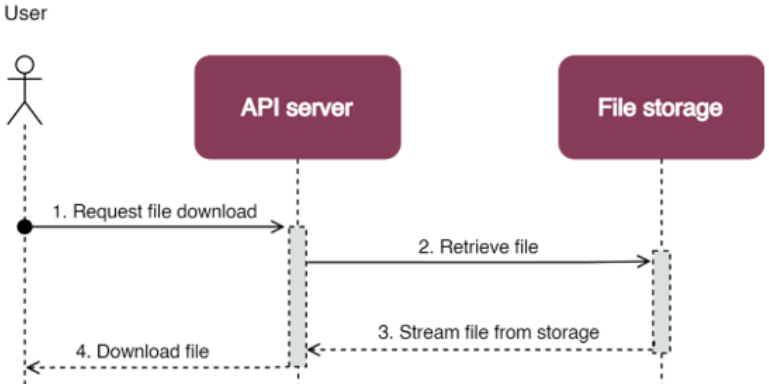


Figure 44. Semantic Repository enabler UC4 (get file with model)

To preview the results of documentation compilation without the need to upload a data model, this **(fifth) use case** allows a user is **to use a documentation "sandbox"** that offers them to interface directly with the stateless documentation compiler.

STEP 1: The user sends an HTTP request to the documentation compiler. The request contains the compressed source files and instructions for which plugin to invoke and with what parameters.

STEPS 2-3: The documentation compiler invokes an appropriate documentation plugin, which returns the compiled documentation files.

STEP 4: The documentation compiler returns the compressed output files to the user.

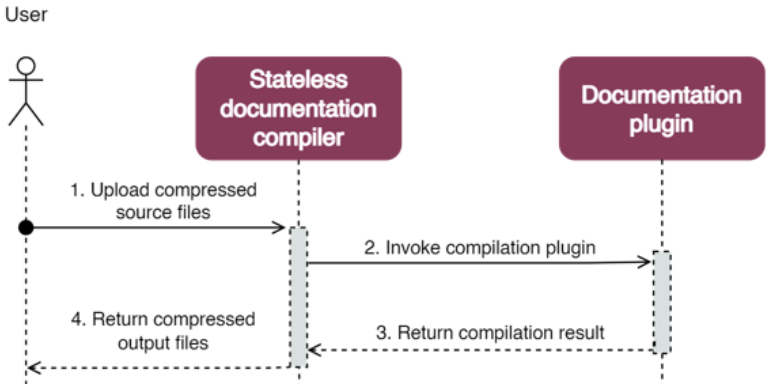


Figure 45. Semantic Repository enabler UC5 (use documentation sandbox)

The **last use case** is related to **the use of the Graphical User Interface (GUI)**, which acts as an entry point for all the functionalities of the enabler. The GUI component interfaces only with the API server.

STEP 1: The user performs an action in the GUI.

STEPS 2-3: The GUI relays the request to the API server, which returns a response to the request.

STEP 4: The GUI is updated with the new information.

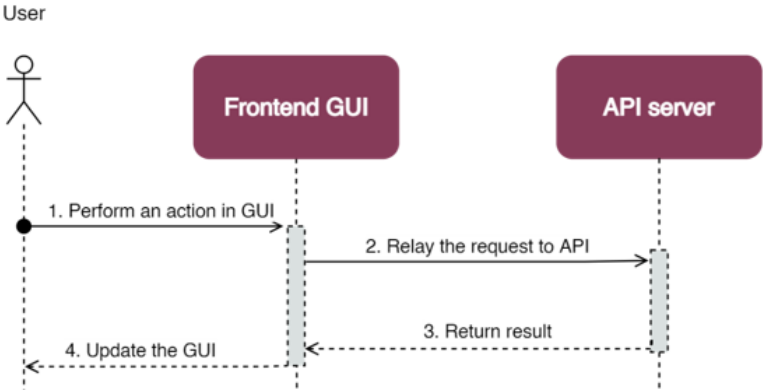


Figure 46. Semantic Repository enabler UC6 (manage the enabler with the GUI)

4.2.1.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_repository_enabler.html

Table 28. Implementation status of the Semantic Repository enabler

Category	Status
Components implementation	The status of the development of components is the following: the API server is under development, at 70%; the database is under development, at ~90%; The file storage is done; the documentation plugins have started but yet immature, and the GUI and the documentation compiler have not started yet.
Feature implementation status	Currently, the enabler allows storing and retrieving models, versioning models and namespacing models. It is still pending: • Namespace ownership and privileges. • Generating documentation.
Encapsulation readiness	All components are encapsulated in Docker images and are deployable through Docker compose. Any Helm chart has been prepared yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.2.2. Semantic Translation enabler

4.2.2.1. Structure and functionalities

Semantic Translation enabler offers a configurable service to change the contents of semantically annotated data following translation rules (so-called "alignments"), or alignment files. The core use case, around which this enabler is designed, is to move data between semantic ontologies (which can be thought of as data schemas or vocabularies) that can express the same information, without changing the meaning of the information.

Flexibility of design and expressivity of configuration files allow for other use-cases, such as semantic reduction (removing selected information, e.g., because of privacy reasons), further annotation (adding additional information based on data content and possibly external variables), or even encoding or encrypting selected data items into a serialised form.

The Semantic Translator supports RDF as the only modern standard for semantic data. By design it supports and promotes the "core ontology" design, in which data transformations are always unidirectional and done to, or from a central ontology, and paired into "translation channels" to achieve bidirectional transformations. In this manner, n-to-n translations can be easily implemented, and the cost of including a new data model in existing deployments does not grow exponentially. Translation services are offered as a "static" API for batch data, or through a publish-subscribe broker for streaming data.

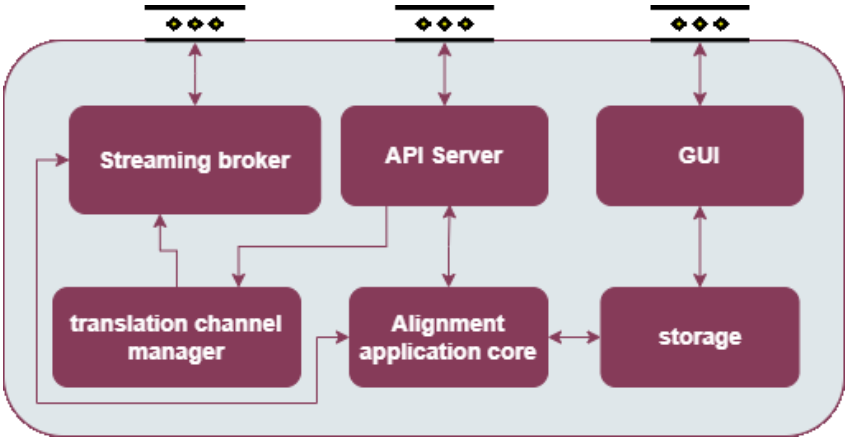


Figure 47. High-level diagram of the Semantic Translation enabler

Implementation technologies

Table 29. Implementation technologies for the Semantic Translation enabler

Technology	Justification	Component(s)
Akka Http	Http server	API Server
Apache Jena	RDF Processing	Alignment application core
Scala	Custom messaging channels management	Translation channel manager
PostgreSQL	Data Persistence	Storage
Apache Kafka	Message Broker	Streaming Broker
Javascript	Web interface	GUI

4.2.2.2. Communication interfaces

Table 30. Communication interfaces (API) of the Semantic Translation enabler – API Server

Method	Endpoint	Description
POST	**/alignments**	Upload new alignment.
GET	**/alignments/[{name}/{version}]**	Get list of stored alignments, or retrieve a specific alignment file.
DELETE	**/alignments/{name}/{version}**	Remove an alignment by name and version.
POST	**/convert**	Convert IPSM-AF 1.0 XML alignment (older format) into IPSM-AF 1.0 RDF alignment
POST	**/convert/TTL**	Convert cells in IPSM-AF 1.0 RDF alignment file from XML into TTL cell format.
POST/GET	**/channels**	Create a new translation channel (POST) or list currently active channels (GET)
DELETE	**/channels/{channelID}**	Remove a channel by ID
GET	**/logging**	Get logging level information
POST	**/logging/{level}**	Set logging level
POST	**/translation**	One-time translation using a sub-list of stored alignments
GET	**/version**	Get version information.
GET	**/swagger**	Display REST API summary with “try it out” options.

Table 31. Communication interfaces of the Semantic Translation enabler – Streaming broker

Method	Endpoint	Description
Pub/Sub	**Multiple topics**	Subscribe to an output topic or publish to an input topic.
Input topic	**Multiple topics**	Messages sent to input topic of any translation channel will enter the streaming core to be semantically translated following the translation channel configuration.
Output topic	**Multiple topics**	Output topic of a translation channel contains only the translated input message.
Monitoring topic	**Multiple topics**	If monitoring is enabled for a translation channel, the monitoring topic will output short timestamp information per each processed message.

4.2.2.3. Use cases

The **first use case** is related to the **definition/storage of an alignment**. Here, a user/client is able to store (compiled) alignment data to the Storage component triggering the following steps:

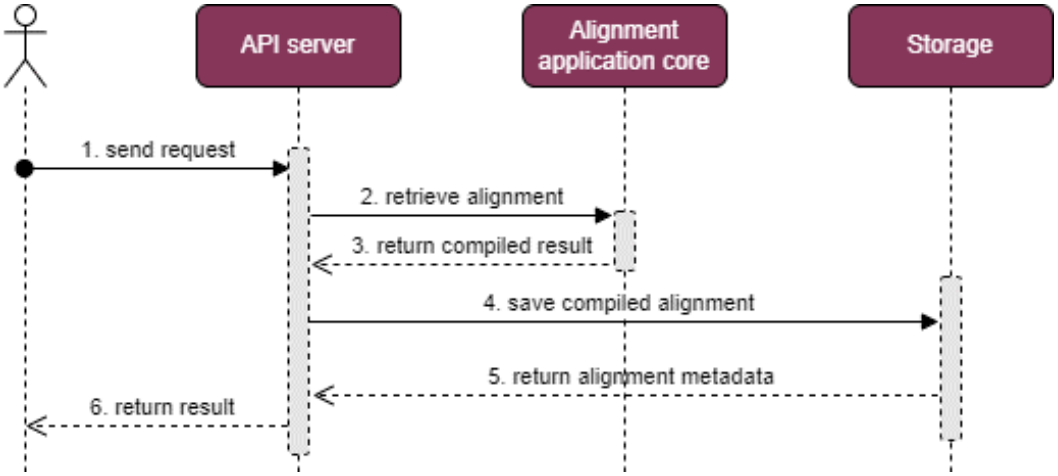


Figure 48. Semantic Translation enabler UC1 (store alignment)

STEP 1: The user/client sends an HTTP request containing the alignment data to the API server. The server validates the request.

STEP 2: The API server sends the alignment data to the Alignment application core component for compilation.

STEP 3: The Alignment application core component returns the compiled alignment to the API server.

STEPS 4-5: The API server saves the compiled alignment data to the Storage component, which the alignment metadata.

STEP 6: The API server returns the metadata to the user/client.

The **second use case** enables a user/client **to read metadata of an alignment** stored in the database. This use case has this sequence diagram:

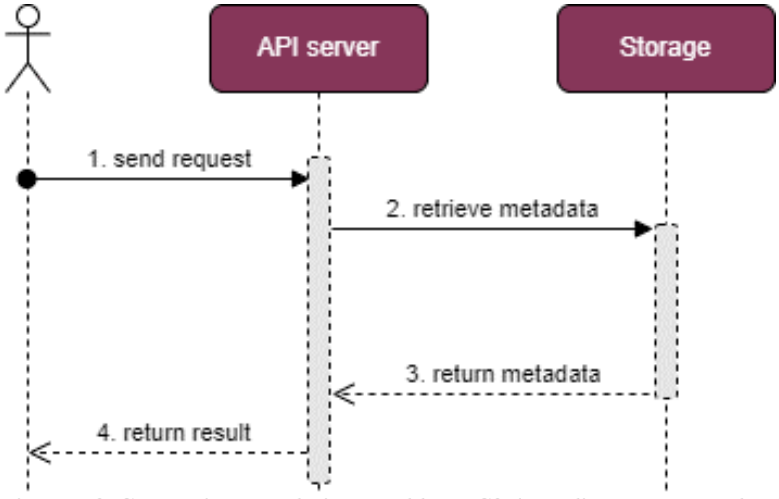


Figure 49. Semantic Translation enabler UC2 (get alignment metadata)

STEP 1: The user/client sends an HTTP request to the API server.

STEP 2: The API server sends an alignment metadata request to the Storage component,

STEP 3: The storage component returns it to the API.

STEP 4: Finally, the API server returns the metadata description to the user/client.

The **last use case** allows a user/client **to define/create a streaming-based translation channel** using available (compiled) alignments.

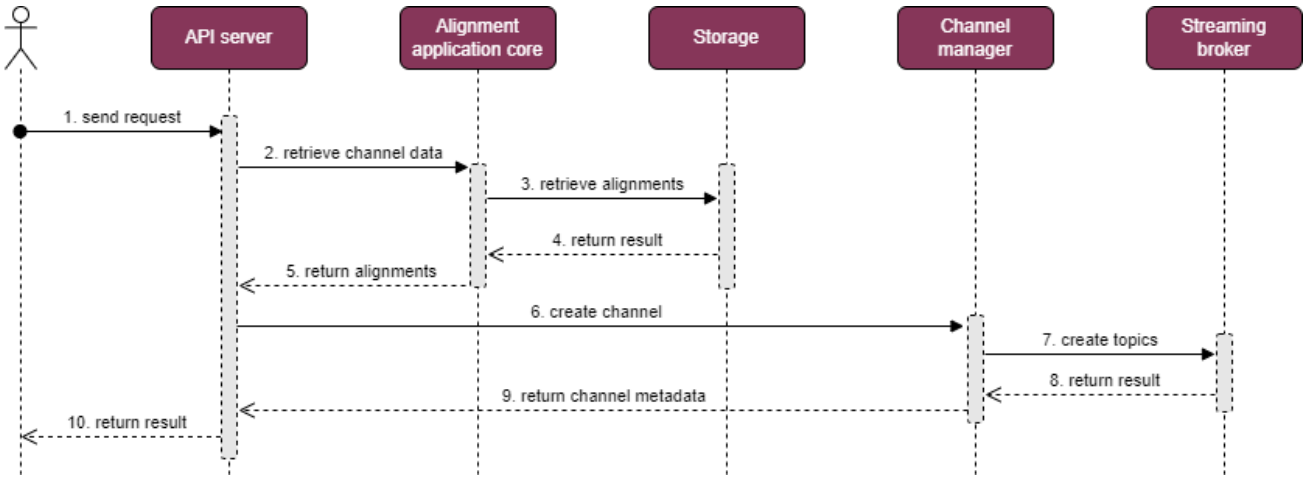


Figure 50. Semantic Translation enabler UC3 (create stream-based translation channel)

STEP 1: The user/client sends channel creation request to the API server.

STEP 2: The API server requests the Alignment application core to retrieve alignments required by the translation channel parameters.

STEPS 3-4: The Alignment application core retrieves the required (compiled) alignments from the Storage.

STEP 5: The Alignment application core returns the resulting translation information to the API server.

STEP 6: The API server asks the Channel manager component to create necessary topics for performing streaming translation.

STEPS 7-8: Channel manager forwards the topic creation request to the Streaming broker, which returns channel data.

STEP 9: The Channel manager the channel metadata to the API server.

STEP 10: The API server sends the result back to the user/client.

4.2.2.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_translation_enabler.html

Table 32. Implementation status of the Semantic Translation enabler

Category	Status
Components implementation	In the current version, the HTTP server, the API, the Streaming translation core and the Channel manager are done. The database storage is under construction (50%), whereas the alignment format v1.0 is done, and a second version is under construction (~30%).
Feature implementation status	Currently, this enabler has implemented streaming and batch translation capabilities. It is still pending: • Translation channel persistence (under construction). • Support for IPSM-AF 2.0.
Encapsulation readiness	All components are encapsulated in Docker images and are deployable through Docker compose. Any Helm chart has been prepared yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.2.3. Semantic Annotation enabler

4.2.3.1. Structure and functionalities

This enabler offers a syntactic transformation service, that annotates data in various formats and lifts it into RDF. Currently supported list of data formats include JSON, CSV and XML. Annotation is configured using RML (RDF Mapping Language) files, with CARML extension for streaming annotation. The core functionality is designed to be integrated into a pipeline before the Semantic Translation enabler, which requires the use of RDF. Semantic annotation lifts data to RDF, which can then be translated into different ontologies.

Batch annotation is done through a stateless REST API, that accepts (i) data to be translated and (ii) an RML file, returning the annotated result. Streaming translation was initially prototyped on Apache Flink, but because of performance concerns, high resource usage, difficult configuration and complicated deployment, the streaming components were re-designed and technology stack was changed. Apache Flink streaming can be still used in the old (beta) release of this enabler. New streaming infrastructure based on CARML Processor and custom Scala components will offer much better usability, and will be able to better manage different persistent configurations for streaming annotation.

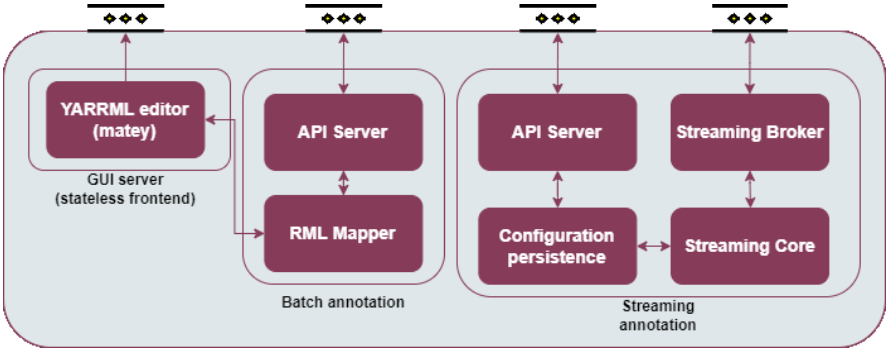


Figure 51. High-level diagram of the Semantic Annotation enabler

Implementation technologies

Table 33. Implementation technologies for the Semantic Annotation enabler

Technology	Justification	Component(s)
Akka Http	*Http server*	Streaming configuration API Server
RML	Mapping language	*All*
Javascript	Frontend RML editor	YARRML
Node.js	Batch annotation REST API	Batch API Server
RML Mapper Java	RML annotation	Batch annotation core
CARML Processor	Streaming annotation	Streaming core
Scala	Custom messaging channels management	Streaming core
PostgreSQL	*Data Persistence*	Configuration Persistence
Apache Kafka	*Message Broker*	Streaming Broker

4.2.3.2. Communication interfaces

Table 34. Communication interfaces (API) of the Semantic Annotation enabler – API server

Method	Endpoint	Description
POST	**:4000/execute**	Send annotation definition (RML) and data to be annotated. Receive annotated data.
GET	**:4000/**	Display Swagger GUI
GET	**:8081/**	Flink Web GUI for configuration of streaming annotation.

Table 35. Communication interfaces of the Semantic Annotation enabler – Streaming broker

Method	Endpoint	Description
Pub/Sub	**Multiple topics**	Subscribe to an output topic or publish to an input topic.
Input topic	**Multiple topics**	Messages sent to input topic of any annotation channel will enter the streaming core to be semantically annotated following the translation channel configuration.
Output topic	**Multiple topics**	Output topic of a annotation channel contains only the translated input message.
Error topic	**Multiple topics**	If error topic is configured for an annotation channel, the error topic will output information about any errors, that prevented publishing the annotated message on the output channel, including invalid data format and other annotation errors.

4.2.3.3. Use cases

The **first use case** involves **authoring RML files using GUI editor**. Preparation of RML files to configure annotation can be done in any text editor offline, or via the provided GUI. Using the GUI is fairly simple, with the option to write annotation files directly in RML, or in YARRML – a more human-readable format, that compiles to RML. Additionally, prepared RML can be tested on example data defined by the user. The GUI also comes with some simple built-in examples, that the user can modify to create their own RML and YARRML. This use case follows the next steps and sequence diagram:

STEP 1: User edits YARRML in the web browser editor, either starting from scratch, or from one of the provided examples.

STEP 2: User defines test data to test, if written YARRML is correct.

STEP 3: User requests annotation of provided example data with the defined YARRML.

STEPS 4-5: YARRML editor compiles YARRML to RML. It uses the RML and test data to an request annotation job at the RML Mapper.

STEPS 6-7: RML Mapper performs the annotation and returns the result, which is displayed by the YARRML editor to the user.

STEPS 8-9: User requests the compiled RML file, which is the compiled from YARRML by the editor.

STEP 10: YARRML editor presents RML to the user, who can then download it, or continue editing, or testing on different data.

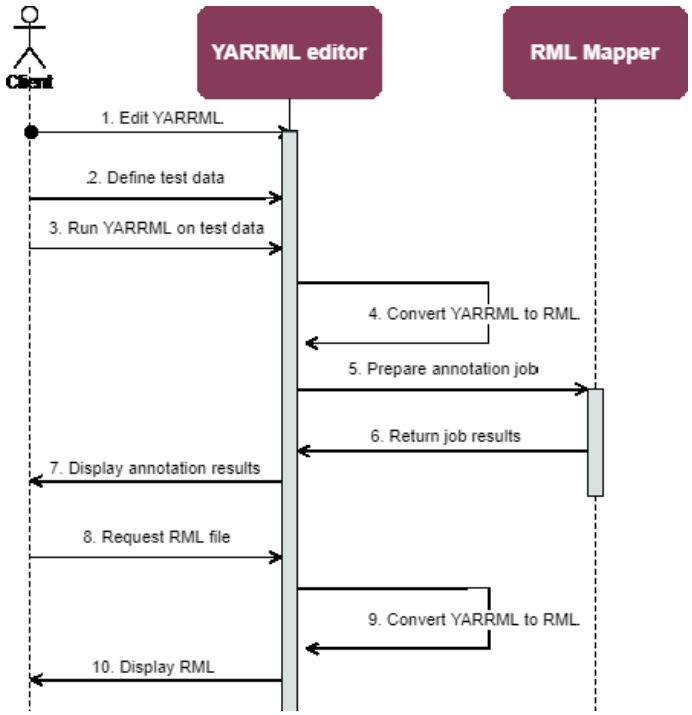


Figure 52. Semantic Annotation enabler UC1 (prepare RML files)

The **second use case** is related to the **use of batch annotation**. Using it is quite straightforward, as the service is stateless and idempotent. All information necessary to perform annotation must be sent in a single request by the user, who then receives the annotated result. The sequence diagram and involved steps are the following:

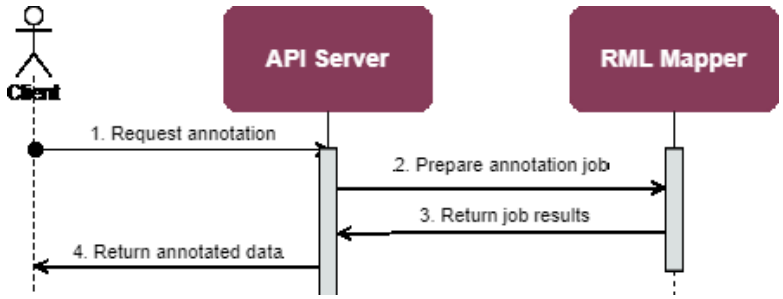


Figure 53. Semantic Annotation enabler UC2 (batch annotation)

STEP 1: User prepares annotation rules in RML and data to be annotated and sends it to the batch API server.

STEP 2: Batch API server prepares annotation job and sends it to RML Mapper.

STEP 3: RML Mapper performs annotation using data from the request and returns results – whether annotation was successful, or resulted in an error.

STEP 4: API server forwards annotated data and any errors to the user.

Before using the streaming annotation, a channel must be configured. Channel configuration specifies topics (input, output, and optional error topic) and annotation file to be used. Annotation files must be uploaded beforehand, and are retrieved, using ID specified in the channel configuration information. This **third use** case follows the next sequence diagram:

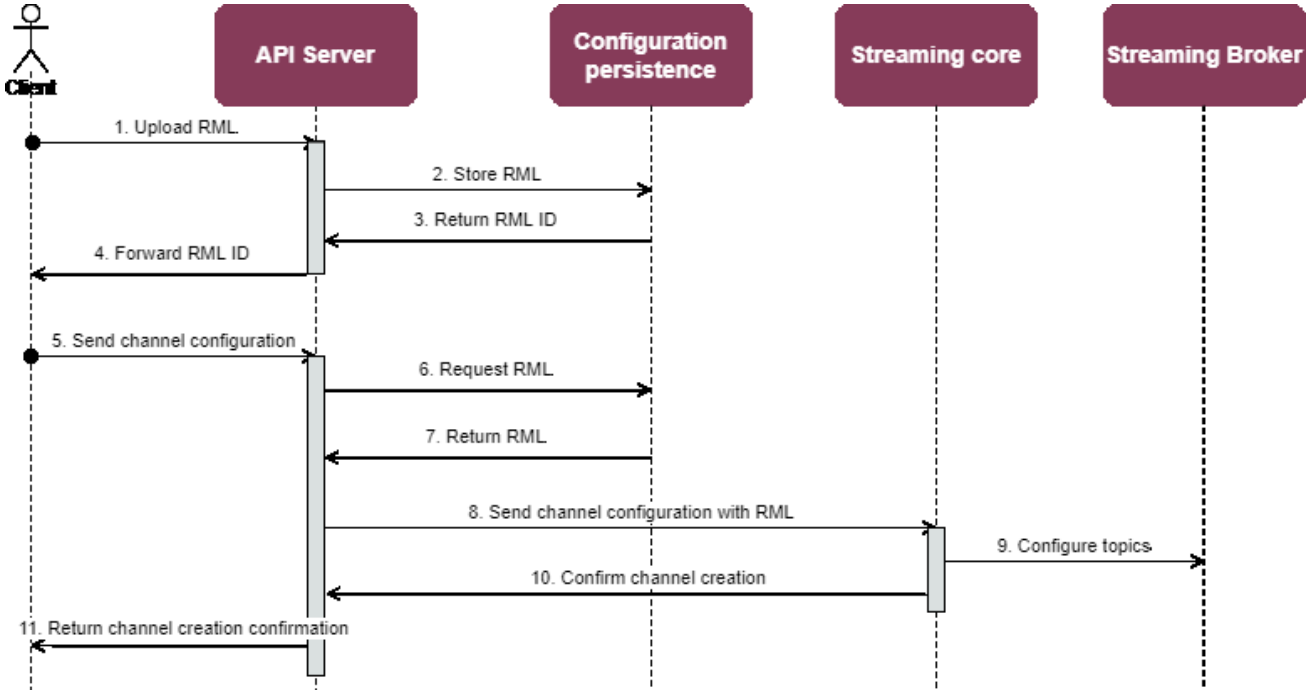


Figure 54. Semantic Annotation enabler UC3 (configure channel for stream annotation)

STEP 1: User uploads RML file to be used later.

STEPS 2-3: The API server uses the Configuration persistence component to store RML file under a given ID, returned by the latter component.

STEP 4: The API server forwards the stored RML ID to the user.

STEP 5: User sends channel configuration, that specifies the ID of the uploaded RML file.

STEPS 6-7: The API server retrieves a previously stored RML file from the persistence component, which returns it (or an error, if there is no RML file stored under the given ID).

STEP 8: The API server sends channel configuration with RML file to the streaming core.

STEP 9: The Streaming core configures topics in the streaming broker and materializes the annotation channel, storing RML and topic configuration in memory.

STEP 10: Streaming core confirms channel creation and returns channel ID and configuration information.

STEP 11: The API server forwards channel ID and configuration information to the user.

The **last (forth) use case** is the use of **streaming annotation capabilities**. This is performed via interaction with the streaming broker, which exposes input, output, and optional error topics. A consumer may subscribe to output topics, and optionally to error topics. In general, channels do not need to have an error topic configured, and error topics can be shared by multiple channels so that errors are aggregated.

Any message published on an input topic passes through the streaming core and is either annotated and published on the output topic, or an error is generated and forwarded to the error topic (if it exists for the given channel). A consumer does not need to have been subscribed to the output topic to subscribe to the error topic. In practice, consumers interested in handling annotation errors are not in the same group of interests, as "regular" clients that publish or receive messages via the annotator. This use case has the following diagram and involved steps:

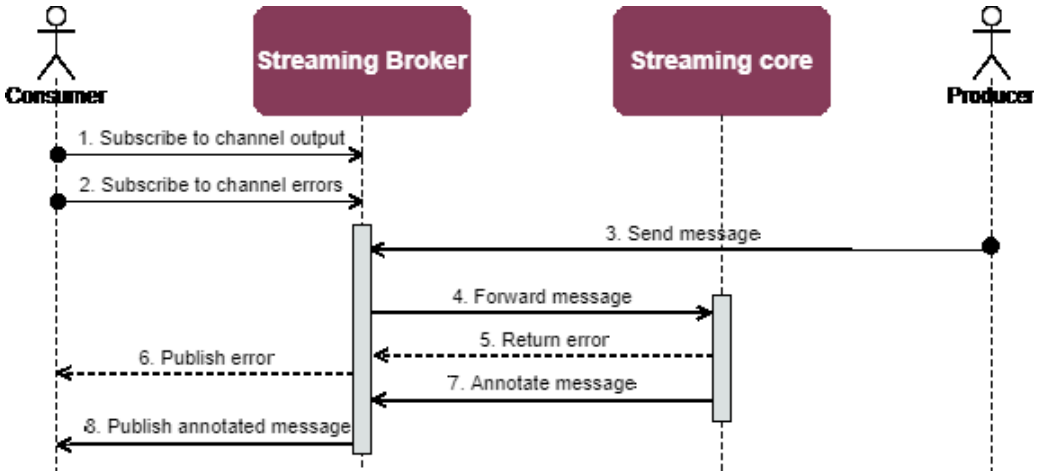


Figure 55. Semantic Annotation enabler UC4 (stream annotation)

STEP 1: Consumer subscribes to an output topic of a previously configured annotation channel.

STEP 2: Consumer (optionally) subscribes to an error topic of an annotation channel that was configured previously.

STEP 3: Producer publishes a message on an input topic of a previously configured annotation channel.

STEP 4: Streaming broker forwards the message to be annotated to the streaming core.

STEP 5: The streaming core attempts to annotate the message, following the configuration of the annotation channel. If there are any errors, they are forwarded to the error topic of the annotation channel.

STEP 6: If there are any errors, they are forwarded to subscribers of the error topic.

STEP 7: If the annotation was successful, the streaming core publishes it on the output topic of the annotation channel.

STEP 8: Streaming broker distributes the annotated message to all subscribers of the annotation channels output topic.

4.2.3.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_annotator_enabler.html

Table 36. Implementation status of the Semantic Annotation enabler

Category	Status
Components implementation	Currently, the following components are finished: HTTP server for batch annotation, API REST for batch annotation and GUI editor. The streaming-related components are in different stages of development: HTTP server for streaming annotation (0%), Streaming translation core (~30%), Streaming Broker (~50%) and Configuration persistence (0%).
Feature implementation status	Batch annotation and GUI editor features are in place, whereas Streaming annotation is still under construction.
Encapsulation readiness	Batch annotation components are encapsulated in Docker images and are deployable through Docker compose. Streaming annotation components are not encapsulated. Any Helm chart has been prepared yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.2.4. Edge Data Broker

4.2.4.1. Structure and functionalities

The Edge Data Broker (EDB) enables the efficient management of data demand and data supply among edge nodes based on a publish/subscribe schema, taking into account load balancing criteria. This enabler distributes data where it is needed for application, services, and further analysis while considered essential only in those deployments that involve IoT architectures.

A group of innovative features are provided by the Edge Data Broker, which facilitates the creation of pipelines, in which data flow can be controlled. Control over the conditional paths is achieved through scriptable broker nodes, while data may be analysed to cause alert events, directed to specific sinks depending on pre-defined conditions. The data broker is scriptable with a highly expressive language to enable a wide range of uses. Figure 56 presents the four components of Edge Data Broker enabler that are used to achieve the operational and intelligent functionalities:

- Subscriptions and messages between the broker and the Edge Nodes,
- Management of message scheduling, routing and delivery,
- Common interfaces for searching and finding information,
- Integration with other data brokers, and
- Distribution load-balancing strategies.

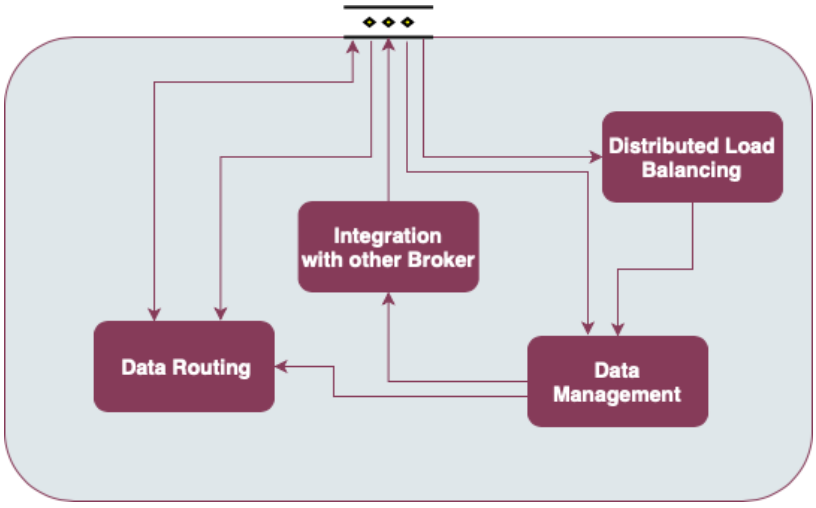


Figure 56. High-level diagram of the Edge Data Broker

Implementation technologies

Table 37. Implementation technologies for the Edge Data Broker

Technology	Justification	Component(s)
VerneMQ	MQTT-based data broker	All
Lua Scripts	Data hooks and routing	Data Management
Kubernetes	Nodes clustering and orchestration	All

4.2.4.2. Communication interfaces

Table 38. Communication interfaces (API) of the Edge Data Broker

Method	Endpoint	Description
GET	**/health**	The health check will return: • 200 when the Edge Data Broker is accepting connections and is joined with the cluster (for clustered setups). • 503 will be returned in case any of the above two conditions are not met.
GET	**/metrics**	Configurable endpoint that provides multiple metrics relevant to monitoring and alerting.

Table 39. Communication interfaces (MQTT) of the Edge Data Broker – Data Routing

Pub Sub	**Multiple topics**	The Pub/Sub interface of the Broker can be used to send/receive data among devices and applications

4.2.4.3. Use cases

Three use cases have been identified for the EDB. The **first use case** represents the **data distribution mechanism**, where an IoT device publishes a message in an ASSIST-IoT Gateway (e.g., GWEN) and the subscriber receives the message from another, located in another area. This functionality is feasible as the EDB is based on clustered nodes deployed in various devices. The sequence diagram and involved steps are:

STEPS 1-2: A first client subscribes to a topic of the Edge Data Broker on Gateway 1. The gateway acknowledges the subscription.

STEPS 3-4: A second client subscribes to the same topic of the Edge Data Broker enabler on Gateway 2. Again, the broker acknowledges it.

STEP 5: The first client publishes a message on the previous topic.

STEPS 6-7: The Edge Data Broker distributes the message to all the workers of the Cluster, and then the second client receives it.

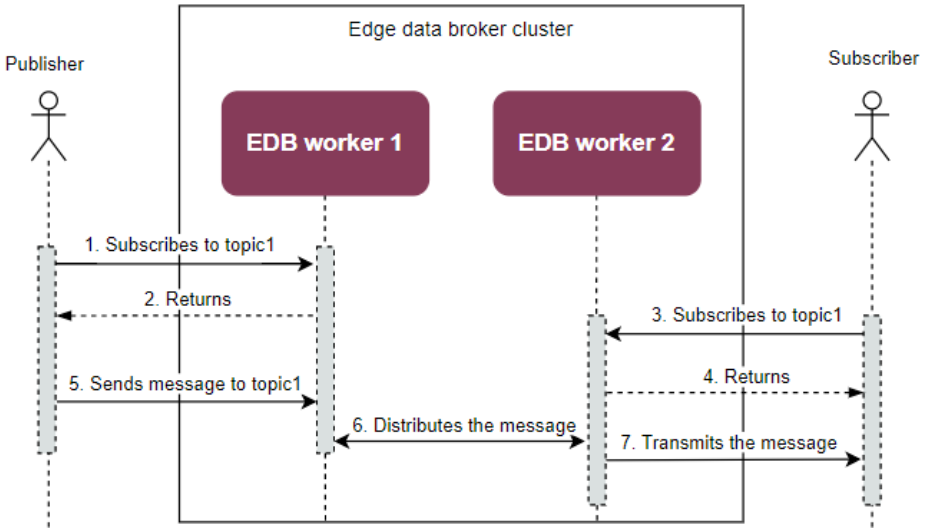


Figure 57. Edge Data Broker UC1 (data distribution)

A key function of the Edge Data Broker enabler is the **rule engine**. This feature can be used to transform the data and republish to different topics, in case some rule is triggered. The **second use case** (related to rules) considers the diagram and steps depicted below:

STEPS 1-2: A first client subscribes to a topic of the Edge Data Broker on Gateway 1. The gateway acknowledges the subscription.

STEPS 3-4: A second client subscribes to an alert topic of the Edge Data Broker enabler on Gateway 2. Again, the broker acknowledges it.

STEP 5: The first client publishes a message on the topic. This message has a value that can trigger a rule.

STEPS 6-8: The Rule Engine transforms this message to produce an alert and publishes this alert on the alert topic.

STEP 9: The Edge Data Broker distributes the alert to all the subscribed workers of the cluster.

STEP 10: The second client receives the published alert message.

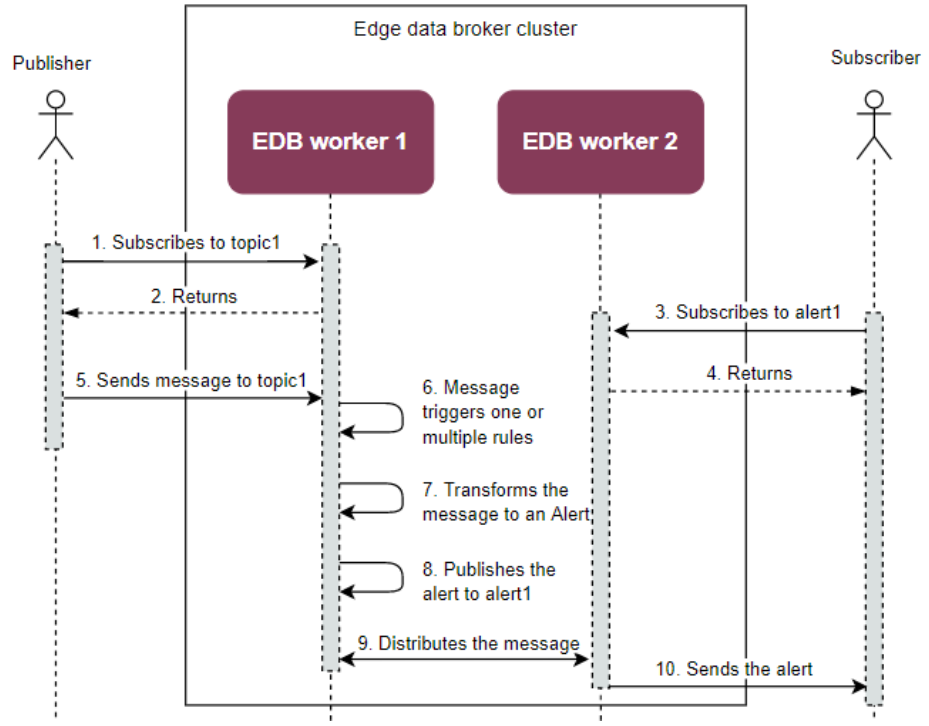


Figure 58. Edge Data Broker UC2 (rule engine)

In the **third use case, data filtering functionality** provides local intelligence capabilities to the Edge Data Broker, improving the efficient management of the data. The received data is filtered in real time and routed based on conditional events so that the subscriber will receive only meaningful data.

STEPS 1-2: A first client subscribes to a topic of the Edge Data Broker on Gateway 1. The gateway acknowledges the subscription.

STEPS 3-4: A second client subscribes to the filtered topic of the Edge Data Broker enabler on Gateway 2. Again, the broker acknowledges it.

STEP 5: The first client publishes a message on the topic. This message has a value that exceeds or is bellow a predefined threshold.

STEPS 6-7: The filtering system receives this message and either deletes it or passes it to the filtered topic based on the filter (threshold).

STEP 8: The second client either receives or not the message.

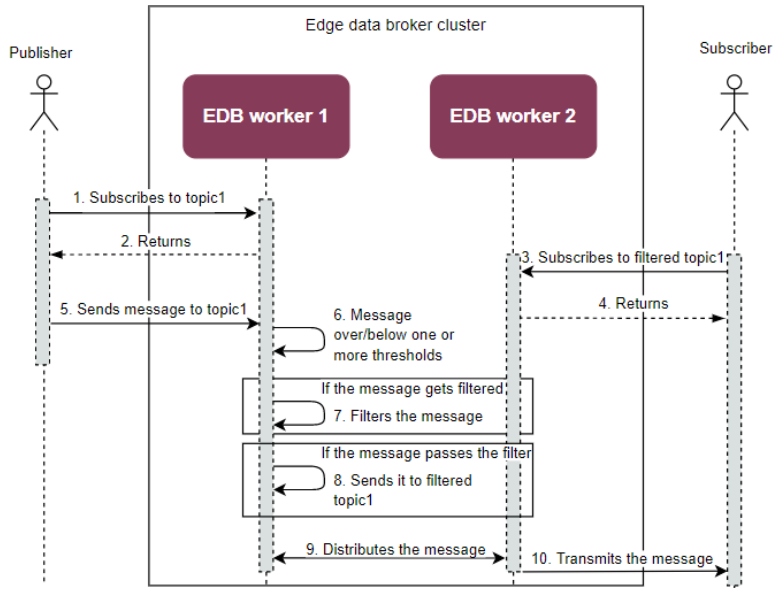


Figure 59. Edge Data Broker UC3 (data filtering)

4.2.4.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/edge_data_broker_enabler.html

Table 40. Implementation status of the Edge Data Broker

Category	Status
Components implementation	A first version of the components is already in place.
Feature implementation status	The operational and management functionalities implemented so far include: (i) routing and distributing pub/sub messaged over the distributed broker, (ii) data filtering based on thresholds, and (iii) rule engine to create alerts. In the following months, the adaption of EDB in different pilots will be explored and customised hooks/rules/deployments will be applied based on use case needs.
Encapsulation readiness	A Docker image for ARM64 has been created and tested. A Helm chart for deploying a cluster of EDB has been created. Both have been tested on a Cluster with two RaspberryPis 4 model B (for ARM64 architecture). Data filtering and rule engine features not encapsulated yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.2.5. Long-term Storage Enabler

4.2.5.1. Structure and functionalities

The role of the Long-Term Storage Enabler (LTSE) is to serve as secure and resilient storage, offering different storage sizes and individual storage space for other enablers (which could request back when they are being initialised in Kubernetes pods). Therefore, it is considered one of the ASSIST-IoT essential enablers, envisioned to be deployed on the cloud rather than the edge. Figure 60 depicts the high-level overview of the LTSE components, which functionalities are also described:

- **LTSE Gateway:** The entrance gate to the LTSE, acting as a proxy from ASSIST-IoT enablers and external services, whose data should be collected either at SQL server databases or NoSQL cluster nodes. To do so, the LTSE Gateway is based on REST API request, with appended SQL/NoSQL endpoints, respectively. Furthermore, the LTSE gateway also guarantees that the data will be kept safe, in face of various kinds of unauthorised access requests, or hardware failures, by only allowing access to the data once the Identity Manager and the Authorisation enablers have confirmed their access rights.

- **LTSE NoSQL cluster:** A group of one or more LTSE NoSQL node instances that are connected together, and carry out the distribution of tasks, searching and indexing, across all the NoSQL nodes. Every NoSQL node in the NoSQL cluster can handle HTTP and transport traffic by default with the external enablers through the LTSE gateway. The transport layer is used exclusively for communication between nodes; the HTTP layer is used by REST clients. The full hierarchy would be therefore, noSQL_Cluster > noSQL_Node > noSQL_Index > noSQL_document. For High Availability (HA), noSQL_document in LTSE_noSQL_Index may be distributed across multiple shards, which in turn are distributed across multiple nodes, if configured.
- **LTSE SQL server:** It manages the SQL databases, formed by different enablers data tables. It performs, hence, backup database actions on behalf of the enablers. The SQL_Server can handle multiple concurrent connections from external enablers via the LTSE Gateway. In general, the full hierarchy is: SQL_Cluster > SQL_Database > SQL_schema > SQL_table > SQL_row. For High Availability, a master database with one or more standby servers could be set up.

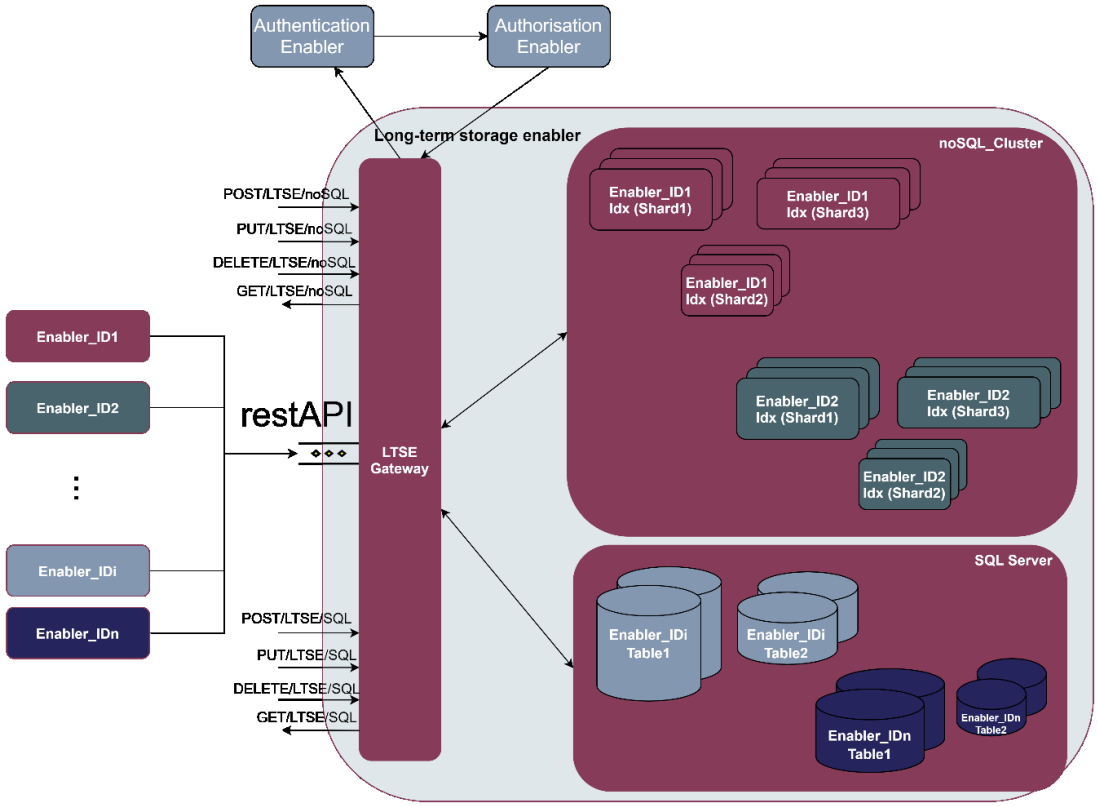


Figure 60. High-level diagram of the Long-term Storage enabler

Implementation technologies

Table 41. Implementation technologies for the Long-term Storage enabler

Technology	Justification	Component(s)
GinGonic	Gin is a web framework written in Go (Golang). It features a martini-like API with much better performance.	LTSE Gateway
PostgREST	A standalone web server that turns a PostgreSQL database directly into a RESTful API. The structural constraints and permissions in the database determine the API endpoints and operations.	LTSE Gateway
Elasticsearch API	Elasticsearch exposes REST APIs that are used by the UI components and can be called directly to configure and access Elasticsearch features	LTSE Gateway
Elasticsearch	A search and analytics engine which is based on Apache Lucene. Completely open source and built with Java, Elasticsearch is a NoSQL database, able to store data in an unstructured way that cannot use SQL to query it.	LTSE NoSQL Cluster
PostgreSQL	Open source object-relational database known for reliability and data integrity. ACID-compliant, it supports foreign keys, joins, views, triggers and stored procedures.	LTSE SQL Server

4.2.5.2. Communication interfaces

Table 42. Communication interfaces (API) of the Long-term Storage enabler

Method	Endpoint	Description
POST	**/SQL/DATABASES/:databasename**	Creates a databasename in the LTSE SQL cluster
POST	**/SQL/DATABASES/:databasename/TABLES/:tableName**	Creates a tablename in the databasename of LTSE SQL server
POST	**/SQL/DATABASES/:databasename/TABLES/:tableName/… DATA/data**	Inserts data into the tablename on the databasename of LTSE SQL server
GET	**/SQL/DATABASES/:databasename/TABLES/:tableName**	Obtains all the data contained within the tableName of the databasename of LTSE SQL server
PUT	**/noSQL/INDEX/<indexName>**	Creates a new index indexName in the LTSE noSQL cluster. When creating an index, you can specify the settings for the index, mappings for fields in the index, and Index aliases
GET	**/noSQL/INDEX/< indexName >**	Returns information about indexName index from the LTSE noSQL cluster
PUT	**/noSQL/INDEX/<indexName>/DOCUMENT/<_id>**	Adds a JSON document to the specified <indexName > index of the LTSE noSQL cluster and makes it searchable with an <_id>
GET	**/noSQL/INDEX/<indexName>/_doc/<_id>**	Retrieves the specified JSON document <_id> from the indexName of the LTSE noSQL cluster.

4.2.5.3. Use cases

There are 4 main use cases that apply in this enabler.

The **first one** is related to the **storage of NoSQL data** of an authorised Enabler **on a NoSQL cluster**, after provisioning an index on it. The diagram with the required steps is summarised below:

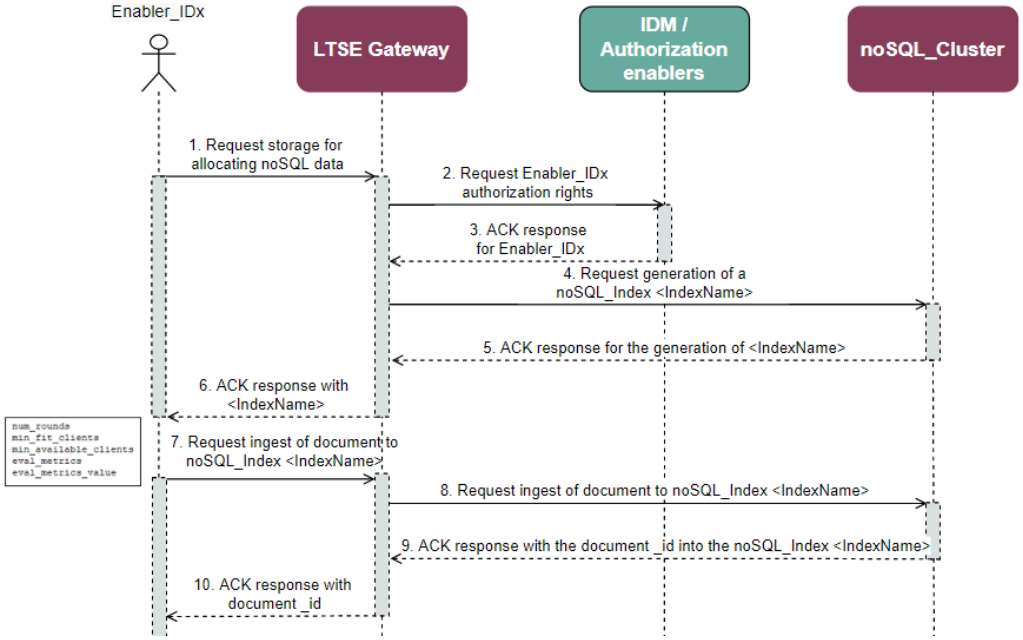


Figure 61. LTSE UC1 (store NoSQL data)

STEP 1: The Enabler_IDx interacts via LTSE gateway with the LTSE, requesting to create a NoSQL storage.

STEPS 2-3: The LTSE Gateway checks the authorization rights of the Enabler_IDx from IdM/Authorization enablers, which confirms or denies Enabler_IDx access to the LTSE noSQL cluster.

STEP 4: If granted, LTSE Gateway request the generation of Enabler_IDx index into LTSE noSQL_Cluster.

STEPS 5-6: LTSE noSQL_Cluster confirms the generation of <IndexName> index and inform to LTSE gateway, which, in turn, forwards the index details to the Enabler_IDx.

STEP 7: The Enabler_IDx requests ingestion of NoSQL data document to LTSE Gateway.

STEP 8: LTSE Gateway request the ingestion of Enabler_IDx NoSQL data document into <IndexName> of the LTSE noSQL_Cluster.

STEPS 9-10: LTSE noSQL_Cluster confirms the ingestion of document _id into the <IndexName> index of the LTSE noSQL_Cluster and informs to LTSE gateway, which, in turn, forwards the document details to the Enabler_IDx.

The **second use case** is related to **the retrieval of NoSQL documents** with a specific <IndexName> from the NoSQL cluster. The diagram and the related steps are the following:

STEP 1: The Enabler_IDx interacts via LTSE gateway with the LTSE, requesting specific data allocated into its NoSQL storage Index.

STEPS 2-3: The LTSE Gateway checks the authorization rights of the Enabler_IDx from IdM/Authorization enablers, which confirms or denies Enabler_IDx access to the LTSE NoSQL cluster.

STEP 4: If granted, LTSE Gateway request the associated information demanded into Enabler_IDx <IndexName> of LTSE noSQL_Cluster.

STEP 5: The LTSE gateway, in turn, forwards the document to the Enabler_IDx.

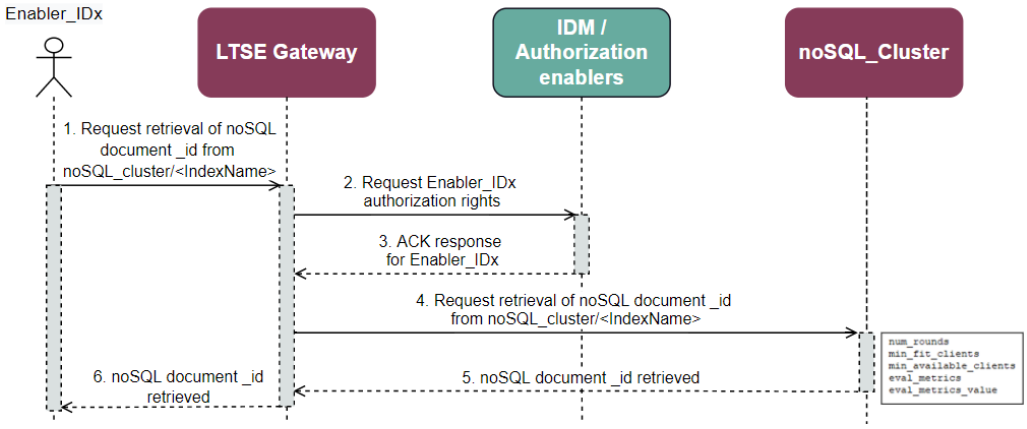


Figure 62. LTSE UC2 (get NoSQL data)

The **third use case** is related to **the storage of SQL data** of an authorized Enabler **on a SQL server**, after provisioning the required database and table. The diagram and the related steps are the following:

STEP 1: The Enabler_IDx interacts via LTSE Gateway with the LTSE, requesting to create a SQL storage.

STEPS 2-3: The LTSE Gateway checks the authorization rights of the Enabler_IDx from IdM/Authorization enablers, which confirms or denies Enabler_IDx access to the LTSE SQL server.

STEP 4: If granted, LTSE Gateway requests the generation of Enabler_IDx database into LTSE SQL_Server.

STEPS 5-6: LTSE SQL_Server confirms to the LTSE Gateway the generation of :databaseName SQL database, which, in turn, forwards the index database details to the Enabler_IDx.

STEPS 7-8: Then, Enabler_IDx requests to the LTSE Gateway the generation of a table into LTSE SQL_Server. The LTSE Gateway forwards this request to the SQL server.

STEPS 9-10: LTSE SQL_Server confirms to the LTSE Gateway the generation of :tableName SQL table, which, in turn, forwards the table details to the Enabler_IDx.

STEP 11-12: The Enabler_IDx requests ingestion of SQL data to LTSE Gateway, which forwards this petition to the SQL_Server (within the table of the database provisioned).

STEPS 13-14: LTSE SQL_Server confirms the ingestion of SQL data into the :databaseName SQL database, and :tableName SQL table of the LTSE SQL_Server and informs to LTSE gateway, which, in turn, forwards the details to the Enabler_IDx.

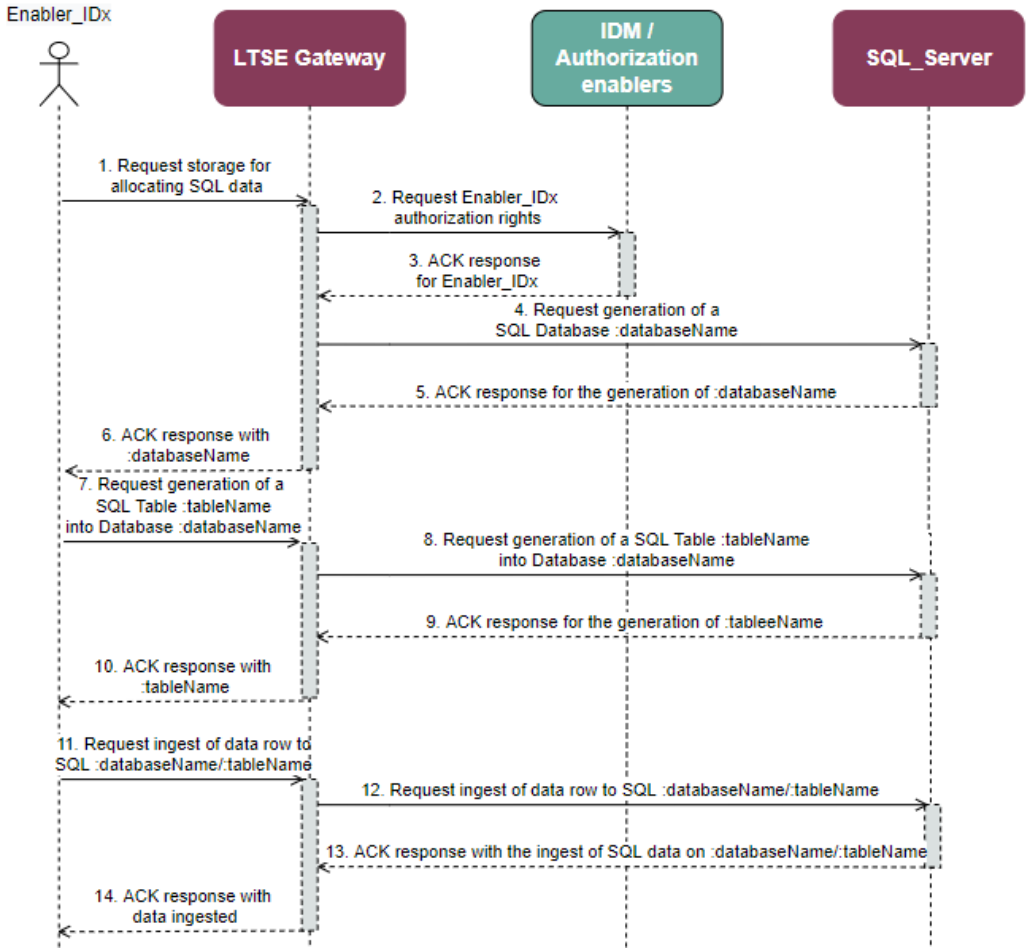


Figure 63. LTSE UC3 (store SQL data)

Finally, the **last use case** is related to **the retrieval of SQL data table** from a specific SQL database of the SQL server. The diagram and the involved steps are the following:

STEP 1: The Enabler_IDx interacts via LTSE gateway with the LTSE, requesting specific data allocated into its noSQL storage Index.

STEPS 2-3: The LTSE Gateway checks the authorization rights of the Enabler_IDx from IdM/Authorization enablers, which confirms or denies Enabler_IDx access to the LTSE server.

STEPS 4-6: If granted, LTSE Gateway requests the associated information demanded into Enabler_IDx :tableName of :databaseName of LTSE SQL_Server, which, in turn, forwards the table to the Enabler_IDx.

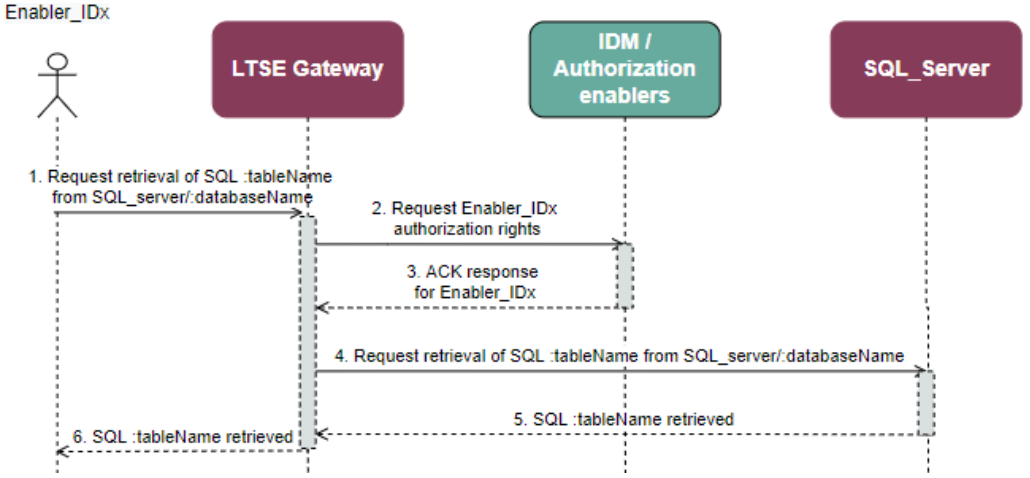


Figure 64. LTSE UC4 (get SQL data)

4.2.5.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/long_term_data_storage_enabler.html

Table 43. Implementation status of the Long-term Storage enabler

Category	Status
Components implementation	The noSQL cluster, and SQL server components are in place. The creation of databases, tables and indexes, as well as the ingest of SQL data and NoSQL documents have been tested in the laboratory.
Feature implementation status	Although supported, it is recommended for the time being to not deploy the LTSE with High Availability, i.e., configure the helm charts with a single PostgreSQL database and a single Elasticsearch node. Other pending features are: • Implementing the remaining endpoints. • Integration with the cybersecurity enablers.
Encapsulation readiness	All the three components are encapsulated in charts, where the LTSE NoSQL and LTSE SQL components are subcharts, i.e., dependencies of the LTSE gateway.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.3. Application and Services enablers

4.3.1. Tactile Dashboard

4.3.1.1. Structure and functionalities

The Tactile Dashboard enabler has the capacity to represent data through meaningful combined visualizations in real time. It also provides (aggregates and homogenizes) all the User Interfaces (UIs) for the configuration of the different ASSIST-IoT enablers, and associated components. It should be noticed that the tactile dashboard is a general GUI generation framework (based on partner PRO's own PUI9 framework).

The PUI9 framework is based on the VueJS framework. It allows the creation of fully reusable web components that can be used to create web pages (SPA) or complex web APPs. In addition, new applications using the tactile dashboard framework have a basic layout with a login screen and a fully configurable menu. The main advantages of the tactile dashboard framework are: modern, responsive and in some cases adaptive design; very good performance; based on web components, responsive; responsive components; each component has its own HTML template, internal JavaScript code, styles, and translations; gentle learning curve, being very easy and quick to start being productive.

Hence, each pilot will implement its own tactile dashboard according to its requirements, but all of them will be based on this framework, which will have in common that it is divided into three main components: Frontend, Backend, and PUI9 database. The following figure sketches the architectural diagram of tactile dashboard components.

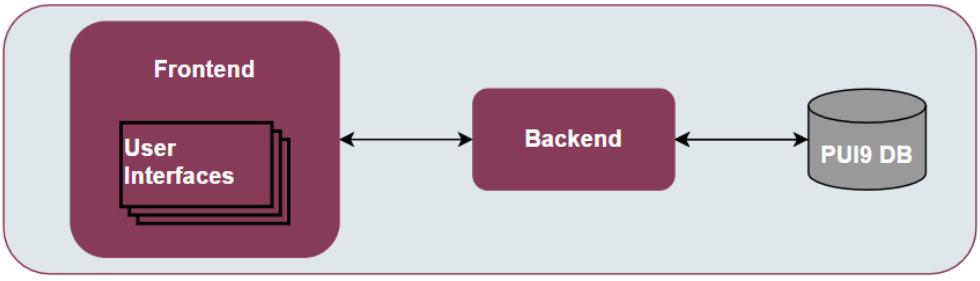


Figure 65. High-level diagram of the Tactile Dashboard

- **Tactile frontend:** The tactile frontend is what the ASSIST-IoT user interacts with. Therefore, it is responsible for most of what a user actually sees, including the definition of the structure of the web page, the look and feel of the web page, and the implementation of mechanisms for responding to user interactions (clicking buttons, entering text, etc.)
- **Tactile backend:** An HTTP server that listens to the requests coming from the tactile frontend in a specific port number, which is always associated with the IP address of the hosting computer. Thus, the tactile backend waits for tactile frontend requests coming to that specific port, performs any actions stated by the request, and sends any requested data via an HTTP response.
- **PUI9 database:** It is the place to store the tactile embedded information so that it can easily be accessed, managed, and updated. It might store information about ASSIST-IoT pilot's users, sensors' data, list of daily instructions, or reports. When a user requests some data to the tactile dashboard frontend webpage, the data inserted into that page comes from the PUI9 database.

Implementation technologies

Table 44. Implementation technologies for the Tactile Dashboard

Technology	Justification	Component(s)
VueJS	Basic development framework.	Frontend
Vuetify	Adds a set of base web components on which PUI9 components are built.	Frontend
Datatables	Allows the display of tabulated information.	Frontend
Axios	To make AJAX requests.	Frontend
NPM	Tool for dependency management.	Frontend
Webpack	Tool for packaging the application.	Frontend
Babel	Tool for transpiling modern JavaScript code.	Frontend
Eslint	Code format validator Webpack loaders.	Frontend
Java 8	Core Java platform, able to improve efficiency in developing and running Java programs	Backend
Spring	Provides flexible Java libraries	Backend
SQL	Tactile dashboard is compatible with several SQL databases (PostgreSQL, Oracle, SQL Server)	Backend

4.3.1.2. Communication interfaces

Table 45. Communication interfaces (API) of the Tactile Dashboard

Method	Endpoint	Description
POST	**/login/signin**	Login
POST	**/model_id/list**	Provides the list of model_id
GET	**/model_id/get**	Provides the model_id
PUT	**/model_id/update**	Updates the model_id
POST	**/model_id/insert**	Creates a new registry into model_id
DELETE	**/model_id/delete**	Deletes a registry into model_id

4.3.1.3. Use cases

Three use cases are envisioned for this enabler. They refer to the user login page, to the data forms listing, and to the access to external enablers APIs. The **first use case** will be instantiated by a user once it opens a web browser and types in the address bar the corresponding IP address/DNS of the instantiated tactile dashboard. Automatically, the Tactile dashboard will prompt the **login webpage** over which the user should introduce his/her credentials, which will be further evaluated in the tactile dashboard backend by querying this information to the embedded database of the application.

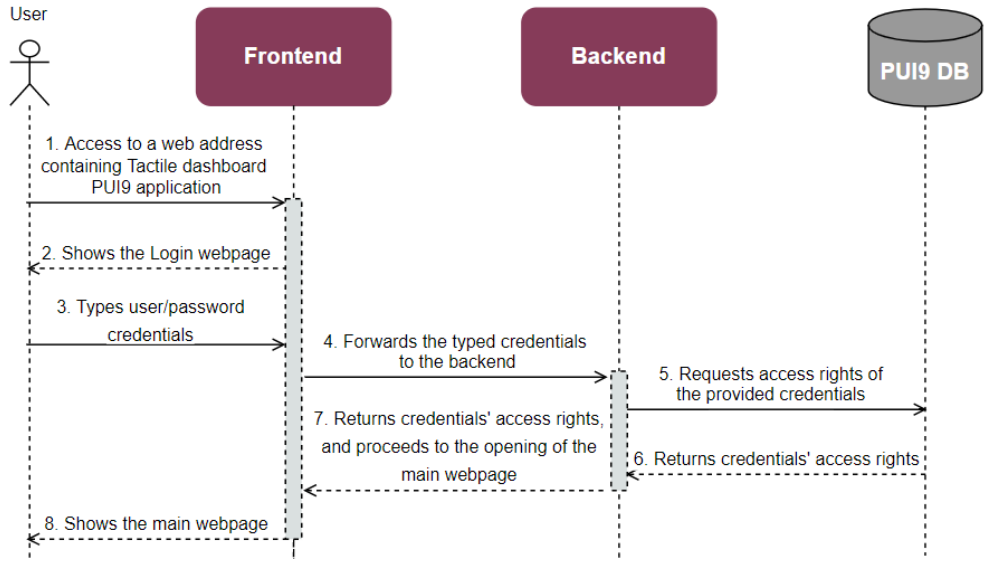


Figure 66. Tactile Dashboard UC1 (login webpage)

STEPS 1-2: The user opens a web browser and navigates to the web address containing the PUI9 application, and then the tactile dashboard frontend prompts the login webpage, demanding users' credentials.

STEPS 3-4: The user types his/her credentials and click on the login/submit frontend button, which forwards the details to the tactile backend.

STEPS 5-6: The backend communicates with the PUI9 database to collect the user's access rights[1] and checks if the user has rights to access the application.

STEPS 7-8: If the user's credentials are approved, the backend requests to the fronted to prompt the main menu webpage of the application to the user.

The **second use case** will be instantiated also by a user once it has been logged in accordingly. The use case is about **listing a specific data** requested by the user in the corresponding **menu of the application**. The diagram and involved steps are summarised below:

STEP 1: The user opens a web form page, and request listing a specific queried data.

STEPS 2-3: The frontend gathers the query, and forwards the details to the tactile dashboard backend, which, in turn, demands to the PUI9 database (either PostgreSQL, Oracle, or SQL Server) the user's requested data.

STEPS 4-5: The PUI9 database receives the backend query, compiles the requested data from the user, and provide the details back to the backend, which in turn, provides it to the fronted.

STEP 6: The tactile dashboard frontend prints in the specific web page form, the user's queried data.

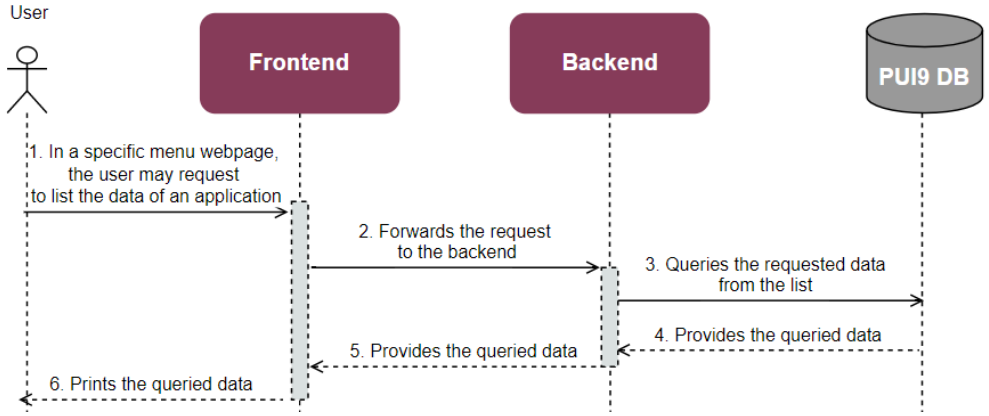


Figure 67. Tactile Dashboard UC2 (show data managed by PUI9 database)

[1] The users' access profiles can be stored within the enabler database or taken from the external, more advanced IdM and authorization enablers databases, accessible by means of API commands from the tactile dashboard backend (see use case 3 of the tactile dashboard).

The **third use case** may (or may not) be instantiated by the user, when he/she **demands additional information which is not collected in the PUI9 database** (e.g., data stored in the LTSE or EDB), or additional graphical functionalities not supported by the tactile dashboard (e.g., charts generation from the Business KPI enabler), but as highlighted in the examples, by other ASSIST-IoT enablers. Therefore, instead of the logical tactile dashboard workflow (frontend – backend – PUI9 database), the backend directly communicates with the API of the associated enabler.

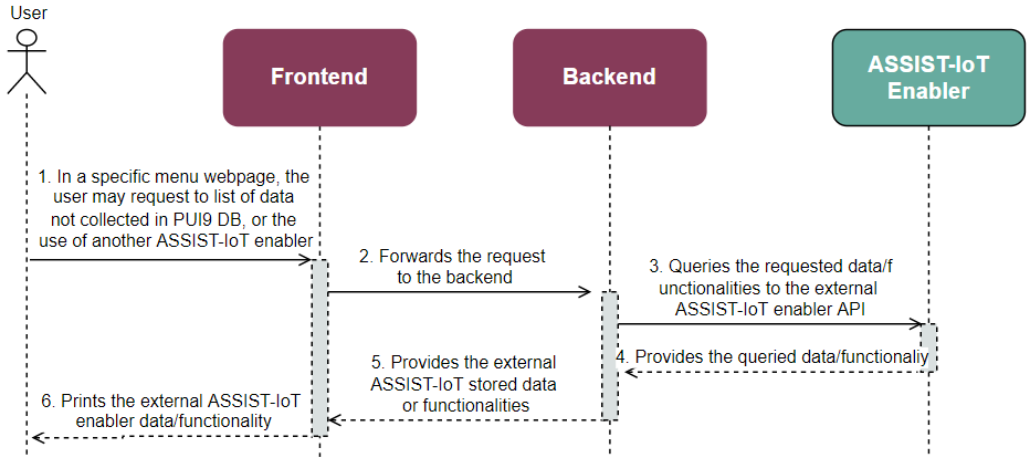


Figure 68. Tactile Dashboard UC3 (show data not managed by PUI9 database)

STEP 1: The user opens a web form page, and request listing a specific queried data/functionality not stored/supported by the tactile dashboard.

STEPS 2-3: The frontend forwards the details to the tactile dashboard backend, which, in turn, communicates with the external ASSIST-IoT enabler API.

STEP 4: The external ASSIST-IoT enabler proceeds internally with the request based on the API command from the tactile dashboard backend, and provide the requested data/functionality.

STEPS 5-6: The tactile dashboard backend receives the external ASSIST-IoT enabler response, and forwards the information to the frontend, which, finally, prints the user's demanded data/graphical functionality.

4.3.1.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/tactile_dashboard_enabler.html

Table 46. Implementation status of the Tactile Dashboard

Category	Status
Components implementation	The tactile dashboard framework is fully operational. However, until now, only one webpage and the manageability enabler webpage have been generated. The tactile dashboards of the pilots and any pages need for configuring/managing other enablers are still missing.
Feature implementation status	As each ASSIST-IoT pilot (and potentially many enablers) will demand a particular tactile dashboard/page to be configured, as well as a set of enablers to be consumed, the required HTTP methods to be supported in order to allow the communication between the tactile dashboard backend and the rest of deployed enablers is still pending to define and implement.
Encapsulation readiness	The example tactile dashboard, as well as the manageability dashboard are already containerized, and a Helm chart has been created for them.
Deployed with the Orchestrator in a laboratory environment	True, some integration tests between the manageability dashboard and the orchestrator have been carried out in UPV laboratories.

4.3.2. Business KPI Reporting enabler

4.3.2.1. Structure and functionalities

All valuable logs, time-series analytics and Key Performance Indicators (KPIs) desired by the end-user should be available for representation in graphs, charts, pies, etc. The Business KPI enabler will allow to embed them as User Interfaces (UIs) within the tactile dashboard. It will facilitate the visualization and combination of charts, tables, and other visualization graphs in order to search for hidden insights. The enabler is composed of (i) a server component containing the business logic engine, accompanied with (ii) a UI component that defines the graphical UI that users interact with, and (iii) a Command Line Interface (CLI) tool especially designed for developers. Figure 69 presents the architectural diagram of the Business KPI reporting enabler and its internal components:

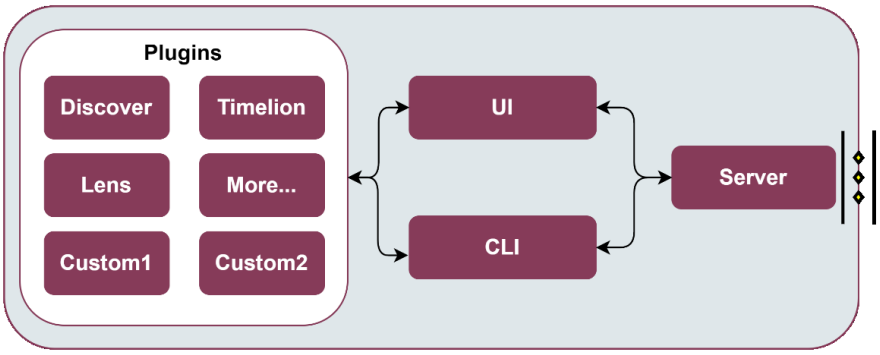


Figure 69. High-level diagram of the Business KPI Reporting enabler

- **Business KPI Server:** Collects data from data collectors (e.g., tactile dashboard PUI9 database, LTSE, or EDB enablers) into a dedicated database and provides access to it to the UI and CLI components via an internal REST API.
- **Plugins:** Business KPI functionalities are implemented through modular plugins (Discover, Tag, Lens, Maps, etc.), which contain the business logic and communicate with the UI and CLI components, based on the data collected in the Business KPI server. Furthermore, if willing to, custom plugins can also be easily integrated if needed, thanks to having a modular approach.
- **Business KPI UI:** Whenever the end-user accesses the Business KPI enabler via the Tactile Dashboard webpage, the UI component loads all server plugins that comprise the core functionalities of the Business KPI enabler. Hence, the UI component provides an editor to create and explore interactive visualizations and a set of functionalities to arrange the visualizations according to ASSIST-IoT end-user goals.
- **Business KPI CLI:** The CLI component enables custom plugins built by 3rd party developers to interact with the Business KPI Server, so that it is reachable from the UI to e.g., provide new data aggregation methods, or to visualize new chart types, colour palettes, etc.

Implementation Technologies

Table 47. Implementation technologies for the Business KPI Reporting enabler

Technology	Justification	Component(s)
Kibana	Open source program specialized in providing data visualization in various convenient formats (histograms, line graphs, pie charts, sunbursts, geospatial map displays, and other common visualization options)	Business KPI server, Plugins, Business UI, Business CLI

4.3.2.2. Communication interfaces

All charts and graphs in the Business KPI enabler are stored as saved-objects (basically a JSON-object that describes which visualizations are included). Therefore, the API methods are not those which allow generating the graphs but are, however, managed with Graphical User Interfaces that connect with a specific database. Nevertheless, the business KPI enabler is formed by spaces and data views, which allow to customize the webpage layout for visualizations.

Table 48. Communication interfaces (API) of the Business KPI Reporting enabler

Method	Endpoint	Description
POST	**/api/spaces/<space_name>**	Create a Business KPI space_name
GET	**/api/spaces/<space_name>**	Retrieve a Business KPI space_name
DELETE	**/api/spaces/<space_name>**	Delete a Business KPI space_name
POST	**/api/data_views/data_view**	Create a data view with a custom title (JSON file)
POST	**/api/saved_objects/data-view/my-view**	Update <my-view> data view (JSON file)
GET	**/api/data_views/data_view/my-view**	Retrieve the data view <my-view>
DELETE	**/api/data_views/data_view/my-view**	Delete a data view <my-view>

4.3.2.3. Use cases

There is a **single use case** that applies to this enabler. It is related to the **generation of graphs from time-series data** stored in the LTSE of ASSIST-IoT deployments. Its diagram and the involved steps are the following:

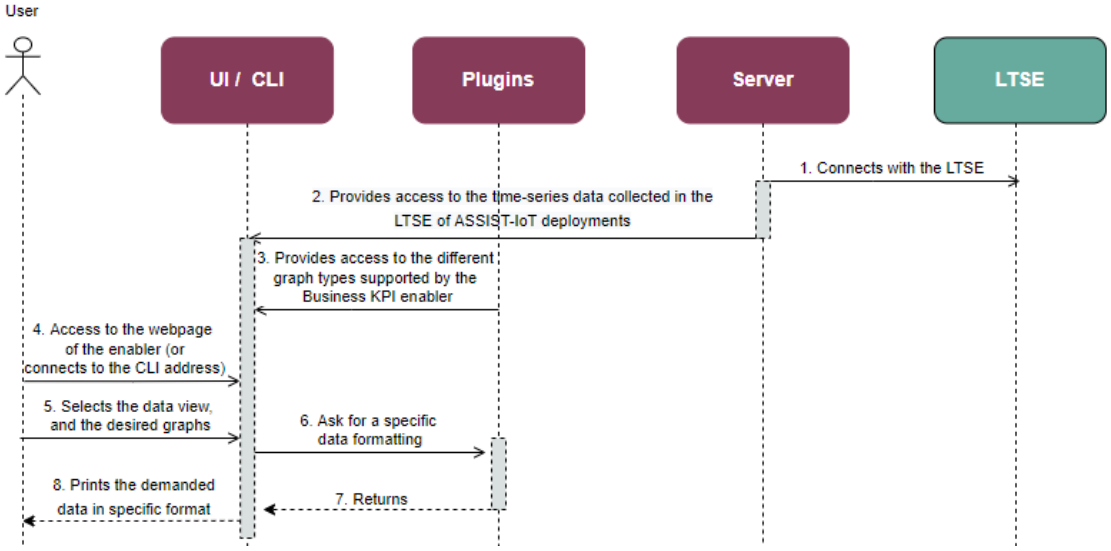


Figure 70. Business KPI Reporting enabler UC (generate graphs from time-series data)

STEP 1: The Business KPI server connects with the LTSE in order to have access to the time-series data produced in the ASSIST-IoT deployment.

STEPS 2-3: The Business KPI server and the Plugins provide access to the time-series data from LTSE, and the different graph types supported by the enabler to the UI/CLI, respectively.

STEPS 4-5: The user accesses to the webpage/menu of the tactile dashboard that allocates the business KPI enabler GUI (or connects to the CLI terminal), and selects visualising data in a specific format.

STEPS 6-8: Thanks to the plugins, the user can observe the data in the demanded format.

4.3.2.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/business_kpi_reporting_enabler.html

Table 49. Implementation status of the KPI Reporting enabler

Category	Status
Components implementation	All the components are already implemented.
Feature implementation status	The integration of the Business KPI enabler with the tactile dashboard has been postponed for the second term of the project.

Category	Status
Encapsulation readiness	The enabler is already encapsulated, and a Helm chart has been provided. However, it has not been properly deployed and tested in a K8s cluster yet.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.3.3. Performance and Usage Diagnosis enabler

4.3.3.1. Structure and functionalities

Performance and Usage Diagnosis (PUD) enabler collects performance metrics from monitored targets by scraping metrics to HTTP endpoints and highlighting potential problems in the ASSIST-IoT platform. This allows to autonomously act in accordance (if enabled) or to notify the platform administrator for fine-tuning the associated machine resources.

The PUD enabler provides an end-to-end approach to the infrastructure and application monitoring, covering all levels with easy instrumentation. The enabler collects performance metrics aiming to maintain operational simplicity while being able to adapt to a variety of scales/levels of the ASSIST-IoT infrastructure. By scraping metrics from endpoints with an HTTP pull model, PUD enabler will stay synchronised with the ASSIST-IoT infrastructure under monitoring. Its structure remains unchanged from its initial design.

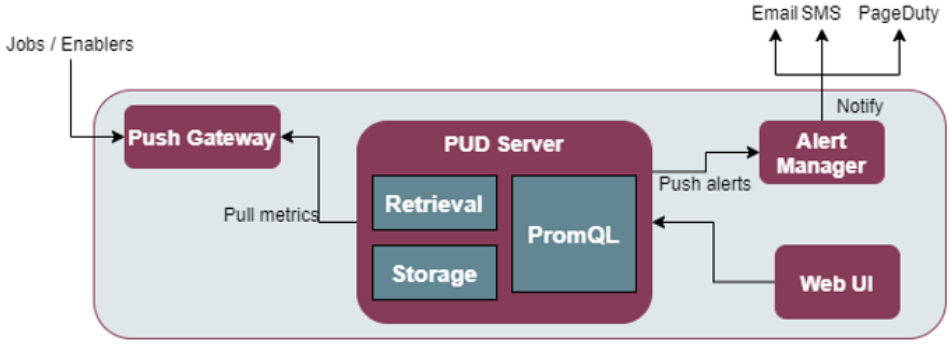


Figure 71. High-level diagram of the Performance and Usage Diagnosis enabler

Implementation Technologies

Table 50. Implementation technologies for the Performance and Usage Diagnosis enabler

Technology	Justification	Component(s)
Prometheus Server	Retrieval, Storage and Querying Metrics	PUD Server
Prometheus Alert Manager	Handles alerts sent by PUD Server	Alert Manager
Prometheus PushGateway	Push time series from short-lived batch jobs to an intermediary job that PUD Server can scrape	Push Gateway
Kube State Metrics	Service that listens to the Kubernetes API server and generate metrics about the state of the objects	-

4.3.3.2. Communication interfaces

Table 51. Communication interfaces (API) of the Performance and Usage Diagnosis enabler

Method	Endpoint	Description
HTTP Pull	**/metrics**	PUD follows an HTTP pull model: It scrapes Prometheus metrics from endpoints routinely. The abstraction layer between the application and PUD is an exporter, which takes application-formatted metrics and converts them to Prometheus metrics.
HTTP Push	**/write**	Remote write feature of PUD allow transparently sending samples and its primarily intended for long term storage.
HTTP Push	**/metrics/job/ some_job**	PUD PushGateway allows ephemeral and batch jobs to expose their metrics. Since these kinds of jobs may not exist long enough to be scraped, they can instead push their metrics to a PushGateway that then exposes those metrics to PUD.

4.3.3.3. Use cases

The **first use case** of PUD enabler is the **monitoring of the whole Kubernetes cluster** that it lives in. This can be achieved with kube-state-metrics, a project under the Kubernetes organization, which generates Prometheus format metrics based on the current state of the Kubernetes native resources. This is done by connecting to the Kubernetes API and gathering information about resources and objects, e.g., Deployments, Pods, Services, and StatefulSets.

The **second use case** of the PUD is the **monitoring the enablers of the ASSIST-IoT architecture** by scraping metrics provided from each enabler's exporter routinely and storing them in Prometheus's server. Both use cases follow the same sequence diagram and steps, summarised as follows:

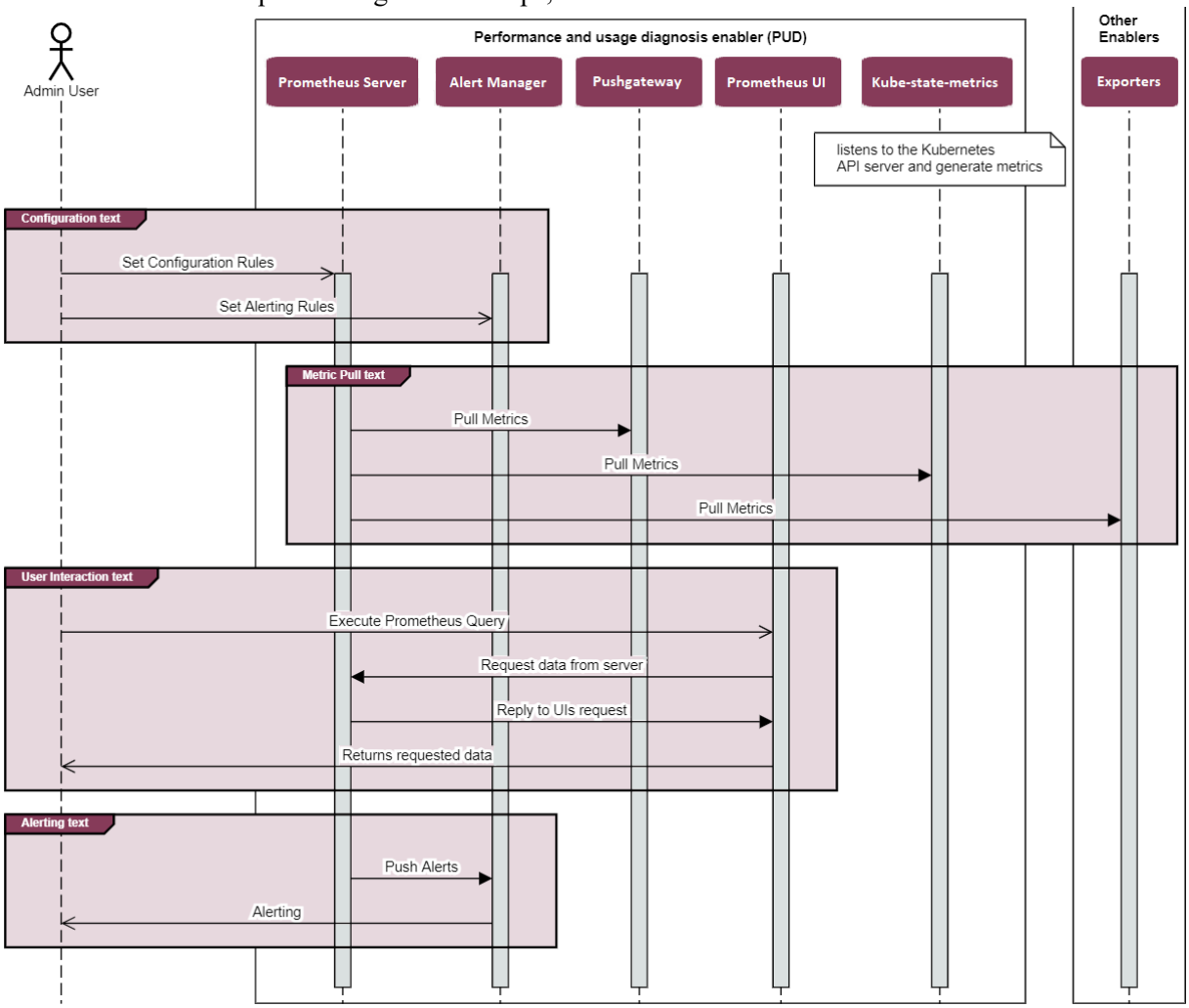


Figure 72. PUD UC (monitoring cluster and enablers)

In both use cases, the system administrator can use PromQL, which allows the users to select and aggregate time-series data in real time. The results of an expression can be shown as a graph in Prometheus's user interface. PUD's alerting system can be used for sending out notifications about firing alerts to an external service, through Prometheus alert manager, whenever an alerting rule has been met.

STEP 1: Initialising PUD by configuring the main server and all of its components via command-line flags or a configuration file.

STEP 2: The PUD server scrapes metrics from endpoints routinely from exposed HTTP endpoints. Once an endpoint is available, it can start scraping numerical data, capture it as a time series, and store it in a local database.

STEP 3: The user can access PUD's User Interface which provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time-series data in real time. The result of an expression can either be shown as a graph, viewed as tabular, or consumed by external systems.

STEP 4: When an alert occurs, the PUD server sends it to an alert manager. The latter then manages those alerts, including silencing, inhibition, aggregation and sending out notifications via methods such as email, on-call notification systems, and chat platforms.

NOTE: Steps 2,3 and 4 may happen concurrently and in a different order.

4.3.3.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/performance_and_usage_diagnosis_enabler.html

Table 52. Implementation status of the Performance and Usage Diagnosis enabler

Category	Status
Components implementation	A first version of the components is already in place.
Feature implementation status	Currently, PUD enabler is configured to scrape Kube-state-metrics and Edge Data Broker endpoints, as well as to monitor Edge Data Broker and Kubernetes clusters. Next release features will include: • Configuring PUD enabler to scrape other enablers endpoints. • Implementing and configuring an adapter for the communication between PUD and LTSE (i.e., use LTSE as persistent storage of PUD metrics). • Integrating a new UI into the enabler (probably Grafana dashboard).
Encapsulation readiness	Docker images and Helm Chart created and tested.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.3.4. OpenAPI Management enabler

4.3.4.1. Structure and functionalities

The OpenAPI Management enabler will be an API manager that allows enablers to publish their APIs, to monitor the interfaces lifecycles, and make sure that needs of external third parties (including granted Open Call projects), as well as applications that are using the APIs, are being met. Hence, the OpenAPI manager will provide the tools to ensure successful API usage in the developers' environment, help end-users for business insight analytics, as well as help ASSIST-IoT admins to preserve platform's security and protection. To achieve this, all ASSIST-IoT enablers should document their API in a common API specification format, which in principle has been identified to be the Swagger-JSON format.

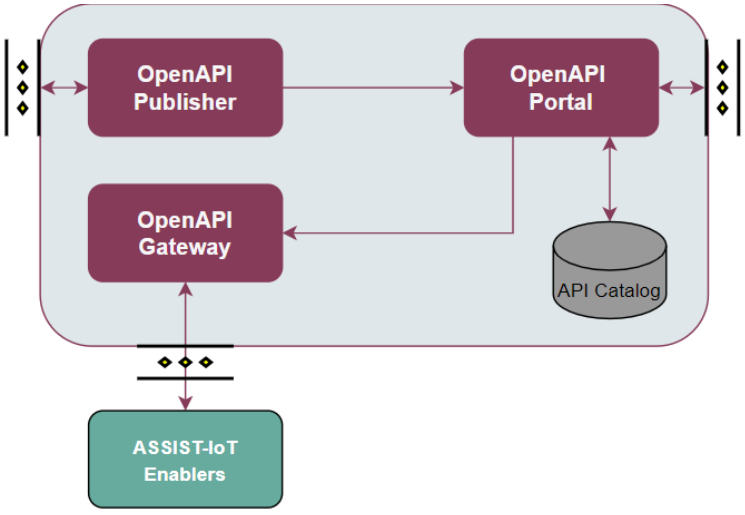


Figure 73. High-level diagram of the OpenAPI Management enabler

The OpenAPI management enabler has the following components:

- **OpenAPI Publisher:** A collection of tools that API providers use to define their APIs. These tools are also responsible to generate API documentation, manage access and usage policies, and can as well be used for testing-debugging purposes.
- **OpenAPI Portal:** A community site that allows users access to documentation, tutorials, software development kits etc. It can also allow to manage subscription keys and obtain support from the API provider if needed.
- **OpenAPI Gateway:** A server that acts as an API front-end, which receives API requests and passes them to the backend. Afterwards it returns the responses back to the requester. It can modify the requests and responses on the fly, and it can also provide the functionality to support authentication, authorization, security, audit and caching.
- **OpenAPI Catalog:** The OpenAPI management enabler has a registry/catalog embedded, to store, update and query the published JSON schemas of each enabler.

Implementation technologies

Table 53. Implementation technologies for the OpenAPI Management enabler

Technology	Justification	Component(s)
Swagger, Swagger UI	Swagger is a set of open source tools for writing REST-based APIs. It simplifies the process of writing APIs by notches, specifying the standards & providing the tools required to write beautiful, safe, performant & scalable APIs.	OpenAPI publisher, OpenAPI portal
KrakenD	KrakenD is an extensible, high performance API Gateway that aid with the adoption of microservices and secure communications. KrakenD is easy to operate and run and scales out without a single point of failure.	OpenAPI Gateway
MongoDB	Small NoSQL database selected for storing the published APIs	API Catalog

4.3.4.2. Communication interfaces

Table 54. Communication interfaces (API) of the OpenAPI Management enabler

Method	Endpoint	Description
GET/POST/PUT/DELETE	**/apis/{enabler_id}**	Get/add/modify/delete a new API design document for an enabler.
GET	**/apis**	Return all the API design document published.

4.3.4.3. Use cases

The **first use case** of OpenAPI management enabler is built around an external user who wants to **consult the API documentation of a specific enabler**. The following flow and steps describe the process:

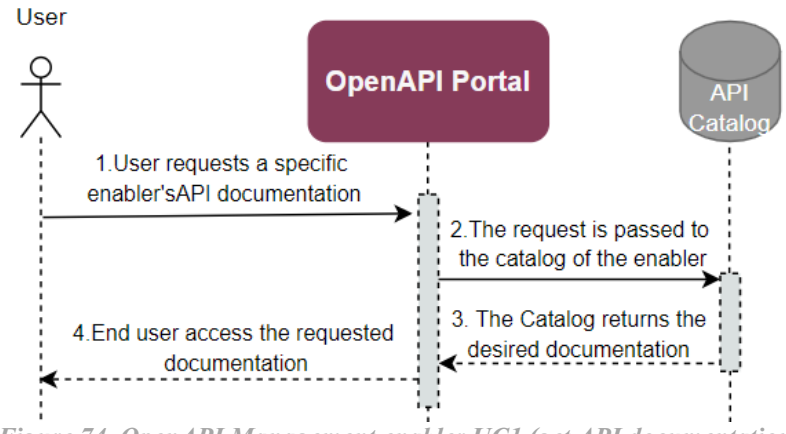


Figure 74. OpenAPI Management enabler UC1 (get API documentation)

STEP 1: An OpenAPI caller requests a specific enabler's API documentation by communicating with the OpenAPI portal.

STEP 2: The portal processes the request and communicates with its API catalog.

STEP 3: The catalog returns the desired documentation.

STEP 4: The enabler outputs the requested API documentation.

The **second use case** is about an ASSIST-IoT admin/developer who wants to **publish** a newly designed **API document**. The enabler process is described below:

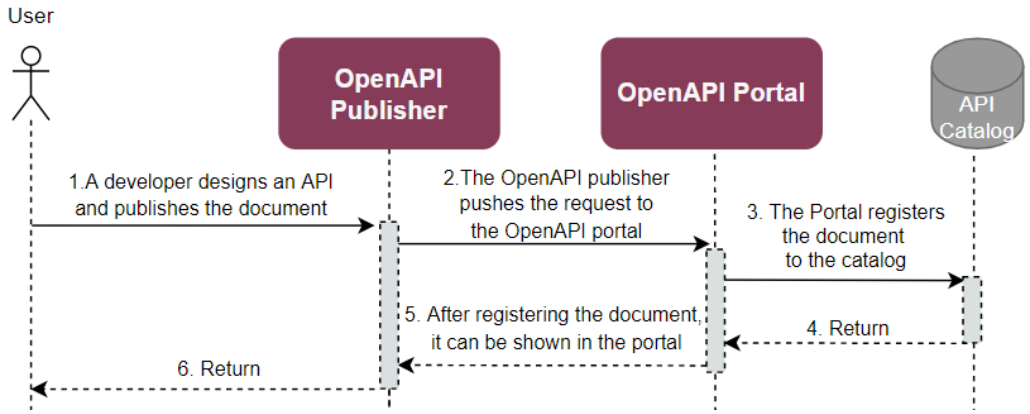


Figure 75. OpenAPI Management enabler UC2 (publish API document)

STEP 1: An ASSIST-IoT admin designs an API document and wants to publish it, starting a communication with the OpenAPI Publisher.

STEP 2: The request is then pushed from the OpenAPI Publisher to the OpenAPI Portal.

STEP 3: The Portal registers the document in the catalog.

STEPS 4-5: After registering the document, this can be shown in the portal.

STEP 6: Finally, the user receives an acknowledgement that the document has been published (or an error message, if an error has occurred).

The **third use** case involves an external entity who wants to **interact with an ASSIST-IoT enabler**. The figure and steps below describe the flow for being redirected to the correct enabler:

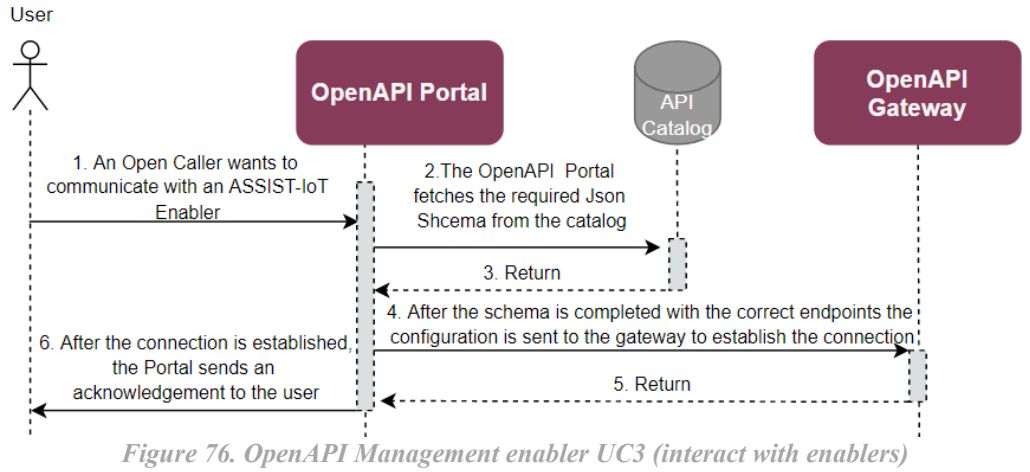


Figure 76. OpenAPI Management enabler UC3 (interact with enablers)

STEP 1: An user starts a connection with the OpenAPI portal to interact with an ASSIST-IoT enabler.

STEPS 2-3: The OpenAPI Portal gets the required JSON Schema from the catalog.

STEP 4: After the schema is completed with the correct endpoints, the configuration is sent to the gateway to establish the connection.

STEPS 5-6: Once the connection is established, the Portal sends an acknowledgement to the requester.

4.3.4.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/openapi_management_enabler.html

Table 55. Implementation status of the Open API Management enabler

Category	Status
Components implementation	A first version of all the components is in place.
Feature implementation status	The basic features of these components are working, but adaptations for the project and integration with the identity management are still missing.
Encapsulation readiness	Components are yet to be encapsulated.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.3.5. Video Augmentation enabler

4.3.5.1. Structure and functionalities

This enabler receives images or video captured either from ASSIST-IoT Edge nodes, or from ASSIST-IoT databases, and using Machine Learning Computer Vision functionalities, performs object detection/ recognition of particular end-user assets (e.g., cargo containers, cars' damages). It should be noticed that in order to carry out the proper object recognition in an operation, an appropriate annotated dataset should be ready and available for training and testing. Figure 77 presents the architectural diagram of the video augmentation enabler and internal components.

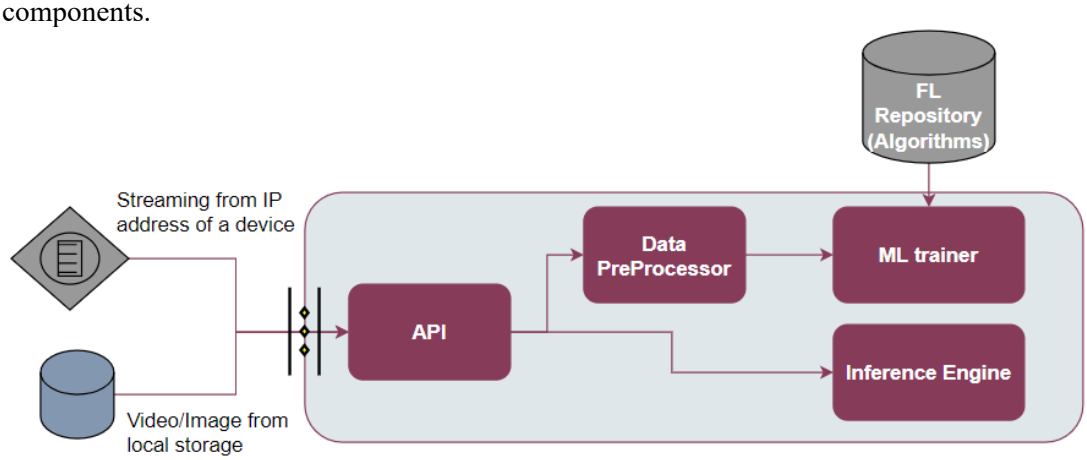


Figure 77. High-level diagram of the Video Augmentation enabler

- **API:** The entrance gate to the video augmentation enabler. It provides a set of restful API endpoints, over which the user can easily interact with the enabler to e.g., run an ML training process, run an ML inference, or get the status of the current training process.
- **Data Pre-processor:** Since the dataset can be collected from various sources such as Cameras or Databases, but it may not be used directly for performing ML analysis processes (e.g., the dataset contains unorganized or noisy data), a data pre-processing can be done. Data pre-processor provides tools for cleaning the raw data such as taking care of missing values, categorical features, and normalization.
- **ML trainer:** An ML model is a function with learnable parameters that maps an input to the desired output. The optimal parameters are obtained by training the model on data. ML Trainer will carry out the process of feeding the network with millions of training data points so that it systematically adjusts the knobs close to the correct values. Although the video augmentation ML trainer already supports some ML models, additional ML models can be retrieved from the FL Repository. Since the training process of images/videos may be computationally intensive, as the data can be passed through Neural Network with several training rounds, it is recommended to be performed on a GPU.

- **Inference engine:** The Inference engine provides the process of running a trained ML over a specific input through an interpreter. The interpreter, based on TensorFlow, is designed to be lean and fast, and uses a static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

Implementation Technologies

Table 56. Implementation technologies for the Video Augmentation enabler

Technology	Justification	Component(s)
Fast API	Widely used technology to developed REST APIs in an easy way	API
Tensorflow	This platform is used because of its well-documented and useful object detection API	Data PreProcessor, ML trainer
OpenCV	Widely-used library and tools for computer vision	Inference engine

4.3.5.2. Communication interfaces

Table 57. Communication interfaces (API) of the Video Augmentation enabler

Method	Endpoint	Description
POST	**/train/{model_id}**	Executes a training session over the annotated data in the Video Augmentation data folder with the ML model {model_id}.
GET	**/train_status**	Provides the status
POST	**/inference_local/{model_id}**	Performs inference or validate process over the stored data (video or image) with the trained model model_id.
POST	**/inference_streaming/{IP_address,model_id}**	Performs inference or validate process over the video being streamed at IP_address with the trained model model_id.

4.3.5.3. Use cases

The two main use cases of this enabler are related to the training and the inference process of a computer vision ML model over a local or streaming image/video set. The **first use case**, i.e., the **training process**, will be initiated by a user, once the labelled data is updated and allocated in the corresponding local folder.

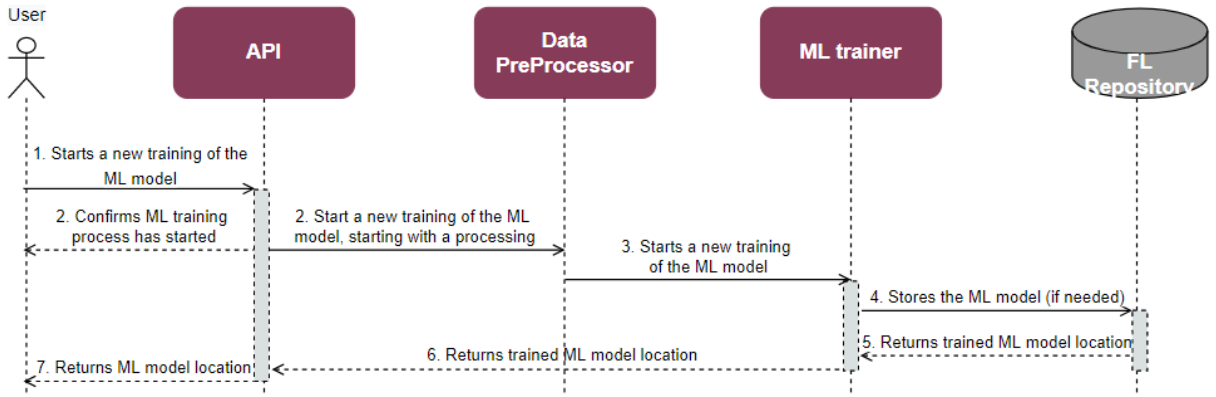


Figure 78. Video Augmentation enabler UC1 (model training)

STEP 1: The user starts a new training process via API command, once the properly annotated data is present in a folder accessible by the training module.

STEPS 2-3: The API communicates with the PreProcessor in order to start a new pre-processing (if required), and afterwards it forwards the processed data to the ML trainer to actually start the training of a new ML model.

STEPS 4-7: When the training process is finished, if expressed, the ML model is stored in the FL repository and notified to the user.

The **second use case** is related to the **inferencing of new video set** (either stored in local folder or received via an HTTP streaming service) with a trained ML model. In this case, the following steps and diagram apply:

STEP 1: The user starts an inference process via API command, making use of model trained previously by the dedicated module. The video format over which the Video Augmentation enabler will perform the inference (local or streaming) is also included in the body of the API endpoint.

STEP 2: The API informs to the Inference engine to start the new process.

STEPS 3-4: The Inference engine starts the process and sends the output video files to a video player user application (outside of the scope of Video Augmentation enabler).

STEP 5: The video player reproduces the inferenced filed in order to be visualised by the user.

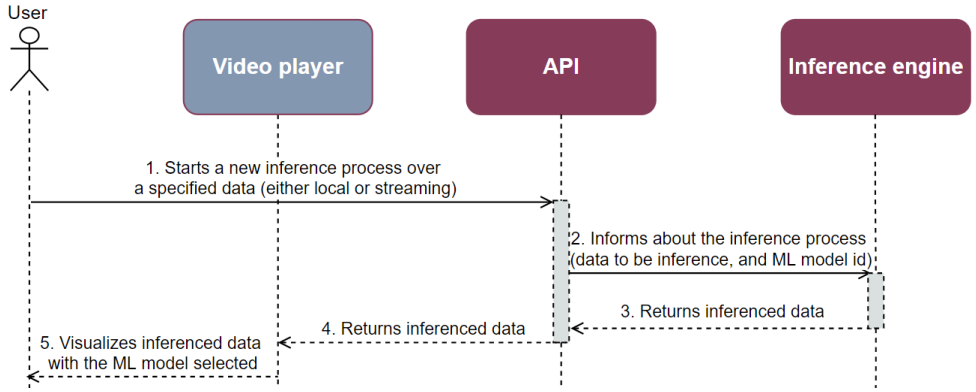


Figure 79. Video Augmentation enabler UC2 (video inference)

4.3.5.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/video_augmentation_enabler.html

Table 58. Implementation status of the Video Augmentation enabler

Category	Status
Components implementation	All the components are already implemented, except for the API, which is still at a 20% of its development.
Feature implementation status	The basic functionalities related to training and inferencing are ready, however, not instantiable from the API. The integration of the video augmentation enabler with the FL repository has been postponed for the second term of the project.
Encapsulation readiness	The video augmentation enabler is already encapsulated in the form of a Docker, but it has not been deployed over a Kubernetes cluster yet, and no helm chart has been generated.
Deployed with the Orchestrator in a laboratory environment	Not yet

4.3.6. MR enabler

4.3.6.1. Structure and functionalities

The novel interface that is used in the MR enabler offers a human-centric interaction through better cooperation of the end-users with the IoT environment. Through the MR enabler, the human effort and decisions are introduced in the loop of every critical action, whenever needed. The MR enabler aids human-friendly haptics and the end-user can receive and provide tactile, real-time and visual feedback as well as data capable of identifying critical improvements, preventions and triggers in long-, short-term, or real-time. Through reporting functions, the MR enabler gathers reliable data to extract information and perform analytics. Decision-making is improved as human flexibility, creativity and expertise, interact with IoT platforms and devices. The functionalities of the MR enabler are summarised as follow:

- Identifying assets (along with relevant data) at close proximity,
- Visualizing rendered (3D) models through the head-mounted MR devices, along with highlighted zones of the same model. The models and all related data come from the long-term storage,
- Receiving alert messages from real-time data streams and displaying them to the user, and
- Capturing and storing media files in order to include them in a report.

The present design has a few changes compared to D4.1, as it includes one additional component. This component manages the configuration of the MR enabler, directly communicating with the ASSIST-IoT platform, where the user can configure it in order to be functional with other components and systems in the IoT ecosystem.

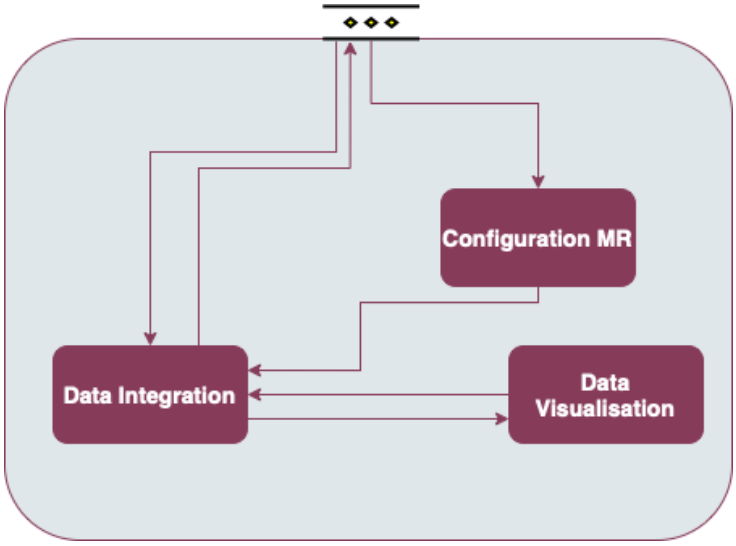


Figure 80. High-level diagram of the MR enabler

Implementation Technologies

Table 59. Implementation technologies for the MR enabler

Technology	Justification	Component(s)
Unity	Parametrisation of the MR enabler	Configuration MR
Unity	Retrieval, Storage and Querying Data	Data Integration
Unity	Information visualisation	Data Publication

4.3.6.2. Communication interfaces

The MR enabler will retrieve, gather and integrate all the necessary data via external REST API endpoints or other services provided. The following endpoints are used to function with the rest of the components:

Table 60. Communication interfaces of the MR enabler

Method	Endpoint	Description
REST API	Not defined yet	The MR enabler will send reports (data and image) to the LTSE.
MQTT	Not defined yet	MQTT messages will be send to MR enabler in order to visualise them.

4.3.6.3. Use cases

The MR enabler supports various use cases in the health and safety process, performed by the end-user (such as, **identification of assets** or elements at close proximity, **visualization of rendered (3D) models** and related content, real-time **alerting**, **notifications** and reporting of required actions). In the following graph, the **main sequence** diagram of the internal components of the MR enabler are presented, while the detailed steps are further described below.

STEP 1: The Configuration component of the MR enabler sends a GET request to the ASSIST-IoT Management Platform, in order to receive the up-to-date configurations that exist online.

STEPS 2-4: If the connection with the Platform is successful, the Management Platform responds with the proper configurations, forwarded to the internal components to apply them.

STEP 5: The data is integrated by the MR enabler and can either be sent to other enablers, or visualized to the Inspector via the MR-HMD (Head Mounted Device).

STEP 6: As soon as an end-user visualises the site data, he/she is able to take an informative action through the graphics interface of the MR enabler.

STEPS 7-8: The action performed by the end-user through the HMD will be integrated by the MR enabler and sent to other enablers, if needed.

STEP 9: A response will return to the graphics interface (MR HMD) of the end-user.

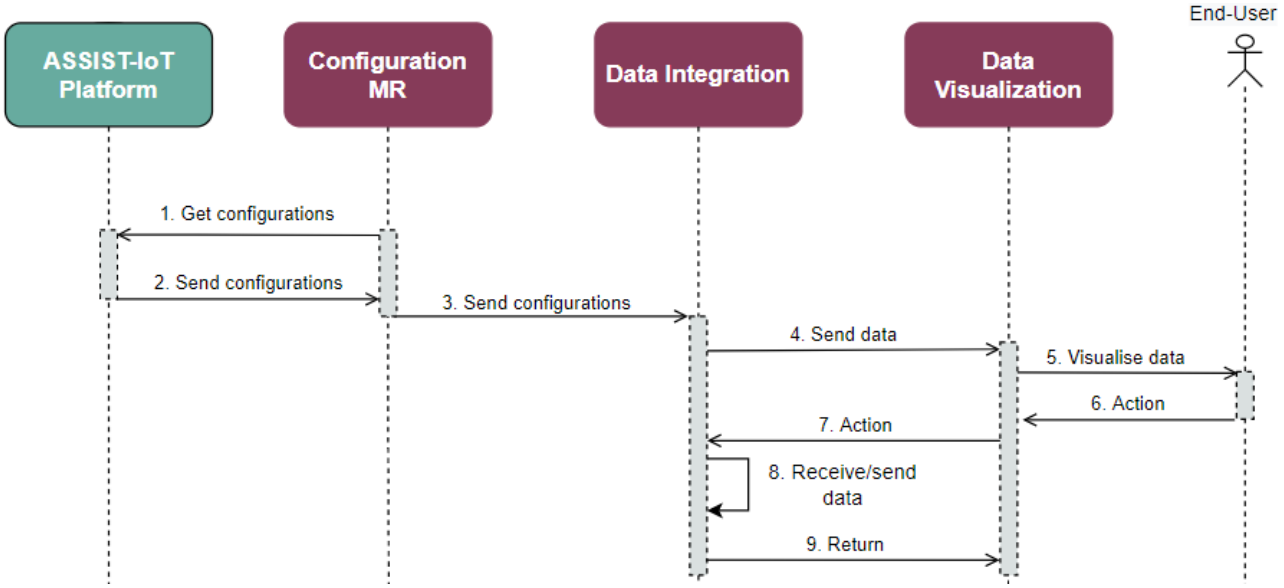


Figure 81. MR enabler UC (3D visualisation, asset identification, alerting and notification)

4.3.6.4. Implementation status

Link to Readthedocs (structure defined in WP6 documentation task):
https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/mr_enabler.html

Table 61. Implementation status of the MR enabler

Category	Status
Components implementation	A first version of the components is already in place.
Feature implementation status	The following features of the MR enabler are ready: (i) visualization and manipulation of the rendered model of an area with the highlighted areas, (ii) creation of reports using attached media, and (iii) integration with Edge Data Broker enabler via MQTT protocol. There are some pending features before closing the development of this enabler: • Integration with other enablers (such as, LTSE, Semantic, Location management system). • Identification of workers at close proximity (pending integration with other enablers to activate this feature)
Encapsulation readiness	This enabler cannot be encapsulated (see deliverable 3.6, Chapter 5.2 Encapsulation exceptions)
Deployed with the Orchestrator in a laboratory environment	This enabler cannot be encapsulated (see deliverable 3.6, Chapter 5.2 Encapsulation exceptions)

5. Future Work

This deliverable encompasses not only the information provided in this document, but also the first software results related to the horizontal planes of the architecture. These software outcomes are in different degree of development: some of them are containerised, others integrated with K8s (manifests ready) or already packaged (in Helm charts), whereas the implementation of a few of them has not started yet. It should be highlighted that the development of the enablers identified as essential has been prioritised for this first release, and hence a functional version is ready. The software material (source code) associated to those advances has been submitted attached to this document (in the form of a compressed file) extracted from the collaborative code repository of the project (GitLab).

In this document, an update and extension of the specifications provided in the first iteration of this deliverable. This includes an update of the functionalities, of the schematic of their internal components, and of the endpoints to be provisioned. Besides, new information is provided, including (i) the **technological selections**, not just candidate ones, (ii) a set of relevant **use case diagrams**, where the interaction from users (or from relevant enablers) with the enablers and its internal interactions are depicted, and (iii) a reporting table of the current development **status**. The enablers developed so far allow for starting and/or continuing efforts related to this and other work packages:

- To finish the development of the components of the enablers (WP4). It should be highlighted that the efforts devoted to SD-WAN related enablers were postponed to be initiated after the finalisation of this deliverable.
- To containerise, and/or generate the K8s manifests required to deploy them in those cases that have not virtualised the overall solution (WP4).
- To test the enablers in a common, staging environment (WP6), following a common testing and integration methodology.
- To perform the needed modifications for ensuring proper interactions with other enablers from WP4 & WP5.
- To package, publish and release the enablers as Helm charts (WP6).
- To start implementing them in pilots for further validation and assessment (WP7), either fully or partially packaged.
- To prepare/fine-tune them (if needed) for their forthcoming usage by the Open Call awarded projects.

In the next (and last) iteration of the deliverable, all the enablers will have a functional packaged version available. They will be accompanied by another document, in which all the modifications and deviations will be reported, as well as an update with the final enabler templates.