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

assist-iot

D4.3 Final Core Enablers Specification and Implementation

Deliverable No.	D4.3	**Due Date**	31-OCT-2023
Type	Report	**Dissemination Level**	Public
Version	1.0	**WP**	WP4
Description	Final specification and implementation status of Smart IoT Devices, GWEN and enablers of the horizontal planes of ASSIST-IoT.		

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 Fornes	P01 UPV	alforlea@upv.es
Francisco Mahedero	P01 UPV	framabio@upv.es
Rafael Vañó	P01 UPV	ravagar2@upv.es
Raul Reinosa	P01 UPV	rreisim@upv.es
Eduardo Garro	P02 PRO	egarro@prodevelop.es
Juan Antonio Pavón	P02 PRO	jpavon@prodevelop.es
Paweł Szmeja	P03 IBSPAN	pawel.szmeja@ibspan.waw.pl
Piotr Sowiński	P03 IBSPAN	piotr.sowinski@ibspan.waw.pl
Konstantinos Flevarakis	P04 CERTH	kostisfl@iti.gr
Evripidis Tzionas	P04 CERTH	tzionasev@iti.gr
Georgios Stavropoulos	P04 CERTH	stavrop@iti.gr
Konstantinos Fragkos	P06 INF	cfragkos@infolysis.gr
Nikolaos Vrionis	P06 INF	nvrionis@infolysis.gr
Johan Schabbink	P09 NEWAYS	Johan.Schabbink@newayselectronics.com
Nasia Balakera	P10 ICCS	nasia.balakera@iccs.gr
Fotios Konstantinidis	P10 ICCS	fotios.konstantinidis@iccs.gr
Tom Papaioannou	P10 ICCS	thomas.papaioannou@iccs.gr
Konstantinos Routsis	P10 ICCS	konstantinos.routsis@iccs.gr
Zbigniew Kopertowski	P15 OPL	Zbigniew.Kopertowski@orange.com
Jaroslaw.Legierski	P15 OPL	Jaroslaw.Legierski@orange.com

History

Date	Version	Change
07-Feb-2023	0.1	ToC presented
30-Sep-2023	0.2	First round of contributions
11-Oct-2023	0.3	Second round of contributions
27-Oct-2023	0.9	Integration of changes from IR
31-Oct-2023	1.0	Official release, final version submitted to EC

Key Data

Keywords	Enablers, GWEN, Smart IoT devices, Implementation
Lead Editor	P01 UPV – Alejandro Fornés
Internal Reviewer(s)	Rafael Borné (P13 - S21Sec), Katarzyna Wasielewska-Michniewska (P03 - IBSPAN)

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 M18-M36, 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 four planes of the ASSIST-IoT reference architecture are as follows: Device and edge plane, Smart network and control plane, Data management plane and Application and services plane. This work package is devoted to the design and implementation of the software and hardware artifacts needed to realise a system based on such reference architecture.

This deliverable reports the final outcomes of the work package, specifically, the realisation of the project hardware (GWEN and smart IoT devices) and the final specifications of the project enablers. For each one of the latter, the following information is provided: table of general data, high-level component diagrams, table of components considered and utilised realisation technologies, endpoints/interfaces, enabler stories, and table with implementation information. Jointly with these specifications, the enablers' code is attached and presented along with this report.

Table of contents

List of tables

List of figures

List of acronyms

Acronym	Explanation
AI	Artificial Intelligence
API	Application Programming Interface
BIM	Building Information Model
BLE	Bluetooth Low Energy
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNF	Cloud Native Network Function
CNI	Container Network Interface
CNN	Convolutional Neural Network
CRD	Custom Definition Resource
CSV	Comma-Separated Values
DB	Database
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EDB	Edge Data Broker
FL	Federated Learning
FR	Filtering & Ruling
GUI	Graphical User Interface
GWEN	Gateway and Edge Node
HAL	Hardware Abstraction Layer
HDMI	High-Definition Multimedia Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IdM	Identity Management
IoT	Internet of Things
IPSec	Internet Protocol Security
IPSM	Inter-Platform Semantic Mediator
JSON	JavaScript Object Notation
K8s	Kubernetes
KPI	Key Performance Indicator
LTSE	Long-Term Storage Enabler

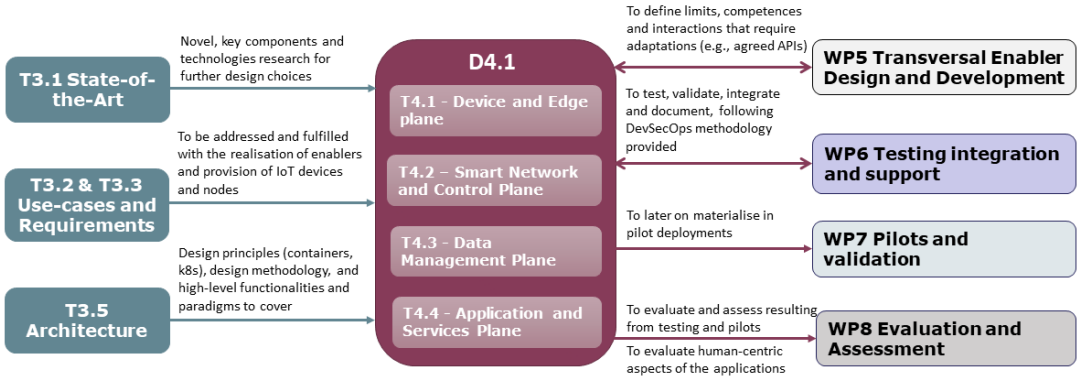
MQTT	Message Queuing Telemetry Transport
ML	Machine Learning
MR	Mixed Reality
NB	NorthBound
NGIoT	Next Generation IoT
NoSQL	Not only SQL
OEM	Original Equipment Manufacturer
OS	Operating System
OSS	Operational Support System
PUD	Performance and Usage Diagnosis
PUI9	Prodevelop's User Interface
QoS	Quality of Service
RDF	Resource Description Framework
REST	REpresentational State Transfer
RML	RDF Mapping Language
RTG	Rubber-Tyred Gantry
RTT	Round-Trip Time
SB	SouthBound
SDN	Software Defined Network
SD-WAN	Software Defined WAN
SIM	Subscriber Identity Module
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator
VNF	Virtualized Network Function
VPN	Virtualized Private Network
WAN	Wide Area Network
XML	eXtensible Markup Language

1. About this document

Two are the main objectives of this deliverable: (i) to **finalise** the **specifications** of the horizontal **enablers** designed, and (ii) to **provide the final functional** of the enablers developed. These enablers are the cornerstone of the project, allowing the design and realisation of Next Generation IoT (NGIoT) systems based on the ASSIST-IoT architecture. With them, the pilots of the project will be implemented and the outcomes, evaluated. Apart from software enablers, D4.3 also includes the final specifications and pictures of the ASSIST-IoT's Gateway/Edge Node (GWEN) and the Smart IoT devices, which have been developed specifically for the project.

This deliverable corresponds to the final document of a series of three iterations. The iterative nature of this report was firstly due to the fact that the requirements and the reference architecture were evolving in WP3 and a result of the interactions with WP5. Also, an agile evolution is needed and desired considering the feedback from the integration activities of WP6, and the use of the artifacts in pilots, under WP7.

1.1. Deliverable context

Keywords	Lead Editor
Objectives	**O2:** D4.3 presents the final specifications of the enablers of the network plane and is presented jointly with their final version. **O3:** Specifications of enablers focused on data (semantics, broker, storage) are provided, as well as their final version. **O5:** Human-centric interfaces for the use cases are presented.
Work plan	
Milestones	This deliverable is directly related with MS7 – Integrated solution, as these enablers are, jointly with WP5's, the main artifacts contributing to the final, integrated solution. In any case, integration efforts are carried out under the scope of WP6, under test in the pilots (WP7).
Deliverables	This deliverable is the natural evolution of D4.2, and receives inputs from D3.3 (requirements and use cases – second iteration) and D3.7 (architecture definition – final iteration). Enablers stemming from this WP feed WP6 for testing, integration, distribution and documentation, being the cornerstone of pilots' implementations of WP7, and key part in the technical evaluation to be performed under the scope of WP8.

1.2. Outcomes of the deliverable

This document reports the **final specifications** of the hardware and software artifacts implemented during the execution of WP4, updating the information provided in D4.1 and D4.2. Along with this document, the code of the enablers, which jointly with the GWEN are the main outcomes of the WP4, will be published in public repositories (and upload to EC portal, as a compressed file), so they are openly available.

1.3. Lessons learnt

During the implementation phase, designs required to adapt to the actual needs of NGIoT systems, represented by the project pilots. These are some of the lessons learnt during the last phase of WP4 activities:

- Although the GWEN was initially designed to act as a Kubernetes master, its relative low RAM made it more suitable to act as worker of another Kubernetes master. In future designs, additional RAM will be needed to have the ability of working as master.
- Persistency in Kubernetes can be problematic in K8s-based environments if not managed properly. Enablers had to be tested in different cases, e.g., in case of pod deletion/restart, power down, and manual chart uninstall.
- All the lessons learnt stated in Section 1.4 of D4.2 apply.

1.4. Deviation and corrective actions

The Consortium dedicated effort to formalise (in D4.1) and materialise (in D4.2) the artifacts of the WP4, however, there are some deviations that have slightly altered the initial plan:

- Cilium was selected as the main K8s CNI plugin for the clusters. However, its low maturity in Yocto systems require adapting the Smart orchestrator to work also with flannel as an alternative for Yocto-based nodes.
- Some enablers have suffered significant design modifications, like the multi-link and the traffic classification enabler. Additional effort had to be devoted to complete them in time.
- Initial tests of the smart orchestrator shown slow installation times, execution errors and incompatibilities. Because of this, although the design did not change significantly, the underlying technologies had to be changed and its code refactored to be usable.
- All the actions stated in Section 1.5 of D4.2 apply.

1.5. Version-specific notes

The following notes apply to this version:

- The final hardware artifacts of the project are now reported, including pictures.
- The specifications of the enablers have been updated, showing the final component diagrams, endpoints and enabler stories, thus making D4.2 obsolete.
- D4.3 can be considered a self-contained document, however, some data has not been included in this report and can be checked at D4.2, such as the rationale for technologies selection to avoid excessive document length. This information has not been deemed important as many enablers components could have been realised considering alternative technologies or programming languages.

2. Introduction

The ASSIST-IoT architecture is based on a multidimensional approach, considering planes and verticals[1]. Planes are collections of functions that can be logically layered on top of one another. Data gathered and potentially pre-processed in the *Device and edge layer*, can be optimally transported in a network managed by the *Smart network and control plane*, processed (routed, stored, transformed) in the *Data management plane*, and consumed by the *Applications and services plane*. Verticals, in turn, represent inherent properties of the system or cross-cutting to the rest of the architecture, as well as functions targeting specific NGIoT properties.

Enablers are the cornerstone of the ASSIST-IoT reference architecture. They offer specific features belonging to the aforementioned planes and verticals. As each business scenario has its own particularities, not all the enablers are needed in all architecture realisations, although some of them are considered essential, i.e., should be part of any system based on ASSIST-IoT.

This deliverable reports the **specifications of the enablers designed and implemented for the four planes of the architecture**. All of them (but the exceptions, like the MR enabler) are packaged and follow the encapsulation principles of the project: they are designed as a set of microservices or micro-applications, realised as containers, only accessible via exposed interfaces, and packaged as Helm charts following the specifications of the project – aligned with current trends. All WP4 enablers have been finalised, and have been published in public repositories (except OpenAPI enablers because of internal policy of the institution in charge, see D6.8). In any case, as all software products, they can be enhanced and evolve in the future.

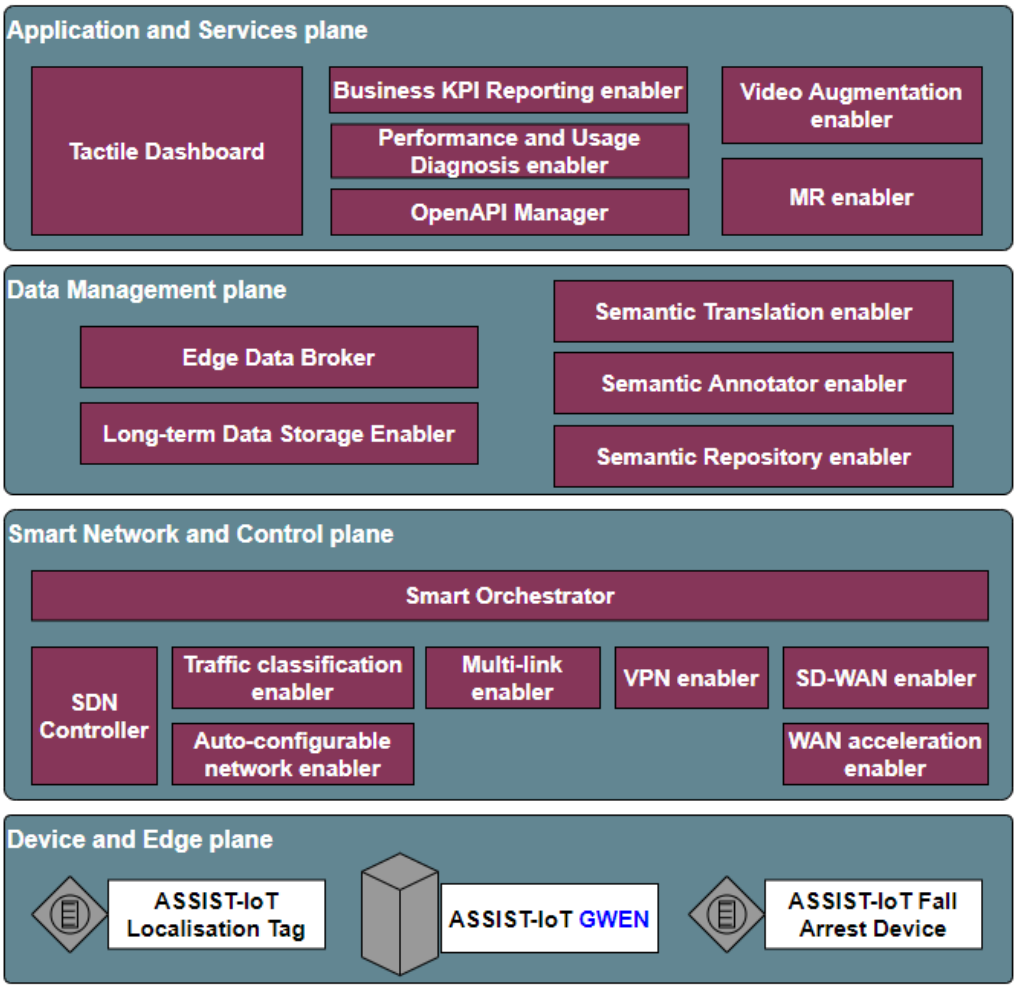


Figure 1. ASSIST-IoT enablers and hardware elements formalised

[1] ASSIST-IoT project, D3.7 – Architecture Definition Final.

3. Devices specifications

The specifications shown in Table 1 represent the environmental conditions in which the devices designed and developed during the execution of the project (the GWEN and the Smart of IoT devices) can be operated and stored. Some iterations are still needed before finalise the industrialisation of the hardware, but in any case devices have been designed to be later on tested considering IEC60068-2-2(Bd) + IEC60068-2-1(Ad), IEC60068-2-29 and IEC60068-2-6 test methods, for environmental, shock and random vibrations, respectively.

Table 1. General equipment specifications

Condition type	Condition and testing methods	Value
Environmental operating	Ambient temperature range, normal operation	-10 to 50 [ºC]
	Relative humidity range, normal operation	20 to 90 [%] non-condensing
Environmental storage	Storage temperature range	-20 to 70 [ºC]
	Storage relative humidity	5 to 95 [%] non-condensing

3.1. GWEN

The ASSIST-IoT Gateway and Edge Node (GWEN) is the edge gateway developed for the ASSIST-IoT project. It is a demonstrator setup with flexible configuration suitable to fit all pilots with adjustable computational power and interfaces. The specifications of the GWEN respond to the block schematic diagram shown in Figure 4 from D4.2. It is important to remind that the gateway has been tested in pilots 1, 2 and 3b, supporting the execution of some of the enablers of the project. Having 2 GB of RAM by default, GWENs had to work clustered with other edge devices, such as RaspberryPis, in order to provide enough processing power in those use cases that required a large number of enablers deployed at the edge.

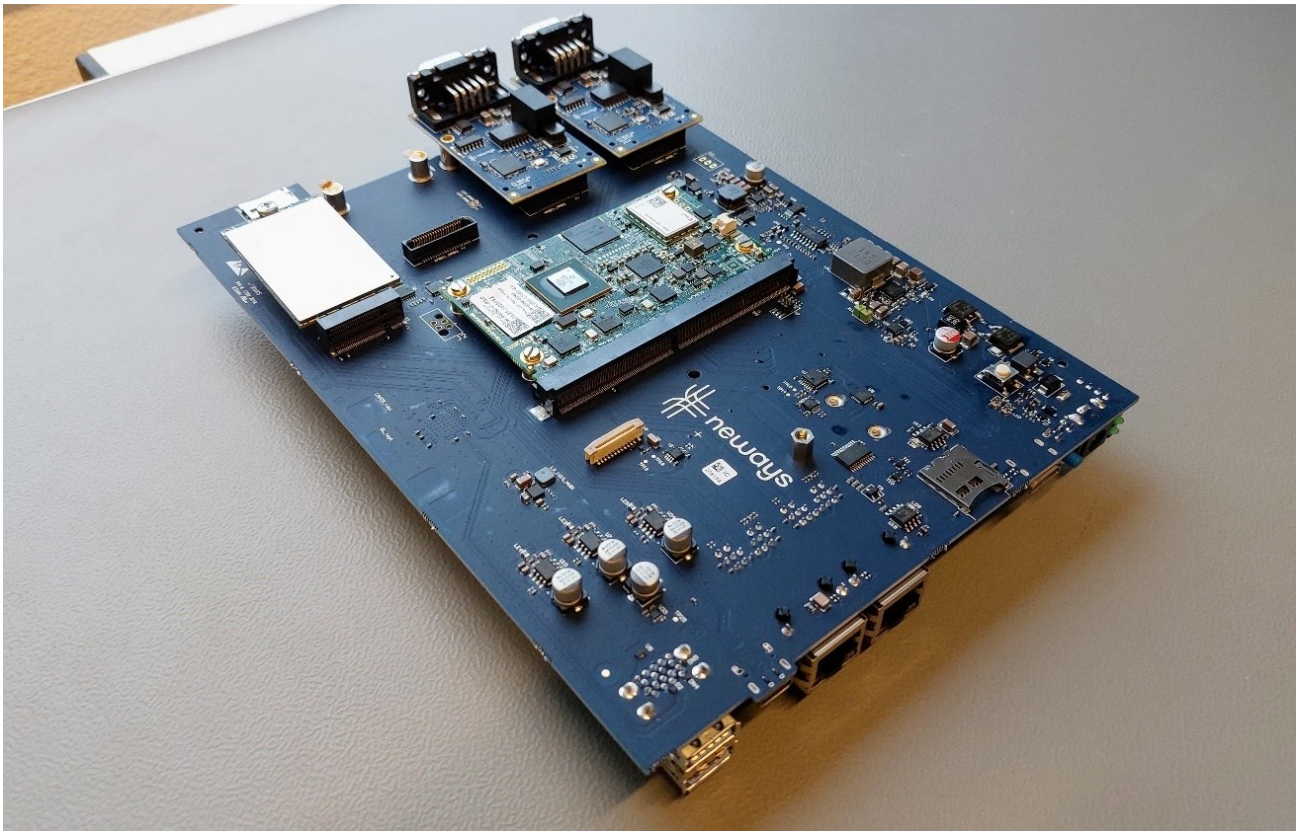


Figure 2. GWEN prototype

Table 2. GWEN specifications

Electronic type	Electronic component	Specification
Wired interfaces	USB	USB2.0 & USB3.0
	Ethernet	2x a 1GB Ethernet port
	SD card	Micro SD card interface
	Power	Barrel connector 12VDC
	HDMI	Mini HDMI interface
	Camera interface	CSI interface for connecting a camera
Wireless interfaces	WiFi	A WiFi interface following IEEE 802.11ac is available
	BLE	Bluetooth Low Energy (BLE) 5.2 (IEEE 802.11ac)
Compute power & storage	Processing module	i.MX 8M Plus. Quad core @1.8 GHz (ARM® Cortex®-A53)
	RAM	2GB & 4GB LPDDR4 (2 variants were developed)
	eMMC	16GB eMMC
Expansion boards		
CAN	Automotive CAN open module	
RS485	An USB to RS485 module is available	
Mobile network module	A M.1 slot is reserved for 3G/4G/5G functionality. A SIM card slot is available on the GWEN.	

The firmware of the GWEN consists of an Operating System (OS) a container runtime and in addition pre-installed software to support enablers will be used. This pre-installed software operates on top of the OS, next to the container runtime so custom containers can use this pre-installed software. Key specifications of the Edge node firmware are given:

- **Operating System (OS):** Yocto, based on Linux, is used as OS. The Yocto Project is an open source collaboration project that helps developers creating custom Linux-based systems regardless of the hardware architecture. The project provides a flexible set of tools and a space where embedded developers worldwide can share technologies, software stacks, configurations, and best practices that can be used to create tailored Linux images for embedded and IOT devices, anywhere a customised Linux OS is needed.
- **Hardware Abstraction Layer (HAL):** The HAL consists of device driver as interface between the electronics and the OS. The Yocto project supports several kinds of peripherals and provides device drivers which implement hardware specific functionality for these peripherals. Besides, not supported peripherals of the Edge node will need own developed device drivers. These are also part of the HAL.
- **Configuration and initialisation:** The configuration and initialisation of the standard interfaces (Ethernet, Serial, etc.), SSH and a default user will be preconfigured on the Edge node, making the node fully functional and ready to run enablers on.
- **Container runtime:** For the container runtime, Docker is used. Docker is a set of platform-as-a-service products that use OS-level virtualisation to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. As the enablers will be implemented as containers, Docker will be preinstalled.
- **Pre-installed software:** The following supportive software will be pre-installed at the Edge node:

- *Python:* Python is a general-purpose programming language that will be used by many enablers. It is used for web development, AI, machine learning, mobile application development, etc. As Python will be used on the Edge node, it will be preinstalled.
- *Kubernetes (K3s):* Kubernetes automates operational tasks of container management and includes built-in commands for deploying applications, rolling out changes to applications, scaling applications up and down to fit changing needs, monitoring applications making it easier to manage applications. Where K3s is a lightweight Kubernetes distribution created by Rancher Labs, and it is fully certified by the Cloud Native Computing Foundation (CNCF). K3s is highly available and production-ready. It has a very small binary size and very low resource requirements.
- *Wazuh:* Wazuh is a free and open source platform used for threat prevention, detection, and response. It is capable of protecting workloads across on-premises, virtualised, containerised, and cloud-based environments.

3.2. ASSIST-IoT localisation tag

The localisation tags and anchors are build up according the following block-diagram:

Table 3. Localisation tag specifications

Electronic component	Specification
UWB Transceiver	Qorvo DWM1001c
Battery	16340 rechargeable battery
Micro controller	STM32F072CBT6 & Nordic nRF52832
Program/Debug	JTAG
Buzzer, LED, pushbutton	The DEV-kit (Qorvo mdek1001c) has several built in components such as a buzzer, 2 LED's and 2 pushbuttons which can be programmed as liked. In the pilots these were used for operator communication and response. The DEV-kit also has a build-in accelerometer which is used for the fall-arrest.

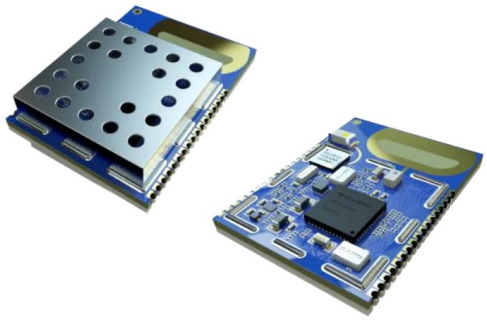


Figure 3. The heart of the module, the Qorvo DWM1001C UWB breakout board

3.3. ASSIST-IoT fall arrest device

Table 4. Fall arrest device specifications

Electronic component	Specification
Fall arrest sensor interface	digital I2C/SPI serial interface
Inertia Measurement Unit (IMU)	LIS2DH12 3-axis accelerometer motion sensor
UWB Transceiver	The accelerometer is part of a build in component of the Qorvo DWM1001c breakout board.

4. Horizontal enablers

All enablers will report their final specifications in the same way. In total, five sections will be included, with the following data:

General specifications and features

Table 5. Template table to report the general information of the enablers

Enabler	***Name of the enabler***
Id	*Short unique identifier/acronym*
Owner and support	*Lead and supporting beneficiaries*
Description and main functionalities	*Functional description of the enabler. Improve current descriptions!!*
Key features	*Bullet points for describing its features, focusing on advancements over SotA (e.g., improvements over Prometheus when developing the PUD, or the EDB w.r.t VerneMQ)*
Plane/s involved	*Horizontal plane or planes on which the enabler's features are delivered*
Requirements mapping	*List of the IDs of the requirements addressed or considered. Update 4.1 using D3.3 data*
Use case mapping	*List of the IDs of the use cases related to this enabler. Update 4.1 using D3.3 data*
Internal components	*List of the internal components of this enabler*

Components and technologies

A high-level schema of the internal microservices or micro-applications is to be included. It should be highlighted that, when implemented, some components can be wrapped in a single container.

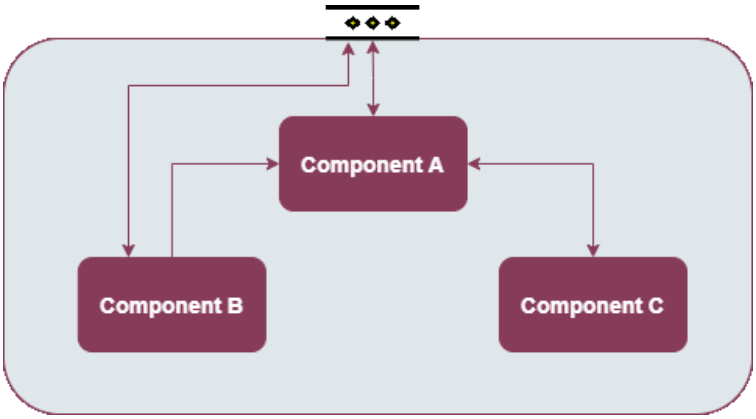


Figure 4. Example of high-level diagram

The description and implementation technologies of each one of the components will be reported in a table as the following one:

Table 6. Template to report the components and implementation technologies of the enablers

Component	**Description**	**Technology/s**
	It is in charge of / Id deals with / It provides …	

NOTE: Apart from components of the encapsulation exceptions, Docker, K8s and Helm have been considered for implementing all the enablers. Additional notes can be added, for complementing the information provided or express any aspect worth to be included.

Communication interface/s

The third section reports the communication interfaces. Generally, this refers to API endpoints, following the table below. In case that other interfaces are present (e.g., MQTT connections, VPN enabler via dedicated TCP/UDP connection, etc.) they will be reported accordingly.

Table 7. Template table to report the API of the enablers

Method	Endpoint	Description
GET/POST/PUT/DELETE		

***** Extended information can be found in the enabler documentation.

Enabler stories

This section updates the use cases in which enablers respond until a specific event or call, presenting the interaction that happens among the components. In this deliverable, they are referred to as **enabler stories**, as "use cases" could be confused with the pilot-related use cases of the project. Also, referring to them as "user stories" would not have been correct as some actions are not triggered by users, but other enablers.

Additional information

The fifth and last section reports additional information related to documentation, encapsulation readiness, integration with other enablers of the project, and features that could be extended in future releases.

Table 8. Template table to report the implementation status of the enablers

Category	Status
Link to ReadtheDocs	*Link to documentation*
Potential features	*Additional features that could be added/extended in the future, now that more knowledge about the enabler is available*
Encapsulation readiness	*Row to explain if an enabler is an encapsulation exception, and why, or if it has a full functional Helm package ready*
Integration with other enablers	*Expresses if the enabler require others to offer all its functions, or if it works in a complete standalone fashion*

4.1. Smart Network and Control enablers

4.1.1. Smart orchestrator

4.1.1.1. General specifications and features

Table 9. General information of the Smart orchestrator

Enabler	***Smart orchestrator enabler***
Id	T42E1
Owner and support	UPV
Description and main functionalities	The Smart Orchestrator Enabler aims to control the lifecycle of enablers in a multi-cluster environment. It not only serves a management function but also provides network security and enabler instantiation automation capabilities. The Smart Orchestrator works with Helm charts as a packaging system, which offers the following benefits: • Simplifies the packaging of software, making it easily customisable. • Enables seamless upgrades of enablers.
Key features	• Allows managing the enablers lifecycle. • Introduces network security and automation capabilities. • Eases the connection of enablers.
Plane/s involved	Smart Network and Control Plane
Requirements mapping	• R-P1-20: Remote latency capabilities (it will be in charge of deploying enablers related to network) • R-P1-22: Multilink wireless network capabilities (it will be in charge of deploying the related CNFs)

Enabler	*Smart orchestrator enabler*
	• R-P3A-11: Connectivity between OEM and fleet (it may/can deploy a ping-based CNF to evaluate connection between fleet and OEM prior to an update, and instantiate those VNFs needed for stablishing the connection) • R-P3A-12: Edge Connectivity (it may/can deploy CNFs to support required latencies)
Use case mapping	This enabler is inherent to an ASSIST-IoT ecosystem and therefore it should be present at all pilots, otherwise it would not be possible to orchestrate VNFs and hence the Smart Network and Control plane would not be present. Among the use cases of the project, the ones with higher need of it are: • UC-P1-6: Wireless remote RTG operation • UC-P1-7: Target visualisation during RTG operation • UC-P2-6: Safe navigation instructions • UC-P3B-1: Vehicle's exterior condition documentation
Internal components	API, Scheduler, Multi-cluster service controller, Metrics server, Orchestrator microservices

4.1.1.2. Structure, components and implementation technologies

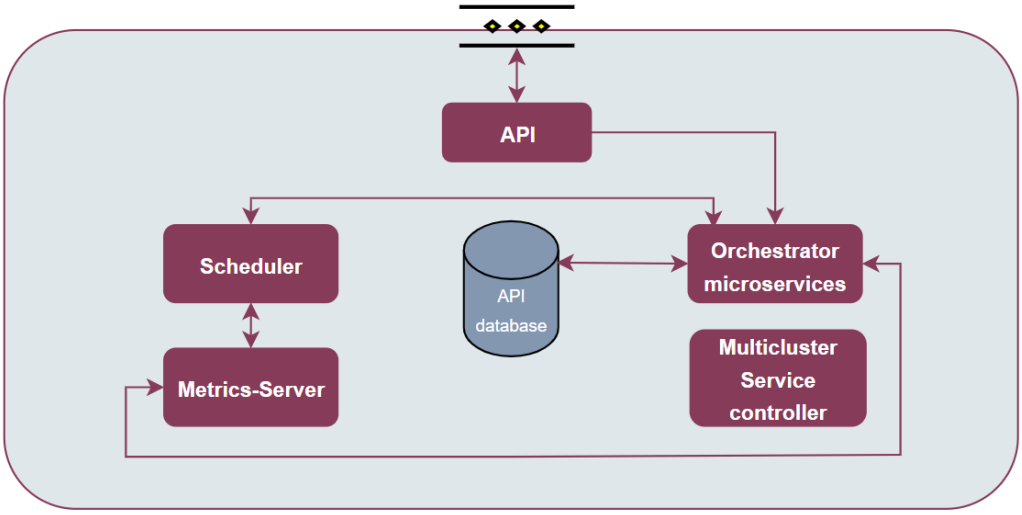


Figure 5. High-level diagram of the Smart orchestrator

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 10. Components and implementation of Smart orchestrator

Component	Description	Technology/s
API	The RESTful API serves as a central gateway to access all orchestrator services. Its primary responsibilities include handling API requests for adding, retrieving, and deleting clusters, repositories, and enablers. Additionally, it incorporates a scheduler function for automating enabler cluster scheduling.	Express
Orchestrator Microservices	These microservices provide the orchestration functions for adding, retrieving, and deleting clusters, repositories, and enablers.	Express
Multicluster service controller	Replicates cloud-based services, allowing them to be accessed by edge services using their respective DNS names	Python, Cilium
Scheduler	Provides automatic cluster election for the enablers instantiation based on the resources available or network traffic.	Python, mck8s, NeuralProphet
Metrics server	Provide the scheduler with the resources from each of the clusters.	Prometheus

4.1.1.3. Communication interfaces

Table 11. API of the Smart orchestrator

Method	Endpoint	Description
GET	**/clusters**	Returns the clusters that have been added.
GET	**/clusters/{id}**	Returns the cluster that has been added by id.
GET	**/clusters/node/{id}**	Returns the nodes in a cluster that have been added.
GET	**/clusters/cloud/find**	Returns the cloud cluster.
POST	**/clusters**	When provided with a body, it attempts to establish a cluster connection and add it into the system.
DELETE	**/clusters/{id}**	Deletes a cluster by id.
GET	**/repos**	Returns the repositories.
GET	**/repos/charts/{id}**	Returns the charts available in a specific repository.
POST	**/repos/public**	Given an URL, description and name, incorporate a public repository into the system.
POST	**/repos/private**	Given an URL, description, name and user credentials, incorporate a private repository into the system.
POST	**/repos/update**	Updates the repositories available charts in each of them.
DELETE	**/repos/{id}**	Deletes a repository by id
GET	**/enabler**	Returns the enablers installed in the clusters.
GET	**/enabler/cluster/{id}**	Returns the enablers installed in a specific cluster.
POST	**/enabler**	When provided with a body, it installs an enabler, either manually or automatically, by applying predefined policies.
POST	**/enabler/upgrade/{id}**	When provided with a body, it upgrades an enabler changing the version.
DELETE	**/enabler/{id}**	Deletes an enabler by id.
DELETE	**/enabler/volumes/{id}**	Deletes the volumes by the enabler id.

4.1.1.4. Enabler stories

Numerous enabler stories can be applicable to this particular enabler. However, it is possible to categorise some of them into groups that share a common objective.

The **first enabler story** depicts how a **cluster addition** to the Smart Orchestrator enabler occurs. This operation consists in a POST request where the request body includes the K8s cluster kubeconfig.

STEPS 1-2: The user sends a POST request with some data related with the credentials and additional fields such as the name, description or the CNI installed in the cluster. These data are captured by the API and sent to the corresponding microservice.

STEP 3: The microservice responsible for the task attempts to stablish connection with the K8s cluster. If the connection is successful, the clustermesh is stablished and the next step is initiated.

STEP 4: After the cluster has been created, in order to enable the scheduler to work, the metrics server must have the newly added cluster registered. Consequently, the microservice proceeds to register it.

STEP 5: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

STEP 6-7: The microservice sends a request to the database for adding or updating the cluster and the DB is in charge of doing it.

STEPS 8-10: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

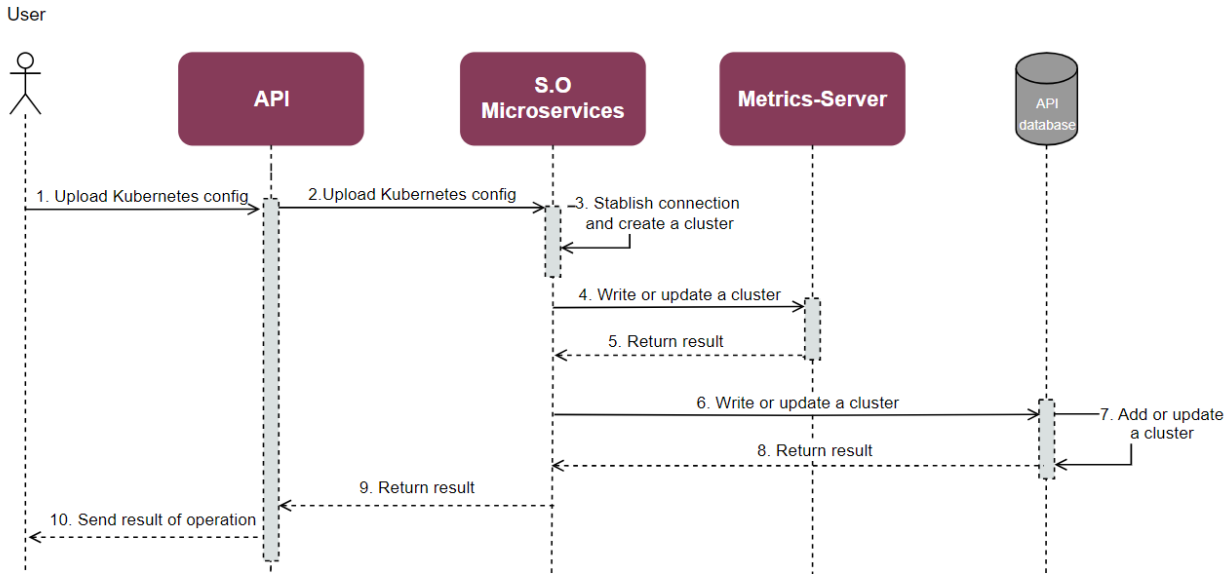


Figure 6. Smart Orchestrator enabler ES1 (add cluster)

The **second story groups all the GET requests (get clusters, get helm repositories, get enablers)**; in this case the example is focused in getting the helm repositories.

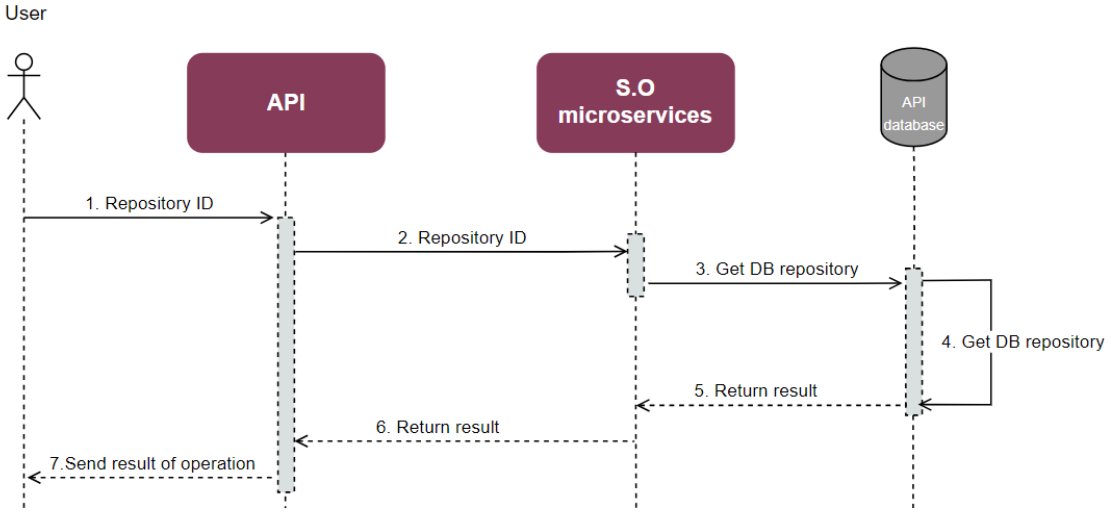


Figure 7. Smart Orchestrator enabler ES2 (list repositories)

STEPS 1-2: The user sends a GET request for getting the repositories data. The request is captured by the API and sent to the corresponding microservice.

STEPS 3-4: The microservice sends a request to the database for getting the helm chart repository and the DB is in charge of doing it.

STEPS 5-7: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

The **third enabler story consolidates all DELETE requests (delete cluster, delete helm repository, delete enabler)**, with the specific example in this case being centered around the deletion of a Helm repository.

STEPS 1-2: The user sends a DELETE request for deleting a specific repository including the ID of it. The request is captured by the API and sent to the corresponding microservice.

STEP 3: The microservice in charge of this task deletes it from the system.

STEPS 4-5: The microservice sends a request to the database. The DB is in charge of deleting it.

STEPS 5-7: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

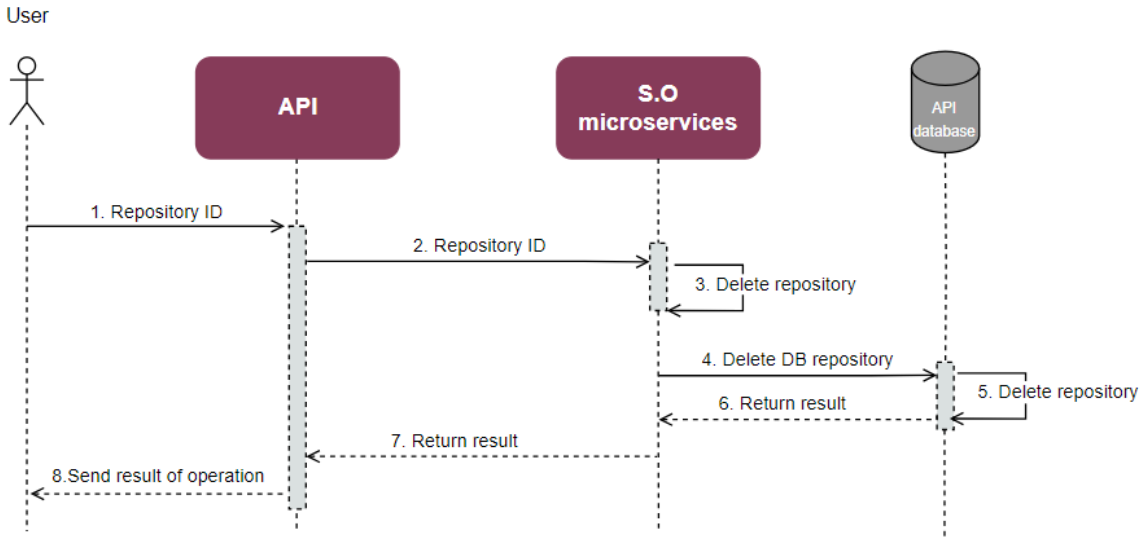


Figure 8. Smart Orchestrator enabler ES3 (delete repository)

The **fourth story** explains how an **enabler is instantiated**. This story may involve two variables: the utilisation of the scheduler and the use of the multicluster service, although they are not mandatory for installing an enabler.

STEPS 1-2: The user sends a POST request with some data related with the name, helm repository and helm chart, values and cluster (placement policy when the scheduler is used). These data are captured by the API and sent to the corresponding microservice.

STEP 3: When the scheduler feature is activated, the microservice sends a request to fetch the available clusters, considering both the resources within the cluster and the resources requested by the enabler.

STEPS 4-5: The scheduler sends a request to the database for getting the K8s clusters kubeconfig. The DB is in charge of getting them.

STEP 6: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

STEP 7: The scheduler selects the cluster by using the placement policy and the resources mentioned earlier.

STEP 8: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

STEP 9: A microservice creates the enabler, installing it from the helm chart saved in the repository selected and in the cluster selected by the scheduler. When an enabler is installed, if it contains an multicluster service, triggers an event.

STEPS 10-11: The microservice sends a request to the database. The DB is in charge of creating the enabler DB record.

STEPS 12-14: If an error occurs, the response will include the error message, which is then transmitted to the user. If no errors occur, a successful response is returned.

If the multicluster service is enabled in the enabler helm chart, the task runs simultaneously to the enabler DB creation.

STEPS 10-11: The multicluster service controller gets the K8s event and gets all the K8s kubeconfig from the DB, except the cloud one.

STEP 12: If an error occurs, the response will include the error message and will print an error log. If no errors occur, a successful response is returned.

STEP 13: The multicluster service controllers deploys the services in all the edge clusters by stablishing connection with them.

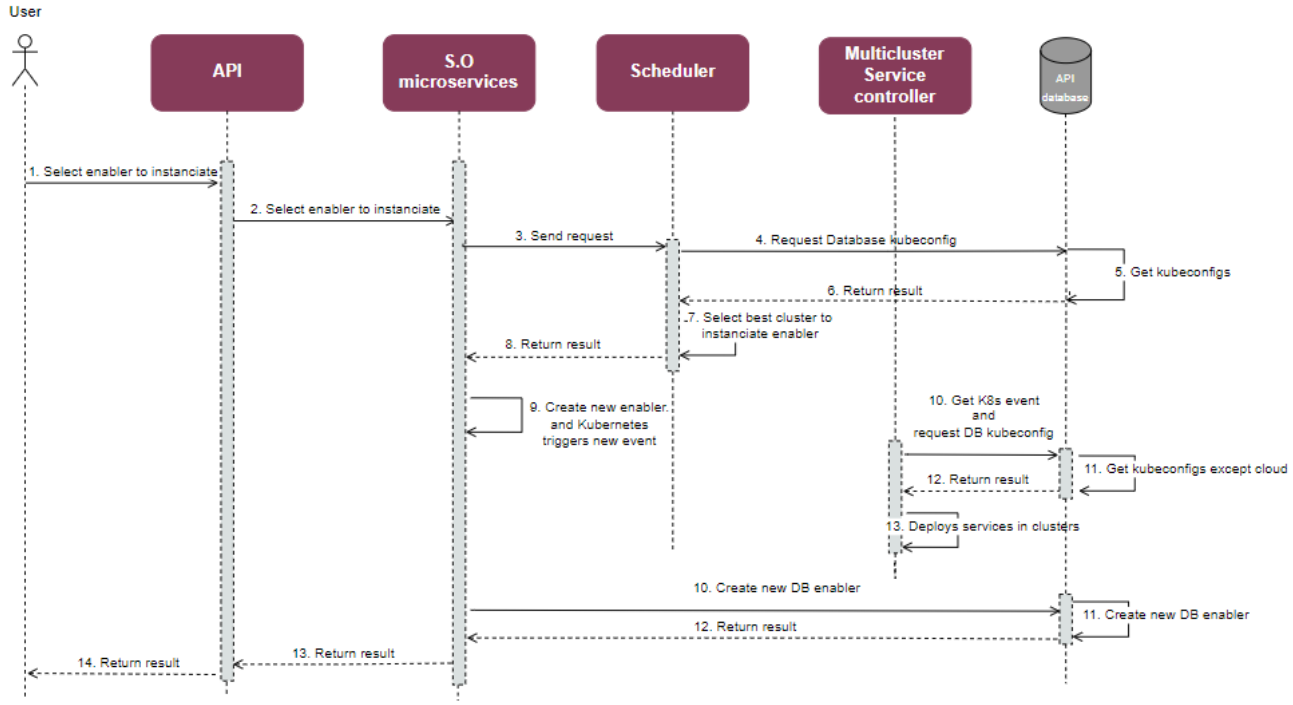


Figure 9. Smart Orchestrator enabler ES4 (add enabler)

4.1.1.5. Implementation information

Table 12. Implementation status of the Smart orchestrator

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/smart_orchestrator.html
Potential features	All the features for the project have been implemented. However, newer versions may incorporate ClusterAPI for provisioning, upgrading, and managing multiple clusters resources.
Encapsulation readiness	Full functional Helm package ready
Integration with other enablers	The enabler is integrated with the PUD to be used as the Metrics Server.

4.1.2. SDN Controller

4.1.2.1. General specifications and features

Table 13. General information of the SDN controller

Enabler	***SDN Controller***
Id	T42E2
Owner and support	OPL
Description and main functionalities	SDN Controller is the part of programmable network management system (control plane) to control network devices i.e., software switches based on OpenFlow protocol, including configuring, monitoring and management of packet flows. The main functionalities are related to network management, operation and maintenance, allowing topology management, network configuration, network control and network operations, among other features.
Key features	• Network topology configuration, • Monitoring of network elements, • Packet flows configuration.
Plane/s involved	Smart Network and Control Plane

Enabler	*SDN Controller*
Requirements mapping	• R-P3A-11: Connectivity between OEM and fleet (it provide connectivity setup and control) • R-P3A-12: Edge Connectivity (it provides network core connectivity for edge systems)
Use case mapping	Not applicable directly in the pilots as it requires SDN equipment. It is envisioned to any of use cases in which SDN network or 5G virtualised edge access will be deployed as well as for other enablers related to programmable network. Among pilot' use cases, it would be involved for mission critical systems: • UC-P1-6: Wireless remote RTG operation • UC-P2-6: Safe navigation instructions • UC-P3B-1: Vehicle's exterior condition documentation • UC-P1-7: Target visualisation during RTG operation
Internal components	• GUI • Northbound API • Configuration, Control, Topology component • Southbound API

4.1.2.2. Structure, components and implementation technologies

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 10), one can distinguish core functional modules like Configuration, Control, Topology, and Northbound (NB) and Southbound (SB) APIs.

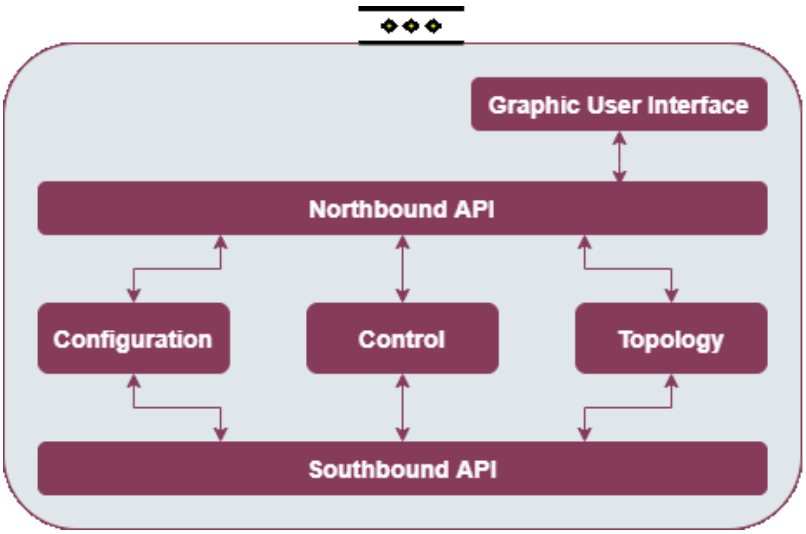


Figure 10. High-level diagram of the SDN controller

For implementation in the project the open source ONOS controller was selected. The version for K8s deployment with Helm chart was developed. The main components are depicted in the table below:

Table 14. Components and implementation of SDN controller

Component	Description	Technology/s
Northbound API	Northbound API provide REST API and new generation interfaces using gNMI, gNOI, P4Runtime, NetDisco. It is needed for developing applications for network control and orchestrations and can be used by other external enablers.	Java
Southbound API	Southbound API provide protocols like NETCONF and new generation interfaces using gNMI, gNOI, P4Runtime, NetDisco. It is needed for network devices control provided by different vendors.	Java

Control Module	This component is responsible for network flow control and meter API. It allows for network routing and traffic management.	Java, REST API
Configuration Module	This component is in charge of configuration of network devices, tracking the changes in the configuration of the network.	Java, REST API
Topology Module	This component is responsible for topology management of the network. It manages and keeps information about the network graph and network devices, links, and hosts.	Java, REST API
Graphic User Interface	This component will expose the functionalities of the internal modules of the SDN Controller for administrative purposes.	Java, REST API

4.1.2.3. Communication interfaces

Table 15. API of the SDN controller

Method	URL	Description
GET/POST/ PUT/DELETE	**/link/ ?{device=deviceId} {port=portNumber} {direction=[ALL,INGRESS,EGRESS]}**	Lists all infrastructure links, creates, update, deletes device
GET/POST/ PUT/DELETE	**/devices/{deviceid}/ports**	Lists all infrastructure devices, creates, update, deletes device
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}/{intent-id}**	Gets the details for the given Intent object. Creates, deletes a new Intent object.
GET/POST/ PUT/DELETE	**/applications/{app-name}**	Gets a list of all installed applications. Activates, deactivates the named application.
GET/POST /DELETE	**/configuration/{component}**	Gets the configuration values for a single component. Adds, removes a set of configuration values to a component

NOTE: Extended information can be found in the enabler documentation.

4.1.2.4. Enabler stories

The usage of SDN controller enabler is envisioned in many applications and for other enablers needs that require the network control and monitoring features. Three exemplary stories are presented below, being the flow almost identical (the major change is on the SDN controller internal function consumed).

The **first enabler story** is related to the **configuration of a programmable network device** (switch**)**, following the sequence and related steps:

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.

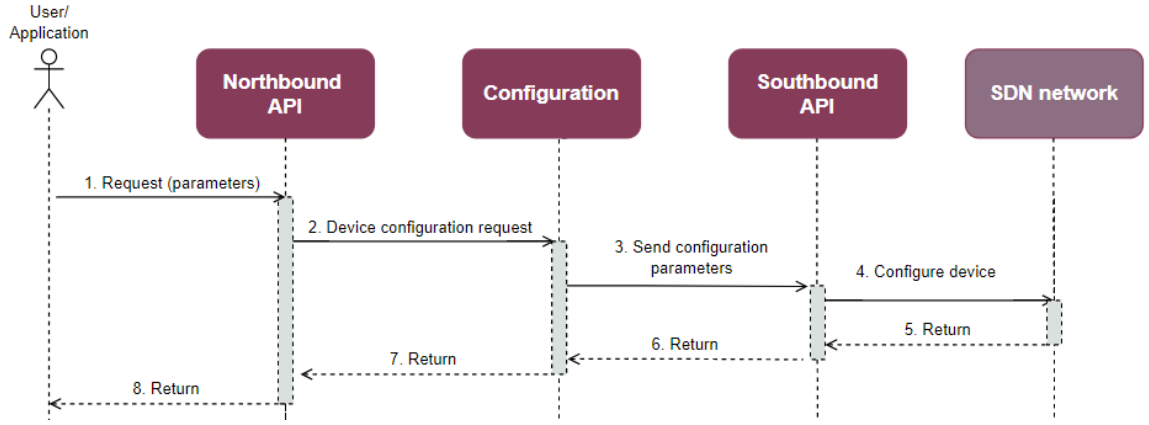


Figure 11. SDN controller ES1 (device configuration).

The **second enabler story** 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:

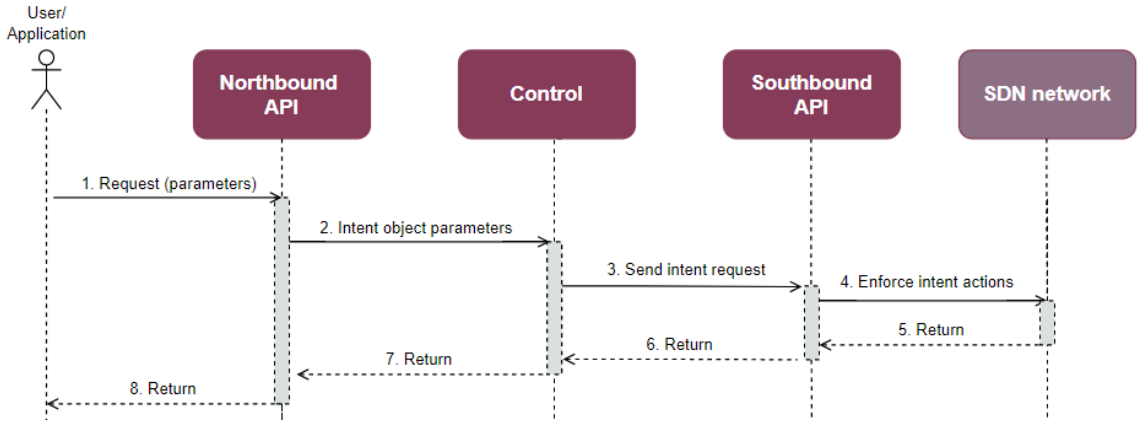


Figure 12. SDN controller ES2 (intent deployment)

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 **third enabler story** depicted is related to **topology discovery**. In this case, the diagram and steps are the ones described below:

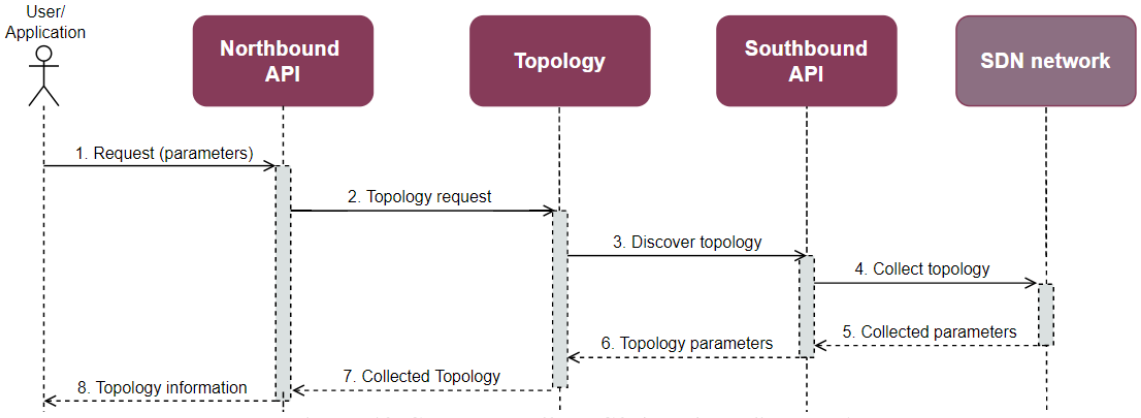


Figure 13. SDN controller ES3 (topology discovery)

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.

4.1.2.5. Implementation information

Table 16. Implementation status of the SDN controller

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/sdn_controller.html
Potential features	This enabler could be combined with other enablers in the project and used for application development to manage the SDN network.
Encapsulation readiness	Controller is encapsulated in Docker images and work in a Kubernetes cluster. Helm chart is ready to use.
Integration with other enablers	Usage with other enablers and integrated with auto-configurable network enabler.

4.1.3. Auto-configurable network enabler

4.1.3.1. General specifications and features

Table 17. General information of the Auto-configurable network enabler

Enabler	Auto-configurable network enabler
Id	T42E3
Owner and support	OPL
Description and main functionalities	This enabler provides optimised network resource management using network routing configuration capabilities of the SDN Controller. Using an AI-based solution, it improves the performance of the network KPI's i.e., traffic load distribution, data losses and transfer latency.
Key features	• AI based policy rules generation, • Monitoring of network parameters (traffic load) and QoS parameters (data losses and latency), • Network resources optimisation for multidimensional KPI's
Plane/s involved	Smart Network and Control Plane
Requirements mapping	• R-P1-20: Remote latency capabilities (this enabler can help prioritising involved traffic) • R-P3A-12: Edge Connectivity (it provides network core connectivity for edge systems)
Use case mapping	Not applicable directly in the pilots as it requires SDN equipment. It is deployable in any of use cases in which SDN network is applied. Among pilot' use cases, it would be involved for mission critical and video streaming systems: • UC-P2-6: Safe navigation instructions • UC-P3B-1: Vehicle's exterior condition documentation • UC-P3B-2: Exterior defects detection support
Internal components	Policy Engine, Monitoring Module

4.1.3.2. Structure, components and implementation technologies

This enabler provides functionalities for optimising network configuration leveraging the SDN Controller in programmable network environment. It assumes generation of the policies and enforces them using the northbound APIs of the SDN Controllers. Polices are set automatically (using AI solutions: Ant Colony mechanism) to improve the performance and quality of selected KPIs of the network (e.g., traffic load distribution, data transfer losses and latency). Enabler provides three different strategies regarding KPI's optimisation: network resources, data transfer losses and latency of data transfer. Moreover, multidimensional optimisation strategy taking into account all mentioned parameters was developed.

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 KPIs 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 and QoS statistics. The internal structure is presented in figure below.

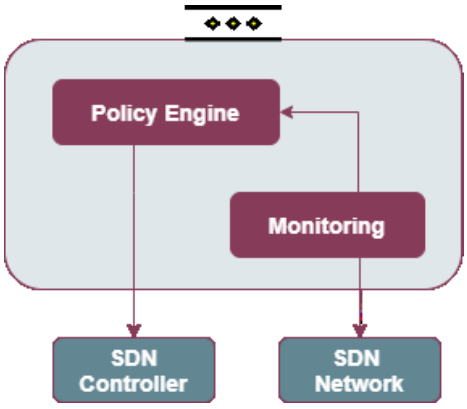


Figure 14. High-level diagram of the Auto-configurable network enabler

For the project K8s deployment with helm chart was developed. The main components are depicted in the table below.

Table 18. Components and implementation of Auto-configurable network enabler

Component	Description	Technology/s
Policy Engine	This component is in charge of creation of polices and its execution in the SDN network for optimising the network traffic and creation of routing paths. It obtains the network information using SDN controller and data traffic and QoS parameters using monitoring module. The optimising algorithms is supported by AI techniques like Deep Learning and Ant Colony algorithm.	Python, Java, REST API
Monitoring Module	This component is responsible for collecting network traffic statistics and QoS parameters. The monitored KPIs are: traffic load per link, data losses and latency per link. Open source rt-sFlow tool was integrated.	Python, Java

4.1.3.3. Communication interfaces

Table 19. API of the Auto-configurable network enabler

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

NOTE: Extended information can be found in the enabler documentation.

4.1.3.4. Enabler stories

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

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

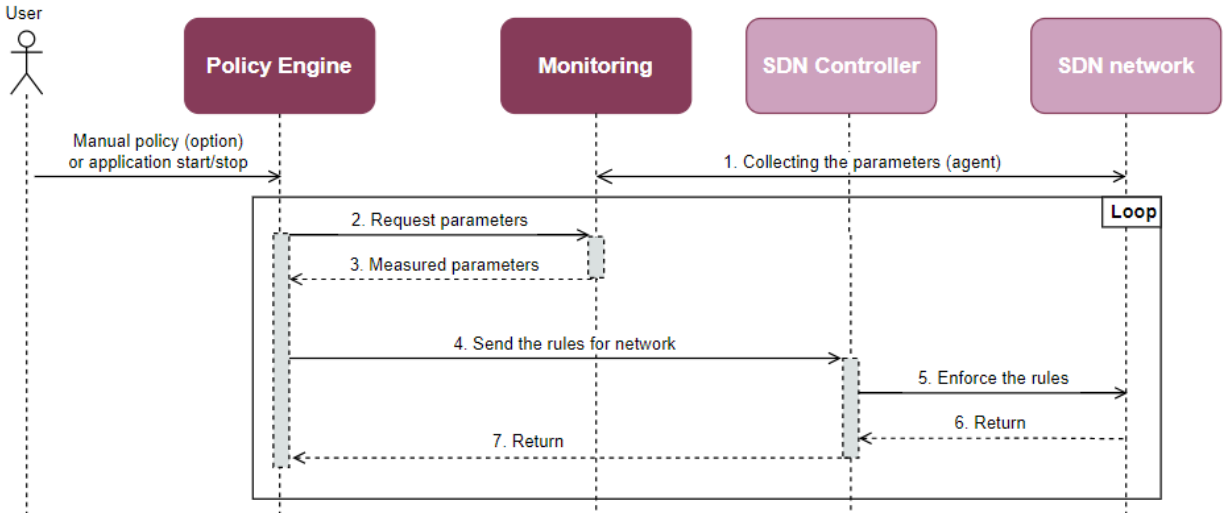


Figure 15. Auto-configurable network enabler ES1 (policy-based network adaptation)

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 works in standalone fashion, triggering itself regularly when the new flow is coming to the network.

4.1.3.5. Implementation information

Table 20. Implementation status of the Auto-configurable network enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/auto_configurable_network_enabler.html
Potential features	This enabler can be used for other application development to optimise the SDN network resources.
Encapsulation readiness	Enabler is encapsulated in Docker image and work in a Kubernetes cluster. Helm chart is ready to use.
Integration with other enablers	Integrated with SDN controller enabler.

4.1.4. Traffic classification enabler

4.1.4.1. General specifications and features

Table 21. General information of the Traffic classification enabler

Enabler	Traffic classification enabler
Id	T42E4
Owner and support	UPV
Description and main 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

Enabler	Traffic classification enabler
	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.
Key features	• Allows using widespread pcap files to train the models offline • Two different types of models to train and use: CNN and Resnet
Plane/s involved	Smart Network and Control Plane
Requirements mapping	Not applicable, as any pilot makes use of SDN equipment. If SDN networks were available, it could be mapped to: • R-P1-20: Remote latency capabilities (this enabler can help prioritising involved traffic) • R-P3A-12: Edge Connectivity (this enabler can prioritise traffic related to PCM calibration updates)
Use case mapping	Not applicable, as any pilot makes use of SDN equipment. If they had, it would fit those use cases in which a particular traffic could be prioritised by the SDN Controller. Among pilot' use cases, it would be involved in those that traffic of either video streams, mission critical systems or image data have priority: • UC-P1-7: Target visualisation during RTG operation • UC-P2-6: Safe navigation instructions • UC-P3B-1: Vehicle's exterior condition documentation • UC-P3B-2: Exterior defects detection support
Internal components	API, Training module, classifier

4.1.4.2. Structure, components and implementation technologies

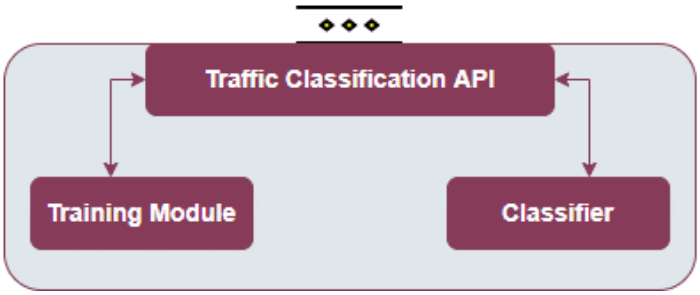


Figure 16. High-level diagram of the Traffic classification enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 22. Components and implementation of the Traffic classification enabler

Component	Description	Technology/s
API	API REST, acting as a central proxy of the operations that are offered by the enabler. It is responsible of managing the API calls related to starting a training and an inference process. It also includes necessary calls for preparing data used for further training.	Flask
Training module	It will be invoked for training the ML models, ideally when an extended or new dataset is available (mandated by a user). Currently, a CNN and a Resnet models are incorporated.	scapy, torch, scikit-learn
Classifier	Contains the functions in charge of executing the inference process, taking a trained model and a set of packet features as inputs.	scapy, torch, scikit-learn

4.1.4.3. Communication interfaces

Table 23. API of the Traffic classification enabler

Method	Endpoint	Description
GET	**/version**	Returns the version of the enabler.
GET	**/health**	Returns status of the enabler (it is considered healthy if its components are deployed and can be communicated).
GET	**/v1/api-export**	Returns the openapi specifications of the enabler.
POST	**/v1/preprocess**	Given a set of .pcap files via volume (in ML_folder/data), these are prepared for further training.
POST	**/v1/create_train_test_set**	Given a set of preprocessed files (in ML_folder/preprocessed), these are split in two sets for training and validation, and parcel files are prepared.
POST	**/v1/train**	Given a set of prepared files (in ML_folder/target), a training process is started. This may take a long time depending on the input data volume
POST	**/v1/cnn_inference_app**	Returns the application of the packets of a .pcap file, considering a previously trained CNN model (present in ML_folder/model).
POST	**/v1/cnn_inference_traffic**	Returns the traffic type of the packets of a .pcap file, considering a previously trained CNN model (present in ML_folder/model).
POST	**/v1/resnet_inference_app**	Returns the application of the packets of a .pcap file, considering a previously trained resnet model (present in ML_folder/model).
POST	**/v1/resnet_inference_traffic**	Returns the traffic type of the packets of a .pcap file, considering a previously trained resnet model (present in ML_folder/model).

4.1.4.4. Enabler stories

Two are the enabler stories that apply to this enabler. The **first enabler story** will be instantiated by a user, once the volume attached to the training module labelled samples of data (in .pcap format) to use for this purpose. The story has been updated to represent the different calls that a user has to make to complete flow. Particularly, the steps related to the first use case are:

STEPS 1-2: Before training, the .pcap files used for training need to be pre-processed. The user starts this process interacting with the enabler API, which forwards this operation to the training module.

STEP 3: Once the operation has been completed, the API is notified, informing the user.

STEPS 4-5: Afterwards, pre-processed packets have to be separated in a training and a validation set. The user starts this process with the respective API call, which is then forwarded to the training module.

STEP 6: Once the operation has been completed, the API is notified, informing the user.

STEPS 7-8: With the sets ready, the user can start the training process, selecting the desired model.

STEP 9: Once the training has been completed, the API is notified, informing the user.

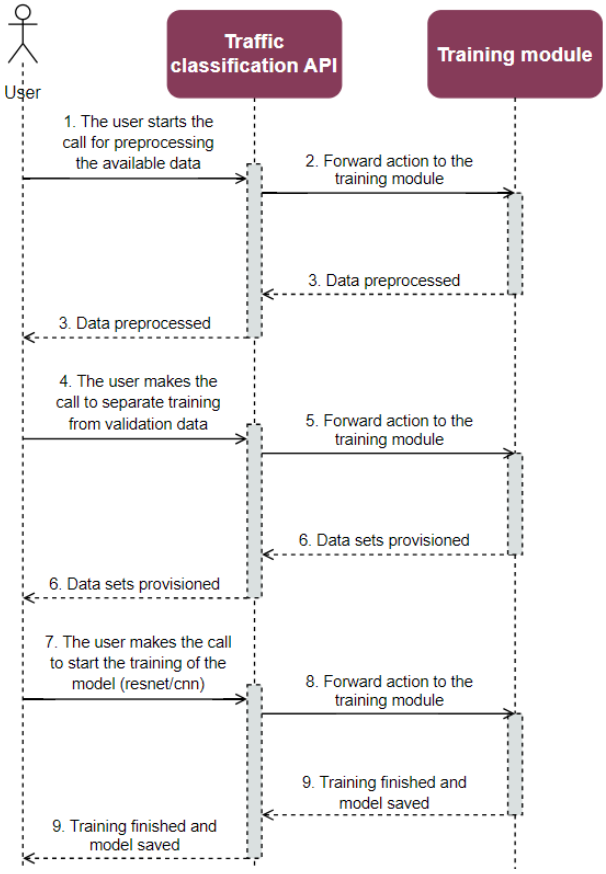


Figure 17. Traffic classification enabler ES1 (train model)

The **second enabler story** can be initiated by a user or by the SDN controller, related to the **classification of a packet or group of packets**. In this case, the next steps are followed:

STEP 1: A external entity (user or SDN Controller) starts an inference process via API command, making use of previously-trained model. The .pcap file to process is attached.

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) and the launcher, notified.

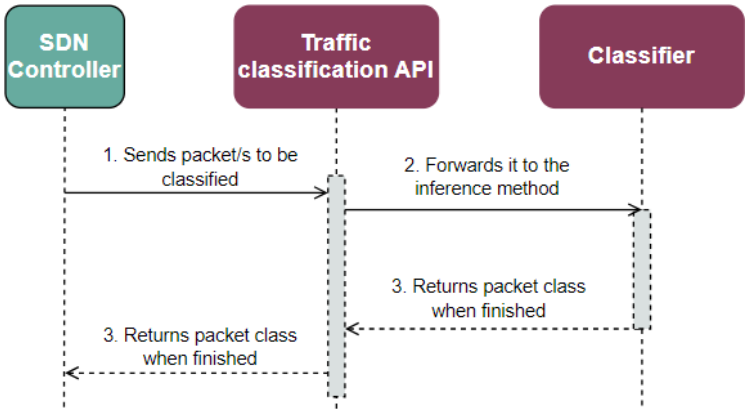


Figure 18. Traffic classification enabler ES2 (packet classification)

4.1.4.5. Implementation information

Table 24. Implementation status of the Traffic classification enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/traffic_classification_enabler.html
Potential features	In the current version, models are stored in a K8s' persistent volume. In future releases, the enabler could be enhanced by using the FL repository enabler with that end. Also, in future API versions, calls can be transformed to be model-agnostic, passed via parameter, to allow the use of additional models.
Encapsulation readiness	Full functional Helm package ready
Integration with other enablers	Any integration has been performed. In case an SDN controller needs it, it should be adapted suitably to consume the Traffic classification API.

4.1.5. Multi-link enabler

4.1.5.1. General specifications and features

Table 25. General information of the Multi-link enabler

Enabler	Multi-link enabler
Id	T42E5
Owner and support	UPV
Description and main functionalities	The main goal of this enabler is to manage different wireless access networks, so in case the configured primary link is down a second one is up without noticing (at least, not by the user) any kind of service disruption. The enabler offers the ability to be reconfigured in the meantime it is running.
Key features	• Allows to maintain connection between two hosts with multiple interfaces and select them in priority order. • Its performance could be changed to support redundancy instead of (or jointly with) reliability.
Plane/s involved	Smart Network and Control Plane

Enabler	Multi-link enabler
Requirements mapping	• R-P1-22: Multilink wireless network capabilities (self-explanatory). • R-P1-21: Remote reliability capabilities (in case one network fails, another can take over, considering redundancy mechanisms)
Use case mapping	• UC-P1-6: Wireless remote RTG operation • UC-P1-7: Target visualisation during RTG operation
Internal components	Client/server API, Bridging component, Bonding component, VPN client/server

4.1.5.2. Structure, components and implementation technologies

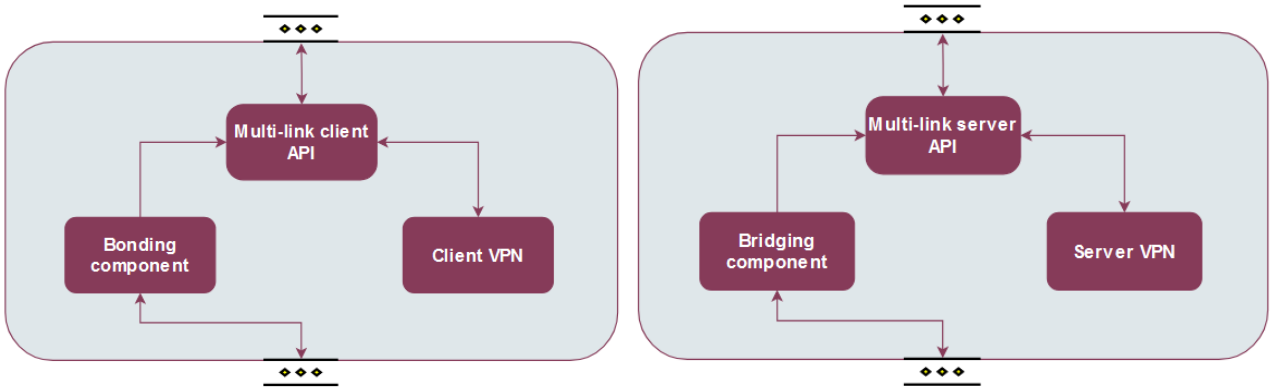


Figure 19. High-level diagram of the Multi-link enabler: client side (left), server side (right)

In the next figure one can see an implementation of the Multi-link enabler between two hosts, in this case there are two links (Ethernet and WiFi) combined by a bond in the client side and a bridge on the server side. The bond monitors the primary link (WiFi) and in case this link fails switch to the backup link (Ethernet). If the primary link connection is restored, it will switch to the primary link. In other words, the devices will be communicating all the time using the primary link except when the connection of the primary link is down.

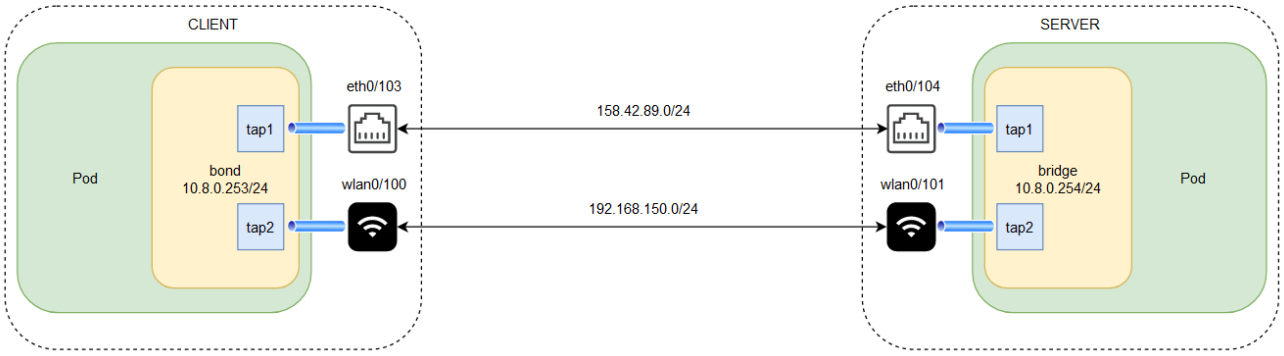


Figure 20. Multi-link client and server example between two hosts

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 26. Components and implementation of the Multi-link enabler

Component	Description	Technology/s
Client/server API	API REST, acting as a central proxy of the operations that are offered by the enabler. It is responsible of managing the API calls related to start and stop the client and server, also calls to reconfigure the bonding component and active testing of the enabler.	Express
Bonding component	It will be invoked to create/configure the bond interface.	Bash
Bridging component	It will be invoked to create/configure the bridge interface.	
VPN Client/Server	Create tunnels of layer 2 (tap) for each link that compounds the bond/bridge.	OpenVPN

4.1.5.3. Communication interfaces

Table 27. API of the Multi-link enabler

Method	Endpoint	Description
GET	**/version**	Returns the version of the enabler.
GET	**/health**	Returns status of the enabler (it is considered healthy if its components are deployed and can be communicated).
GET	**/v1/api-export**	Returns the Open API specifications of the enabler.
GET	**/v1/server/key**	Returns the key used by the tap tunnels
GET	**/v1/server/status**	Returns if the server side is running or not
GET	**/v1/client/status**	Returns if the client side is running or not
GET	**/v1/client/bond_params/{interface}**	Returns the bond configuration of the interface provided in the URL as parameter
POST	**/v1/ping_test**	Execute ping to the IP provided in the JSON of the request body and returns if it was successful or not.
POST	**/v1/server/start**	Start the server side of the multilink creating the bonding component and the tap tunnels specified in the JSON body of the request.
POST	**/v1/server/stop/{bridging_component}**	Stop the server side of the multilink deleting the bonding component specified as parameter in the URL.
POST	**/v1/client/start**	Start the client side of the multi-link creating the bonding component and the tap tunnels specified in the JSON body of the request.
POST	**/v1/client/stop/{bonding_component}**	Stop the client side of the multi-link deleting the bonding component specified as parameter in the URL.
POST	**/v1/client/bond_params/{interface}**	Change the configuration parameters related to the bond interface.
POST	**/v1/tap_up/{tap}**	Bring up the tunnel interface indicated as parameter in the URL.
POST	**/v1/tap_down/{tap}**	Bring down the tunnel interface indicated as parameter in the URL.

NOTE: Extended information can be found in the enabler documentation. Specially the requests available by each side (client and server).

4.1.5.4. Enabler stories

Although there are many operations, some of them follow the same communication schema, so they will be grouped. The **first enabler story** is related with the **start/stop of the server side of the multi-link enabler**. With this operation it is possible to enable/disable the server side.

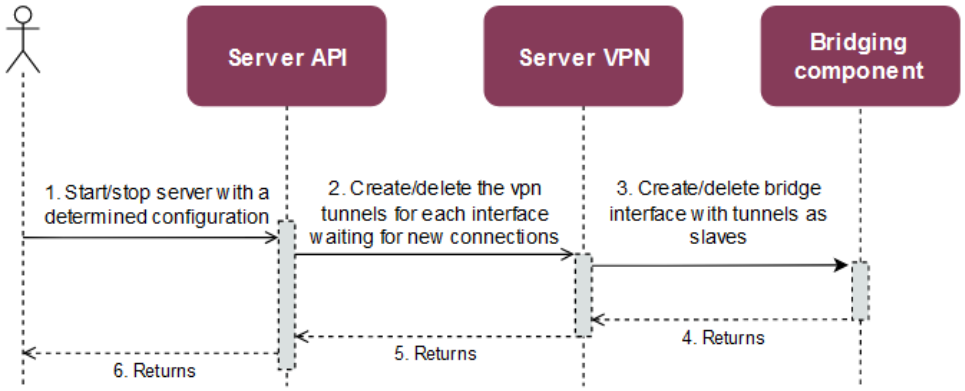


Figure 21. Multi-link enabler ES1 (server-side start/stop)

STEP 1: The user consumes the API of the Multilink-server to start/stop the server side. In the case of the start operation the configuration of the client must be in the request body as JSON following the schema in the OpenAPI. If the request is for stop the server, it is needed the name of the bridging component as a parameter in the URL.

STEP 2: The VPN server component creates/deletes the tap tunnels specified in the bond configuration and leave this connection opened and waiting for a connection request by the client.

STEP 3: Create/delete the bridge interface with tunnels as slaves.

STEPS 4-6: If there is any error, return error response showing the error log, if everything is correct return successful response.

The **second enabler story** refers to the same as the previous one but applying it to the client side (**multi-link client start-stop**).

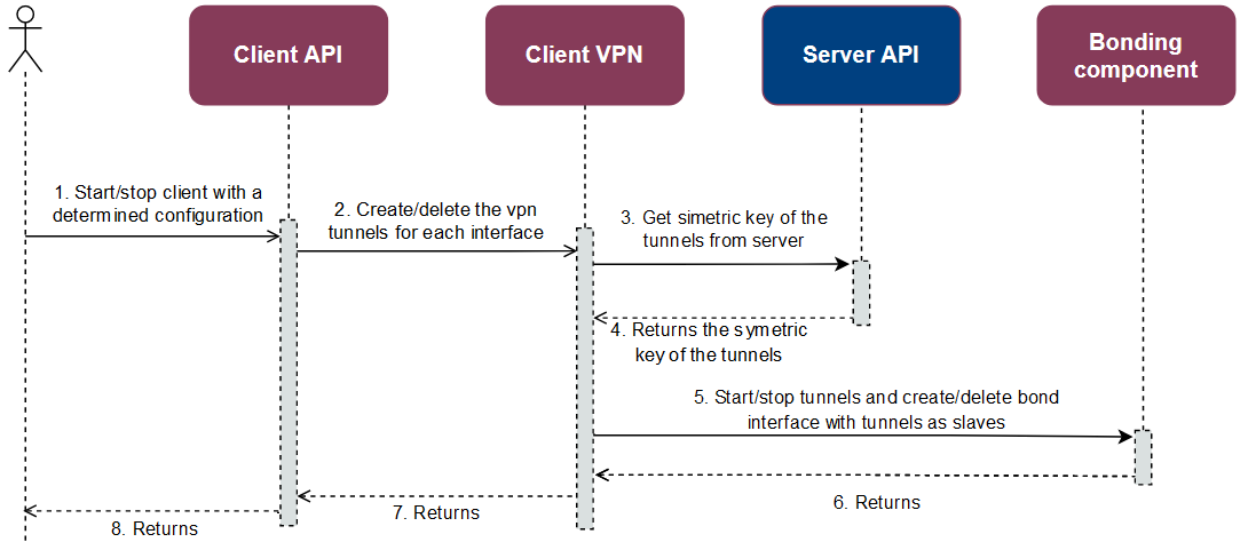


Figure 22. Multi-link enabler ES2 (client-side start/stop)

STEP 1: The user consumes the API of the Multi-link client to start/stop the client side. In the case of the start operation the configuration of the client must be in the request body as JSON following the OpenAPI schema.

STEP 2: The VPN client component creates/deletes the tap tunnels specified in the bond configuration.

STEP 3: Request to the server side the key of the tunnels and start the tunnels connection.

STEP 4: The server key is received and stored in the client side.

STEP 5: The bond interface is created/deleted adding/removing the tap tunnels as slaves.

STEPS 6-8: If there is any error, return error response showing the error log, if everything is correct return successful response.

The **third enabler story** explains how to **change the bonding parameters** (only client side has bonding component). The steps are:

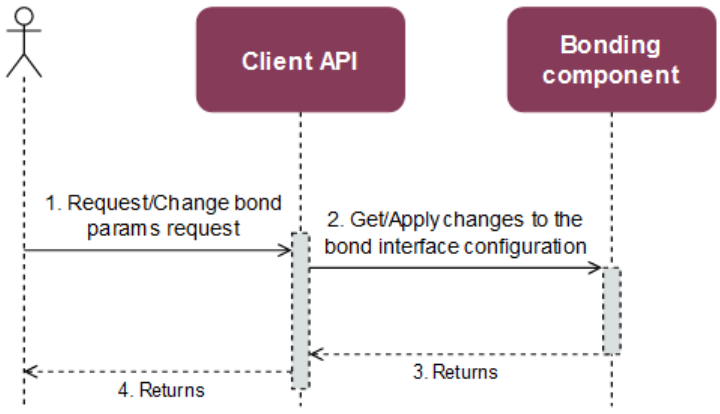


Figure 23. Multi-link enabler ES3 (change bonding parameters)

STEP 1: The user consumes the API of the Multi-link client to get/change the client side. In the case of changing parameters, the configuration of the bond parameters that want to be changed must be in the request body as JSON following the schema in the OpenAPI.

STEP 2: Apply the changes to the bonding component.

STEPS 3-4: If there is any error, return error response showing the error log, if everything is correct return successful response.
The **fourth enabler story** is referred to **bring up or down tunnels (tap interfaces)** in the client side. It could be interesting for test the correct behaviour of the bond, selecting the correct links in case of failures.

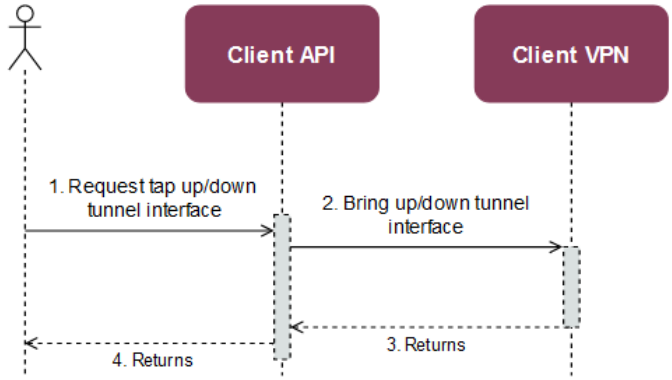


Figure 24. Multi-link enabler ES4 (bring up/down tunnel interfaces)

STEP 1: The user queries the API of the Multi-link client to bring up/down a tunnel interface. The tunnel interface selected is in the URL as parameter.

STEP 2: Bring up/down the interface.

STEPS 3-4: If there is any error, return error response showing the error log, if everything is correct return successful response.
The **fifth enabler story** corresponds to a procedure to **check the connectivity** between client or server side to another host. This request has been implemented to test the connection between the client and server side.

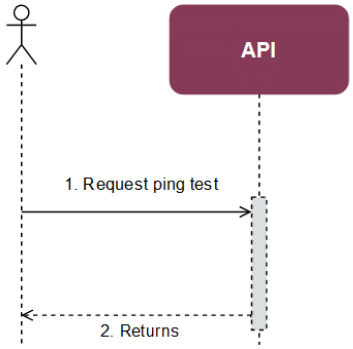


Figure 25. Multi-link enabler ES5 (ping test)

STEP 1: The user queries the API of the Multi-link client to make a ping test. The IP to test the connection with is in the request of the query.

STEP 2: Ping to the host.

STEP 3-4: If there is any error, return error response showing the error log, if everything is correct return successful response with the ping log.

The **sixth enabler story** refers to the request to know the **client/server status**.

STEP 1: The user queries the client API to know the status of the server.

STEP 2: Check the status.

STEPS 3-4: If there is any error, return error response showing the error log, if everything is correct return successful response with the status of the client/server (running or not).

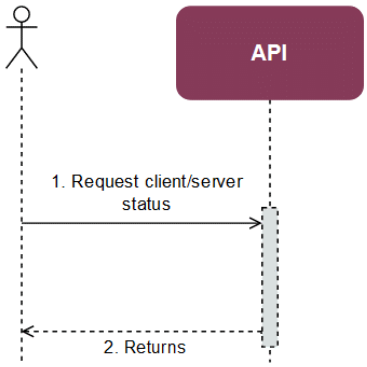


Figure 26. Multi-link enabler ES6 (client/server status)

4.1.5.5. Implementation information

Table 28. Implementation status of the Multi-link enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/multi_link_enabler.html
Potential features	In the current version, the pod that creates, configures and reconfigures the bond interface (create-bond-and-taps) needs superuser privileges due to create, modify and delete interfaces it's a must and it is needed the kernel of the host to do such actions. There has been explored the alternative of doing such actions with a Kubernetes CNI plugin. In the test realised it has been figured it out that it is not as reliable as the actual implementation so it could be developed in a future.
Encapsulation readiness	Full functional Helm package ready
Integration with other enablers	The enabler doesn't have a direct integration with any other enabler.

4.1.6. SD-WAN enabler

4.1.6.1. General specifications and features

Table 29. General information of the SD-WAN enabler

Enabler	SD-WAN enabler
Id	T42E6
Owner and support	UPV
Description and main functionalities	The objective of this enabler is to provide access between nodes from different sites based on SD-WAN technology. This enabler implements 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)
Key features	• Provides tunnelling feature, for securing network connections between sites. • Facilitates the implementation of application-level QoS policies. • Easy configuration of a complex technology
Plane/s involved	The SD-WAN enabler is in the Smart Network and Control plane of the ASSIST-IoT architecture. It belongs to the building block related to self-contained networks, which are the ones used for provisioning private networks over public ones.
Requirements mapping	• R-C-10: Transmission security • R-C-11: Network optimisation
Use case mapping	This enabler will grant a secure and optimised connection for applications and services from different sites. Since Pilot 3b, that could be the target of this enabler, has a cloud managed by a third-party, the enabler cannot be exploited as some network prerequisites cannot be met.

Enabler	SD-WAN enabler
	• UC-P3B-1: Vehicle's exterior condition documentation • UC-P3B-2: Exterior defects detection support
Internal components	• SD-WAN controller • Rsync • NoSQL Database • Etcd

4.1.6.2. Structure, components and implementation technologies

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 SDWAN edge must 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 automatically controlling the SD-WAN edges and hubs, enabling and securing the connections. Its structure is presented in Figure 27, consisting of the following elements:

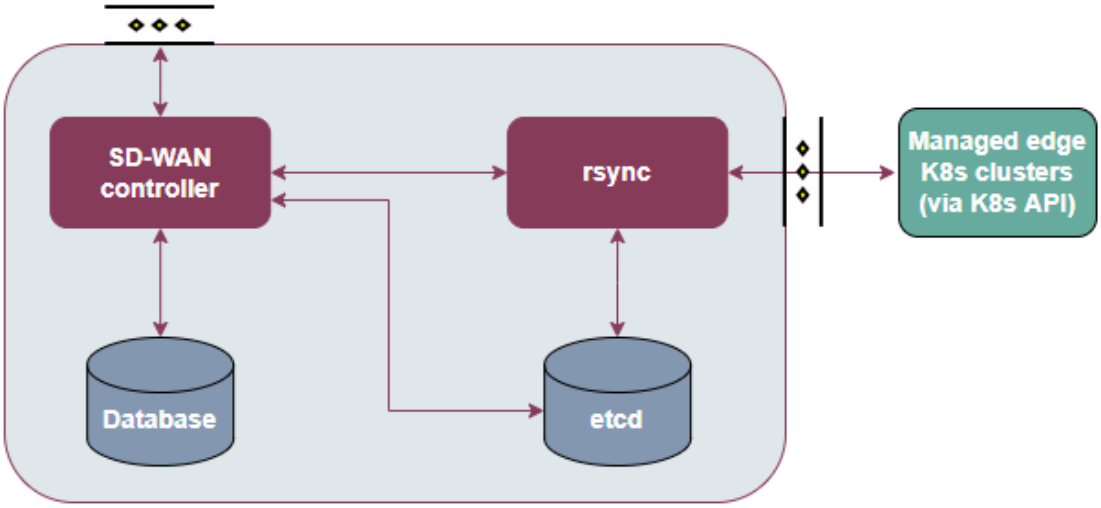


Figure 27. High-level diagram of the SD-WAN enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 30. Components and implementation of the SD-WAN enabler

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

4.1.6.3. Communication interfaces

Table 31. 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.
	/overlays/{id}/proposal	Endpoint in charge of defining IPSec proposals that can be used for tunnels in an overlay.
	/overlays/{id}/hubs	Defines a traffic hub in an overlay. Requires certificate and kubeconfig file to be able to manage it.
	/overlays/{id}/ipranges	Defines the overlay IP range used for the edge clusters.
	/overlays/{id}/devices	Defines an edge cluster location (with WAN Acceleration enabler). Among other input, it required kubeconfig file and certificate information.
	/overlays/{id}/hubs/{id}/devices{id}	Defines a connection between a hub and an edge cluster.

4.1.6.4. Enabler stories

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

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

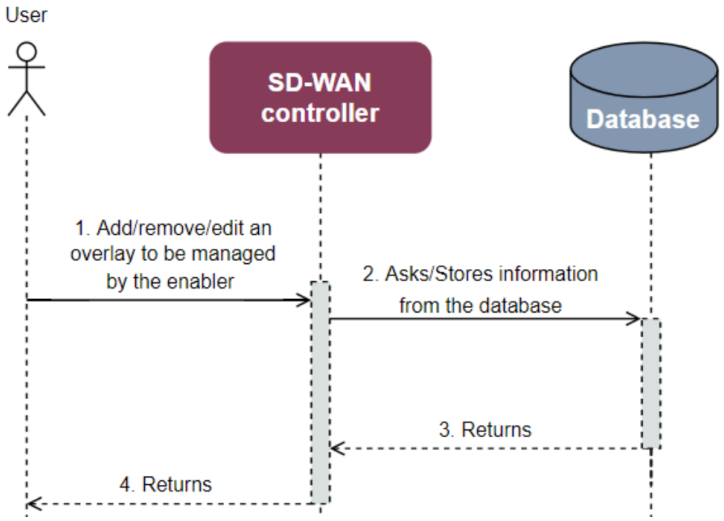


Figure 28. SD-WAN enabler ES1 (overlay management)

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 **enabler stories** related **to definition of IP ranges** to be used for the connections, and **IPSec configuration proposals** for an overlay.

The **second enabler story** 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.

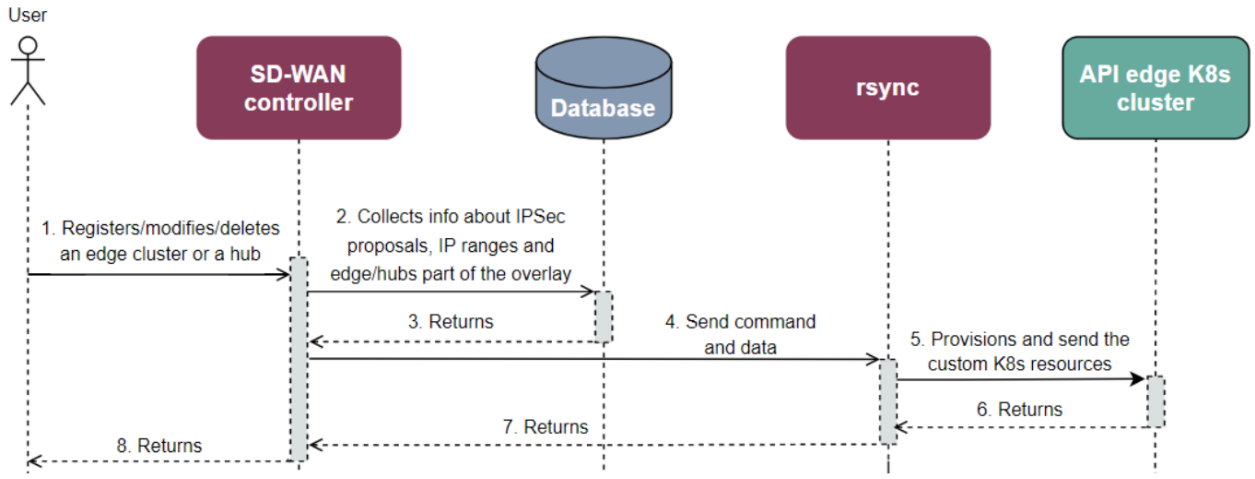


Figure 29. SD-WAN enabler ES2 (tunnel establishment)

The **third and last enabler story** 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.

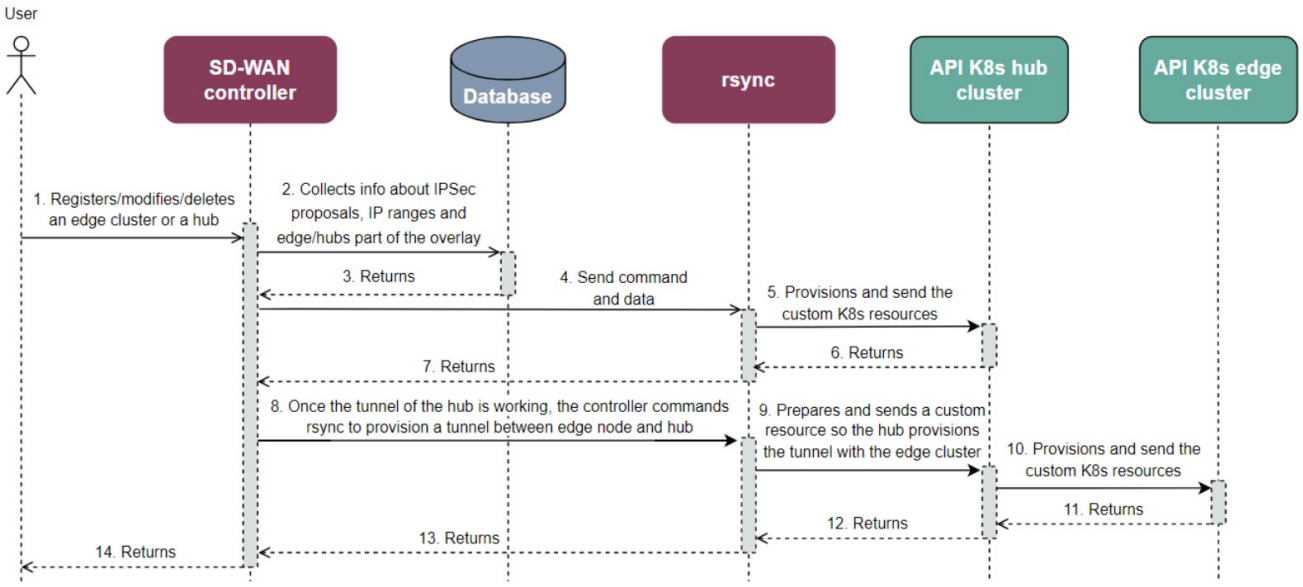


Figure 30. SD-WAN enabler ES3 (connection of hubs with edge cluster)

4.1.6.5. Implementation information

Table 32. Implementation status of the SD-WAN enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/sd_wan_enabler.html
Potential features	This enabler cloud be combined with other enablers in the project to manage inter-cluster communication.
Encapsulation readiness	All components are encapsulated in a helm chart ready to deploy the enabler.
Integration with other enablers	This enabler was specifically designed to integrate with the WAN Acceleration enabler.

4.1.7. WAN acceleration enabler

4.1.7.1. General specifications and features

Table 33. General information of WAN acceleration enabler

Enabler	WAN acceleration enabler
Id	T42E7
Owner and support	UPV
Description and main 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.

Enabler	WAN acceleration enabler
Key features	Works jointly with the previous enabler (T42E6) to provide the same features (see table 29)
Plane/s involved	The WAN Acceleration enabler is located in the Smart Network and Control plane of the ASSIST-IoT architecture. In particular, it belongs to the building block related to VNFs, specifically (i) for provisioning private networks over public ones, jointly with the SD-WAN enabler, and (ii) for supporting VNFs chaining (containerised, thus CNFs).
Requirements mapping	• R-C-10: Transmission security • R-C-11: Network optimisation
Use case mapping	This enabler will grant a secure and optimised connection for applications and services from different sites. Since Pilot 3b, that could be the target of this enabler, has a cloud managed by a third-party, the enabler cannot be exploited as some network prerequisites cannot be met. • UC-P3B-1: Vehicle's exterior condition documentation • UC-P3B-2: Exterior defects detection support
Internal components	• SD-WAN CRD controller • SD-WAN CNF • API

4.1.7.2. Structure, components and implementation technologies

The structure diagram of the enabler is presented in the Figure 31. High-level diagram of WAN acceleration enabler. 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. In addition, the enabler has an API to interact with it.

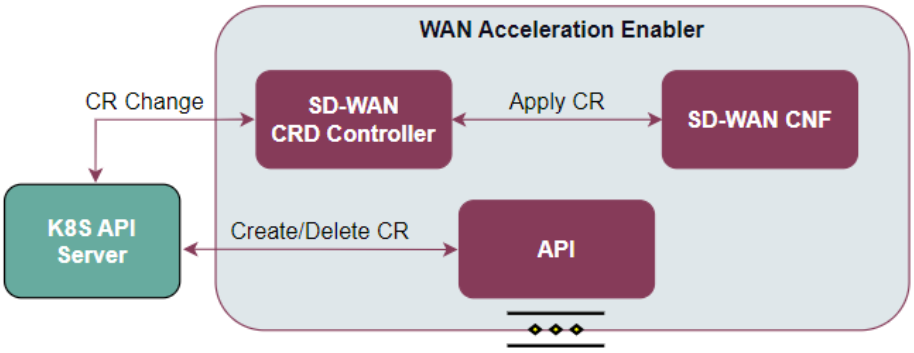


Figure 31. High-level diagram of WAN acceleration enabler

As aforementioned, the WAN Acceleration enabler is composed of three main elements, as one can see in the figure below:

Table 34. Components and implementation of the WAN acceleration enabler

Component	Description	Technology/s
SD-WAN CRD Controller	Component that will receive API calls from the K8s API of the cluster to configure the CNF component.	ovn4nfv-k8s-plugin, k8s custom resource definition controller
SD-WAN CNF	The CNF will embed functions to setup aspects such related to IPSec, firewalling, DNS, DHCP, and WAN link management, exposing and API to be controller/queried.	OpenWRT, IPSec
API	The API component contains an easy-to-use interface to create, list or delete all configuration related to internal management, such as firewall rules or mwan3 policies. This component interacts directly with the K8s API server rather than with other components.	Python

4.1.7.3. Communication interfaces

Table 35. API of the WAN acceleration enabler

Method	Endpoint	Description
GET/POST/DELETE	**/api/v1/firewall/zones/{zone-name}**	To create, list or delete firewall zones in which to include rules for inter-cluster traffic.
GET/POST/DELETE	**/api/v1/firewall/snats/{snat-name}**	To create, list or delete firewall snat for cluster configuration.
GET/POST/DELETE	**/api/v1/firewall/dntas/{dnat-name}**	To create, list or delete firewall dnat for cluster configuration.
GET/POST/DELETE	**/api/v1/firewall/forwarding/ {forwarding-name}**	To create, list or delete firewall forwarding for cluster configuration.
GET/POST/DELETE	**/api/v1/firewall/rules/{rule-name}**	To create, list or delete firewall rules for cluster configuration.
GET/POST/DELETE	**/api/v1/mwan3/policies{policy-name}**	To create, list or delete mwan policies for the cluster configuration.
GET/POST/DELETE	**/api/v1/mwan3/rules/{rule-name}**	To create, list or delete mwan rules for the cluster configuration.
GET	**/api/v1/version**	Get version of the enabler deployment
GET	**/api/v1/health**	Get health status of the enabler deployment
GET	**/api/v1/api-export**	Get API swagger

4.1.7.4. Enabler stories

The **first enabler story** is related to the depicted endpoints will always follow the same flow, either for configuring or querying the CNF components. The uses cases are related to WAN interfaces, policies, firewall and MWAN3 as explained before and the diagram involved steps are the following:

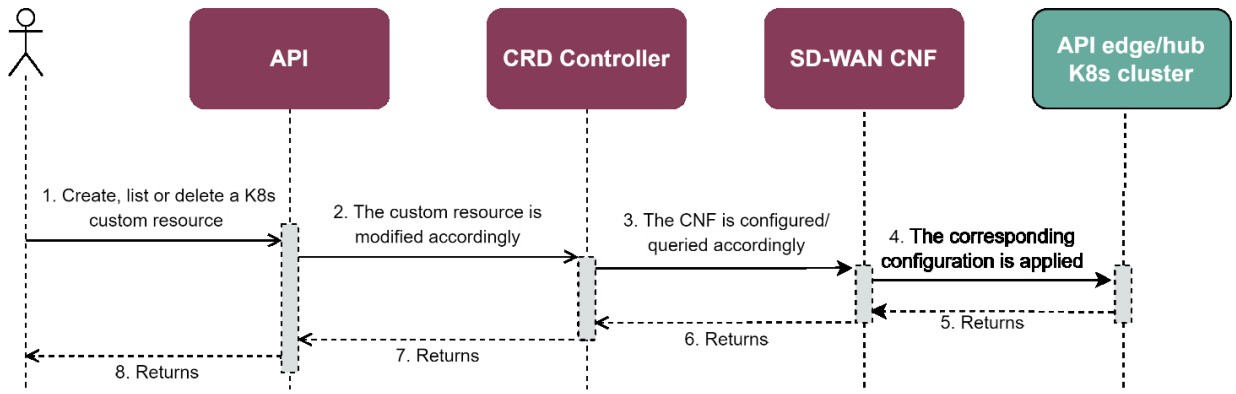


Figure 32. WAN acceleration enabler ES1 (configuring/querying the CNF)

STEP 1: The user interacts with the WAN Acceleration API to create, list or delete firewall/mwan3 custom resource developed for the enabler.

STEP 2: The petition is sent from the API to the CRD controller and modified accordingly.

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

STEP 4: The WAN Acceleration enabler applies the configuration via the API edge/hub K8s cluster.

STEPS 5-8: Once the process has finished, the WAN Acceleration API will return the confirmation message, based on the response from K8s cluster API message.

The **second enabler story** is related to get the version, health and API swagger of the enabler (common endpoints). The steps are the following:

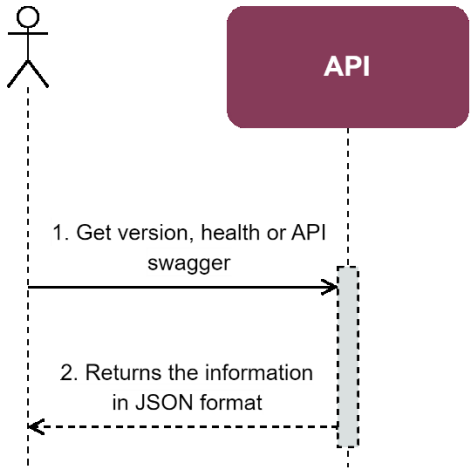


Figure 33. WAN acceleration enabler ES2 (querying the common endpoints)

STEP 1: The user sends a request to the API endpoint to receive the common endpoints data such as the versions available, health status or API swagger information of each version.

STEP 2: The API returns the information to the user in JSON format.

4.1.7.5. Implementation information

Table 36. Implementation status of the WAN acceleration enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/wan_acceleration_enabler.html
Potential features	This enabler cloud be combined with other enablers in the project to manage inter-cluster communication.
Encapsulation readiness	All components are encapsulated in a helm chart ready to deploy the enabler.
Integration with other enablers	This enabler was specifically designed to integrate with the SD-WAN enabler.

4.1.8. VPN enabler

4.1.8.1. General specifications and features

Table 37. General information of the VPN enabler

Enabler	VPN enabler
Id	T42E8
Owner and support	UPV
Description and main functionalities	This enabler will facilitate the 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. The site's network will be considered trusted, so VPNs will not be needed to connect nodes or devices that belong to it. It should be highlighted that SD-WAN enabler will be the primarily choice for connecting sites' networks while VPN will (primarily) connect particular external elements to the site's network since VPN lacks both network and application-level performance optimisation, and it requires extensive manual effort to add different sites to the entire WAN.
Key features	• Secure access of devices to a site's network • High scalability deployment for enabling the integration of a very large number of devices
Plane/s involved	Smart network and control plane

Enabler	VPN enabler
Requirements mapping	• R-C-10: Transmission security • R-C-25, Holistic security/privacy approach
Use case mapping	Project's use cases will be executed primarily with on-site networks. However, in case that any device with external connectivity is needed to be integrated, the VPN enabler will be used to boost security.
Internal components	API, VPN server

4.1.8.2. Structure, components and implementation technologies

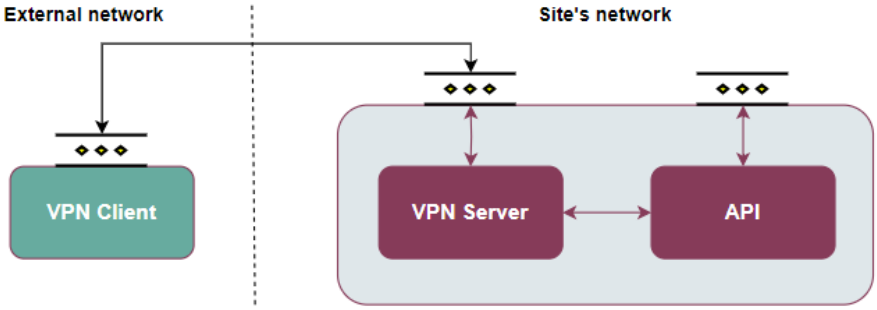


Figure 34. High-level diagram of the VPN enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 38. Components and implementation of the VPN enabler

Component	Description	Technology/s
API	This component allows users to configure the VPN server and manage the clients, interfaces and keys.	Node.js, Express
VPN server	Component that will setup the VPN tunnels with the clients and support the traffic from/to the connected clients.	Wireguard
VPN client	Installed in the devices that want to connect to the network site, it has to be compatible with the technology used for the server side.	Wireguard

NOTE: Wireguard has been selected for performance and scalability reasons, however, other technologies could have been chosen.

4.1.8.3. Communication interfaces

Two interfaces are exposed in this enabler, the API so users can configure and manage it, and the VPN server itself, which must be exposed to accept the connections from the external devices.

Table 39. 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 40. Communication interface (UDP) of the VPN enabler

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

4.1.8.4. Enabler stories

The enabler stories remain unchanged with respect to D4.2. The **first enabler story** 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.

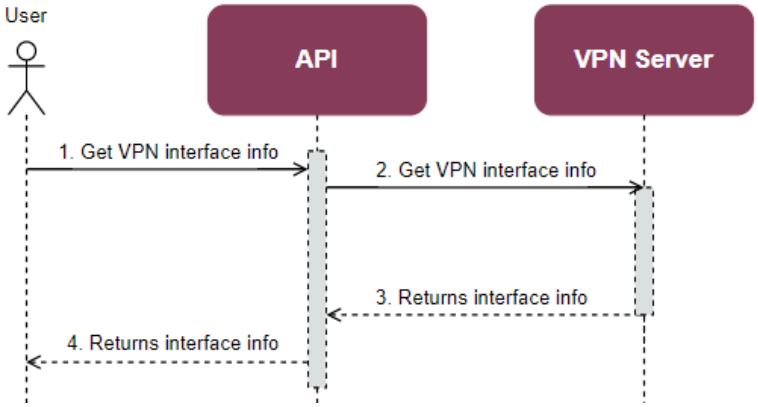


Figure 35. VPN enabler ES1 (get network interface information)

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

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

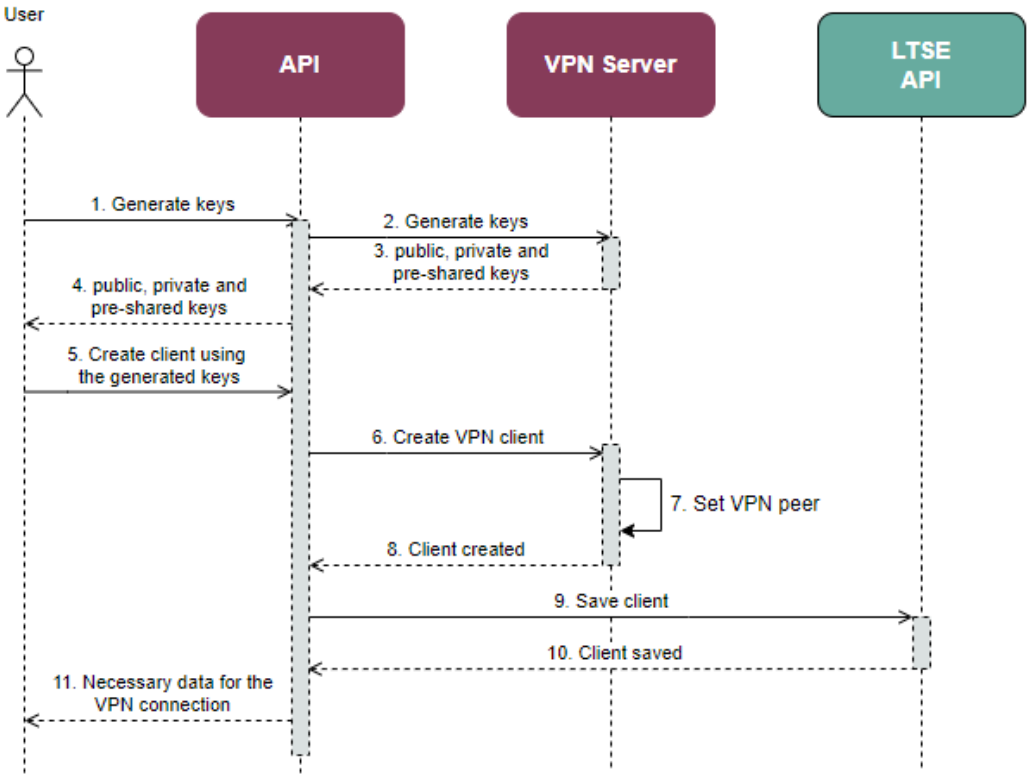


Figure 36. VPN enabler ES2 (create client)

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.

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

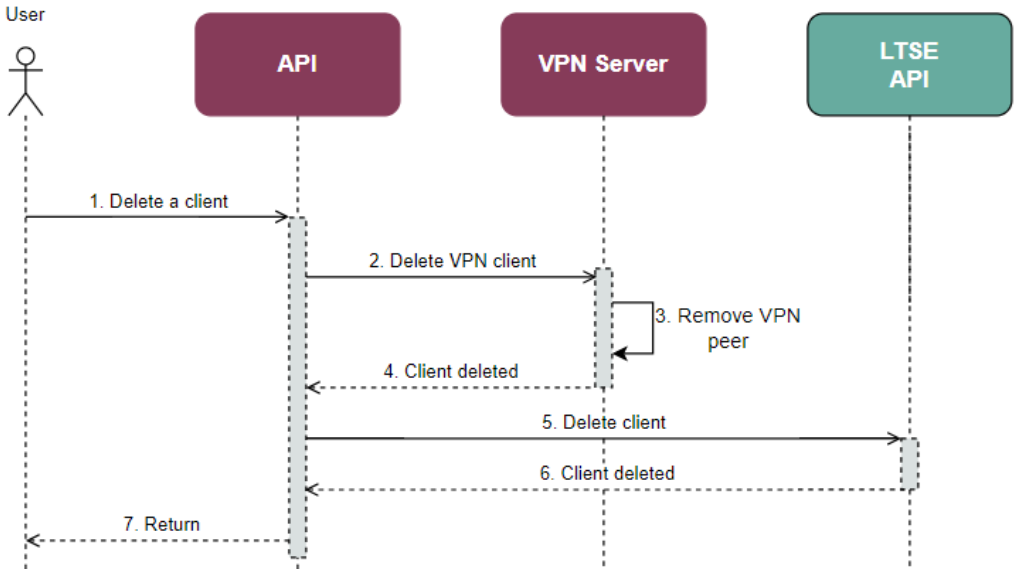


Figure 37. VPN enabler ES3 (delete client)

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.

The **fourth enabler story** 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:

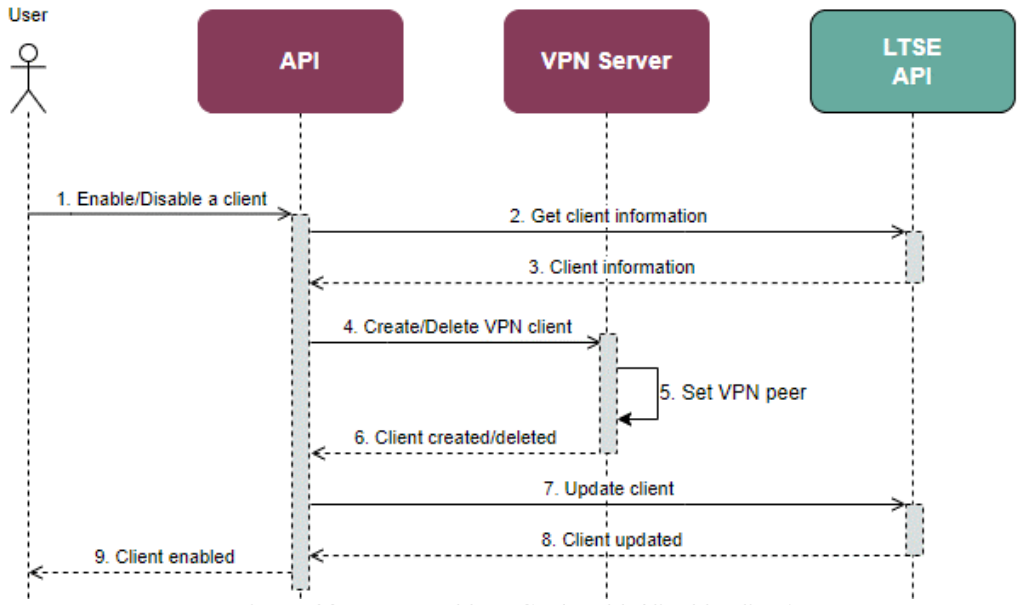


Figure 38. VPN enabler ES4 (enable/disable client)

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 data, 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.

The **fifth (last) enabler story** 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:

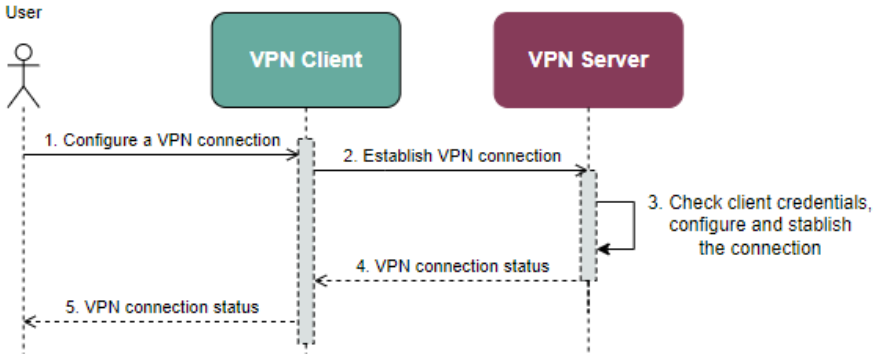


Figure 39. VPN enabler ES5 (connect client)

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.

4.1.8.5. Implementation information

Table 41. Implementation status of the VPN enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/vpn_enabler.html
Potential features	Any additional feature is foreseen to improve this enabler.
Encapsulation readiness	Full functional Helm package ready
Integration with other enablers	The VPN enabler can work in a standalone fashion, however, an integration with the LTSE has been made to use it as backup of the tunnel and clients information.

4.2. Data Management enablers

4.2.1. Semantic repository enabler

4.2.1.1. General specifications and features

Table 42. General information of the Semantic repository enabler

Enabler	Semantic Repository (SemRepo)
Id	T43E1
Owner and support	SRIPAS
Description and main 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. It supports files that describe data models or data transformations, such as ontologies, schema files, semantic alignment files, etc. Also, human-readable documentation for the models is served. This enabler is designed as a public (i.e. network-wide) source of data models (with metadata) to facilitate data interoperability and sharing in a semantic ecosystem.
Key features	• Versioning: different versions of data models, • Metadata: arbitrary information about the models can be stored. • Provision & search: data models are public and browsable, • Documentation: automatically compiled from source files to HTML and served to end users. • Webhooks: notifying other actors about changes to the models.
Plane/s involved	Data Management Plane
Requirements mapping	R-P2-15, R-C-1, R-C-2, R-C-6, R-C-14
Use case mapping	UC-P2-1, UC-P2-2, UC-P2-3, UC-P2-4, UC-P2-5, UC-P2-6
Internal components	API server, database, file storage

4.2.1.2. Structure, components and implementation technologies

The Semantic repository has three components, where the API server serves as a gateway to the rest of the enabler.

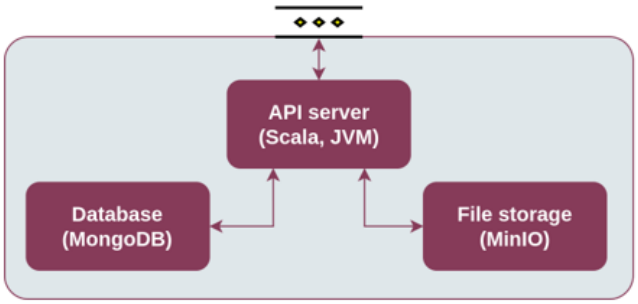


Figure 40. High-level diagram of the Semantic repository enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 43. Components and implementation of the Semantic repository enabler

Component	Description	Technology/s
API server	Provides a high-performance streaming HTTP API for the enabler, based on REST principles. Handles all user requests and performs background maintenance tasks.	Scala, Akka, Akka Streams, Java Virtual Machine
Database	Stores the information about the models, documentation, metadata, webhooks, and other. Highly scalable.	MongoDB
File storage	Stores the actual models. Supports storage tiering and is highly scalable.	MinIO

4.2.1.3. Communication interfaces

This enabler communicates through the REST API, organised by namespaces and model identifiers. Each model (i.e. the stored piece of data) is contained within a namespace, has a version ID, and may include more than one file format (for a single data model). 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.

As the Semantic Repository enabler has a rich API, only the most representative portion of it is presented here. The full REST API documentation can be found in the enabler documentation: https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_repository_enabler.html#rest-api-reference.

Table 44. API of the Semantic repository enabler

Method	Endpoint	Description
GET	**/v1/m**	Lists available repositories.
POST/PUT/ DELETE	**/v1/m/{namespace id}**	Creates (POST), updates (PATCH), or removes (DELETE) a specified namespace and its settings.
GET	**/v1/m/{namespace id}**	Returns the settings of the namespace and lists models in it.
GET	**/v1/m/{namespace id}/{model id}**	Returns the metadata of the model and lists the available versions of the given model.
POST/PUT/DELETE	**/v1/m/{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	**/v1/m/{namespace id}/{model id}/{version id}**	Returns the metadata of the given model version.
POST/ DELETE	**/v1/m/{namespace id}/{model id}/{version id}/content? format={data format}**	Sets (POST) or removes (DELETE) a specified file from the server.
GET	**/v1/m/{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	**/v1/m/{namespace id}/{model id}/{version id}/doc_gen?plugin={plugin}**	Uploads source files for documentation compilation with the specified compilation plugin.
GET	**/v1/m/{namespace id}/{model id}/{version id}/doc/{file name}**	Returns the documentation for a model.
DELETE	**/v1/m/{namespace id}/{model id}/{version id}/doc**	Deletes the documentation for a model.
POST	**/v1/doc_gen?plugin={plugin}**	Requests a compilation of a set of documentation source files in « sandbox » mode, with a given

Method	Endpoint	Description
		compilation plugin. The ID of the new job is returned to the user.
GET	**/v1/doc_gen/{job_id}**	Returns the status of a documentation compilation job with the given ID.
GET	**/v1/doc_gen/{job_id}/content/{file}**	Returns the content of a given output file for a documentation compilation job.
POST	**/v1/webhook**	Creates a new webhook.
GET	**/v1/webhook**	Retrieves the list of registered webhooks.
GET/DELETE	**/v1/webhook/{webhook_id}**	Retrieves the details of a given webhook (GET) or deletes it (DELETE).

4.2.1.4. Enabler stories

The **first enabler story** 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:

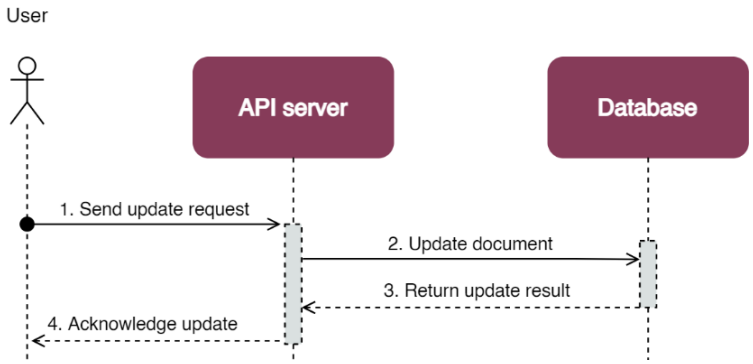


Figure 41. Semantic repository enabler ES1 (modify metadata)

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 enabler story** 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.

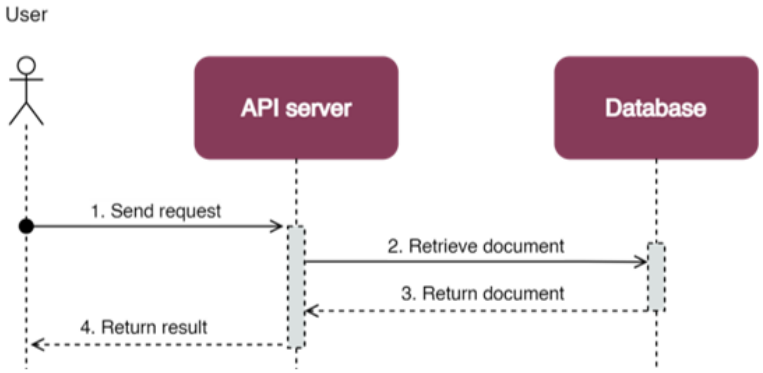


Figure 42. Semantic repository enabler ES2 (get metadata)

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 enabler story** 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.

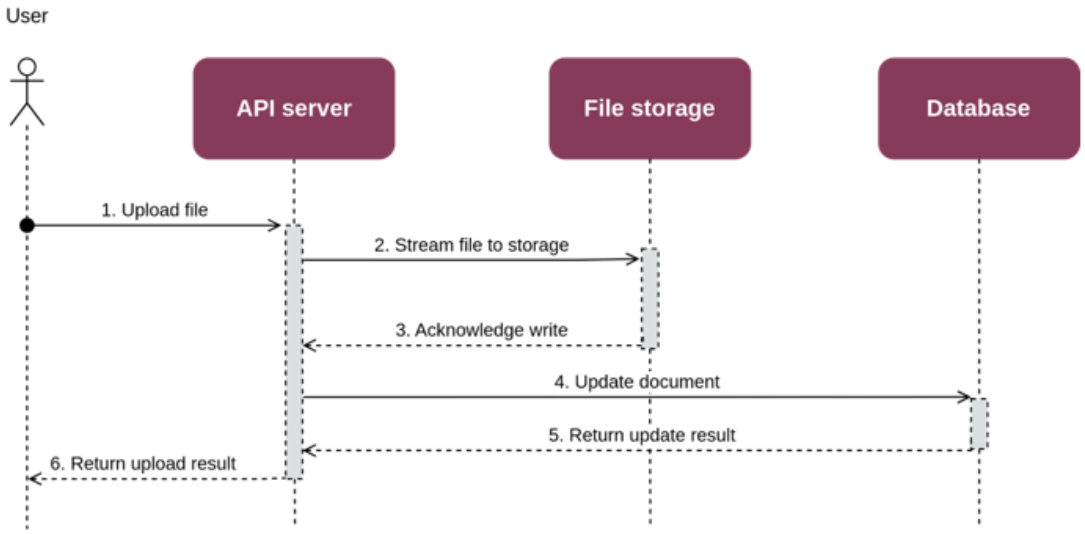


Figure 43. Semantic repository enabler ES3 (upload file with model)

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).

The **fourth enabler story** 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.

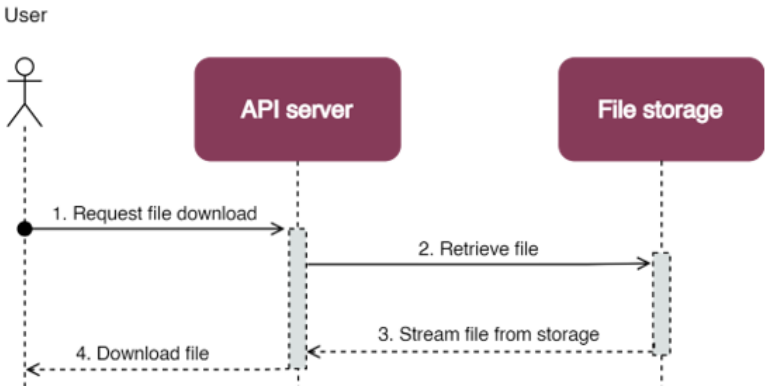


Figure 44. Semantic repository enabler ES4 (get file with model)

The **fifth enabler story** is about **uploading source documentation files**. The steps are the same for both documentation assigned to a specific model and for the documentation sandbox. The only difference is in the

used API endpoints and the internal data structures in the database. Although the process described here relies on a series of steps, in reality the system heavily uses asynchronous messaging and pipelined streams. Thus, the whole process is very efficient and avoids caching the documentation files in memory.

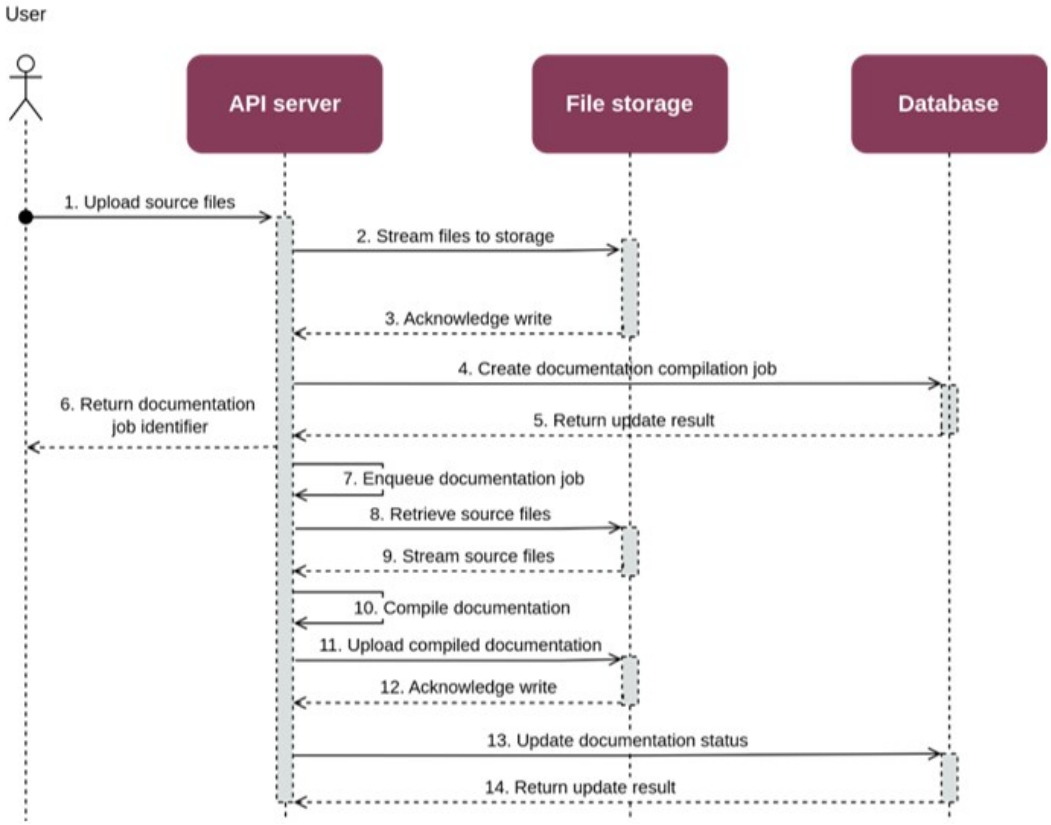


Figure 45. Semantic repository enabler ES5 (upload documentation)

STEP 1: The user sends an HTTP request to the API server. The request contains the compressed source files and instructions for which plugin to invoke.

STEPS 2-3: The source files are saved to file storage for future use.

STEPS 4-5: A new documentation compilation job is created in the database. The job is assigned a random, unique ID.

STEP 6: The user gets the response acknowledging the start of documentation compilation. The response includes the compilation job ID, which can later be used to check the status of the job (use case 6).

STEP 7: The API server adds the compilation job to the immediate job queue.

STEPS 8-9: The appropriate compilation plugin picks up the job from the queue and retrieves the needed files from storage (streaming).

STEPS 10-12: The plugin compiles the documentation and streams the result to file storage.

STEPS 13-14: The status of the documentation job is updated in the database.

The last **(sixth) enabler story** is related to retrieving the status of a previously requested documentation compilation job. This is applicable to both documentation associated with a specific model and the documentation sandbox.

STEP 1: The user requests the status of a documentation job with a given ID. The ID was previously given to the user when requesting the job (use case 5).

STEPS 2-3: The API server checks the status of the documentation job in the database.

STEP 4: The status is returned to the user, including the information on whether it is in progress, succeeded, or failed. If the job failed, an error message is included.

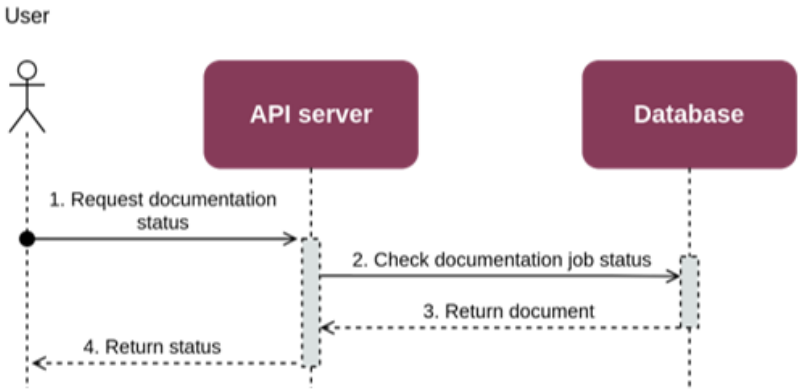


Figure 46. Semantic repository enabler ES6 (check documentation job status)

4.2.1.5. Implementation information

Table 45. Implementation status of the Semantic repository enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_repository_enabler.html
Potential features	The enabler is considered feature-complete for the purposes of the project. However, in the future the security, interoperability and documentation pluggability mechanisms could be expanded or improved.
Encapsulation readiness	A single Helm chart for the whole enabler, including all components.
Integration with other enablers	Enabler can be used in standalone mode, without other enablers.

4.2.2. Semantic translation enabler

4.2.2.1. General specifications and features

Table 46. General information of the Semantic translation enabler

Enabler	Semantic translation enabler (SemTrans)
Id	T43E2
Owner and support	SRIPAS
Description and main functionalities	This enabler offers a configurable service to change the contents of semantically annotated data following translation rules (so-called "alignments"). The core use case is to translate data semantics between 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 pub-sub broker for streaming data
Key features	• Transformation of semantic data

Enabler	Semantic translation enabler (SemTrans)
	• Uses RDF as a standard semantic file format • Supports translation via streaming or REST API • Flexible architecture supporting n-to-n translation
Plane/s involved	Data Management Plane
Requirements mapping	R-P2-1, R-P2-2, R-P2-7, R-P2-11, R-P2-15
Use case mapping	UC-P2-1, UC-P2-2, UC-P2-3, UC-P2-4, UC-P2-5, UC-P2-6
Internal components	List of the internal components of this enabler

4.2.2.2. Structure, components and implementation technologies

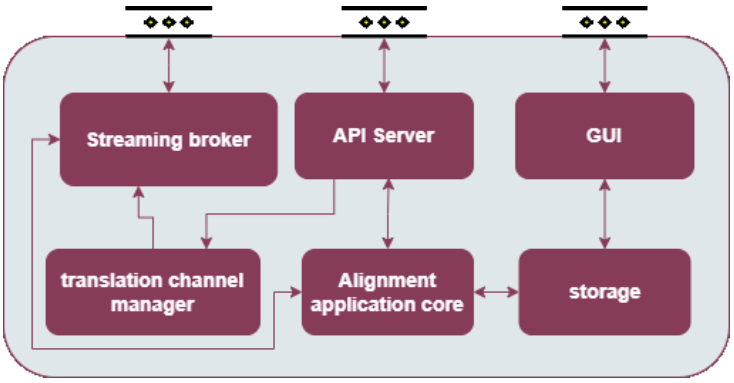


Figure 47. High-level diagram of the Semantic translation enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 47. Components and implementation of the Semantic translation enabler

Component	Description	Technology/s
API Server	Http server for REST API	Pekko Http
Alignment application core	RDF Processing core responsible for parsing and executing alignment files over data	Apache Jena
Translation channel manager	Custom messaging channels management	Scala
Storage	Data Persistence for translation channels configuration and alignment files	PostgreSQL
Streaming Broker	Streaming message broker	Apache Kafka + Verne MQTT
GUI	Web interface	Javascript

4.2.2.3. Communication interfaces

Table 48. API of the Semantic translation enabler – API Server

Component	Description	Technology/s
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

Component	Description	Technology/s
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 49. 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.

4.2.2.4. Enabler stories

The **first enabler story** 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:

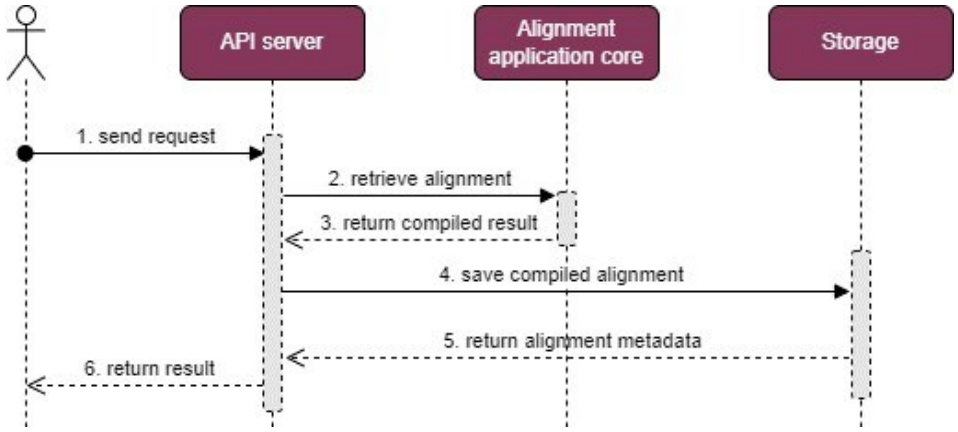


Figure 48. Semantic translation enabler ES1 (store alignment)

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 enabler story** enables a user/client **to read metadata of an alignment** stored in the database. This enabler story has this sequence diagram and steps:

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,

STEPS 3-4: The storage component returns it to the API. Finally, the API server returns the metadata description to the user/client.

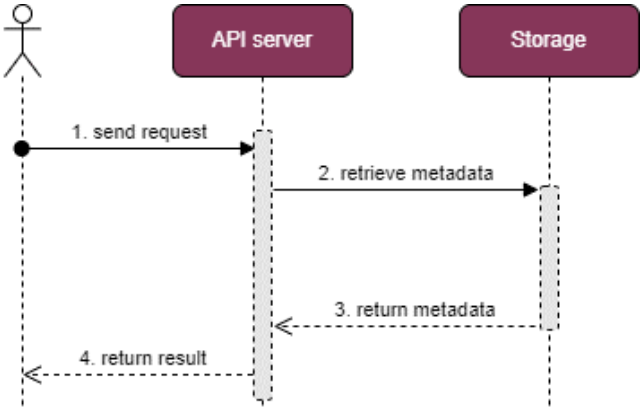


Figure 49. Semantic translation enabler ES2 (get alignment metadata)

The **third enabler story** allows a user/client **to define/create a streaming-based translation channel** using available (compiled) alignments.

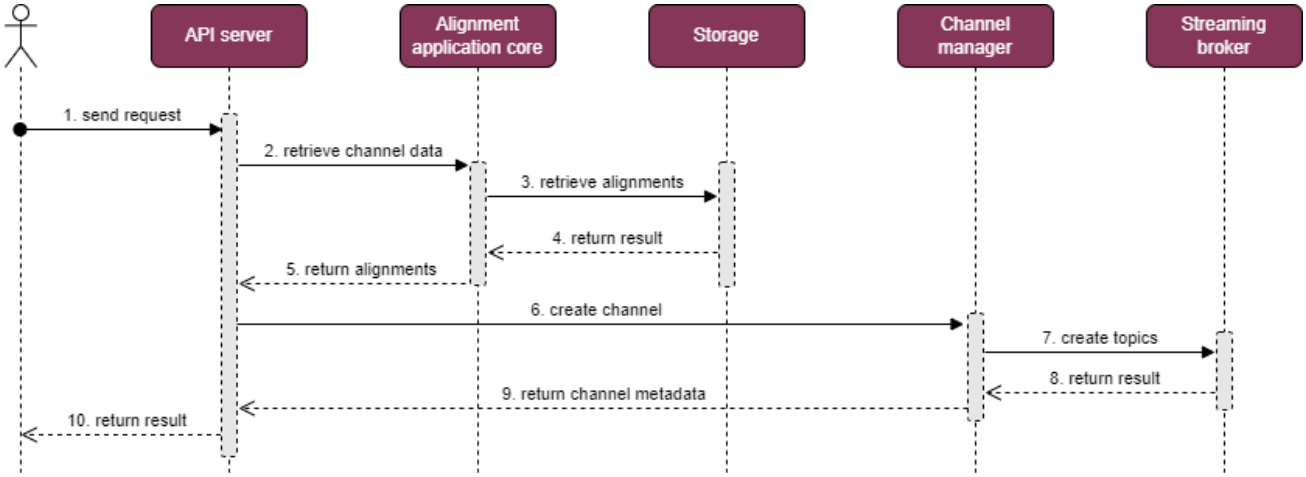


Figure 50. Semantic translation enabler ES3 (create stream-based translation channel)

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.

4.2.2.5. Implementation information

Table 50. Implementation status of the Semantic translation enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_translation_enabler.html
Potential features	Additional features that could be added/extended in the future, now that we have more knowledge about it.

Category	Status
Encapsulation readiness	A single Helm chart for the whole enabler, including all components, with alternative supplementary Docker compose file.
Integration with other enablers	Can be used standalone, or connected with any other enabler that uses the supported streaming technologies, in particular the Semantic Annotation enabler.

4.2.3. Semantic annotation enabler

4.2.3.1. General specifications and features

Table 51. General information of the Semantic annotation enabler

Enabler	Semantic Annotation enabler (SemAnn)
Id	T43E3
Owner and support	P03 IBSPAN
Description and main functionalities	This enabler offers a syntactic transformation service, that annotates data in various formats (JSON, CSV and XML) and lifts it into RDF. Annotation is configured using CARML – a dialect of RML (RDF Mapping Language) designed for streaming annotation. The core functionality is designed to be integrated into a streaming pipeline before the Semantic Translation enabler. However, the enabler can be used by itself to annotate data through streaming technologies, or REST API. A custom annotation channel architecture supports quick creation of lightweight channels with optional outputs for error and monitoring. For very constrained devices, this enabler can be used without the persistence module reducing storage requirements, at the trade-off of not persisting configuration between restarts.
Key features	• Annotation of JSON, CSV and XML into RDF • Lightweight annotation channels architecture for streaming annotation • Standalone mode that reduces storage requirements
Plane/s involved	Data Management Plane
Requirements mapping	R-P2-1, R-P2-2, R-P2-7, R-P2-11, R-P2-15
Use case mapping	UC-P2-1, UC-P2-2, UC-P2-3, UC-P2-4, UC-P2-5, UC-P2-6
Internal components	API Server, Streaming core, Streaming Broker, Configuration Persistence (database)

4.2.3.2. Structure, components and implementation technologies

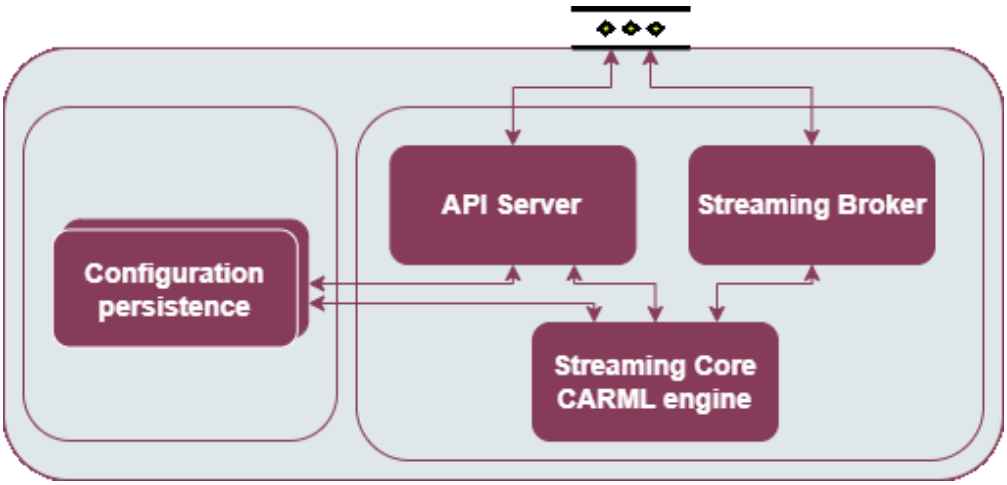


Figure 51. High-level diagram of the Semantic annotation enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 52. Components and implementation of the Semantic annotation enabler

Component	Description	Technology/s
Streaming configuration API Server	Http server for the REST API	Akka Http
CARML engine	Central component responsible for parsing and applying CARML files to transform data	Scala + Java
Streaming core	Custom component that manages the streaming channels	Scala + Akka streams
Configuration Persistence	Database for persisting configuration	MongoDB
Streaming Broker	Streaming Message broker. This enabler can be connected to an external broker and works with MQTT and/or Kafka, supporting both even in a single annotation channel (i.e. input topic and output topic do not need to use the same streaming broker, and can be set up independently on MQTT and/or Kafka).	VerneMQ/Kafka

4.2.3.3. Communication interfaces

The table below presents an overview of the REST interface for the Semantic Annotation Enabler. Full details with all parameter types, example values and detailed descriptions of every endpoint are in the Swagger documentation.

Table 53. API of the Semantic annotation enabler – API server

Method	Endpoint	Description
GET	**/swagger/**	View the REST API documentation and Swagger interface.
GET	**/channels/({channelId})**	Retrieve information about all channels, or a single channel (if the optional channelId is provided). Additional parameters can be used to retrieve information selectively, e.g. ?settings=true returns settings information, ?status=true returns status. Parameters may be used together.
POST	**/channels**	Add a new channel definition, and optionally materialise and start the channel. This endpoint accepts channel configuration in JSON. Depending on initial status (written in the configuration file), the channel may be added, but not started.
PATCH	**/channels/{channelId}**	Updates the channel status with values provided in the channel status object provided in the request body. With the channel status object, the channel can be started or stopped, or error/monitoring topic settings updated (see channel architecture below).
PATCH	**/channels/{channelId}/restart**	Stop and then start a channel.
DELETE	**/channels/{channelId}**	Stop and remove a channel.
GET	**/annotations/({annotationId})**	Retrieve information about all annotations, or a single annotation (if the optional annotationId is provided). Additional parameters are supported, similar to the GET /channels endpoint.
POST	**/annotations**	Add a new annotation to storage.
DELETE	**/annotations/{annotationId}**	Remove an annotation from storage.
GET	**/version**	Returns software version information.
GET	**/status**	Returns global status, including errors, if there are any.
GET	**/settings**	Returns current global settings.

This enabler uses a streaming channel architecture that annotates the messages between a series of topics (see figure below). Messages sent to the input topic are pushed through the channel and processed at different stages. The first "input monitoring" stage outputs a simple message for any message that passes through in order to confirm that a message was received. The processing and error stage attempts to transform the message using

an annotation configuration file (CARML). If unsuccessful, errors are output on the error topic. Otherwise the annotated message passes the "output monitoring" stage (with equivalent functionality to the "input monitoring" stage) and is finally written to the output topic. Input and output topics are always active, provided that the channel is not stopped. Monitoring and error topics can be optionally enabled or disabled, even if the channel is running.

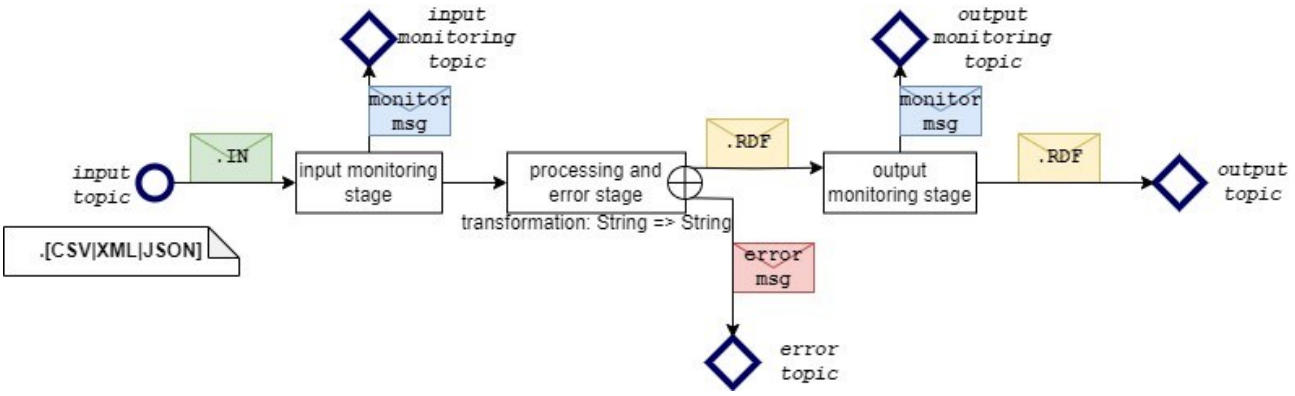


Figure 52. Semantic annotation enabler – annotation channel architecture overview

Table 54. 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 an annotation channel contains only the annotated 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.
Input/Output Monitoring topics	**Multiple topics**	These topics output (independently from each other) information about received messages (currently only timestamp).

4.2.3.4. Enabler stories

The **first enabler story** 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:

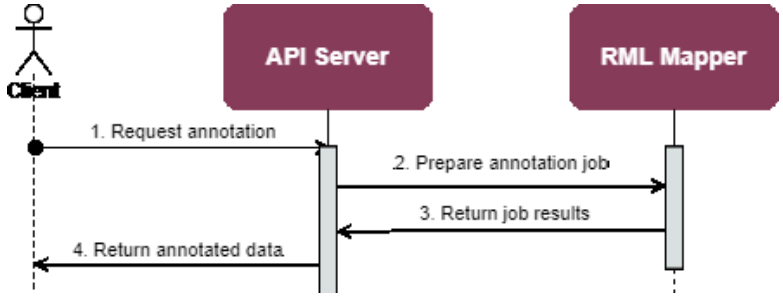


Figure 53. Semantic annotation enabler ES1 (batch annotation)

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 **second enabler story** follows the next sequence diagram:

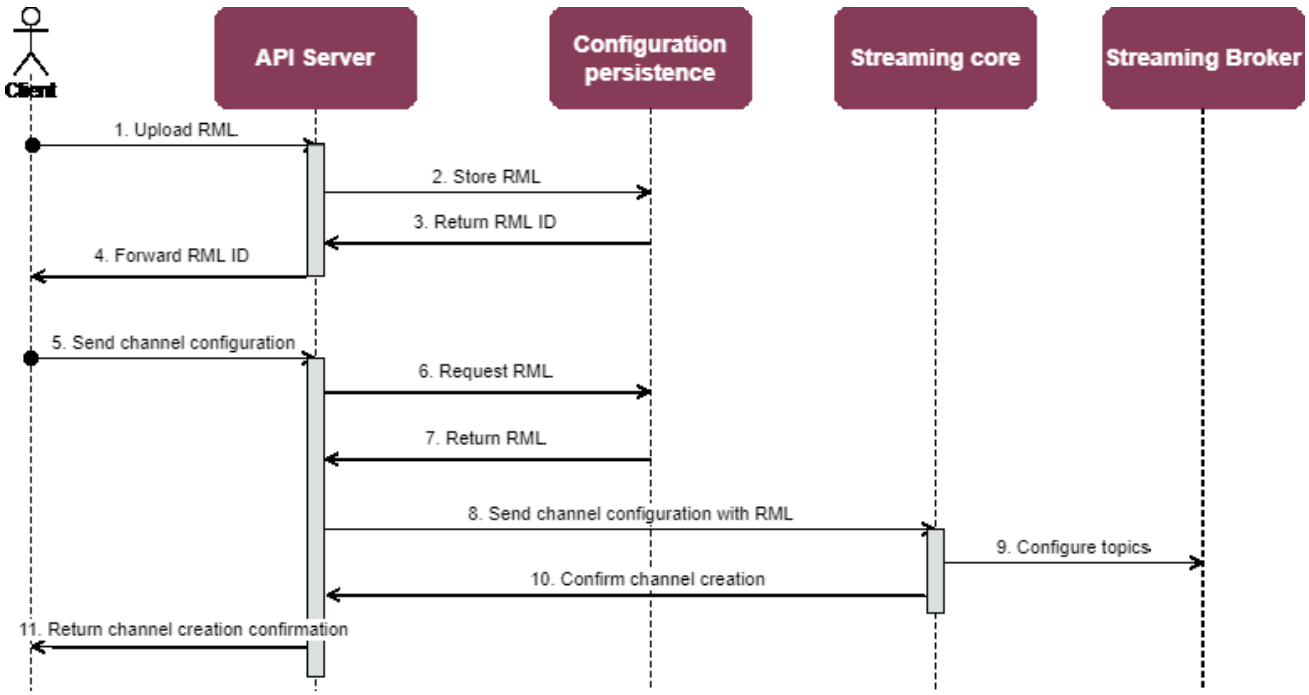


Figure 54. Semantic annotation enabler ES2 (configure channel for stream annotation)

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 materialises 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 **third enabler story** 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 enabler story has the following diagram and involved steps:

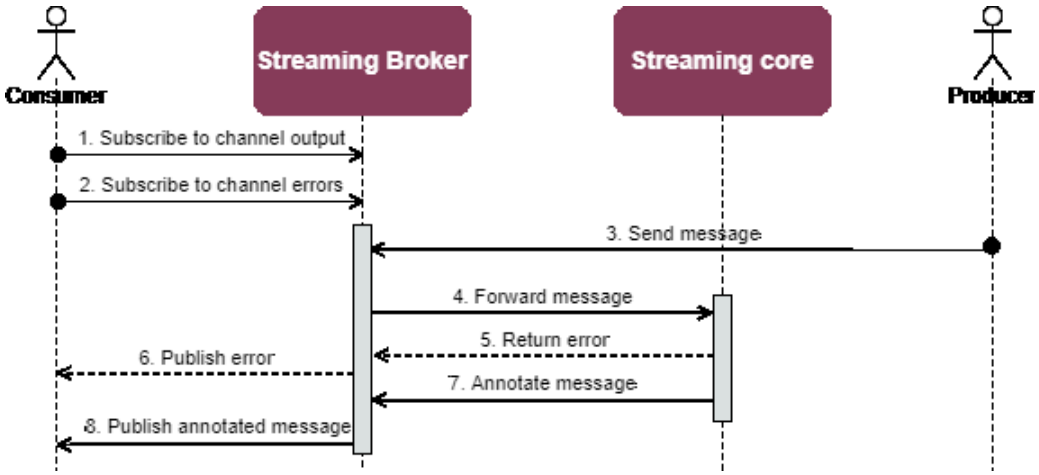


Figure 55. Semantic annotation enabler ES3 (stream annotation)

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.

4.2.3.5. Implementation information

Table 55. Implementation status of the Semantic annotation enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/semantic_annotator_enabler.html
Potential features	Additional features that could be added/extended in the future, now that we have more knowledge about it.
Encapsulation readiness	A single Helm chart for the whole enabler, including all components, with alternative supplementary Docker compose file.
Integration with other enablers	Can be used standalone, or connected with any other enabler that uses the supported streaming technologies, in particular the Semantic Translation enabler.

4.2.4. Edge data broker

4.2.4.1. General specifications and features

Table 56. General information of the Edge data broker

Enabler	Edge Data Broker (EDB)
Id	T43E7
Owner and support	ICCS

Enabler	Edge Data Broker (EDB)
Description and main functionalities	The Edge Data Broker enables the efficient management of data demand and data supply among edge nodes based on a publish/subscribe schema, taking 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.
Key features	• Subscriptions and messages between Edge Devices through the Edge Data Broker enabler • Management and distribution of messages using delivery mechanisms • Common interfaces for filtering messages • Integration with other data brokers if needed
Plane/s involved	Data Management Plane
Requirements mapping	R-C-2: Data governance
Use case mapping	All pilots make use of this enabler. Particularly, the use cases UC-P1-1 to 5 from Pilot 1, all UCs from Pilot 2 and 3a, and UC-P3B-1 from Pilot 3b.
Internal components	MQTT Broker, FR-Script (Filtering & Ruling Script), Auth Database, MQTT-Explorer

4.2.4.2. Structure, components and implementation technologies

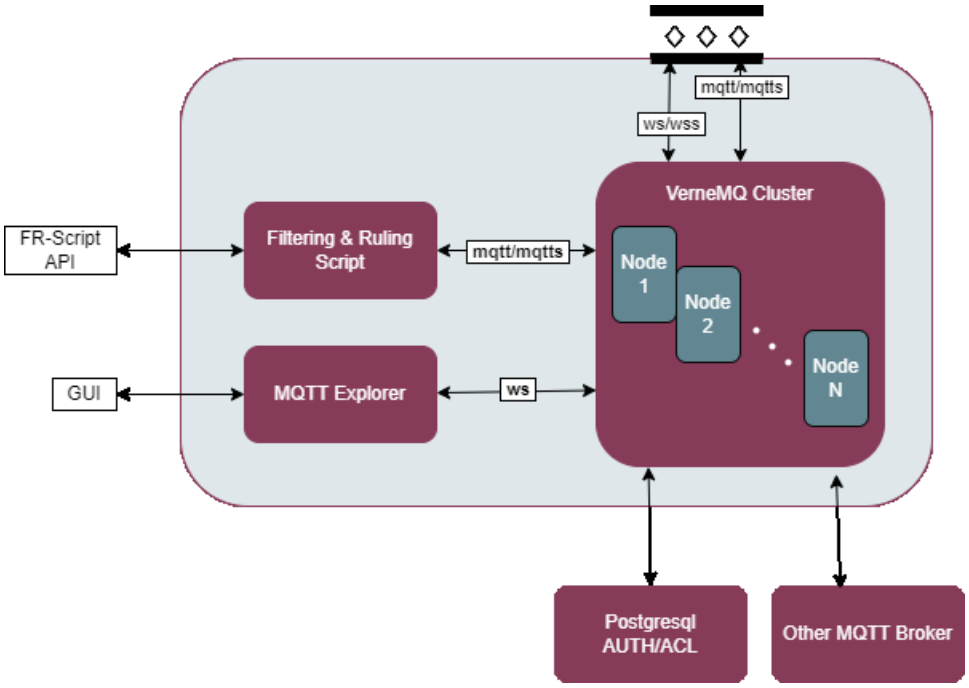


Figure 56. High-level diagram of the Edge data broker

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 57. Components and implementation of the Edge data broker

Component	Description	Technology/s
MQTT Broker	A high-performance, distributed MQTT broker	VerneMQ
FR-Script	A custom script that provides the ability to filter selected topics based on conditions and logical operation rules defined by the user.	Python
Auth Database	Relational database for storing user credentials and Access Control Lists	PostgreSQL
MQTT-Explorer	A GUI that provides a structured overview of MQTT topics	Typescript

4.2.4.3. Communication interfaces

Table 58. Communication interfaces of the Edge data broker – MQTT Broker

Method	Endpoint	Description
Pub/Sub	**Multiple topics (mqtt, mqtts)**	Subscribe to an output topic or publish to an input topic through MQTT or MQTTS protocols
Pub/Sub	**Multiple topics (ws, wss)**	Subscribe to an output topic or publish to an input topic through websockets (ws) or secured websockets (wss)
GET	**:8888/status**	Web GUI for VerneMQ's Cluster and Node(s) status
GET	**:8888/metrics**	VerneMQ's metric exporter for PUD's Prometheus

Table 59. API of the Edge data broker – FR-Script

Method	Endpoint	Description
GET	**:8000/metrics**	FR-Script's metric exporter for PUD's Prometheus
GET	**:9877/docs**	Display FR-Script's Swagger GUI (Listing the bellow APIs)
GET	**/**	Get all filters & rules
POST	**/**	Create filters & rules
GET	**/filters**	Get all filters
GET	**/filter/{id}**	Get filter
PATCH	**/filters**	Update filters
DELETE	**/filter/{id}**	Delete filter
GET	**/rules**	Get rules
GET	**/rule/{id}**	Get rule
PATCH	**/rules**	Update rules
DELETE	**/rule/{id}**	Delete rule

4.2.4.4. Enabler stories

The **first enabler story** describes the usage of Edge Data Broker enabler's FR-Script and its **filtering capabilities**. In this scenario's example we have four external clients connected to EDB of which two are publishers and two are subscribers.

STEP 1: All clients get connected to EDB's VerneMQ cluster. FR-Script works as Publisher/Subscriber client, subscribe to topic (#).

STEP 2: Subscribers 1 and 2 subscribe to topics "test/+/alert" and "test/+" respectively.

STEP 3: Publisher 1 publishes a message to topic "test/1". The message(topic(test/1)) is sent to FR-Script and subscriber 2.

STEP 4: FR-Script check its filtering statements of the corresponding topic and if the conditions set by the user, result true FR-Script creates a new topic appending a new subtopic in the existing topic tree and publishes a new or the same payload, depending on its configuration. In this case the conditions' result is False, so nothing happens.

STEP 5: Publisher 2 publishes a message to topic "test/2". The message(topic(test/2)) is sent to FR-Script and subscriber 2.

STEP 6: FR-Script repeats STEP 4 and this time the conditions results True. FR-Script creates topic "test/2/alert" as it is configured by the user and publishes a message.

STEP 7: The message is sent to FR-Script and Subscriber 1 subscribed to topics "#" and "test/+/alert".

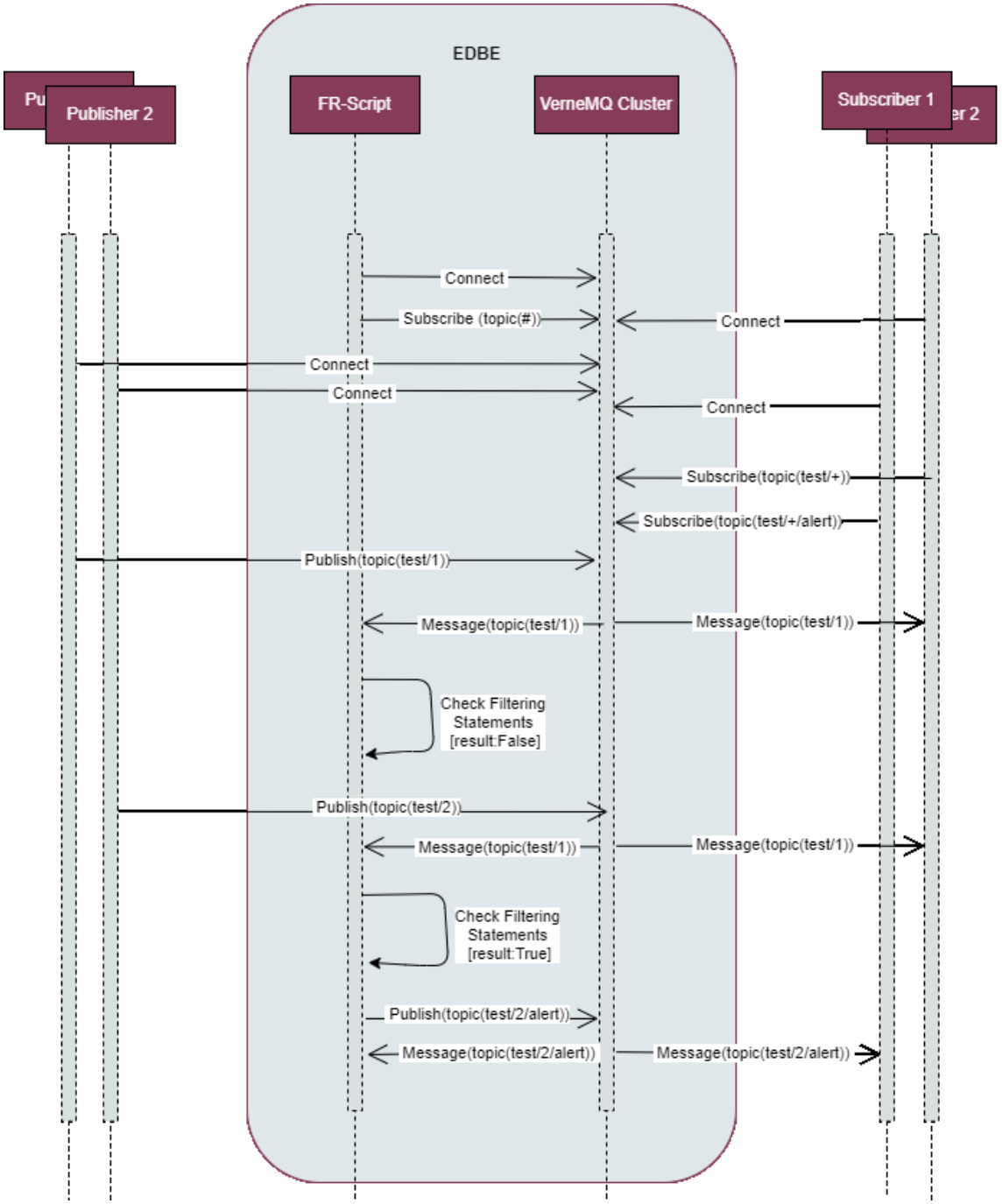


Figure 57. Edge data broker ES1 (filtering)

The **second enabler story** describes the usage of Edge Data Broker enabler's FR-Script and its **ruling capabilities**. In this scenario's example we have five external clients connected to EDB of which three are publishers and two are subscribers.

STEP 1: All clients get connected to EDB's VerneMQ cluster. FR-Script works as Publisher/Subscriber client, subscribe to topic(#).

STEP 2: Subscribers 1 and 2 subscribe to topics "!action/1" and "!action/2" respectively.

STEP 3: Publisher 1 publishes a message to topic "test/1". The message(topic(test/1)) is sent to FR-Script.

STEP 4: FR-Script check its ruling statements of the corresponding topic (in our case can be ether "test/+" or "test/#"). Then saves the message's topic and payload and checks if there are other defined topics in its statements. If there are and there are no saved payloads for them, waits for a message published on the rest of them. So, it waits.

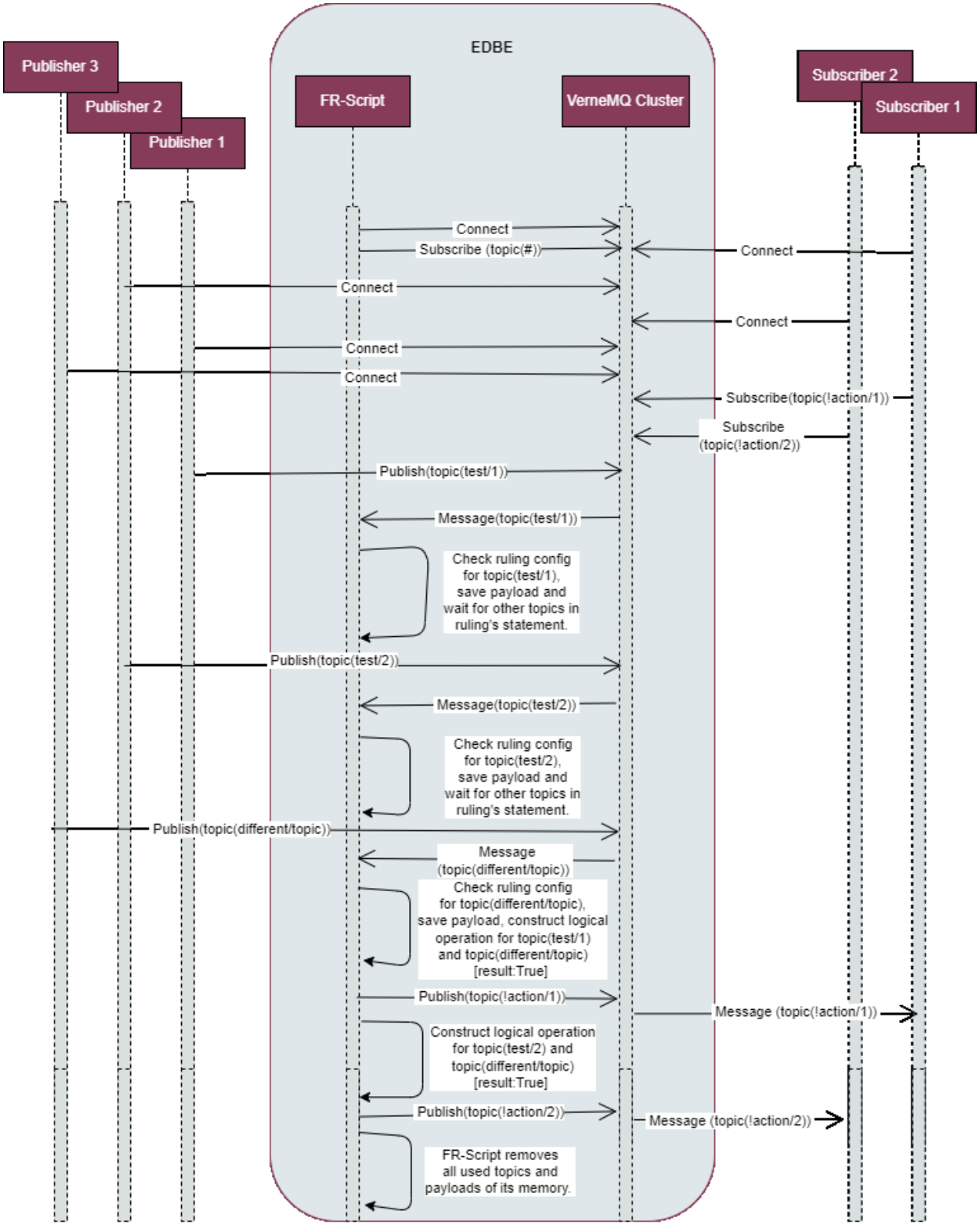


Figure 58. Edge data broker ES2 (ruling)

STEP 5: Publisher 2 publishes a message to topic "test/2". The message(topic(test/2)) is sent to FR-Script.

STEP 6: Repeat STEP 4.

STEP 7: Publisher 3 publishes a message to topic "different/topic". The message(topic(different/topic)) is sent to FR-Script.

STEP 8: FR-Script check its ruling statements of the corresponding topic. Then saves the message's topic and payload and checks if there are other defined topics in its statements. This time all defined topics have corresponding payloads saved on FR-Script so it constructs a logical operation based on logic conditions set by the user in its configuration for topics "test/1" and "different/topic". The logical operation results True.

STEP 9: FR-Script publishes a message on topic "!action/1" (both configured by the user in FR-Script configuration) and the message is sent to subscriber 1.

STEP 10: FR-Script constructs a logical operation based on logic conditions set by the user in its configuration for topics "test/2" and "different/topic". The logical operation results True.

STEP 11: FR-Script publishes a message on topic "!action/2" (both configured by the user in FR-Script configuration) and the message is sent to subscriber 2.

STEP 12: FR-Script removes all used topics and their payloads of its memory.

4.2.4.5. Implementation information

Table 60. Implementation status of the Edge data broker

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/edge_data_broker_enabler.html
Potential features	One additional feature for EDB could be an AI-based ruling configuration based on resource constrains.
Encapsulation readiness	Two Helm charts for the whole enabler, including all components, one specialised for ARM Architectures and Edge Devices (GWEN, Raspberry Pi, etc.) and one specialised for x64 Ubuntu Architectures.
Integration with other enablers	EDB is able to function both independently and in connection with other enablers to support communication between enablers and enablers with Edge devices. It is also able to be bridged with other (MQTT) Data brokers. Integration with LTSE's SQL Database (PostgreSQL) used as Auth Database for storing user credentials and Access Control Lists.

4.2.5. Long-term storage enabler

4.2.5.1. General specifications and features

Table 61. General information of the Long-term storage enabler

Enabler	Long-term Storage Enabler
Id	T43E8
Owner and support	PRODEVELOP and UPV
Description and main functionalities	LTSE is the main long-term storage enabler of the project, offering different storage sizes and individual storage spaces for other enablers (which could request back when they are being initialised in Kubernetes pods), as well as for pilots-related data.
Key features	• NoSQL storage • SQL storage • User control access • Single point of management by REST API endpoints
Plane/s involved	Data Management Plane
Requirements mapping	R-C-6, R-C-10, R-C-14, R-C-15, R-P3A-3, R-P3A-5, R-P3B-3
Use case mapping	All pilots make use of this enabler. Particularly, UC-P1-1, UC-P2-1, UC-P2-2, UC-P2-4, UC-P2-5, UC-P3A-1, UC-P3A-2, UC-P3B-1
Internal components	LTSE Gateway, LTSE NoSQL cluster, LTSE SQL server

4.2.5.2. Structure, components, and implementation technologies

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 59 depicts the high-level overview of the LTSE components, which functionalities are also described:

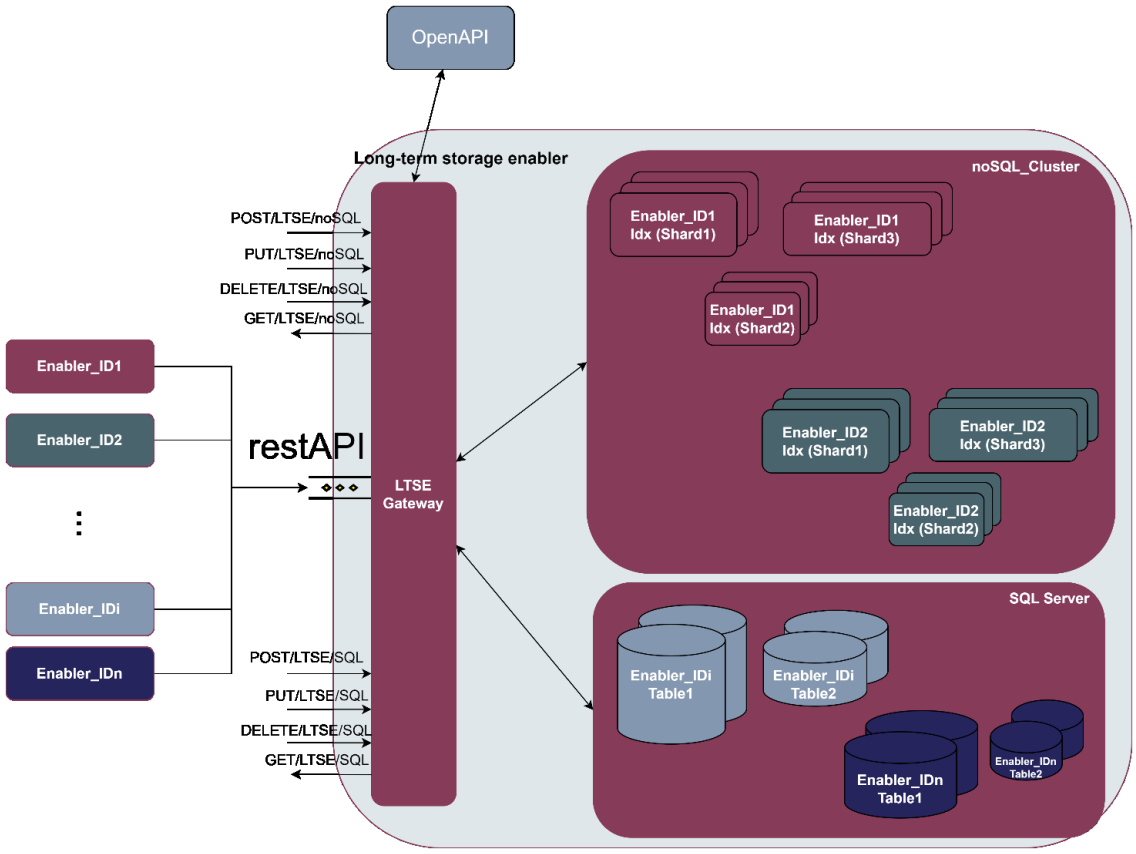


Figure 59. High-level diagram of the Long-term storage enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 62. Components and implementation of the Long-term storage enabler

Component	Description	Technology/s
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 access is managed by the OpenAPI endpoint configurations, which defines which enablers and users have access to which data stored in LTSE.	GinGonic
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.	Elasticsearch

Component	Description	Technology/s
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_Server > SQL_Database > SQL_schema > SQL_table > SQL_row. For High Availability, a master database with one or more standby servers could be set up.	PostgreSQL, PostgREST

4.2.5.3. Communication interfaces

There are a lot of REST API endpoints directly available from the ElasticSearch and PostGREST documentation. Additional endpoints have been created for the initial management of the LTSE. They are listed in the table below.

Table 63. User Communication interfaces of the Long-term storage enabler

Method	Endpoint	Description
GET	**/sql/schemas**	Lists the schemas available in the LTSE SQL server
POST	**/sql/schemas/{schemaname}**	Creates a schemaname in the LTSE SQL server
PUT	**/sql/schemas/{schemaname}**	Activates or deactivates the schema in PostGREST
POST	**/sql/schemas/{schemaname}/tables/{tableName}**	Creates a table tablename into the schema schemaname of the LTSE SQL server
PUT	**/sql/schemas/{schemaname}/tables/{tableName}**	Truncates data from table tablename of the schema schemaname of the LTSE SQL server
DELETE	**/sql/schemas/{schemaname}/tables/{tableName}**	Deletes table tablename of the schema schemaname of the LTSE SQL server
POST	**/sql/api/{tablename}/**	Inserts data into the tablename on the LTSE SQL server (from PostGREST)
PUT	**/sql/api/{tablename}/**	Modifies filtered or all (defined in the body) data into the tablename on the LTSE SQL server (from PostGREST)
DELETE	**/sql/api/{tablename}/**	Deletes filtered or all (defined in the body) data into the tablename on the LTSE SQL server (from PostGREST)
GET	**/sql/api/{tablename}/**	Gets filtered or all (defined in the body) data from the tablename on the LTSE SQL server (from PostGREST)
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**	Adds a JSON document to the specified indexName index of the LTSE noSQL cluster
GET	**/nosql/index/{indexName}/document/{id}**	Retrieves the specified JSON document <id> from the indexName of the LTSE noSQL cluster.

4.2.5.4. Enabler stories

There are 4 main enabler stories that apply in this enabler.

The **first enabler story** 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:

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

STEP 2: If granted, LTSE Gateway request the generation of Enabler_IDx index into LTSE noSQL_Cluster.

STEPS 3-4: 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 5: The Enabler_IDx requests ingestion of NoSQL data document to LTSE Gateway.

STEP 6: LTSE Gateway request the ingestion of Enabler_IDx NoSQL data document into <IndexName> of the LTSE noSQL_Cluster.

STEPS 7-8: 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.

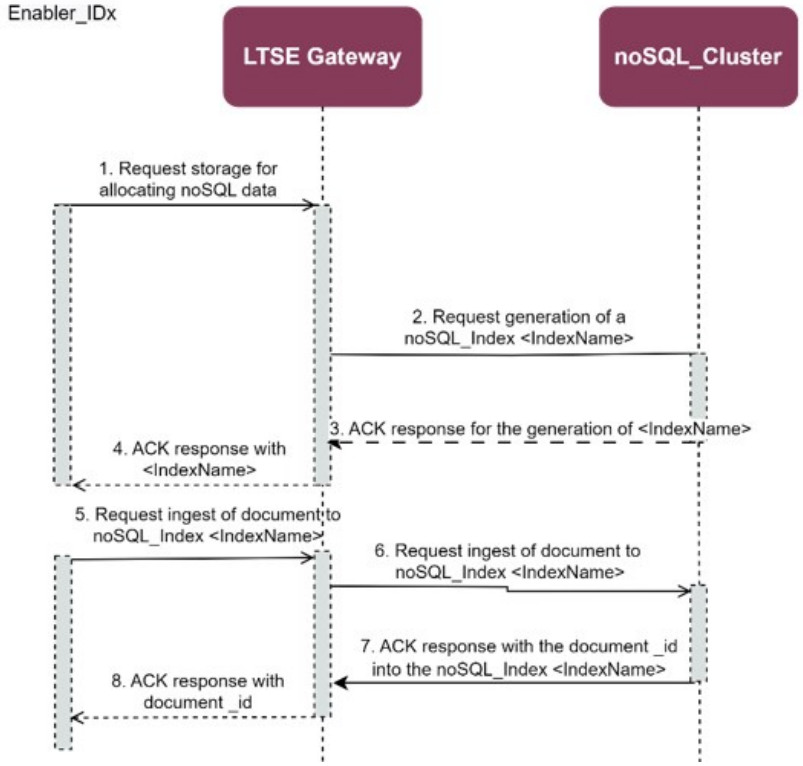


Figure 60. Long-term storage enabler ES1 (store NoSQL data)

The **second enabler story** is related to **the retrieval of NoSQL documents** with a specific <IndexName> from the NoSQL cluster (as well as performing complex queries). The diagram and the related steps are the following:

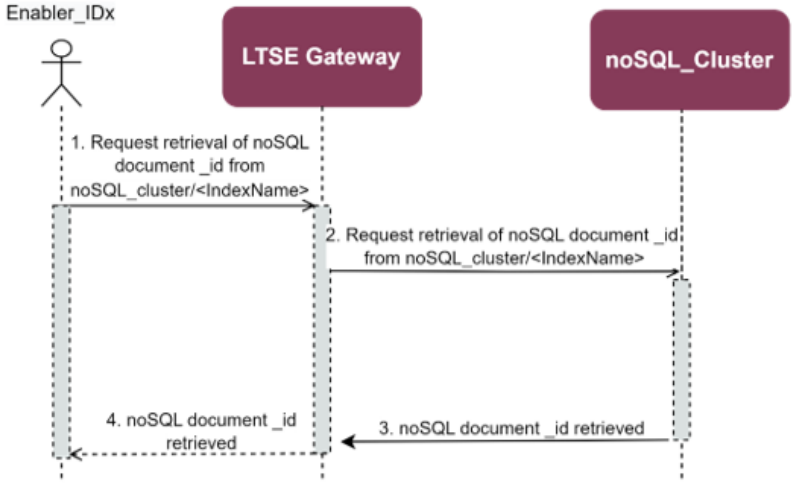


Figure 61. Long-term storage enabler ES2 (get NoSQL data)

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

STEP 2: If granted, LTSE Gateway request the associated information demanded into Enabler_IDx <IndexName> of LTSE noSQL_Cluster.

STEP 3-4: The LTSE gateway, in turn, forwards the document to the Enabler_IDx.

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

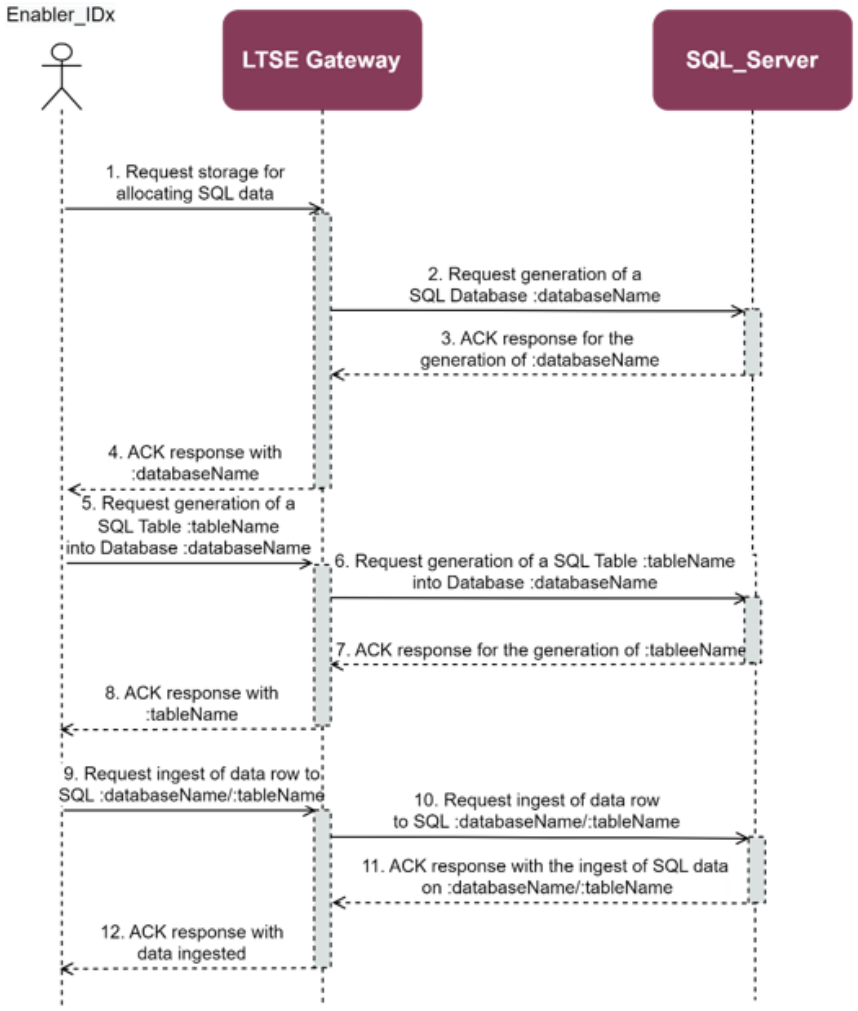


Figure 62. Long-term storage enabler ES3 (store SQL data)

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

STEP 2: If granted, LTSE Gateway requests the generation of Enabler_IDx database into LTSE SQL_Server.

STEPS 3-4: 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 5-6: 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 7-8: 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 9-10: 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 11-12: 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.

Finally, the **fourth and last enabler story** 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 authorisation rights of the Enabler_IDx from IdM/Authorisation 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.

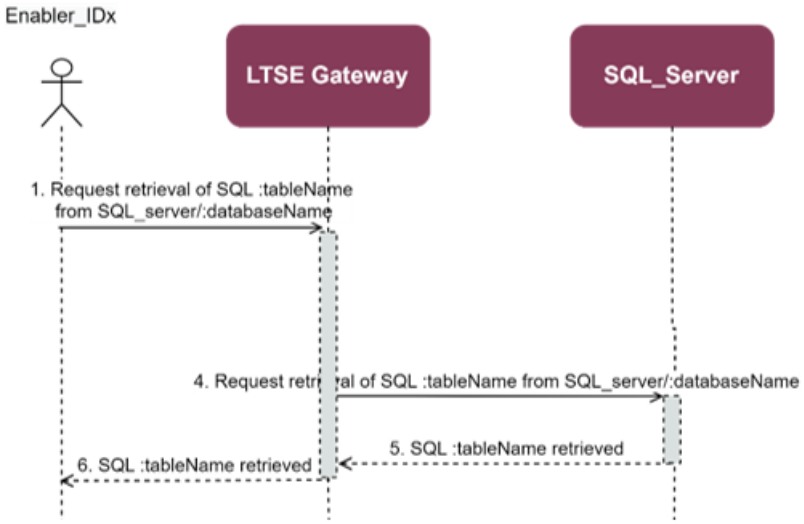


Figure 63. Long-term storage enabler ES4 (get SQL data)

4.2.5.5. Implementation information

Table 64. Implementation status of the Long Term storage enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/datamanagement/long_term_data_storage_enabler.html
Potential features	The enabler is considered feature-complete for the purposes of the project. However, the control access could be expanded or improved in the future.
Encapsulation readiness	A single Helm chart for the whole enabler, including all components.
Integration with other enablers	Enabler can be used in standalone mode, without other enablers, but its integration with Business KPI and EDB have been successfully tested.

4.3. Application and Services enablers

4.3.1. Tactile dashboard

4.3.1.1. General specifications and features

Table 65. General information of the Tactile dashboard

Enabler	Tactile dashboard
Id	T44E1
Owner and support	PRODEVELOP
Description and main functionalities	The Tactile Dashboard enabler has the capacity to represent data through meaningful combined visualisations in real time. Therefore, it allows the creation of fully reusable web components that can be used to create web pages (SPA) or complex web APPs. It also provides (aggregates and homogenises) all the User Interfaces (UIs) for the configuration of the different ASSIST-IoT enablers, and associated components. It is based on Prodevelop's own open source PUI9 framework.
Key features	• Modern, responsive and in some cases adaptive web-design.

Enabler	Tactile dashboard
	• Based on web components, which have their own HTML template • Embedded User control access • Gentle learning curve, but very easy to start being productive
Plane/s involved	Application and services plane
Requirements mapping	R-P3A-10, R-P3B-5, R-P3B-6
Use case mapping	It is used in all pilots. Particularly, in UC-P1-3, UC-P1-5, UC-P2-1, UC-P2-2, UC-P2-4, UC-P3A-1, UC-P3A-2, UC-P3B-1
Internal components	Frontend, Backend, PUI9 DB

4.3.1.2. Structure, components and implementation technologies

The following figure sketches the architectural diagram of tactile dashboard components.

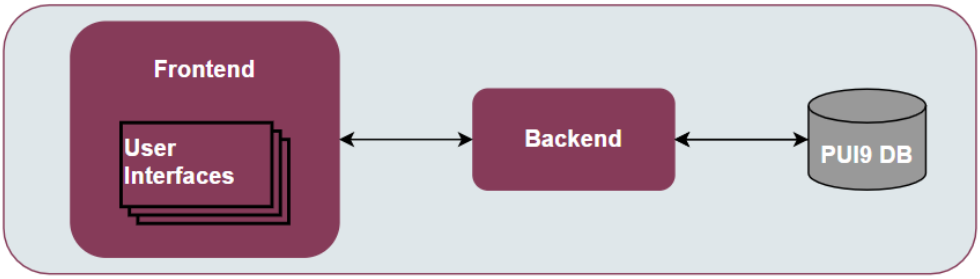


Figure 64. High-level diagram of the Tactile dashboard

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 66. Components and implementation of the Tactile dashboard

Component	Description	Technology/s
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.).	VueJS, Vuetify, Datatables, Axios, NPM, Webpack
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.	Java 8, Spring
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.	SQL databases (compatible with PostgreSQL, Oracle, SQL Server)

4.3.1.3. Communication interfaces

Table 67. User Communication interfaces of the Tactile dashboard

Method	Endpoint	Description
POST	**/login/signin**	Default login of the tactile dashboard
POST	**/loginAutzIdm/signin**	Login to the tactile dashboard by means of the Idm and Authz enablers
POST	**/component_id/{data}**	Inserts into the component_id the specified {data}
GET	**/component_id/{data}**	Gets the component_id {data} stored in the tactile dashboard database

Method	Endpoint	Description
POST	**/login/signin**	Default login of the tactile dashboard
POST	**/loginAutzIdm/signin**	Login to the tactile dashboard by means of the Idm and Authz enablers
POST	**/component_id/{data}**	Inserts into the component_id the specified {data}
PUT	**/component_id/{data}**	Updates the component_id {data} stored in the tactile dashboard database
DELETE	**/component_id/{data}**	Deletes the component_id {data} stored in the tactile dashboard database

4.3.1.4. Enabler stories

Three enabler stories 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 enabler story** 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.

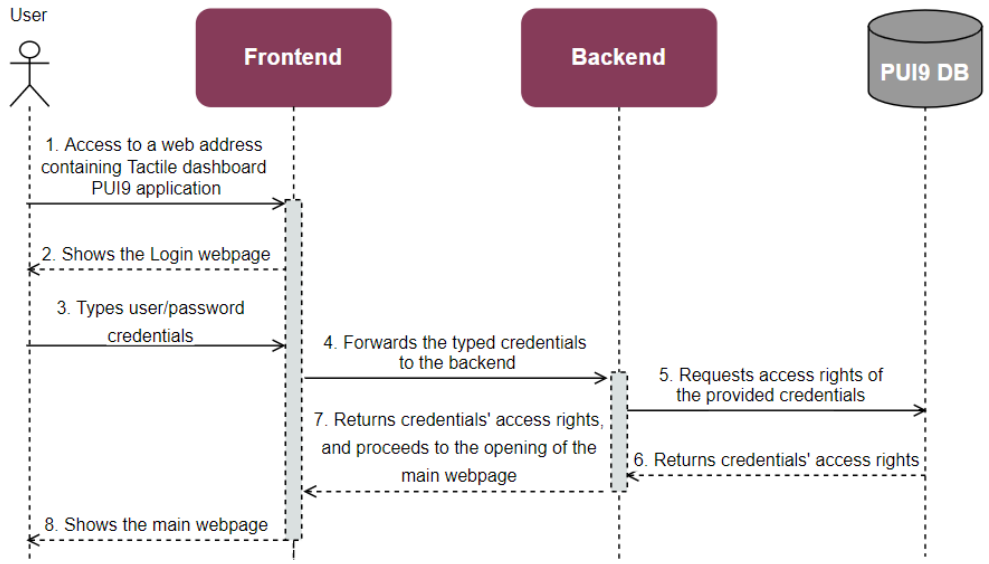


Figure 65. Tactile dashboard ES1 (login webpage)

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[2] 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 enabler story** 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:

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

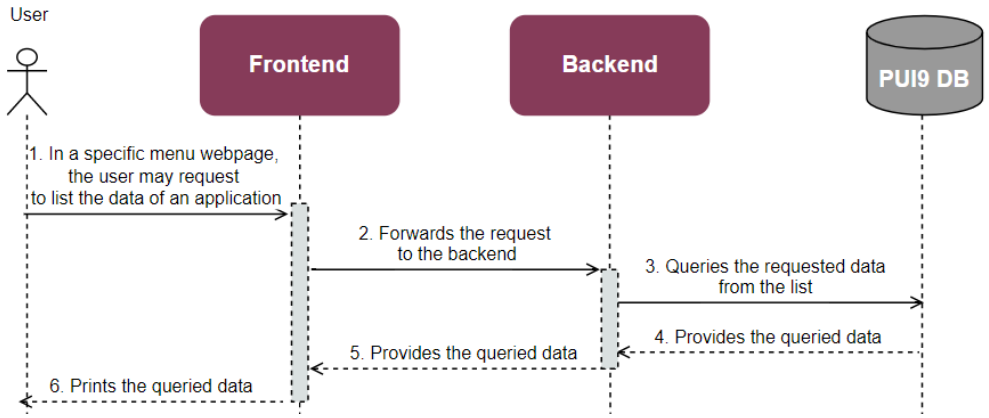


Figure 66. Tactile dashboard ES2 (show data managed by PUI9 database)

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.

The **third enabler story** 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.

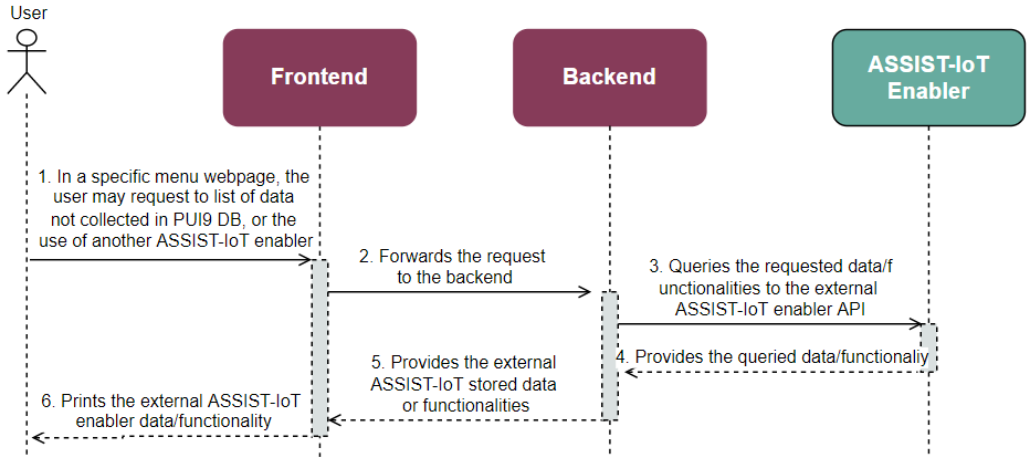


Figure 67. Tactile dashboard ES3 (show data not managed by PUI9 database)

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.

4.3.1.5. Implementation information

Table 68. Implementation status of the Tactile dashboard

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/tactile_dashboard_enabler.html
Potential features	The enabler is considered feature-complete for the purposes of the project.
Encapsulation readiness	The example tactile dashboard, as well as the manageability dashboard are already containerised, and a Helm chart has been created for them.
Integration with other enablers	Enabler can be used in standalone mode, without other enablers, but its integration with Business KPI, PUD, IdM, Authz, and Manageability enablers have been successfully tested.

4.3.2. Business KPI reporting enabler

4.3.2.1. General specifications and features

Table 69. General information of the Business KPI reporting enabler

Enabler	Business KPI Reporting enabler
Id	T44E2
Owner and support	PRODEVELOP
Description and main functionalities	The Business KPI enabler will allow to embed time-series analytics data and Key Performance Indicators (KPIs) desired by the end-user as User Interfaces (UIs) within the tactile dashboard in the form of graphs, charts, pies, etc.
Key features	• Web-based visualisation graphs and templates. • Full and seamless integration with LTSE • Embedded development tools for testing
Plane/s involved	Application and services plane
Requirements mapping	R-C-17, R-P3A-10, R-P3B-5, R-P3B-6
Use case mapping	All pilots will implement this enabler. Particularly, UC-P1-3, UC-P2-1, UC-P2-1, UC-P2-4, UC-P3A-1, UC-P3A-2, UC-P3B-1
Internal components	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

4.3.2.2. Structure, components, and implementation technologies

Figure 68 presents the architectural diagram of the Business KPI reporting enabler and its internal components:

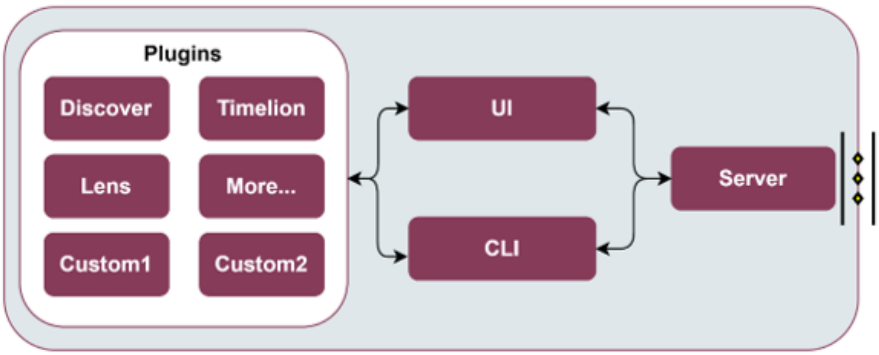


Figure 68. High-level diagram of the Business KPI reporting enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 70. Components and implementation of the Business KPI reporting enabler

Component	Description	Technology/s
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.	Kibana
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 visualisations and a set of functionalities to arrange the visualisations 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 visualise new chart types, colour palettes, etc.	

4.3.2.3. Communication interfaces

The business KPI enabler is formed by spaces and data views, which allow to customise the webpage layout for visualisations. All graphs in the Business KPI enabler are stored as saved-objects (basically a JSON-object that describes which visualisations 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. Given Business KPI enabler is based on Kibana, all the supported REST API endpoints are available at Kibana documentation. The most relevant ones for the initial configuration are listed in the table below.

Table 71. User Communication interfaces 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>

4.3.2.4. Enabler stories

There is a **single enabler story** 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:

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.

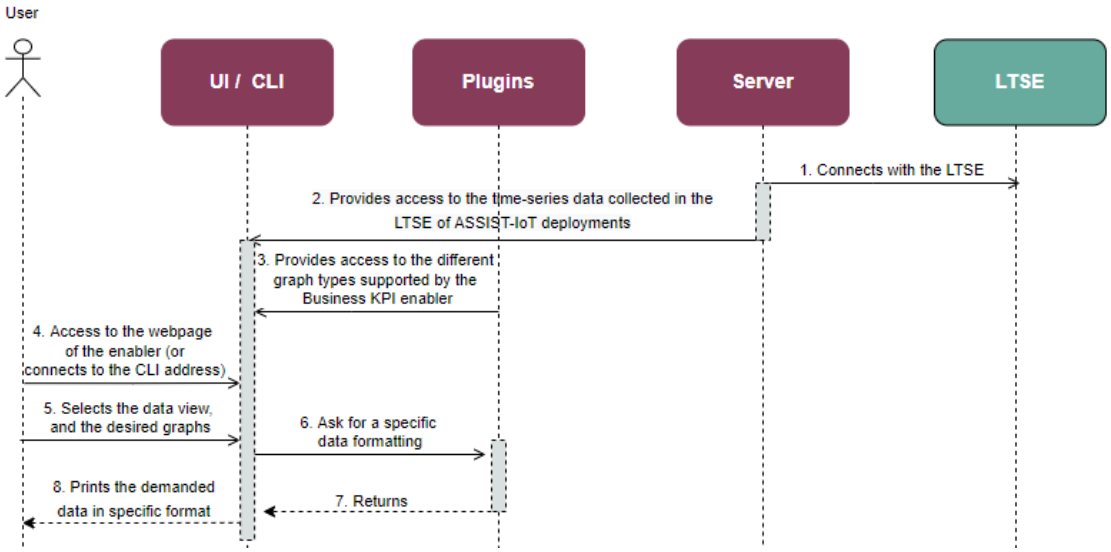


Figure 69. Business KPI reporting enabler ES1 (generate graphs from time-series data)

4.3.2.5. Implementation information

Table 72. Implementation status of the Business KPI reporting enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/business_kpi_reporting_enabler.html
Potential features	The enabler is considered feature-complete for the purposes of the project.
Encapsulation readiness	The enabler is already encapsulated, and a Helm chart has been provided
Integration with other enablers	Enabler require data collected in the LTSE noSQL cluster. Its integration with Tactile dashboard and LTSE enablers have been successfully tested.

4.3.3. Performance and usage diagnosis enabler

4.3.3.1. General specifications and features

Table 73. General information of the Performance and usage diagnosis enabler

Enabler	Performance and Usage Diagnosis (PUD)
Id	T44E3
Owner and support	ICCS
Description and main functionalities	PUD enabler aims at collecting performance metrics from monitored targets by scraping HTTP endpoints on them and highlighting potential problems in the ASSIST-IoT platform. Supported "targets" include kube-state-metrics for monitoring every kubernetes cluster used in the project, node-exporter metrics for monitoring hardware, OS metrics exposed by *NIX kernels, as well as other important metrics for the rest of the enablers used in the architecture.
Key features	• Utilising the pull model to retrieve metrics over HTTP in regular intervals from exporters that expose their metrics on an "/metrics" endpoint. • PromQL, a very flexible query language that can be used to query the metrics in the Prometheus dashboard. Also, the PromQL query will be used by Prometheus UI and Grafana to visualise those metrics. • Exporters are libraries which converts existing metric from third-party apps to Prometheus metrics format. There are many official and community Prometheus exporters. One example is, Kube State metrics, a service which talks to Kubernetes

Enabler	Performance and Usage Diagnosis (PUD)
	API server to get all the details about all the API objects like deployments, pods, daemonsets etc. • Uses time-series database for storing all the retrieved data.
Plane/s involved	Application and Services Plane
Requirements mapping	R-C-7, R-P1-16, R-P1-5, R-P2-12, R-P2-18
Use case mapping	ALL
Internal components	Prometheus Server, Prometheus-es-adapter, TargetAPI, Kube-state-metrics, Grafana Dashboard, Node_exporter

4.3.3.2. Structure, components and implementation technologies

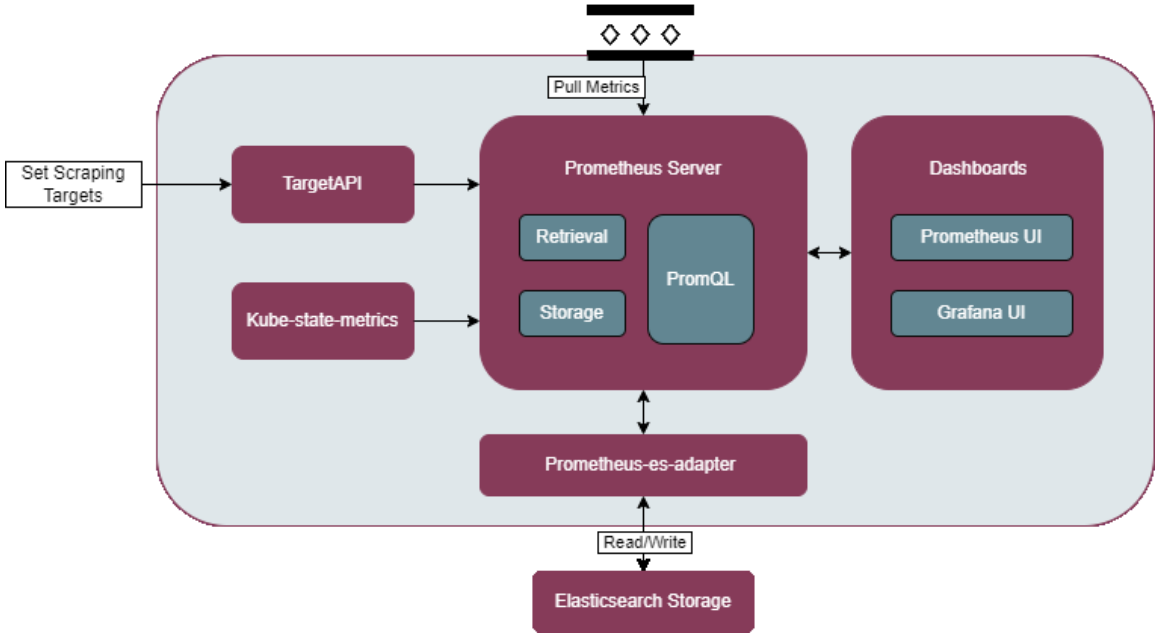


Figure 70. High-level diagram of the Performance and usage diagnosis enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 74. Components and implementation of the Performance and usage diagnosis enabler

Component	Description	Technology/s
Server	An open-source monitoring framework. It provides out-of-the-box monitoring capabilities for the Kubernetes container orchestration platform.	Prometheus Server
Prometheus-es-adapter	A read and write adapter for integrating LTSE's elastic search as prometheus' persistent storage.	Go
TargetAPI	An API that provides the ability to add, update and delete targets for PUD's Prometheus consumption dynamically.	Python
Dashboard(s)	A GUI that provides an interactive visualisation web application composed of charts, graphs and dashboards.	Grafana / Prometheus UI
Kube-state-metrics	A listening service that generates metrics about the state of Kubernetes objects through leveraging the Kubernetes API	Go
Node_exporter	An exporter for hardware and OS metrics exposed by *NIX kernels, is installed separately in every GWEN and Ubuntu device. The node_exporter is designed to monitor the host system	Go

4.3.3.3. Communication interfaces

Table 75. User Communication interfaces of the Performance and usage diagnosis enabler (GUIs)

Method	Endpoint	Description
GET	**:9090/**	Display Prometheus UI
GET	**:3000/**	Display Grafana Dashboard UI

Table 76. API of the Performance and usage diagnosis enabler - TargetAPI

Method	Endpoint	Description
GET	**:5000/docs**	Display TargetAPI's Swagger GUI (Listing the bellow APIs)
GET	**/**	Get all target groups
GET	**/{id}**	Get target group by id
DELETE	**/{id}**	Delete target group by id
POST	**/**	Create targets
PATCH	**/**	Update targets
GET	**/targets**	Get all targets
GET	**/labels**	Get all labels
GET	**/lengths**	Get number of targets and number of labels

4.3.3.4. Enabler stories

In the **first and only PUD's use case** presents the interactions between PUD enabler's components as well as how the enabler should be configured and used from an admin privileged user in order to **monitor the whole ASSIST-IoT system** and all of its enabler components, devices, clusters etc.

STEP 1: The user should manually install node_exporter in all devices (server and edge node devices) of the system in order to monitor hardware and OS.

STEP 2: The user should also install kube-state-metrics in other kubernetes clusters that might reside on the system in order to monitor them along the cluster that PUD resides in which kube-state-metrics is installed by default.

STEP 3: The user should list all available metric exporter endpoints and use the TargetAPI through its rest API in order to add them to PUD for Prometheus consumption.

STEP 4: Prometheus server fetches the posted targets through an HTTP-based service.

STEP 5: Prometheus collects metrics from the beforementioned targets by "scraping" /metrics HTTP endpoints implementing an HTTP Pull model.

STEP 5: Prometheus server reads and writes its data using ElasticSearch as remote persistent storage utilising prometheus-es-adapter.

STEP 6: User accesses Prometheus UI to check the state of the available exporter endpoints and graph the metrics using PromQL.

STEP 6: User accesses Grafana Dashboards to import or create dashboards, tables and graphs for real time monitoring.

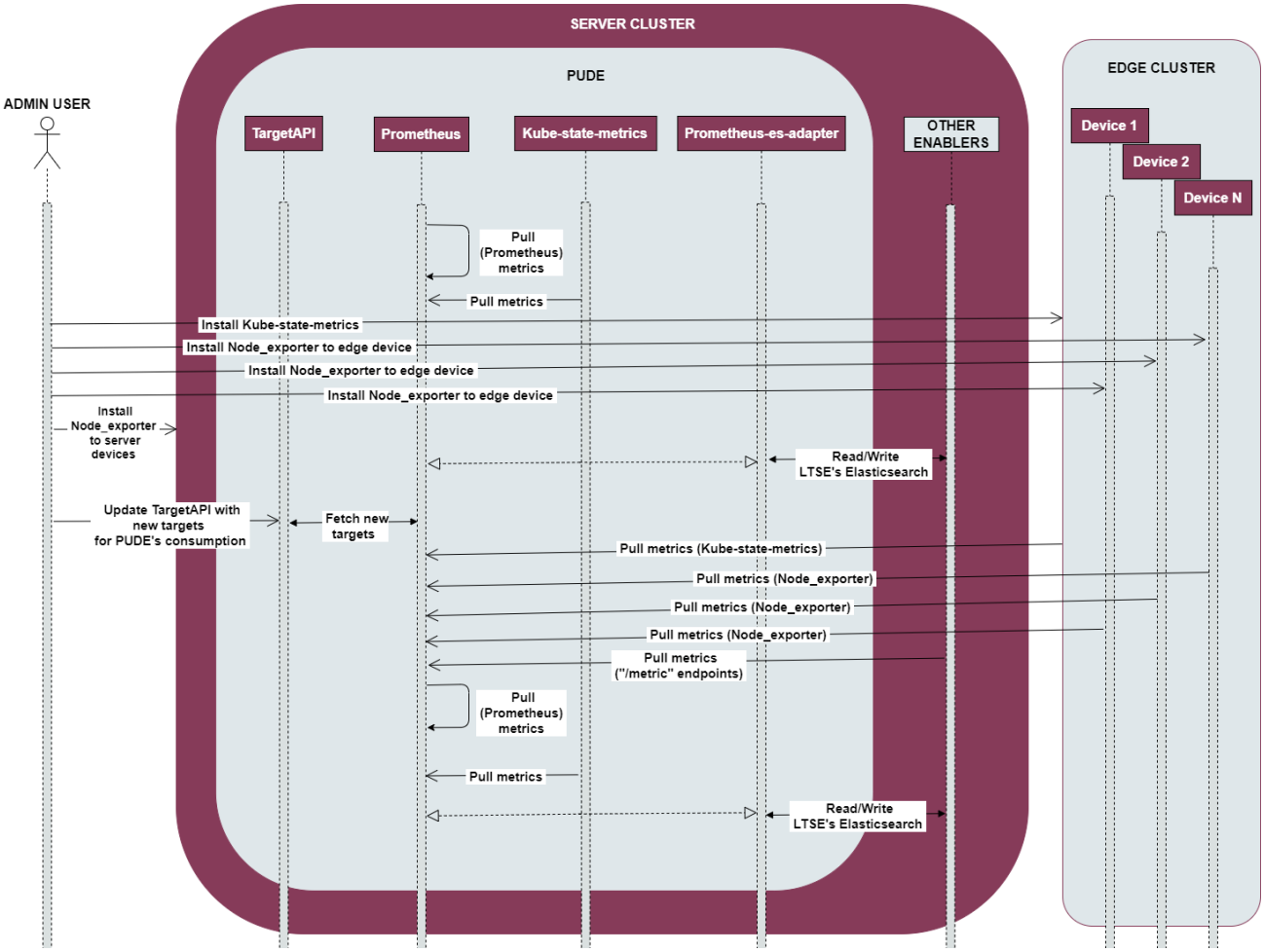


Figure 71. Performance and usage diagnosis enabler ES1 (metrics gathering and presentation)

4.3.3.5. Implementation information

Table 77. Implementation status of the Performance and usage diagnosis enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/performance_and_usage_diagnosis_enabler.html
Potential features	One additional feature for PUDE could be a service that automatically search the whole cluster for "/metric" endpoints and make them available for PUD's Prometheus consumption, without the need of an admin user and the TargetAPI.
Encapsulation readiness	A single Helm chart for the whole enabler, including all components, except Node_exporter which is designed to monitor host systems and requires to be installed in the host system itself.
Integration with other enablers	PUD is able to be used standalone, and get other enablers' metrics over HTTP utilising their exposed metrics on an "/metrics" endpoint in order to monitor them. Integration with LTSE's NoSQL Database used as time-series database for storing all the data.

4.3.4. OpenAPI management enabler

4.3.4.1. General specifications and features

Table 78. General information of the OpenAPI management enabler

Enabler	OpenAPI management enabler
Id	T44E4
Owner and support	CERTH

Enabler	OpenAPI management enabler
Description and main functionalities	The OpenAPI Management enabler is the enabler responsible for managing the APIs in the Assist-IoT project by allowing the enablers of the project to publish their APIs, monitor their lifecycles and make sure that that the needs of external third parties, as well as applications that using the APIs, are being met. Hence, the main functionalities that it serves are to collect all the APIs that are used by the Assist-IoT enablers in order to proxy them through the API gateway to the external users, to be used as an API portal from which the developers can push their OpenAPI documentations to the API-gateway,store them in an API library and interact with them through SwaggerUI.
Key features	• Allows developers to publish OpenAPI definitions through the API gateway. • Endpoints of the client are secured with IdM. • A front-end portal application acting as a library of OpenAPI definitions. • Integrated with the API publisher, enabling developers to push definitions through the front-end. • Provides immediate interaction with definitions through Swagger UI. • Utilises Kong API Gateway (OSS) as the chosen OpenAPI gateway for Assist-IoT. • Integrates the kong-oidc open-source plugin to link Kong Gateway with IdM. • Ensures security for registered endpoints. • Used as an Ingress controller to proxy services outside the Kubernetes cluster.
Plane/s involved	Application and Services Plane
Requirements mapping	• R-C-7: Edge-oriented deployment • R-P1-6: Terminal data access • R-P1-16: Open/Accessible remote capabilities • R-P1-17: Customisable remote desktop • R-P2-5: Wristband pairing with other devices capability • R-P3A-11: Connectivity between OEM and fleet
Use case mapping	This enabler is inherent to the ASSIST-IoT ecosystem and, therefore, it should be present at all pilots without a specific use case in mind yet. Otherwise, it would not be possible to allow external granted Open Callers to integrate and communicate their developments with ASSIST-IoT platform.
Internal components	API Gateway, API Portal, API Publisher

4.3.4.2. Structure, components and implementation technologies

The OpenAPI management enabler consists of three main components, the OpenAPI Publisher, the OpenAPI Portal, and the OpenAPI Gateway.

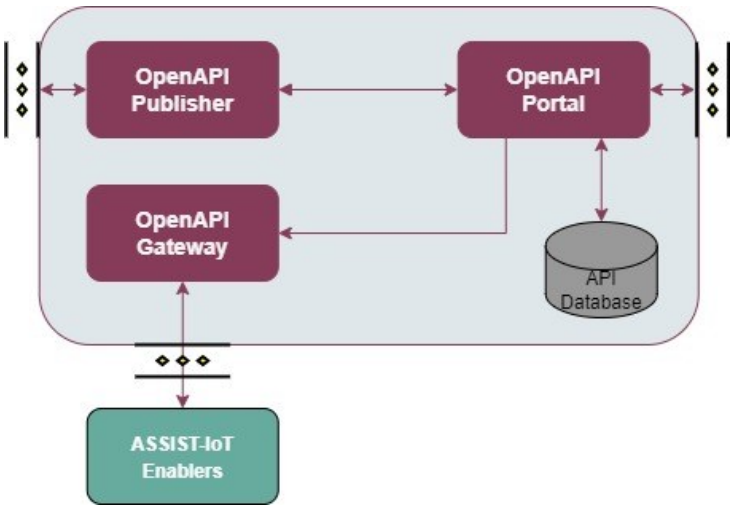


Figure 72. High-level diagram of the OpenAPI management enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 79. Components and implementation of the OpenAPI management enabler

Component	Description	Technology/s
OpenAPI Publisher	A back-end client that enables developers to securely publish OpenAPI definitions through the API gateway using standard HTTP methods.	Python FastAPI, PostgreSQL
OpenAPI Portal	A front-end application serving as a library for OpenAPI definitions, allowing developers to push and interact with their definitions through Swagger UI with IdM authentication.	Swagger UI, ReactJS
OpenAPI Gateway	The OpenAPI gateway is powered by Kong API Gateway (OSS) and offers a lightweight, fast, and flexible solution for managing API traffic, with integration capabilities for IdM and secure endpoint registration. It is used as an ingress controller in the Kubernetes cluster to proxy services and components outside of it.	Kong API Gateway

4.3.4.3. Communication interfaces

Table 80. 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.

4.3.4.4. Enabler stories

The **first enabler story** 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:

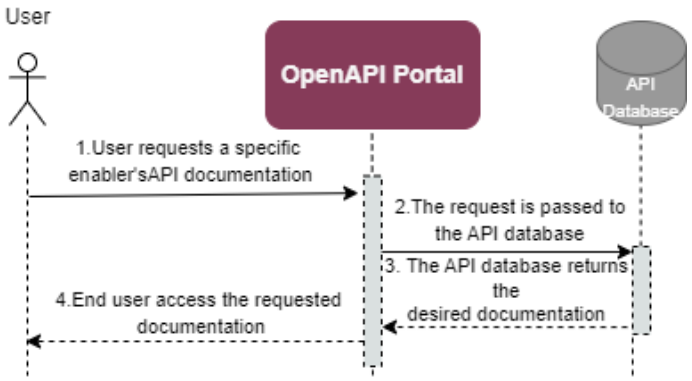


Figure 73. OpenAPI management enabler ES1 (get API documentation)

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 the API database.

STEP 3: The API database returns the desired documentation.

STEP 4: The enabler outputs the requested API documentation.

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

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 database.

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).

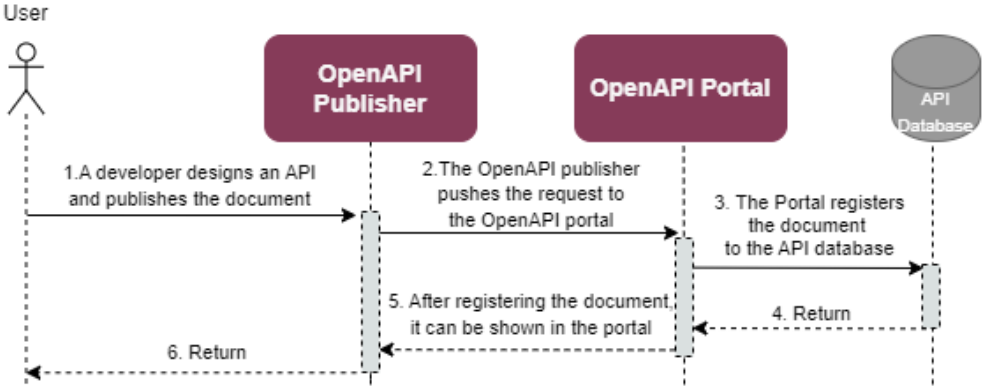


Figure 74. OpenAPI management enabler ES2 (publish API document)

First and second stories are demonstrating the use of the API Portal and publishers as isolated components. The **third enabler story** involves an external entity who wants to **interact with an ASSIST-IoT enabler**. This enabler story is also showing the integration between the OpenAPI gateway and the IdM and the use of Kong as a Kubernetes Ingress Controller in the ASSIST-IoT cluster. The figure and steps below describe the flow for being redirected to the correct enabler:

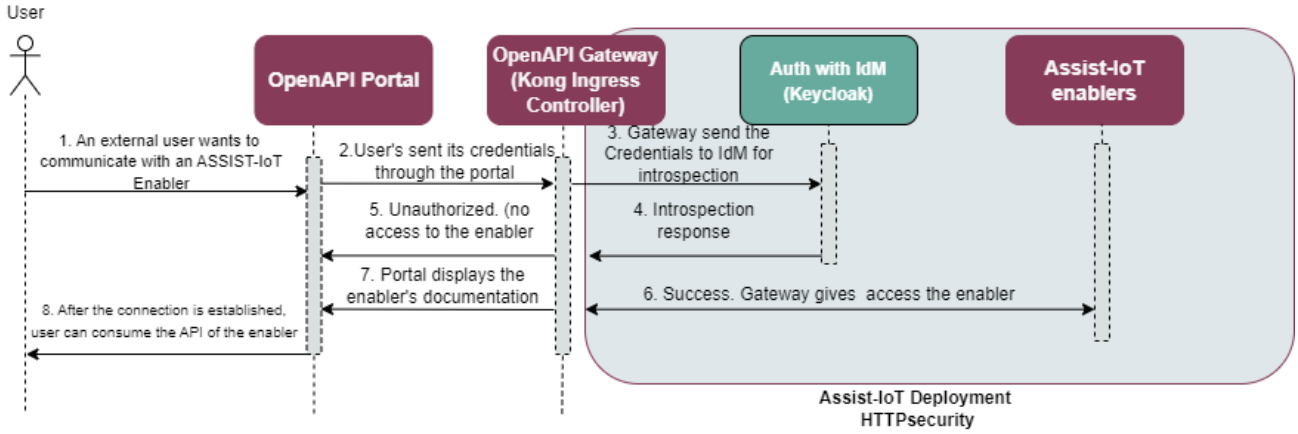


Figure 75. OpenAPI management enabler ES3 (interact with enablers)

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

STEPS 2-3: The Portal gets user's credentials and send them to IdM through the OpenAPI Gateway.

STEP 4: IdM introspects credentials and sends back its response.

STEPS 5: If user's credentials are not correct, user is unauthorised to access Assist-IoT enablers.

STEPS 6-7: Successful connection. The OpenAPI Gateway is proxying enabler's API outside of the cluster and the Portal displays the OpenAPI documentation of the enabler.

STEP 8: User consumes the API of the enabler.

4.3.4.5. Implementation information

Table 81. Implementation status of the OpenAPI management enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/openapi_management_enabler.html
Potential features	The enabler has all the required features to meet the project's objectives.

Category	Status
Encapsulation readiness	The enabler has a working Helm Chart version that will be updated continuously until the end of the project.
Integration with other enablers	Integrated with IdM.

4.3.5. Video augmentation enabler

4.3.5.1. General specifications and features

Table 82. General information of the Video augmentation enabler

Enabler	Video Augmentation enabler
Id	T44E5
Owner and support	PRODEVELOP
Description and main 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.
Key features	• Support of ML-based object detection and recognition models. • Flexible configuration of GPU/CPU computing • Easy-to-use API management
Plane/s involved	Application and services plane
Requirements mapping	R-P1-5, R-P1-23, R-P3A-13, R-P3B-1, R-P3B-2, R-P3B-13
Use case mapping	UC-P1-7, UC-P2-2, UC-P3A-2
Internal components	REST API, ML trainer service, Inference engine

4.3.5.2. Structure, components, and implementation technologies

The following figure presents the architectural diagram of the Video augmentation enabler and its internal components:

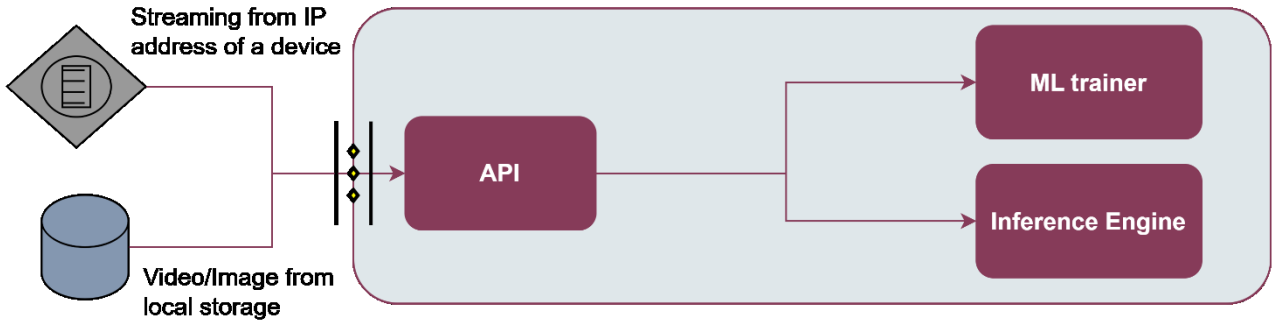


Figure 76. High-level diagram of the Video augmentation enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 83. Components and implementation of the Video augmentation enabler

Component	Description	Technology/s
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.	Fast API
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	Tensorflow OpenCV

Component	Description	Technology/s
	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 installed. 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, initialisation, and execution latency.	

4.3.5.3. Communication interfaces

Table 84. User Communication interfaces 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.

4.3.5.4. Enabler stories

The two main enabler stories of the Video Augmentation 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 enabler story**, i.e., the **training process**, will be initiated by a user, once the labelled data is updated and allocated in the corresponding local folder.

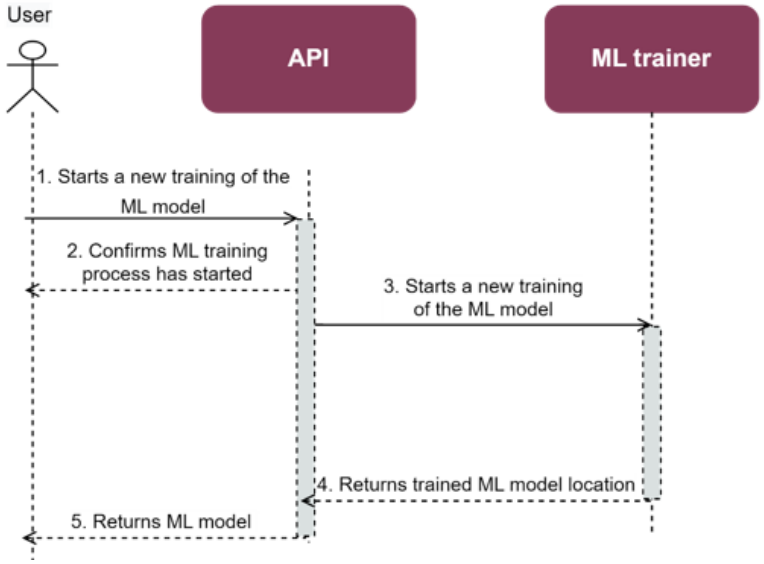


Figure 77. Video augmentation enabler ES1 (model training)

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 ML trainer in order to start a new training of either new or pre-trained ML model available in its framework. And confirms to the user that the ML model training has started.

STEPS 4-5: When the training process is finished, the ML model is stored in the ML trainer database, and notified to the user where it can be downloaded.

The **second enabler story** 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:

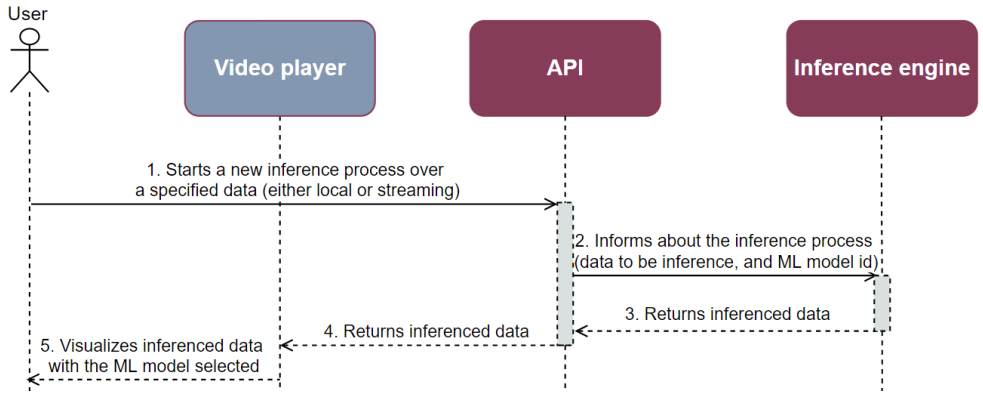


Figure 78. Video Augmentation enabler ES2 (video inference)

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.

4.3.5.5. Implementation information

Table 85. Implementation status of the Video augmentation enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/video_augmentation_enabler.html
Potential features	The enabler is considered feature-complete for the purposes of the project. Additional features foreseen in the future include the support of inference solutions for streaming videos.
Encapsulation readiness	The enabler is already encapsulated, and a Helm chart has been provided
Integration with other enablers	Enabler can be used in standalone mode, without other enablers.

4.3.6. Mixed reality enabler

4.3.6.1. General specifications and features

Table 86. General information of the MR enabler

Enabler	Mixed Reality (MR) enabler
Id	T44E6
Owner and support	ICCS
Description and main functionalities	The MR Enabler processes data, coming from other enablers, adapting it into a format optimised for immersive visualisation using head-mounted Mixed Reality (MR) devices. Data, which may come from long-term storage or real-time data streams, are requested according to its relevance to the user. The MR Enabler ensures that authorised users are

Enabler	Mixed Reality (MR) enabler
	presented with pertinent data through their MR devices, allowing for a personalised and secure experience. Furthermore, the enabler supports user interaction with the virtual content and view customisation.
Key features	• Visualises the model of the construction site through the head-mounted MR devices, along with the danger zones of the site. The model of the site and all its related data come from the long-term storage • Visualises the location of the workers of the construction site, along with their crucial information • Receives alert message from real-time data streams and display their details to the user • Provides the ability to create and send new reports • Captures and stores media files in order to include them in a report
Plane/s involved	Application and Services Plane
Requirements mapping	UC-P2-7
Use case mapping	Pilot 2
Internal components	Configuration MR, Data Integration and Data Visualisation

4.3.6.2. Structure, components and implementation technologies

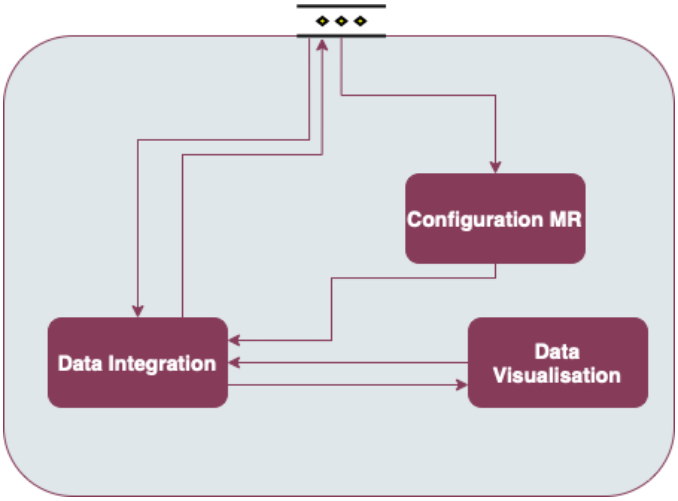


Figure 79. High-level diagram of the MR enabler

Specifically, a description of each one of the components depicted is provided in the table below, along with the technologies used for implementing them:

Table 87. Components and implementation of the MR enabler

Component	Description	Technology/s
Configuration MR	Receives current IoT ecosystem configurations through REST API	C#
Data Integration	Deserialises the information coming from different enablers through REST API and MQTT protocols.	C#
Data Visualisation	Displays properly the incoming information to the user	Unity / MRTK2

4.3.6.3. Communication interfaces

Table 88. API of the MR enabler

Method	Endpoint	Description
GET	**/metrics**	Receive MR enabler's metrics

4.3.6.4. Enabler stories

The **first enabler story** is related to the **examination of the BIM model of the construction site**. Here, the user is able to view and inspect the BIM model by following the steps bellow:

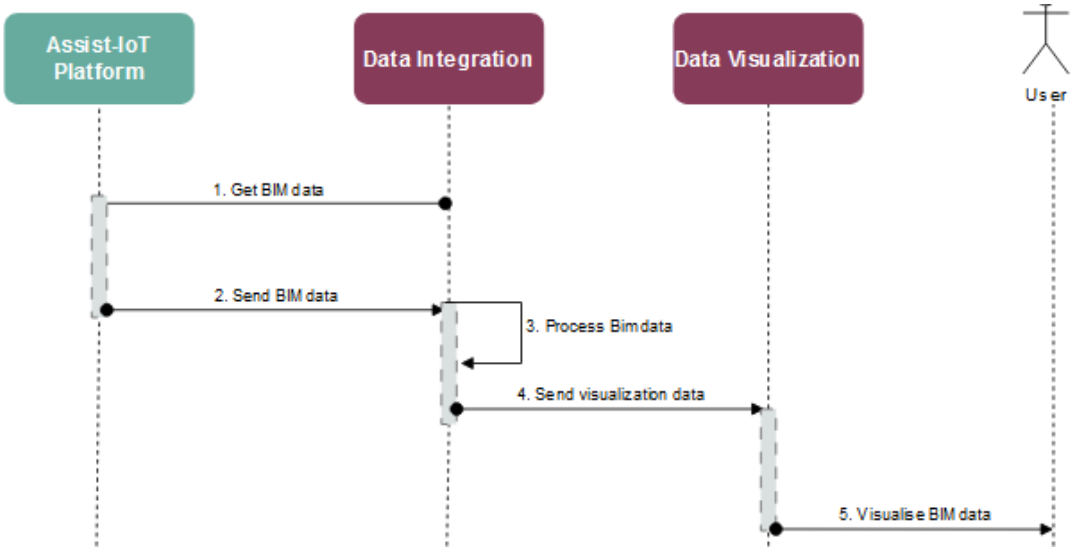


Figure 80. MR enabler ES1 (fetch and visualisation of the BIM model)

STEP 1: The MR enabler sends an HTTP request to receive the BIM model file from the Assist-IoT Platform.

STEP 2: The Assist-IoT Platform sends the data of the BIM model back to the MR enabler.

STEP 3: The MR enabler process the data and creates a 3D presentation of the data.

STEP 4: The BIM model presentation is being sent to the GUI of the user.

STEP 5: The user inspects the BIM model.

The **second enabler story** enables users **to send a report to the database**. It has this sequence diagram and steps:

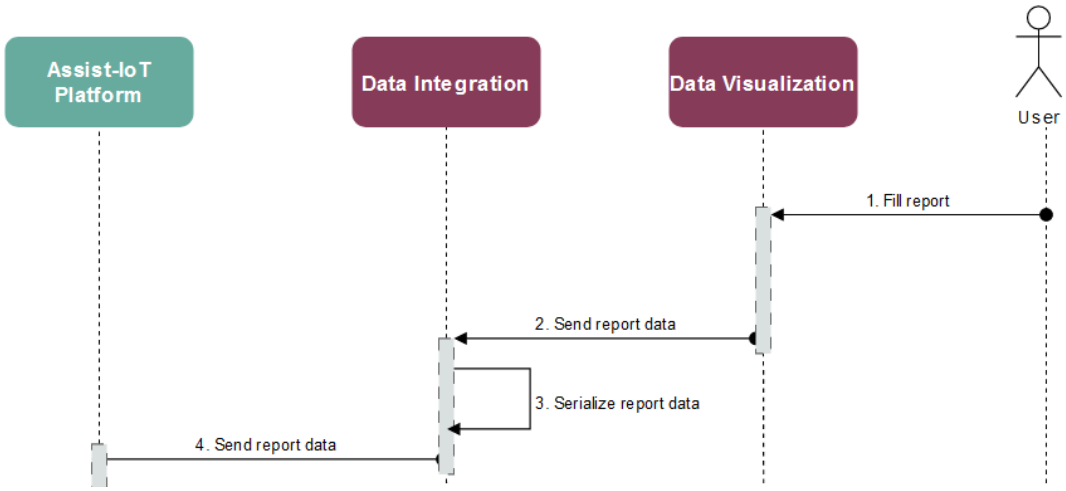


Figure 81. MR enabler ES2 (send report)

STEP 1: The user fulfills a report through the Graphical User Interface of the MR enabler.

STEP 2: The report is being sent to the Data Integration component for preparation.

STEP 3: The MR enabler serialises properly the fields of the report.

STEP 4: The MR enabler sends the report through a REST API request to the Assist-IoT Platform.

The **third and final enabler story** allows a user **to receive a real-time alert** while using the MR enabler and learn more information about the alert from the GUI, by following those steps:

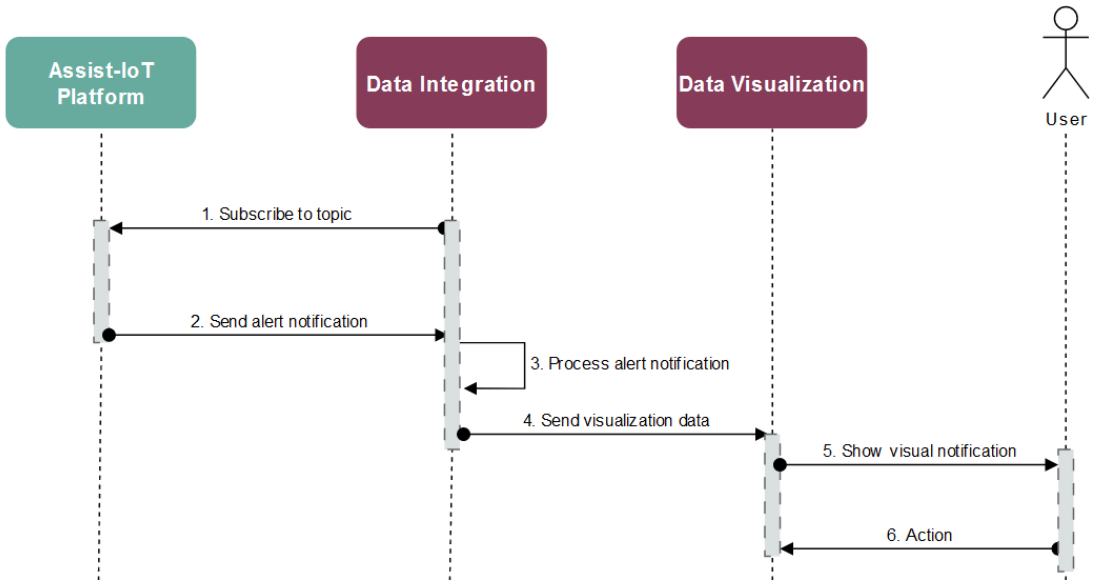


Figure 82. MR enabler ES3 (receive notification)

STEP 1: The MR enabler subscribes to an alert topic of the corresponding enabler, utilising MQTT protocols.

STEP 2: The MR enabler receives a real-time notification through the Assist-IoT Platform.

STEP 3: The MR enabler process the alert notification.

STEP 4: The MR enabler creates a visual component to notify the user.

STEP 5: The MR enabler shows the new notification to the user.

STEP 6: The user can use the Graphical User Interface of the MR enabler to read more details about the incoming alert.

4.3.6.5. Implementation information

Table 89. Implementation status of the MR enabler

Category	Status
Link to ReadtheDocs	https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/application/mr_enabler.html
Potential features	An additional feature to the MR enabler could involve harnessing point cloud data, empowering better interaction between virtual objects and the physical world.
Encapsulation readiness	Excluded (D3.7 - ASSIST-IoT Architecture Definition – Final, page 63)
Integration with other enablers	MR enabler is not able to be used as standalone and depends on connecting with other enablers (Edge Data Broker, Semantic, Long Term Storage enablers) to be fully functional inside the ASSIST-IoT ecosystem.

5. Enabler's Technical Documentation and Demo Videos

The Technical Documentation (https://assist-iot-enablers-documentation.readthedocs.io/en/latest/) for all the aforementioned enablers is available on the Read the Docs platform. This documentation is encapsulated in the final Deliverable D6.6 – Technical and Support Documentation – Final, which represents a comprehensive analysis of the provided documentation (updates are also provided through the final WP6 deliverable D6.8). Its primary objective is to furnish users with essential information concerning the deployment and utilisation of ASSIST-IoT enablers across both the horizontal and vertical facets of the ASSIST-IoT architecture. The Technical Documentation is thoughtfully structured around the overarching ASSIST-IoT architecture, adhering to a general approach encompassing the following key sections: Introduction, Features, Placement within the Architecture, User Guide, Prerequisites, Installation, Configuration Options, Developer Guide, Version Control and Release, Licensing, and Notices.

Selected enablers are also showcased through videos available on the official YouTube channel of the ASSIST-IoT project (https://www.youtube.com/@assist-iot). The primary objective of this endeavour is twofold: firstly, to illustrate the accomplishments achieved and to provide external audiences with insights into the technical intricacies implemented within the project at enablers level, thereby expanding our YouTube channel views and subscribers base. Secondly, these videos aim to highlight the consortium's preparations for the final pilot trials and the practical execution of pilot work, bridging the gap between theory and application. Ultimately, these videos serve as a testament on how these enablers can be integrated into other components, projects or even within a commercial service/product context.

6. Conclusion

With this report and the code of the enablers developed conclude the activities of WP4. It is worth mentioning that additional information of the enablers in terms of installation, usage, integration with other enablers etc. can be found in the official documentation of the project, available in the ReadTheDocs page. Also, highlighting that the intention has been to produce a self-contained document, so that there is no need to consult previous iterations of the deliverable (D4.1, D4.2).

In any case, rather than in updating the enablers' specifications, major effort has been put in finalizing the features of the enablers (as well as manufacturing and delivering to the pilots), which code can be found in the official project repositories.