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

D3.7 – ASSIST-IoT Architecture Definition – Final

Deliverable No.	D3.7	**Due Date**	31-JUL-2022
Type	Report	**Dissemination Level**	Public
Version	1.0	**WP**	WP3
Description	Final specification of the ASSIST-IoT technical architecture and its components.		

Copyright

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

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

Disclaimer

Authors

Name	Partner	e-mail
Alejandro Fornés	P01 UPV	alforlea@upv.es
Ignacio Lacalle	P01 UPV	iglaub@upv.es
Carlos E. Palau	P01 UPV	cpalau@dcom.upv.es
Eduardo Garro	P02 PRO	egarro@prodevelop.es
Paweł Szmeja	P03 IBSPAN	pawel.szmeja@ibspan.waw.pl
Piotr Lewandowski	P03 IBSPAN	piotr.lewandowski@ibspan.waw.pl
Katarzyna Wasielewska-Michniewska	P03 IBSPAN	wasielk@ibspan.waw.pl
Maria Ganzha	P03 IBSPAN	Maria.Ganzha@ibspan.waw.pl
Anastasia Blitsi	P04 CERTH	akblitsi@iti.gr
Evripidis Tzionas	P04 CERTH	tzionasev@iti.gr
Theoni Dounia	P06 INF	tdounia@infolysis.gr
Fotios Konstantinidis	P10 ICCS	fotios.konstantinidis@iccs.gr
Konstantinos Routsis	P10 ICCS	konstantinos.routsis@iccs.gr
Oscar López	P13 S21SEC	olopez@s21sec.com
Saioa Ros Jiménez	P13 S21SEC	sros@s21sec.com

History

Date	Version	Change
14-Jun-2022	0.1	ToC presented
13-Jul-2022	0.2	First round of contributions integrated
20-Jul-2022	0.3	Second round of contributions integrated.
26-Jul-2022	0.4	Third round of contributions integrated. Version sent to IR
28-Jul-2022	0.9	Version with comments from IR integrated. Version sent to PIC
31-Jul-2022	1.0	Submitted version after PIC review

Key Data

Keywords	Reference architecture, views, enablers
Lead Editor	P01 UPV – Alejandro Fornés
Internal Reviewer(s)	P04 CERTH – Georgios Stavropoulos, Iordanis Papoutsoglou P12 FORD-WERKE – Klaus Schusteritz

Executive Summary

This deliverable is written in the framework of WP3 – Requirements, Specification and Architecture of ASSIST-IoT project under Grant Agreement No. 957258. The document is the last of a series that are devoted to formalising ASSIST-IoT reference architecture. More specifically, it aims at outlining the guiding principles of ASSIST-IoT architecture, altogether with the description of the Views that compose it and a set of considerations for a further system realisation.

This deliverable reports the work done towards defining the final version of ASSIST-IoT Reference Architecture, including guiding principles and key considerations for its realisation. The initial version (D3.5) sketched the 2D-structure and its design principles, introducing the concept of enablers (i.e., the Cloud-native, encapsulated innovative functionalities that ASSIST-IoT provides) and dividing them into Planes and Verticals, with the first set of Views; the intermediate report (D3.6) aimed at going more into detail with regards to the use of containerisation and the rationale behind the selection of Kubernetes framework as the main orchestrator of the system, improving the previous content with the insights gained during the execution of technical work packages. This last iteration focuses on polishing the previous versions, with the goal of offering more curated and clear information while easing architects that leverage this Reference Architecture the process of enablers design and selecting the most suitable technologies for developing their Next Generation IoT systems.

In summary, this document, apart from presenting the design principles of the Reference Architecture, provides its guidelines, recommendations and best practices in the form of Views (each of them expressed from the perspective of specific system concerns, illustrating how the architecture addresses them): the Functional view, which shows the functionality required to fulfil the user needs in different Planes (tackled with a set of enablers); the Node view, representing the underlying infrastructure element unit (hardware, pre-installed software) needed to deploy the envisioned features; the Development view, which is introduced for the first time to guide the process of an enabler design and realisation; the Deployment view, which provides guidelines for deploying a system and the enablers that it will host; and the Data view, presenting a high-level perspective on data collection, processing, and consumption. These Views are complemented with a collection of vertical capabilities (which address all-encompassing concerns, properties and transversal functionalities, such as Security and Manageability, among others) and additional considerations realising Next Generation IoT systems. This final architecture document concludes the WP3 activities of the project.

.

Table of contents

List of tables

List of figures

List of acronyms

Acronym	Explanation
ACL	Access Control List
AI	Artificial Intelligence
AIOTI	Alliance for the Internet of Things Innovation (SDO)
AMD	Advanced Micro Devices (CPU processor)
API	Application Programming Interfaces
AR	Augmented Reality
ARM	Advanced RISC Machine (CPU processor)
BIM	Building Information Model
BKPI	Business KPI reporting enabler
CAD	Computer-Aided Design
CAP	Consistency, Availability, Partition Tolerance
CD	Continuous Delivery/Deployment
CI	Continuous Integration
CNI	Container Network Interface

CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CRI-O	CRI plus Open Container Initiative (container runtime)
DevOps	Development and Operations
DevSecOps	Development, Security and Operations
DLT	Distributed Ledger Technology
DMZ	Demilitarized Zone
DNS	Domain Name System
DoS	Denial of Service
EDBE	Edge Data Broker enabler
ETSI	European Telecommunications Standards Institute (SDO)
FaaS	Function as a Service
FCAPS	Fault, Configuration, Accounting, Performance, Security
FL	Federated Learning
FPGA	Field Programmable Gate Arrays
GDPR	General Data Protection Regulation
GPS	Global Positioning System
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Calls
GWEN	Gateway/Edge Node
HLA	High-Level Architecture (from AIOTI)
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IdM	Identity manager enabler
IEC	International Electrotechnical Commission (SDO)
IEEE	Institute of Electrical and Electronics Engineers (SDO)
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization (SDO)
ISP	Internet Service Providers
IT	Information Technology
JSON	JavaScript Object Notation
K8s	Kubernetes
KPI	Key Performance Indicators
LSP	Large-Scale Pilots

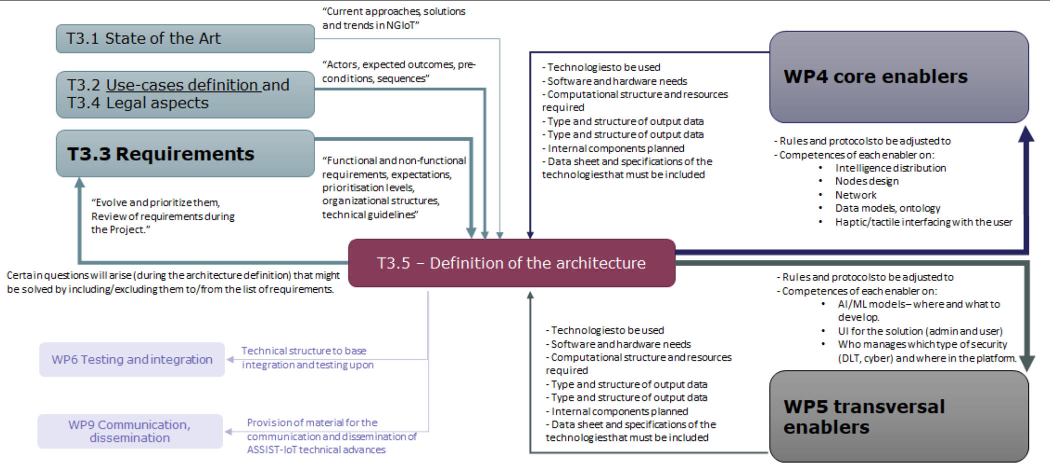
LTSE	Long-Term Storage enabler
MANO	Management and Orchestration (from NFV)
ML	Machine Learning
MQTT	MQ Telemetry Transport
MR	Mixed Reality
NFV	Network Functions Virtualisation
NGIoT	Next Generation IoT
NoSQL	Not only SQL
OS	Operating System
OSI	Open Systems Interconnection
OWASP	Open Web Application Security Project
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Point
PLC	Programmable Logic Controller
PRP	Policy Repository Point
PUD	Performance and Usage Diagnosis enabler
PUI9	Prodevelop User Interface
RA	Reference Architecture
RAM	Random Access Memory
REST	Representational State Transfer
SemVer	Semantic Versioning
SD	Secure Digital (memory card)
SD-WAN	Software-Defined WAN
SDN	Software-Defined Networking
SDO	Standards Developing Organisations
SIM	Subscriber Identity Module
SNMP	Simple Network Management Protocol
SOA	Service-Oriented Architecture
SPA	Single-Page Application
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security

TOGAF	The Open Group Architecture Framework
TPU	Tensor Processing Units
UI	User Interface
UML	Unified Modelling Language
UWB	Ultra-Wide Band
VIM	Virtualised Infrastructure Management
VM	Virtual Machine
VNF	Virtual Network Functions
VPN	Virtual Private Network
VR	Virtual Reality
WAN	Wide Area Network
XACML	Extensible Access Control Markup Language
YAML	YAML Ain't Markup Language

1. About this document

The main objective of this document is **to present the final specification of the ASSIST-IoT Reference Architecture**. Considering that the most prominent outcome of the project is precisely this, this deliverable is key for understanding the rest of the technical content of ASSIST-IoT. More specifically, D3.7 aims at enhancing the content depicted in D3.6 (delivered in M15) considering the continuous research conducted under the scope of T3.5, as well as the knowledge gained during the technical development of the project during the period M6-M21. Particularly, the enablers' development and implementation allowed the fine-tuning of the previous iterations of the architecture, thus being now more concrete, useful, and usable. This document sets the ground for realising Next Generation IoT systems, including a set of design principles, guidelines, and recommendations with that aim. The current document is the final delivery of the project with respect to the Reference Architecture and marks the end of the activities of WP3 – Requirements, Specifications and Architecture.

1.1. Deliverable context

Item	Description
Objectives	**O1**: D3.7 is the first definition of the architecture, that is the outcome of objective 1. **O2**: Smart network's Functional View is provided with overview of its core enablers. In any case, its realisation is under the scope of WP4 **O3**: Security and Privacy vertical capabilities are described with overview of main enablers, still, technical outcomes are delivered by WP5. **O4**: Federation of smart AI enablers is preliminary outlined, being inherent part of different enablers of the Planes and Verticals of the architecture.
Work plan	 T3.1 State of the Art; "Current approaches, solutions and trends in NGIoT"; T3.2 Use-cases definition and T3.4 Legal aspects; "Actors, expected outcomes, pre-conditions, sequences"; T3.3 Requirements; "Functional and non-functional requirements, expectations, prioritisation levels, organizational structures, technical guidelines"; "Evolve and prioritize them, Review of requirements during the Project."; Certain questions will arise (during the architecture definition) that might be solved by including/excluding them to/from the list of requirements.; T3.5 – Definition of the architecture; WP4 core enablers; - Technologies to be used - Software and hardware needs - Computational structure and resources required - Type and structure of output data - Type and structure of output data - Internal components planned - Data sheet and specifications of the technologies that must be included; - Rules and protocols to be adjusted to - Competences of each enabler on: • Intelligence distribution • Nodes design • Network • Data models, ontology • Haptic/tactile interfacing with the user; - Rules and protocols to be adjusted to - Competences of each enabler on: • AI/ML models – where and what to develop. • UI for the solution (admin and user) • Who manages which type of security (DLT, cyber) and where in the platform.; WP5 transversal enablers; WP6 Testing and integration; Technical structure to base integration and testing upon; WP9 Communication, dissemination; Provision of material for the communication and dissemination of ASSIST-IoT technical advances
Milestones	This deliverable marks the completion of *MS5 – Final architecture defined*
Deliverables	This deliverable is fed by its former iterations (D3.5 and D3.6, Architecture definition initial and intermediate versions). It will also be feeding into the technical outcomes under development on D4.3, D5.4 and D5.5.

1.2. The rationale behind the structure

Deliverable D3.7 is the last iteration of the ASSIST-IoT Reference Architecture, aiming at being a reference for the Next Generation IoT. In contrast to D3.6, its aim is to present the final architecture as a self-contained document, without having to go through previous iterations to retrieve all the key information. In addition, Section 1.6 provides a digest of the updates and refinements from the previous iteration, and a summarised version of the architecture is presented in Appendix B to ease its sharing.

The core document starts with Section 2, devoted to presenting a refined and re-arranged version of the ASSIST-IoT approach, this is, its high-level architecture and its design principles. Then, Section 3 formalises the five Views that make up the Reference Architecture, extending previous content from insights and knowledge gained during the execution of the project (including a new view related to development of enablers). Section 4 depicts the Verticals, which complement the previous Views with a set of transversal features and properties that are part of the architecture due to its design choices and the introduction of specific enablers. These two previous sections are crowned by Section 5, which introduces different aspects that should be considered before an ASIST-IoT system realisation. Finally, conclusions are summarised in Section 6.

The architecture is accompanied by two appendices: Appendix A, where content complementing the Data view is presented for data pipeline realisation, and Appendix B, which, as aforementioned, serves as a summarised version of the core document. Despite not providing additional/complementary information, it serves as a brief introduction to the architecture so potential users get familiar with it and decide whether it fits their needs or not without having to go through the whole document (in the scope of the action, it will be used for potential Open Callers that aim at integrating their solution with ASSIST-IoT, so they know what does it provide and what is expected from them).

1.3. Outcomes of the deliverable

This section aims at summarising and highlighting the more relevant results of the deliverable. First, it should be stressed that the information provided is strongly based on the inputs already provided in the previous iterations of the architecture. Instead of presenting it as a document that points to the previous versions and just delves into the novelties, the consortium decided to develop a self-contained report of the final Reference Architecture, so future readers have all the key information in a single document without needing to consult all three versions simultaneously (only the State-of-the-Art-related data has been left aside and referenced when needed).

The document starts with an introduction of the approach followed for defining the Reference Architecture. The conceptual, layered, **high-level architecture** is presented as a set of horizontal Planes crossed by Verticals, which provides and includes the functionalities and properties expected for Next Generation IoT systems. This architecture is rooted in different **design principles**: apart from considering a layered approach, it adopts concepts of architectures based on microservices; it also follows Cloud-native principles adapted to the edge, realising the functionalities by means of containers due to their benefits for edge-oriented ecosystems (lightweight, portability, ease of updating, maturity, etc.); introduction of the **enabler concept**, which represents a collection of (usually) software components working together to provide a specific functionality for the system, considering encapsulation principles to facilitate integration and include inherent security; and leveraging a container orchestration framework, specifically Kubernetes, to manage all the former to ease their adoption.

Guidelines, recommendations, and best practices for stakeholders are presented by means of architecture **Views**, each of them addressing specific concerns of the Reference Architecture. Specifically, the following Views are included:

- **Functional view**: it shows the functionality required to fulfil the user needs and address the stakeholders' concerns. This view presents the features incorporated into the Reference Architecture, logically separated into horizontal Planes, namely the Device and edge plane, the Smart network and services plane, the Data management plane, and the Application and services plane. More significant changes with respect to previous iterations have been included in the Device and edge plane, with more detailed data with respect to the enablers that can be framed in it.
- **Node view**: it represents the underlying infrastructure element unit needed to deploy the envisioned enablers. It provides updated information about the hardware and pre-installed services required (e.g., Operating System, virtualisation technology, container orchestration framework and related add-ons), and it has been extended to include additional considerations with respect to processing elements (e.g., CPU architectures, GPU leveraging, etc.).
- **Development view**: this view is introduced in this iteration of the Reference Architecture. It offers high-level recommendations and guidelines to ease enablers' design and development process, from

conception to realisation. It complements the information provided by the DevSecOps methodology, which takes over once an enabler has been properly designed.

- **Deployment view**: it provides guidelines and recommendations to facilitate a Next Generation IoT system realisation in a specific environment, involving different aspects such as the network provisioning, the Kubernetes setup, the deployment of enablers and their integration to address specific uses cases (i.e., provide more advance services in comparison to those offered by a single enabler). In this version, each aspect has been refined and separated into a particular sub-section to facilitate their comprehension.
- **Data view**: this view considers the specific actions that a set of sequential enablers perform upon data. This view is realised by means of *data pipelines*, which are descriptions with visual and textual components of how data flows in distinct parts of Next Generation IoT deployments (in terms of data collection, processing, and consumption).

The aforementioned Views are complemented with a set of **Verticals**, which address all-encompassing concerns, properties and functionalities that are transversal to them. They can be inherent properties of the system, included by enablers from other Planes/Verticals or coming from the design principles, or realised by means of dedicated enablers. The five Verticals envisioned are: Self-*, Interoperability, Security (plus Privacy and Trust), Scalability and Manageability. In this document, the information provided in D3.5 is extended and improved, and the list of enablers and provided features envisioned for each of them is updated.

Lastly, **complementary information to the Views for realising an ASSIST-IoT architecture** is provided. They include different aspects, such as (i) the introduction of the essential enablers integral for all (or at least, the majority of) Next Generation IoT systems; (ii) the set of common endpoints that every enabler should incorporate to ease their integration; (iii) the decentralised approaches for monitoring and logging both the enablers and the system itself (high-level architecture and realisation example); and (iv) the introduction of encapsulation exceptions (i.e., enablers that cannot follow the principles expected from them due to their individual characteristics). The Reference Architecture is complemented with two appendices, being one a summary of the core report to be shared with third parties as an introductory document.

1.4. Lessons learnt

During the execution of the involved task and in WP3 in general, there are some knowledge and insights that have influenced the definition of the ASSIST-IoT Reference Architecture as well as the developments that have emanated from the execution of the technical work packages. The following lessons have helped to develop a more valuable blueprint that can be used for constructing modern IoT systems that address the needs of use cases from different sectors:

- Despite the fact that there are models and technologies that can link the physical and digital worlds, they lack consistent interfaces and Reference Architectures, which can hinder the commercial success of Next Generation IoT solutions in the absence of a shared standards framework.
- In the terms of innovation technologies, tactile systems, cloud/edge computing continuum, and AI analytics are key ones that a Next Generation IoT systems should embrace. Nevertheless, there are several security, safety and privacy concerns that should be addressed before their operational deployment.
- ASSIST-IoT pursuits create some legal challenges that are mainly related to the use of Artificial Intelligence in relation to safety and security, as well as privacy and requirements of GDPR. Reliability and validity of the datasets used for training purposes are crucial in terms of ensuring safety in the deployment of the ASSIST-IoT AI-based algorithms.
- The Device and edge plane was considered mostly from the pure hardware perspective in previous versions of the document. In this final iteration, information about the enablers that can be encompassed within the scope of this plane is depicted.
- A new view has been included (Development view), as there was a lack of guidelines regarding how enablers have to be designed at early stages (at later ones, DevSecOps methodology should be followed).

- Clearer information about Distributed Ledger Technologies and security schemas for decentralised, IoT environments have been included to cope with the characteristics of this kind of systems.
- Alternative solutions and design choices were proposed for areas (such as specific hardware or self-* schemas) that face difficulties in encapsulation, which may also guide the design of future software, whose inherent properties hamper following encapsulation principles. This version includes the new encapsulation exceptions found.
- Common conventions for enablers, which include their endpoints (following a specific structure and a set of mandatory features to be included), how to retrieve and store logs, and how to monitor the status of all enablers. Initial descriptions had to be modified to be adapted for Kubernetes, Cloud-Native environments.
- Within the considerations for systems realisation, special direction is given to develop health metrics for enablers. While generic solutions can be used and adapted for general enablers, custom, adapted solutions are needed for encapsulation exceptions (i.e., working out of Kubernetes systems).

1.5. Deviation and corrective actions

Since the intermediate version dated from just 6 months ago, which were also the last six months of WP3 execution, there have not been many deviations to document. The following ones can be highlighted:

- Any generic enabler for Device and edge plane has been fully formalised. As corrective action, (i) examples of which kind of enablers belong have been included, and (ii) pilot-specific enablers are under development within the scope of pilot-related activities.
- The process of how enablers are designed lacked of dedicated information. In an effort to mitigate this, a fifth view (Development view) has been included in the Reference Architecture.
- The previous strategy for enablers' logging was not scalable (nor realistic) enough; hence, a new strategy has been envisioned and formalised, accompanied by examples for its realisation.
- New encapsulation exceptions were found within developments in technical work packages; they have been formalised, along with the underlying motivation (i.e., what has motivated these exceptions).

1.6. Version-specific notes

This section contains a summary of new or significantly-updated content with respect to the information provided in the previous iterations of the deliverable (D3.5 and D3.6), separated by sections.

ASSIST-IoT approach

- The conceptual architecture and the design principles remain valid. An arrangement of the content has been made to avoid presenting the enabler concept twice, improving readability.
- Comparison of containers with respect to Unikernels and Serverless (FaaS) virtualisation included, extending the previous comparison with respect to Virtual Machines.
- The micro-application concept has been introduced, as enablers might be composed of functional units larger than microservices (hence, enablers consist of a group of micro-applications, which might or might not be microservices, providing a specific functionality for the architecture).
- State-of-the-art-related information (concepts, available architecture paradigms, comparison with influencing architectures) pointed to previous iterations, not included here.

Architecture views

- The **Functional view** has not suffered major changes. The most important element is the addition of information related to enablers that can be encompassed under the scope of the Device and edge plane, as none have been extensively described. Enablers from this plane will come from WP7 activities, although they will be pilot-specific.
  - Enablers providing features of the Device and edge plane are now classified into three functional blocks, namely: "edge intelligence", "pre-processing functions" and "communication capabilities" (from previous versions, analytics, AI capabilities and

enhancement of IoT devices smartness have been grouped into a single block, i.e., edge intelligence).

    - A generalisation of the content has been made with respect to the hardware that could act as an ASSIST-IoT gateway/node, as previous versions were too centred on the project's GWEN.
    - Regarding the Smart network and control plane, the functional blocks have been modified, eliminating the "VNF" block (there was not any clear distinction from the rest, as enablers from other blocks could be realised as VNFs) and introducing the "Access network management" block instead. Content has been refined, stressing the fact that additional enablers can be incorporated.
    - The Data management plane remains largely unchanged; more extensive information has been given for the enablers of the semantic block, introducing the "Semantic suite" as a high-level tool composed of the three semantic-related enablers.
    - In the initial designs, this plane promised to include security-related data functions, implemented with the use of Distributed Ledger Technologies. Those include, for instance, non-repudiation and integrity verification, auditing, tamper detection, etc. It was decided that, because of the cross-cutting nature of these technologies, they should not be advertised as a set of plane functions, but rather as a vertical capability.
    - The Application and services plane sub-section has extended the descriptions provided for the defined enablers, but the main content remains valid. Again, it is stressed that additional enablers could be considered for this plane (as well as for the rest), especially with respect to the AR/MR/VR block.
- The **Node view** has been extended, allocating a sub-section dedicated to CPU processing architectures and GPU usage within the context of containerised workloads.
    - This is important as some enablers might not work in some environments of the computing continuum if container images have not been prepared carefully, or might not have good performance without hardware acceleration capabilities enabled.
    - Also, more clear information about the pre-installed software and tools needed to be present at the nodes is given, with concrete realisation examples (taken from the technologies considered within the technical work packages).
- The **Development view** has been introduced in the architecture. It aims at providing some guidelines for designing enablers (from the requirements elicitation to components definitions, interactions among them, data-related compliance with privacy and ethical regulations, endpoints and methods, and user stories) and to consider DevSecOps methodology to realise them.
- The content of the **Deployment view** is similar to the previous iteration, with some refining. Major changes occurred with respect to the organisation of the content, dividing it into different sub-sections to add clarity. Also, larger information has been provided on the options related to enablers deployment, aiming at not forcing a specific packaging and deployment tool (in any case, its selection affects the implementation of the Smart orchestrator, in charge of controlling the enablers' lifecycle).
- Finally, the **Data view** has been extended, including more textual information and clear examples of how data pipelines should be formalised, including wrong examples. Additional content has also been included in the related appendix.

Verticals

- Verticals were not discussed in the intermediate iteration (although some sections related to Security, Distributed Ledger Technologies and Federated Learning strategies were included in it); hence, refinements were lengthier and more detailed in this part in comparison to other sections.
- The introduction of **Self-*** has been condensed, while the scope of the five enablers defined tackling this vertical has been greatly refined and contextualised with respect to self-* properties. It has been remarked that additional enablers bringing system autonomy or semi-autonomy (with a certain degree of intelligence) could belong to this vertical, not being limited to the ones under development within the project.

- **Interoperability** does not have any enabler directly devoted to it. In any case, it is a property present in different parts of the architecture, and in this final version, several examples are provided to prove this statement.
- The vertical related to **Security, Privacy and Trust** is the one that contains more enablers (if Federated Learning-related enablers are assigned to the Privacy property, a total of 12 would belong to it). Apart from a new introduction to each of these properties (and the enablers defined), some sub-sections have been included:
    - Some aspects related to security identity and access management are discussed within the scope of decentralised ecosystems, focusing on policy sharing mechanisms.
    - Security considerations with respect to MQTT and third-party access are also presented and analysed, as the implemented mechanisms should be tailored to the characteristics of the addressed business scenarios.
    - Finally, detailed insights about Distributed Ledger Technologies are given: how they fit in decentralised environments, which data should be stored, process of assessment, how enablers are integrated with the ledger, etc.
- **Scalability**, as interoperability, is not addressed by means of any specific enablers. This property is (or should be) part of a Next Generation IoT system thanks partially to the modular characteristics of the considered hardware and software (e.g., Kubernetes). This version stresses that enablers can (and should) be developed with this property in mind, and an example of how Federated Learning enablers address it is provided.
- Finally, the scope of the **Manageability** vertical has been refined, reducing verbosity and bringing clarity. With respect to the first iteration, two of the enablers have been merged (those related to enablers orchestration and output management), and the naming and scope of the three remaining ones, refined.
    - The scope of the Composite services manager, which aims at facilitating the definition and instantiation of agents to integrate enablers (in terms of protocol and basic data transformations), has been slightly reduced to facilitate the realisation of data pipelines.
    - Other manageability tools and mechanisms present in the rest of the Planes/Verticals are also briefly depicted here.

Considerations for architecture realisation

- The list of essential enablers for an architecture realisation has been reduced by one, as the Logging and auditing enabler is no longer part of it. The system administrator should decide if this is needed based on the analysis performed following the guidelines exposed in the related sub-section of the Security, Privacy and Trust vertical.
- The common endpoints have been kept from previous iterations, with some refinement coming from practical insights gained during the execution of the technical work packages. Specifically, /api-export has been the one that has seen more changes, with the introduction of two sub-endpoints.
- The monitoring and logging approaches have been modified to be more technologically agnostic; still, the implementations carried out in the technical work packages are depicted as potential realisation examples. The idea, not just in this section but in the whole document, is not to force any specific technology but give tested examples of implementation (which can then be considered or not).
    - The high-level logging architecture has been modified as the one included in the previous version was not scalable enough (in which a dedicated agent was included for each enabler). Therefore, in the final logging stack, an agent should be present per computing node, and not per enabler.
- In the previous version, only hardware-related encapsulation exceptions (i.e., enablers that cannot follow enablers' design principles) were presented. In this final version, exceptions related to encapsulation infeasibility (or worthlessness) are also explained, with two enablers fitting in this category (self-healing and cybersecurity monitoring agent), which should be installed over a node's Operating System.

2. ASSIST-IoT approach

2.1. Introduction

According to [1], the Internet of Things (IoT) is an *infrastructure of interconnected objects, people, systems and information resources together with intelligent services to allow them to process information of the physical and the virtual world and react*. It is, in its wider sense, a combination of existing technologies that are integrated to fulfil things-enabled applications' requirements rather than a technology per se [2], involving multiple disciplines to solve connectivity, data processing, interfacing, regulatory and security aspects, among others. With IoT consolidated in several application domains, the Next Generation IoT (NGIoT) appeared to address more ambitious and complex use cases. According to NGIoT action, a Coordination and Support Action for the Next Generation Internet of Things, the following technologies have been identified as key enablers towards this evolution [2]: Edge Computing, 5G (including NFV features), Artificial Intelligence and analytics, Augmented Reality and Tactile Internet, Digital Twin and Distributed Ledgers.

As already happened with IoT, there is no single architecture that can suit all the technical and non-technical needs of all the NGIoT systems: there are simply too many topologies, access networks, protocols, devices, data types and technologies involved. Still, a set of requirements, guidelines and recommendations are necessary to guide the design, development and implementation to facilitate these NGIoT realisations, especially now that the complexity of the requirements (and hence, the number of involved technologies) has increased. Reference Architectures (RAs) appeared to this end, to provide a set of basic functionalities, information structures and mechanisms [3] to serve as a blueprint for developing and implementing successful IoT architectures. They should be flexible enough to be applied in different domains and implementable considering different technologies, but without becoming so abstract as to be impractical.

One of the main objectives of ASSIST-IoT is precisely to deliver a RA for the NGIoT, which is the scope of the present document. Aiming at avoiding a high degree of abstraction, a clear set of design principles are presented in this section, following a Cloud-Native paradigm adapted to the edge-cloud continuum as a baseline for its conception.

2.2. Conceptual architecture

ASSIST-IoT Conceptual Architecture was not created from scratch, but rather designed considering multiple inputs including (i) the current trends towards integration between IoT-related technologies with complementary technologies (such as Edge Computing, Artificial Intelligence and SDN/NFV paradigms), (ii) the expertise of the Consortium partners in different technological areas, (iii) the outcomes of previous and concurrent projects, as well as of Standards Developing Organisations (SDOs), and an (iv) extensive research of innovative concepts to improve current performance and scalability of IoT architectures or integrate novel functionalities. More specifically, the ASSIST-IoT RA is influenced mostly by three architectures, namely the ones provided by the IoT European Large-Scale Pilots (LSP) programme [4], the OpenFog consortium [5], and AIOTI HLA [6] (**Note**: a comparison with them was provided in previous versions of the deliverable).

As presented in D3.5 [7], several architecture paradigms can be used or combined for building a RA or a system (layered patter, microkernel, event-driven, space-based, Serverless, based on services, etc.). One of the first design choices made in the initial phases of the ASSIST-IoT was to conceive a layered approach for its NGIoT blueprint. This was driven mainly by its simplicity to represent the functionalities and properties expected from it. In particular, the conceptual architecture is rooted in a multidimensional approach, in which **horizontal Planes** are intersected by **Verticals**, allowing for a higher level of modularity. Planes represent collections of functions that can be logically layered on top of one another. For example, observation data originating from a sensor must pass through the Smart network and control plane and be processed on the Data management plane, before being presented to an end-user in a graphical interface on the Applications and services plane. Not all information must always pass through all Planes – in fact, one common function of an edge device belonging to the Device and edge plane is to filter out only necessary data, or aggregate data, so that only relevant information is passed on. It is worth mentioning that the ASSIST-IoT concept of Planes must not be confused with the traditional (OSI-like) approach of the protocol stack but rather as a smart classification of logical functions that fall under the scope of diverse plane domains.

Verticals, on the other hand, represent inherent properties of the system or cross-cutting to the rest of the architecture, as well as functions targeting specific NGIoT properties. For example, although one may deploy a fully secure identity and authorisation stack, the security should be extended through all the Planes, from the network to application code. A high-level description of the ASSIST-IoT RA can be shown in Figure 1.

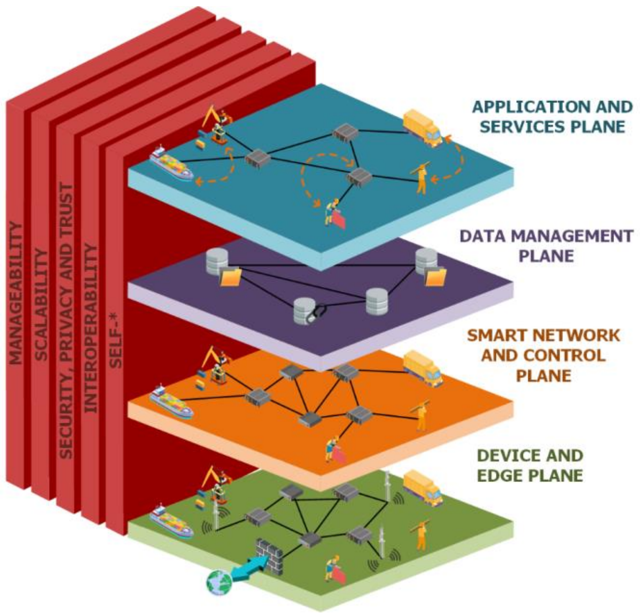

MANAGEABILITY
SCALABILITY
SECURITY, PRIVACY AND TRUST
INTEROPERABILITY
SELF-*
APPLICATION AND SERVICES PLANE
DATA MANAGEMENT PLANE
SMART NETWORK AND CONTROL PLANE
DEVICE AND EDGE PLANE

Figure 1. ASSIST-IoT Conceptual Architecture

The ASSIST-IoT horizontal Planes as well as Verticals are briefly summarised next, with additional information provided in the Functional view and the Verticals section of the RA (Section 0 and 4, respectively):

Horizontal Planes

- **Device and edge plane** describes the collection of functions that can be logically appointed to physical components of IoT, including, but not limited to, smart devices, sensors and actuators, wearables, edge nodes, as well as network hardware, such as hubs, switches and routers. Hence, apart from including the physical elements that realise the computing and networking infrastructure, it encompasses those functionalities required to either perform local intelligent analysis/actions, or pre-process and lift data to services from upper Planes.
- **Smart network and control plane** manages virtual and wireless aspects of network connectivity. The key functions handled on this plane are encompassed by technologies that deliver software-related and virtualised networks (e.g., SD-WAN, NFV, MANO). The plane follows an access-network-agnostic approach, in which the network connections are highly flexible. Features such as dynamic network configuration, routing, tunnelling and high-level intelligent firewalling are provided by this plane.
- **Data management plane** handles all functions related to a virtual shared data ecosystem, in which data are acquired, delivered and processed to provide key data-related functions. These include all those mechanisms related to data routing (i.e., moving data between computation nodes and/or services) interoperability (semantics) and storage, among others.
- **Application and services plane** crowns the Planes with end-user and admin functions and services. It delivers a layer of abstraction that benefits from the services provided by the lower Planes. Also, it combines them to provide synergistic value for the whole system. It offers the gained insights through a set of human-centric and/or tactile interfaces, to grant users and 3rd-party systems access to the system.

Verticals

- **Self-*** encompasses a set of features that provides autonomous or semi-autonomous capabilities alongside different dimensions. In particular, self-* vertical involves different capabilities such as self-diagnosis and self-healing for autonomously detecting and fixing faulty elements, self-configuration and self-provisioning for autonomously configuring and provisioning resources before an eventually increased demand (based on statistical predictions), among others.

- **Interoperability** is a characteristic of the system that ensures that its internal products and services can work among them, and that the system as a whole can work properly with others. There are different interoperability levels that should be contemplated, including technical (technologically possible), syntactic (so data can be exchanged in case that the interface and programming languages are different) and semantic (so these data can be understood in a similar way, implying a precise and unambiguous meaning of the exchanged information.
- **Security, Privacy and Trust** vertical aims at providing the following key functionalities along the architecture, including authorised registration of devices, trusted access for sharing data on multiple domains, secure and private storage, and monitoring and actuation mechanisms for responding to potential cyberthreats, among others. These architecture traits should be carefully analysed, as any flaw may hinder or even preclude a system's adoption.
- **Scalability** is an essential property for adapting to different workloads, performance, costs, and other business needs that can appear in an NGIoT deployment. A system should be flexible enough to adapt to demands or requirements coming from any given business needs, and could involve different hardware (introduction/removal of nodes, network interfaces, storage spaces, etc.), software (automatic scheduling of resources, allowing different topological options, etc.) and communications aspects.
- **Manageability** handles the configuration of different elements of the systems, from the management of computation nodes to the deployment, configuration and termination of the functionalities provided by the rest of Planes and Verticals, considering ease of use as a key adoption factor. Also, this vertical should ensure that the different features are interfaced correctly so complex services for solving use cases can be built by combining them.

Since it is not possible to capture all the functional or cross-cutting features, recommendations and guidelines in a single model, the ASSIST-IoT architecture has resulted in different views, namely the Functional, Node, Development, Deployment, and Data view (presented in Section 3).

2.3. Design principles

The design principles of the ASSIST-IoT RA have been considered following the pillars of (i) decentralisation, (ii) scalability, (iii) software lightness and modularity, (iv) adoption potential and (v) production-readiness. IoT and edge ecosystems are not comparable to the cloud; computing resources are lower, less stationary (they might be increased, decreased or reallocated) and might be geographically scattered. Hence, it is mandatory that the selected principles grant the possibility of having different types of system topologies and are flexible enough ahead of modifications in terms of hardware resources and usage.

The NGIoT brings more appealing applications but also additional complexity. There are many new services and technologies that must be now considered (DLT for the edge, technologies for supporting AR/MR/VR requirements, AI techniques adapted to the edge, etc.), and that might be present or not depending on the particular use cases addressed. Due to that, software modularity should also guide the design principles, as new technologies have appeared or evolved much faster nowadays than in the past and there should be mechanisms that ease adaptation while avoiding disruptions in the services provided.

The last, but likely the most critical part, is related to adoption. Devising a comprehensive, thorough RA that covers all the expected topics with clarity and offers suitable guidelines and recommendations but is not considered/used, brings as low value as a misguided, poorly-conceived RA. The design principles must consider the actual and foreseen trends in the market, in terms of needs, implementation, and usability to minimise the barriers that architects may perceive from leveraging it. In summary, the design principles of ASSIST-IoT are: (i) the split of the different functionalities of the architecture into **micro-applications**, (ii) the instantiation of the former into **containers**, (iii) the introduction of the **enablers abstraction**, and (iv) the orchestration of these enablers considering small **K8s footprint**.

Microservices

As aforementioned, different architecture paradigms can be considered and combined for building a system or, in this case, a RA. The ASSIST-IoT RA, apart from rooting in a layered multidimensional approach, is also based on services, in which a complex problem (like the development of an NGIoT system) is broken into a series of simpler ones [8] (**Note**: The rationale for selecting this paradigm was provided in D3.5, Section 2.2

[7]). Among the different subtypes of architectures based on services (i.e., monolithic, Service-Oriented Architecture – SOA and microservices), ASSIST-IoT leverages concepts from **microservices-based architectures**, in which an overall system (in this case, the RA) is divided into services that communicate among them leveraging their own, well-defined API.

The extended rationale is provided in Sections 2.2 and 3.1 from D3.5, but essentially this selection has been made as: (i) internal **services** are **loosely-coupled**, so they can be maintained, deployed and scaled in an independent way; (ii) there is **no need** of any kind of **communication middleware** or bus to communicate the different services; (iii) the overall system becomes much more **flexible**, as new services can be added more easily and, since they are (quite) decoupled, an error within a service does not jeopardise the operation of the others; and (iv) they allow **coding** services **using different programming languages**. In particular, the goal of the microservices-based architecture of ASSIST-IoT is to build a set of micro-applications (called enablers) that are each responsible for executing one function (as opposed to the monolithic way of building one application that executes everything), and to let each of them be autonomous, independent, and self-contained.

Containerisation

As IoT deployments become more numerous, scalable architectural solutions are crucial for meeting and sustaining the demand for large, expanding, and elastic networks. While former IoT applications simply control and analyse signal data from edge devices in a centralised manner, NGIoT applications are moving towards smart analytical applications, requiring horizontal scalability of device additions, including a reduction of instantiation times, or migrating the actual computing and intelligence towards the edge. To promote rapid onboarding, ASSIST-IoT proposes the use of **containers** as the standard unit that **packages the code and dependencies** needed for a micro-application to work. Other options, like installing them as a function/service over a given Operating System (OS), as a Virtual Machine (VM), as a Unikernel, or as a function over a FaaS (Function as a Service) environment, etc., have also been considered.

This choice is motivated by several factors. Firstly, it is about lightness. Containers can virtualise at the OS level, and then multiple containers can be running over the OS kernel directly. Hence, containers are far more **lightweight** than VMs, they can be **deployed much faster**, and use a fraction of the memory. This is critical as the available computing resources might be scarce. Secondly, containers are very flexible and can be installed over several host OSs and CPU architectures, facilitating the execution of workloads over different types of equipment. Lastly, maturity is also key. Most development teams know how to work around them, having their DevOps processes prepared to embrace them.

As outlined, the advantages of containers with respect to VMs come from their lightweight, deployment time and ease of execution. Regarding Unikernels[1], these require high technical knowledge about OS kernel functions to be properly designed and developed. They, as occur with VMs, expose less attack surfaces as the OS kernel is incorporated on them (which brings a higher level of isolation), yet containers are widely considered a secure virtualisation technology. Serverless (FaaS) platforms[2] adapted to the edge could be interesting, but complex services can be hard to set up and manage. Overall, the two latter approaches are interesting for the edge-oriented ecosystems, but need expertise currently scarce in the market, which could impose an adoption barrier. A comparison of these aspects for the considered technologies is given in Table 1.

Table 1. Containers vs Virtual Machines vs Unikernels vs FaaS

	Containers	VMs	Unikernels	FaaS
Overhead	🟩	🟥	🟩	🟩
Isolation	🟨	🟩	🟩	🟨
Deployment time	🟩	🟥	🟩	🟩
Technological maturity	🟩	🟩	🟥	🟨

[1] Simplified versions of VMs, where only those kernel functions actually needed by the function/service offered are included
[2] They allow deploying functions that interact among them based on declared events

Enablers abstraction

Enablers are the cornerstone of ASSIST-IoT architecture. The introduction of the "enablers abstraction" responds to the realisation of a modular architecture to deliver the functions promised by ASSIST-IoT innovations and future capabilities. In essence, an enabler is a **collection of software** (and possibly hardware) **components - running on computation nodes - that work together to deliver a specific functionality of a system**. ASSIST-IoT enablers are not atomic but presented as a set of interconnected components, bound together in a single package. In particular, an enabler component is an artifact that can be viewed as an internal part of an enabler that performs some action necessary to deliver the functionality of an enabler as a whole.

Enablers often correspond to specific architectural blocks and deliver functionality that is a part of a given plane or vertical. Hence, depending on the characteristics of a specific scenario, it may occur that some enablers are optional, and should be deployed when needed, while others are critical for delivering the full scope of ASSIST-IoT. Hence, the RA's modularity is achieved by splitting functionalities into enablers, so only those relevant to a particular scenario need to be deployed. Enablers that must be deployed for a system to comply with ASSIST-IoT are called *essential* (see Section 5.1).

In another vein, it should be noted that multiple enablers may be used in a system to address a particular use case, leveraging and combining their provided features. One of the most important design principles that distinguish components from enablers is that enablers should not directly communicate with components of other enablers, unless explicitly allowed. This is referred to as encapsulation. Following the previous description, a high-level mock generic enabler is presented in Figure 2. A set of guidelines and recommendations for designing and developing enablers can be consulted in the Development view (Section 3.3).

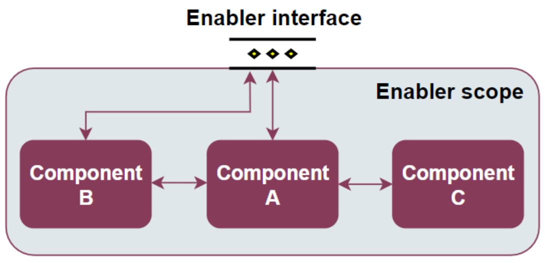


Figure 2. Mock enabler overview

Kubernetes

Designing and developing enablers considering concepts from microservices and being realised as a set of containers is a good starting point, still, the breach between laboratory and production-readiness would be quite large. A **container orchestration framework** brings too many advantages to be neglected. It facilitates the execution of different tasks related to deploying and managing containerised applications, and are specially needed for managing systems with a (potentially) large number of containers instantiated in different servers (e.g., a cluster environment). Solutions such as Kubernetes[3] (K8s), Docker Swarm[4] and Apache Mesos[5] are in charge of (i) managing the creation of containers, (ii) verifying their operation, and (iii) offering correct management of errors.

In particular, K8s is strongly recommended as the main technology for containers' orchestration, and thus, for enablers orchestration. The main motivation for choosing this option lies in it being the *de facto* standard in current trend towards Cloud-Native implementations in contrast to other alternatives, as well as the evolution of K8s distributions with lower memory footprint to cope with the constraints posed by edge-oriented ecosystems. Specifically, K8s **automates rollouts and rollbacks** and monitors the health of workloads to prevent bad rollouts before things start to fail. It also **runs health checks** periodically against deployed services, restarting containers that fail. Also, K8s will automatically scale services up or down based on utilisation, ensuring more **optimal use of resources**. Overall, having K8s as the underlying orchestration platform will close the gap for further market adoption of a given NGIoT system, enabling a set of appealing features for production environments that would not be present without it.

[3] https://kubernetes.io/
[4] https://docs.docker.com/engine/swarm/
[5] http://mesos.apache.org/

3. Architecture views

As defined in ISO/IEC/IEEE 42010 standard [9], **Views** are a work product to express an architecture from the perspective of specific system concerns[6], illustrating how the architecture addresses them. In other words, borrowing a sentence from TOGAF documentation[7], "*just as a building architect might create wiring diagrams, floor plans, and elevations to describe different facets of a building to its different stakeholders (electricians, owners, planning officials), so an IT architect might create **Views** of an IT system for the stakeholders who have concerns related to these aspects*". The different Views described in the following subsections compose, altogether, the whole scope of the architecture; also, separately, they are useful to understand a concern/domain of use-building of it, providing useful guidelines, recommendations and best practices while allowing a large degree of freedom with respect to actual realisations.

While previous iterations of the architecture introduced the Logical/Functional, Node, Deployment and Data views, in this final version a Development view has also been included. **A View** of the ASSIST-IoT RA **aims at representing an observation prism of the whole specification fit to the wills of a determined stakeholder group**. Whereas all the previous seek to align with the guidelines of the building of a Reference Architecture according to the aforementioned standard, its interpretation in realistic, pragmatic terms might seem a bit loose. The goal of this section **is to illustrate how those Views** (and what they represent) **have an actual impact on the creation of the architecture and also on its usage**. In the next table, a summary of the stakeholder-concern-view pointing is provided:

Table 2. Practical application of architecture Views – a relations summary

View of the RA	Stakeholder to whom is useful	Concern of interest and message transmitted
Functional view	Developers and maintainers	Functionalities provided; enablers to be developed and how are they catalogued/framed
	Acquirers/users	Understand the functionalities (core and vertical) which the architecture is composed of
Node view	HW developers	Create/enhance hardware to deliver a suitable ASSIST-IoT edge node
	Edge device/gateways providers	Select hardware to deliver a suitable ASSIST-IoT edge/node; pre-installed software needed to be allocated
	Developers and maintainers	Know the pre-installed functions and engines that enablers will rely on for their execution
Development view	Developers and maintainers	Understand how an enabler should be designed and developed to include a new functionality
Deployment view	Network/system administrators	Put the network in place and set up the K8s' topology properly
	Developers and maintainers	Understand how enablers can be deployed and integrated to address and solve a specific use case leveraging a set of them
Data view	Data engineers	Understand data transformations to be happening within the architecture and plan their enablers accordingly
	Developers and maintainers	

To further illustrate the practical meaning of each view, some examples (applied to real uses of the RA in ASSIST-IoT) are outlined below:

Practical examples of Functional view: one team requires a 5G core network to manage their own 5G access network. First of all, the acquirers can check whether this feature is already part of the provided functionalities or not. In the latter case (most common case, as 5G core networks are not usually integrated part of an NGIoT system since these are managed by third-party Internet Service Providers – ISPs), developers can be informed of other enablers that can support or should interact with it, e.g., the Smart orchestrator, which would be the one in charge of deploying it (and thus, the 5G core enabler should be packaged accordingly), the VPN and the SD-WAN enablers, which might provision tunnels over it, etc. and how should they exposed the configuration and

[6] A "concern" represents a topic of interest to one or more stakeholders belonging to an architecture. A summary of the key **concepts** related to RA definitions can be found in D3.5, Section 2 [7].
[7] https://pubs.opengroup.org/architecture/togaf8-doc/arch/chap31.html

data interfaces. In addition, this View is also the tangible space where the enabler must be properly informed about the architecture in global (components used, technologies, API description, etc.). Once integrated, an acquirer will be able to observe that the new functionality is already provided by the system.

Practical example of Node view: a company is interested in manufacturing a novel edge device that may become an element of an NGIoT deployment. The Node view allows the company to know: (i) the minimum performance and characteristics to be included, (ii) the physical interfaces to be equipped with, (iii) the operating systems that it can/must support, thus the processor architecture to be selected, and (iv) the pre-installed software that must be considered (e.g., from virtualisation-related technologies like K8s distribution and related add-ons, and the container runtime to drivers required to leverage GPUs/FPGAs etc. from container scope). All the previous information is dismissed in all Views but in the Node view, whose essential goal is to provide these kinds of information. Considering all this, the company is then able to purchase/manufacture hardware based on ASSIST-IoT developments accordingly, or a non-hardware specialised entity could select the most suitable device out of a catalogue. In addition, developers can acknowledge the restrictions in terms of processing architecture, available memory, storage and processing power to support their workloads.

Practical example of Development view: a developer needs to include a custom functionality for the system, for example, to aggregate and filter data from different sources streaming their data before being ingested in a persistent storage (to reduce network and storage overload). This view describes a series of steps that should be followed to define and formalise the enabler, including the rationale process, so the required features, components, endpoints and methods, and user stories are documented, so once the development process starts, the baseline information is consistent. The (iterative) development and operation activities should consider security aspects, and to this end, the project has presented its own DevSecOps methodology to implement NGIoT artifacts.

Practical example of Deployment view: the system administrator of a company willing to deploy an NGIoT system based on ASSIST-IoT RA (i.e., in the IT department of a stakeholder) wishes to know where to install a specific enabler within the network topology of the company. They may also wish to, for instance, (i) analyse how many –and which– enablers are necessary to address a use case, (ii) analyse whether enough nodes are put in place or if new edge equipment must be purchased to support it, (iii) identify where AI inference is taking place in the topology and, in such case, analyse if any virtualisation hardware is present/required, (iv) manually balance the distribution of the enablers among the available nodes, or (v) identify which services should be exposed outside the network for third-party access. Necessary input to take those decisions is provided by the Deployment view. Also, general network and K8s-related guidelines and requirements to set up the infrastructure for network administrators are provided in this view (network and system administrator might be different actors within a deployment).

Practical example of Data view: an engineer is going to develop an AI algorithm encapsulated as an enabler to, e.g., predict the position of cranes (or any element) in the yard (or any other environment). Data are provided by sensors from different vendors and in different formats. For the algorithm to work, they need that data are ingested following a specific data model, regardless of the device that has generated them. To get these data in such format, they must (with or without the aid of developers or a system administrator) ensure that data can be consumed by their service, so any needed enabler for data routing (e.g., for implementing a pub/sub mechanism), storage, semantic translation, etc. first have to be described in a data pipeline representation and then deployed and configured accordingly.

3.1. Functional view

The Functional View, also sometimes referred to as Logical View, has the primary objective of **showing the functionality required to fulfil the user needs and address the stakeholders' concerns**. It describes the main system's functional elements, their responsibilities, interfaces, and primary interactions [10]. In the following subsections, a functional decomposition is presented for each horizontal plane of the conceptual architecture, introducing the enablers that belong to each of them with a brief description of the expected features. Within the scope of the technical work packages, the project is formalising their design, developing (or adapting) and integrating these enablers to facilitate the realisation of NGIoT architectures. A graphical summary of the functional view is presented in Figure 3; in any case, it is worth highlighting that additional enablers can be envisioned in the different Planes.

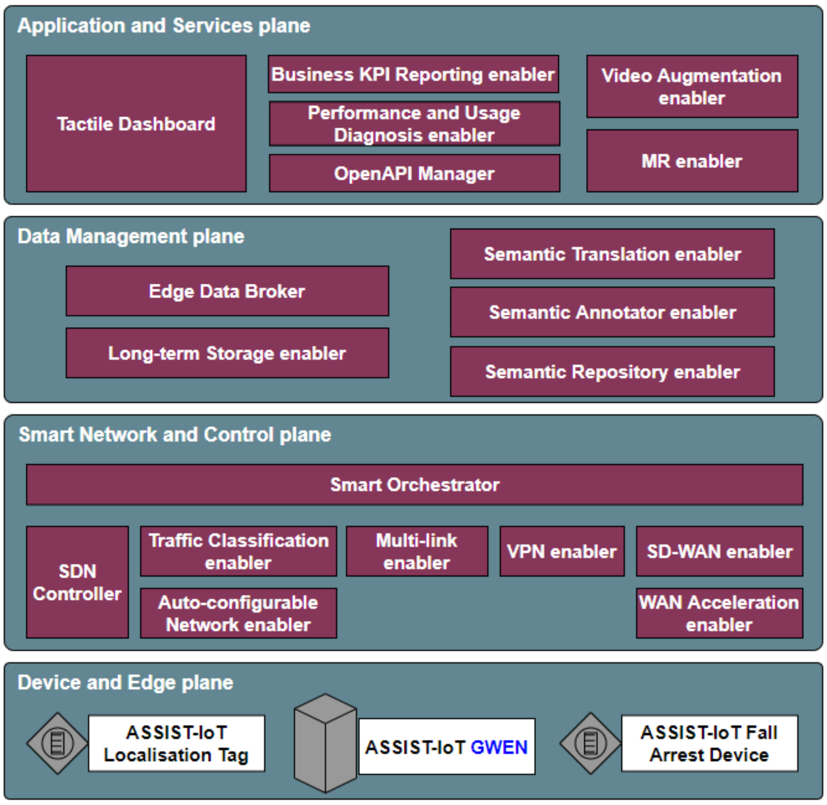

Application and Services plane
Tactile Dashboard
Business KPI Reporting enabler
Performance and Usage Diagnosis enabler
OpenAPI Manager
Video Augmentation enabler
MR enabler
Data Management plane
Edge Data Broker
Long-term Storage enabler
Semantic Translation enabler
Semantic Annotator enabler
Semantic Repository enabler
Smart Network and Control plane
Smart Orchestrator
SDN Controller
Traffic Classification enabler
Multi-link enabler
VPN enabler
SD-WAN enabler
Auto-configurable Network enabler
WAN Acceleration enabler
Device and Edge plane
ASSIST-IoT Localisation Tag
ASSIST-IoT GWEN
ASSIST-IoT Fall Arrest Device

Figure 3. Functional view summary representation

3.1.1. Device and edge plane

The *Device and edge plane* is the plane in charge of providing the functionalities needed for **interacting with the end devices, sensors and actuators and integrating them with the rest of the architecture**. Being at the lower plane, apart from inherent enablers, it includes the infrastructure elements, i.e., the computing nodes (see Node view – Section 3.2) and the network-related elements. So far, the previous iteration of the architecture focused primarily on the Gateway/Edge Node (**GWEN**). There, decisions were made that led to consider it the central element of this plane, as the interfacing point between (IoT, network-related) devices and enablers from upper planes. General-purpose servers and network elements also belong to this plane, even though not being a target of ASSIST-IoT innovation.

As a matter of fact, the project's GWEN has been designed with enough hardware capabilities (in terms of memory, compute power, and storage), including baseline functions depicted in the Node view (i.e., firmware, open OS, K8s/K3s, pre-installed plugins), and also incorporating the most common interfaces required in industrial scenarios. It has been designed focusing on hardware modularity and ease of adaptation: baseline hardware elements are in place, but they can be expanded through expansion boards and SD slots; also, the OS is open and with pre-installed virtualisation technologies and ASSIST-IoT basic firmware, so platform architects have large tailoring options to adapt it to the target business scenarios. In any case, any gateway (or set of devices) with the required interfaces and with enough computing power to run enablers can be part of a realisation of this plane, there is no obligation to use the project's GWEN.

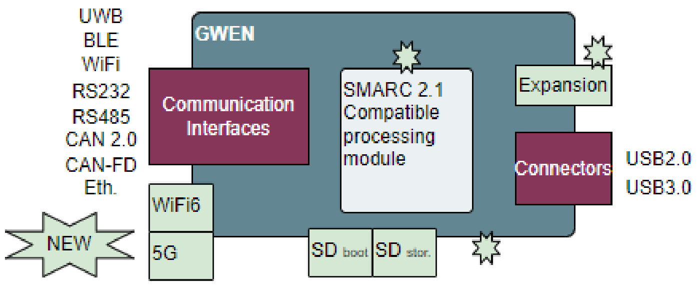


Figure 4. GWEN design to fulfil needs of NGIoT systems

Apart from the computing and networking equipment, the plane also groups a set of features which can be incorporated into the system as hardware/firmware extension and/or enablers. As one can observe in Figure 5, these features can be classified into three functional blocks, namely: “edge intelligence”, “pre-processing functions”, and “communication capabilities” (**Note**: from previous versions, analytics, AI capabilities and enhancement of IoT devices smartness have been grouped into a single block, i.e., edge intelligence, and a new block, i.e., pre-processing functions, has been introduced).

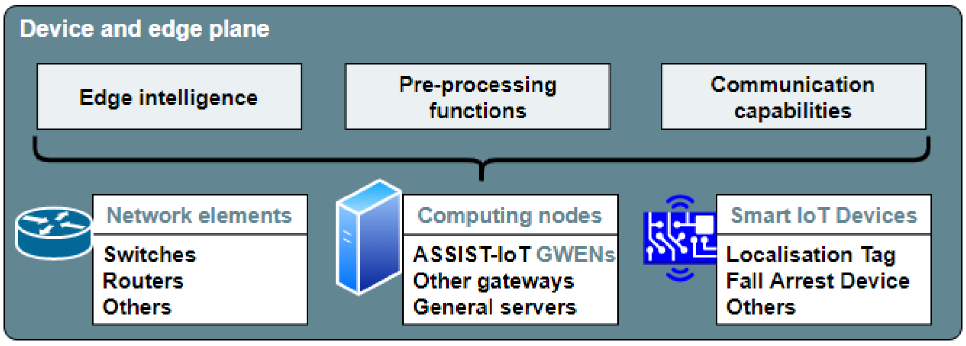


Figure 5. Infrastructure elements and functional blocks of the Device and edge plane

In contrast with other ASSIST-IoT planes, where features are always offered as software packages, the Device and edge plane has a strong focus on hardware, which is more difficult to produce in the case of the IoT domain – or even use-case-specific. Because of that, no ASSIST-IoT enabler has been extensively formalised for this plane. Any pre-processing or edge intelligence enablers are expected to be coupled to the needs of a given use case, and no hardware outside of GWEN was deemed generic enough to warrant inclusion in ASSIST-IoT (although a combination of existing hardware can provide equivalent features) besides domain-specific hardware. Nevertheless, some potential Device and edge plane enabler examples are given in Table 3.

Table 3. Functional blocks and potential enablers of the Device and edge plane

Functional block	Description	Potential enablers
Edge intelligence	Includes all those enablers (and needed hardware) that provide some local analytics or AI capabilities for supporting a use case or enhancing IoT devices smartness	Local analytics service, with custom functions adapted to a use case; local AI framework with needed hardware-implemented algorithms (and required physical components, e.g., GPU, FPGA); AR/VR/MR supporting capabilities; local context broker; internal message broker; local event processor; complex event processors close to the data origin; fall arrest detector mechanism, etc.
Pre-processing functions	Groups those enablers that make some local operations to reduce the overload of the network and ease the integration of data with the rest of the NGIoT system	Transport protocol adapter, local storage (e.g., SQL/NoSQL database), local data filtering/aggregation service
Communication capabilities	Includes all those elements needed to interface a given computing element with the network and/or Smart IoT devices	Extension in the node for communication protocols, e.g., Zigbee, LoRa, etc. (interface – incl. dongles, firmware, local servers, virtual gateways, etc.)

Note: within the scope of the action, some pilot-driven developments that would fall under the scope of these functional blocks are being implemented. The intention is to wrap and deploy them following the enablers’ development guidelines (i.e., following the Cloud-native approach and encapsulation principles).

As a final remark, the Composite services manager from the manageability vertical (see Section 4.5) will help to design and deploy **edge agents** that belong to the Device and edge plane, particularly to the pre-processing functional block. These agents provide two features: transport protocol adaptation and data formatting, aiming at facilitating basic integration and transfer of data with the rest of the architecture.

3.1.2. Smart network and control plane

The *Smart network and control plane* is in charge of **communication aspects as well as orchestration of virtualised functions**. In particular, this plane must ensure that communication between workloads is established as securely, optimised, and transparently for the user as possible. To that end, four functional blocks can be found: “Orchestration”, “Software-defined networks”, “Self-contained networks realisation”, and “Access networks management” (**Note**: In previous versions, a VNF functional block was defined; still, as they can be applied for realising the other blocks and there was not any clear distinction, it has been removed and the enablers, redistributed). This plane encompasses eight enablers, depicted in Figure 6. It is worth mentioning that additional features could have been included as enablers; some examples and a rationale of why they have not been included in the functional view are briefly given for each functional block.

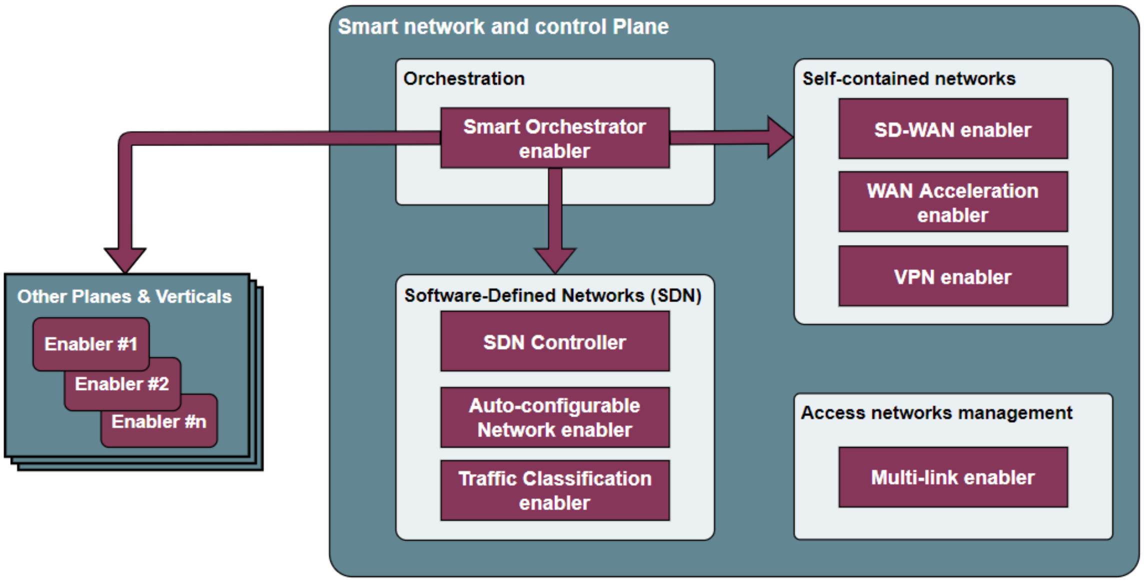

Smart network and control Plane
Orchestration
Smart Orchestrator enabler
Self-contained networks
SD-WAN enabler
WAN Acceleration enabler
VPN enabler
Other Planes & Verticals
Enabler #1
Enabler #2
Enabler #n
Software-Defined Networks (SDN)
SDN Controller
Auto-configurable Network enabler
Traffic Classification enabler
Access networks management
Multi-link enabler

Figure 6. Enablers and functional blocks of the Smart network and control plane

Orchestration

The **Smart Orchestrator** is the main enabler of the plane. Designed following ETSI MANO specifications[8] [11], it is typically in charge of managing the lifecycle of network functions. Still, as many of the NGIoT features are non-network related, this feature of the enabler is also extended to manage the lifecycle of non-network functions. In the scope of the project, it manages the enablers themselves, which as mentioned before, consist of a group of micro-applications delivering a specific feature for the architecture. Besides, being a Cloud-native architecture, it should work using K8s as underlying VIM (Virtualised Infrastructure Management) technology, although enabling it for allowing OpenStack as coexisting VIM would be straightforward, as ETSI MANO contemplates it[9].

Apart from controlling the enablers’ lifecycle, from instantiation to configuration and termination, the smart orchestrator is also in charge of other aspects, such as (i) scheduling of the enabler workloads’ placement within the managed computing continuum, based on selected policies among the available ones (the user could also specify it manually), and (ii) automatic application of network-related rules, to only allow communication between exposed interfaces and prevent any kind of communication between internal components of different enablers. For fulfilling the latter, the Smart orchestrator controls the underlying K8s’ Container Network Interface (CNI) plugin (layers 3-4), and also communicates with the underlying K8s’ service mesh technology (layer 7) for adding security (e.g., encryption and forensics) and observability during the management of the enablers’ lifecycles.

[8] https://www.etsi.org/technologies/nfv

[9] While K8s is selected as VIM as cloud-native workloads (containers) are optimally managed by this orchestration technology, OpenStack is the preferred option when workloads are deployed within Virtual Machines (VMs), which have prevalence in the telco ecosystem.

SDN

The second functional block is devoted to Software-Defined Networking (SDN). This paradigm decouples the network control from the forwarding functions, enabling the control part to be programmable and the network itself to be abstracted from the actual physical resources. In turn, this allows a central management, managed by a controller, without needing to configure the underlying switches one by one, but leveraging policies so the controller can configure them automatically. This block involves the more *traditional* approaches to SDN, as one could (correctly) argue that the **automatic K8s' networking rules** applied by the Smart orchestrator and SD-WAN technology, belonging to the block related to self-contained networks realisation, are also forms of SDN. The main enabler of this block is the **SDN controller**. It can control all the physical and virtual switches of the managed infrastructure compatible with OpenFlow[10] protocol (alternatives to it like REST APIs or other management protocols, like SSH, NETCONF or SNMP, could be considered but are not that efficient). It typically exposes two APIs: the Northbound (NB), so applications and users can interact with the controller's subsystems, and the Southbound (SB), to interact with the SDN-enabled devices.

The other two defined enablers are the **Auto-configurable network** enabler and the **Traffic classification enabler**. The first communicates with the SDN controller via the NB API to set policies on it; these policies (which can be set manually or automatically) aim at improving the performance of specific KPIs of the network (such as load distribution, traffic prioritisation, data transfer losses, latencies, etc.), and to be able to do this, it should collect network statistics (from the controller and from elements of the network itself). The second enabler receives a set of features belonging to packets from the controller and classifies them into a set of types, so the controller can apply the right rules to them. For instance, knowing that a packet belongs to a (e.g.) video application, the path it will follow over the network can be optimised so the bandwidth is maximised.

In this RA, the SDN functional block has not been considered essential as many IoT (and NGIoT) deployments do not leverage SDN-enabled equipment, which is typically considered for the telecommunication industry. It should also be stressed that, although the auto-configurable network enabler and the traffic classification enabler were the selected ones to support the operation of the SDN controller, other enablers that consume the NB API of the controller could be envisioned to provide additional supporting features over the network.

Self-contained networks

This functional block is related to self-contained networks realisation. It responds to the potential necessity of provisioning a private communication channel over a public network, ensuring anonymisation and security of communication. Two technologies have been identified for addressing block: VPN and SD-WAN.

First, the **VPN enabler** is in charge of establishing a secured network to transmit data in an encrypted form between two network elements, usually to facilitate the connection of a device to a network different to the one it belongs to (remote access), although it can also be used to connect two networks (site-to site VPN). They can be implemented by means of technologies such as IPSec or SSL. Besides, **SD-WAN enabler**, working jointly with **WAN acceleration enablers**, is the proposed solution to connect networks over wide geographic areas[11] (considering SD-WAN paradigm). Apart from secure and programmable connectivity, it provides centralised monitoring and control with flexibility: the SD-WAN enabler establishes the connections and manages traffic from a centralised location, while the WAN acceleration enabler, installed once per cluster, actually realises the secured tunnels and also provides additional possibilities such as firewalling or application-level prioritisation.

Any additional enabler is foreseen for this particular functional block, still, the specific features incorporated by the SD-WAN enabler and the WAN acceleration enabler can vary among implementations.

Access networks management

The last functional block comprises all those enablers that manage, or aid at managing, the access networks to connect external devices or nodes. Currently, a single enabler is envisioned, as it has been considered useful for fostering the adoption of wireless networks in NGIoT deployments. In particular, the **Multi-link enabler** is in charge of bonding different wireless networks to work as a single logical one, enabling redundancy and

[10] OpenFlow protocol is a standard for SDN that defines the communication between an SDN controller and the managed devices. In particular, it can update switches' flow tables, but it cannot be used to modify the configuration of a switch.
[11] The specific SD-WAN architecture implemented in the project can be seen in the official documentation (https://assist-iot-enablers-documentation.readthedocs.io/en/latest/horizontal_planes/smart/sd_wan_enabler.html) and will be formalised in D4.3.

reliability mechanisms in real time (e.g., it could consider already-provisioned 5G and WiFi to communicate hosts so, if the main connection fails, the second comes up without significant disruption for end users).

ASSIST-IoT is considered network agnostic, so it does not go far with respect to enablers that could belong in this block. In any case, there are clear examples that could belong to this block, such as the 5G core of a 5G network (in fact, some open source solutions are already Cloud-Native, so they could be incorporated almost seamlessly).

3.1.3. Data management plane

The Data management plane manages all those **services related to a virtual shared data ecosystem, in which data are acquired, routed and processed** to provide key data-related functions. Current enablers have been classified into two functional groups, namely "semantics" and "data governance", as one can see in Figure 7. Functionalities of the Data Management plane are highly independent, and each enabler can be used separately or together in a single system; still, as data is managed, security, privacy and ethics principles and supporting vertical enablers must be considered for supporting the enablers of the plane. Data functions are also very general and are, in principle, not restricted to any single kind of data. Again, it should be reminded that non-contemplated, complementary data-related features that are not directly consumed by an end user could be implemented in a system realisation as an additional enabler.

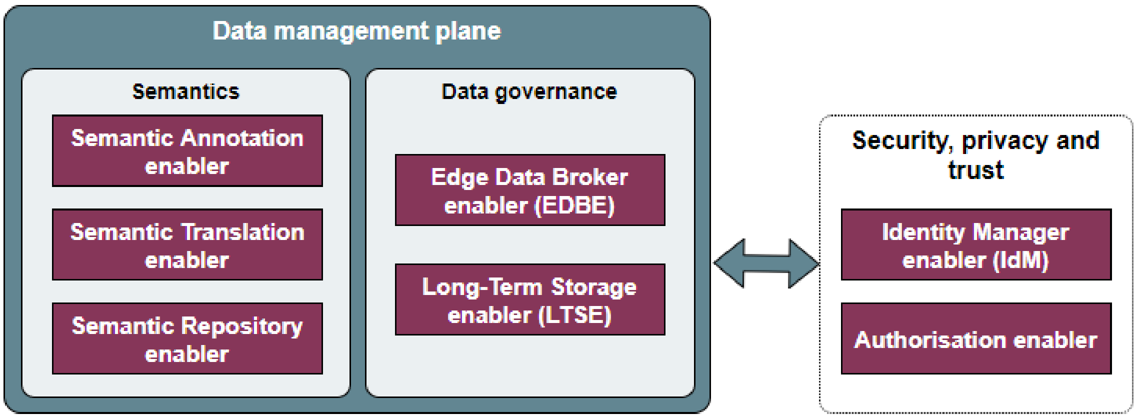

Data management plane
Semantics
Semantic Annotation enabler
Semantic Translation enabler
Semantic Repository enabler
Data governance
Edge Data Broker enabler (EDBE)
Long-Term Storage enabler (LTSE)
Security, privacy and trust
Identity Manager enabler (IdM)
Authorisation enabler

Figure 7. Enablers and functional blocks of the Data management plane

Data governance

An important group of features related to data transportation are delivered by the **Edge Data Broker enabler (EDBE)**, which facilitates the creation of pipelines, in which data flow can be controlled (also see Section 3.5). Scriptable broker nodes enable control over the conditional paths that data will take and the side-effects that it may cause. Depending on particular needs, data may be analysed and conditionally cause alert events, directed to specific consumers depending on pre-set conditions, or be blocked from further transmission (e.g., for security reasons). Hence, EDBE is an important tool for the integration of enablers (and any other software compatible with streaming brokers) through data exchange. It provides a relatively easy way to design decentralised systems with complex data paths and is the backbone of the RA streaming approach. Features of the streaming broker itself, such as replicability and high-availability, allow usage of one or more instances of EDBE to construct resilient and reliable data delivery. This lets system designers ensure that data will arrive where it needs to be, at the time it needs to get there, whether for processing and further transport or storage (archiving).

Besides, the need for dependable data storage in a dynamic, distributed and highly virtualised environment is answered by the **Long-Term Storage enabler (LTSE)**. It supports SQL and NoSQL data models for purposes of secure and highly available archiving and temporary data persistence. A single LTSE can be used independently by many clients (users, services, enablers, etc.). The data functions delivered by this solution aim at alleviating the risk of loss of data in the face of hardware failure, or any other unforeseen circumstances that may lead to otherwise unrecoverable data.

Semantics

Independently from data storage and transportation, the RA proposes specific enablers to process, share and present the data in NGIoT deployments, focusing on its semantics. Semantic descriptions are supported, although not required, for all data managed in an architecture realisation. Two solutions for data interoperability are defined: semantic lifting and semantic translation. The former, performed by the **Semantic annotation enabler**, is the process of changing the data format to fit a specific semantic form, regardless of whether the original format contained any semantic (meta)data. Semantic translation involves mediating the "meaning" of data, so that it is understood for its consumer. It requires input that is already in a semantic format. The **Semantic translation enabler** delivers this functionality and can be configured on-the-fly to support mediation between data packets for any set of data models or schemas. Semantic data interoperability functions can be used in streaming or for bulk data.

To support the usage of semantic interoperability, the plane includes a **Semantic repository enabler**. This hub of data models, schemas and ontologies enables sharing of common data models with relevant documentation (e.g., usage examples), so that the meaning of data is understood not just by semantically enabled software but also by users. This supporting function is important to promote the understanding of semantic data, and to help explain how the semantic data interoperability mechanisms are used in practice.

Together, the three enablers form a **semantic suite**. When used together, the suite offers semantic lifting and translation, supported by a repository to ease the work of semantic engineers and data integrators. Files stored in the Semantic repository can be used not only to inform users about the semantics of the data but also to configure semantic annotation and translation. Note that any service offered by the semantic suite can be omitted, if it is not necessary in a particular use case. For example, the Semantic Translator does not need to be deployed if a system has uniform semantics. Naturally, any ASSIST-IoT deployment may evolve over time, and the enabler may be added and connected to an existing data stream at a later date. An example overview of a semantic suite pipeline is presented in Figure 8.

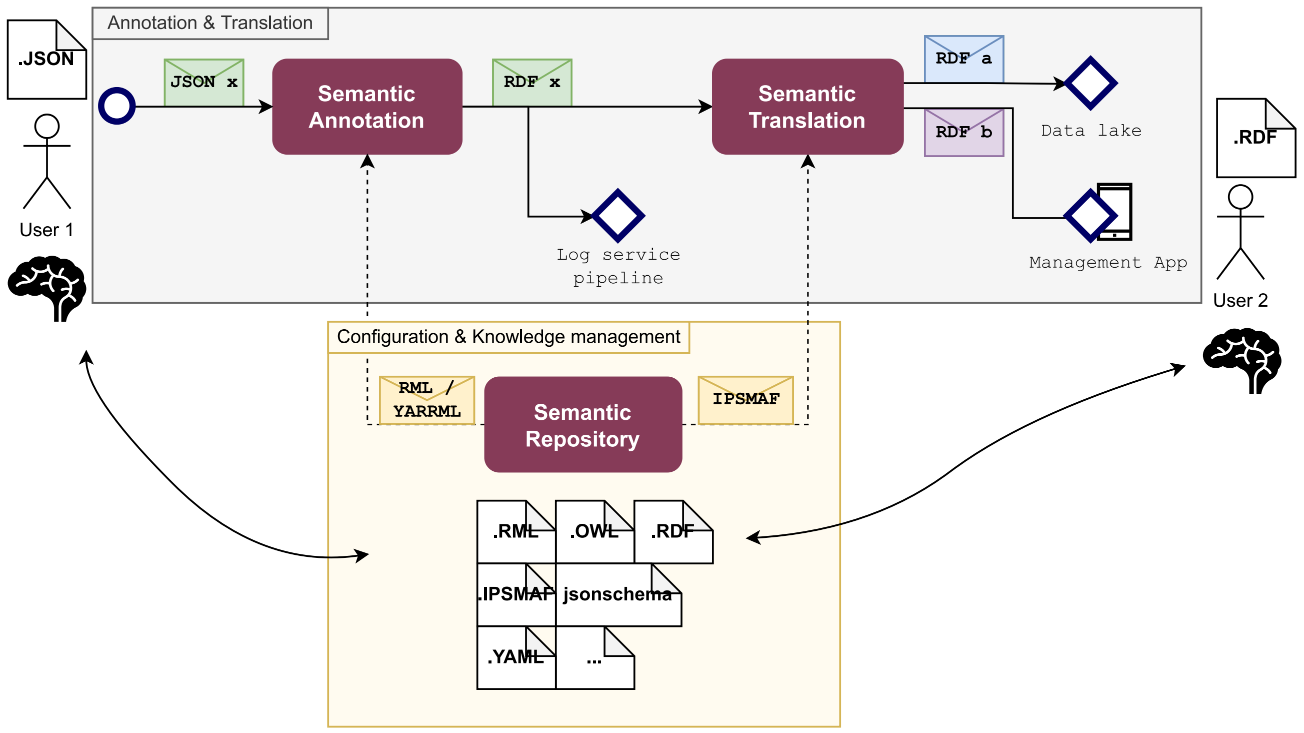


Figure 8. Semantic suite pipeline example

As a supplement to the data processing functions, ASSIST-IoT also proposes a set of data processing rules and guidelines aimed at data administrators. This artifact is intended to be a reference on how to design specific data-related processes, whenever sensitive (e.g., private or confidential) data is involved. Because the data processing rules are not architectural or technical in nature, they have been presented in deliverable D2.4 [12].

3.1.4. Applications and services plane

The *Application and services* plane is intended to **provide access to data, focusing on human centricity and user experience**. Three main working lines (functional blocks) are envisioned in this plane: the development of client-side applications (also called frontends or dashboards), the introduction of novel AR/MR/Computer Vision interfaces (taking into account potential hardware encapsulation requirements), and the provision of graphical toolsets via Open APIs for enabling the open experimentation over ASSIST-IoT deployments. To accomplish the aforementioned objectives, the project partners have identified six enablers, as shown in the figure below, and described next.

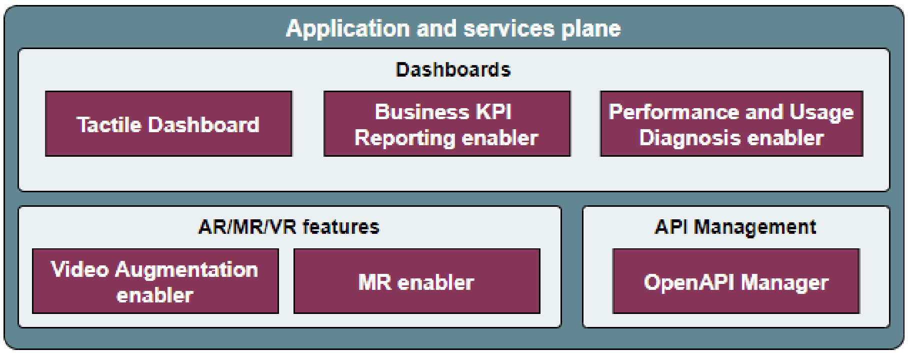

Application and services plane
Dashboards
Tactile Dashboard
Business KPI Reporting enabler
Performance and Usage Diagnosis enabler
AR/MR/VR features
Video Augmentation enabler
MR enabler
API Management
OpenAPI Manager

Figure 9. Enablers and functional blocks of the Application and Services plane

Dashboards

The **Tactile dashboard** has the capability of representing data through meaningful combined visualisations in real time. It also provides (aggregates and homogenises) all the User Interfaces (UIs) for the configuration of the different ASSIST-IoT enablers and associated components. Thus, it allows the creation of fully reusable web components that can be used to create Single-Page Applications (SPAs) or complex web applications. It makes use of Prodevelop's open-source PUI9 framework, which in turn, is based on VueJS. With it, the new applications can have a basic layout with a login screen and a fully configurable menu. The main advantages of the tactile dashboard framework are: (i) modern and adaptive design, (ii) very good performance, (iii) based on web components, (iv) responsive components, which have their own HTML template, internal JavaScript code, styles, and translations, and (v) thanks to using VueJS, it is relatively easy and quick to start being productive. Because of being such a general and open enabler, each deployment will implement its own tactile dashboard according to its requirements.

The **Business KPI reporting enabler (BKPI)** is in charge of representing all valuable logs, time-series analytics, and Key Performance Indicators (KPIs) desired by the end-user in graphs, charts, pies, etc. Hence, this enabler provides a graphical and intuitive interface to make custom graphics, allowing to choose among several types of graphs, the collected data from one or more storages and databases of ASSIST-IoT deployments in order to extract business insights, and allow to focus on only the KPI that are relevant to the organisation. The BKPI will include the possibility of having its UIs embedded within the Tactile dashboard. The last enabler of this category, the **Performance and Usage Diagnosis enabler (PUD)**, provides an end-to-end approach to infrastructure and application monitoring. Based on Prometheus, the enabler is in charge of monitoring (i) the whole K8s clusters managed by the system as well as (ii) the deployed enablers of the architecture. To do it, the PUD enabler collects performance metrics by scraping metrics from endpoints with an HTTP pull model, allowing the detection of potential problems and act in accordance (if enabled) or to notify the platform administrator for fine-tuning the associated machine resources. As the BKPI enabler, the PUD also includes the possibility of being embedded as an additional UI in the form of iFrame within the Tactile dashboard.

API management

The **OpenAPI management enabler** is an API manager that allows enablers to publish their APIs, monitor the interfaces lifecycles, and ensure that the needs of external third parties, as well as applications that are using the exposed APIs, are being met. Hence, this enabler provides the tools to ensure successful API usage in the developers' environment, help end-users with business insight analytics, as well as help platform admins to preserve its security and protection. To achieve this, all the enablers have to document their API in a common

API Swagger-JSON format. It will be the main mechanism for third-party access, precluding the direct exposure of internal enablers to the outside, thus reducing the attack surfaces.

AR/MR/VR features

Two enablers belonging to this functional block have been envisioned. First, the **Video augmentation enabler** performs computer vision functionalities over captured images or video streams either from Edge nodes or ASSIST-IoT databases. On the one hand, a Machine Learning (ML) trainer carries out the process of feeding a neural network with a pre-annotated dataset to detect/recognise objects, and, on the other hand, an Inference Engine acts as a visual interpreter, executing the trained ML model over a specific input. The training process might be computationally intensive, so performing it over a GPU is recommended (in such case, some configuration over the host has to be applied so containers can leverage it). And second, the **MR enabler** offers a human-centric interaction through better cooperation of the end-users with the IoT environment. In particular, an end user can receive tactile, real-time and visual feedback as well as data (long-, short-term, or real-time) from the environment (including rendered 3D models) and, if needed, they can report alerts or send data to the system. Decision-making is improved as human flexibility, creativity and expertise are boosted thanks to enabling interaction with IoT platforms and devices in a more immersive way. It should be stressed that these two particular enablers focus on specific augmentation capabilities, additional ones could be envisioned for supporting additional features from these kinds of tactile and/or haptic interfaces.

3.2. Node view

The View of the Node in ASSIST-IoT responds to **a structural aspect** of the architecture. Per definition, *ASSIST-IoT enablers (composed of components) are instanced to run on a single or on multiple nodes of the deployment's infrastructure.* Therefore, a node in the architecture is the element that undertakes the execution of enablers.

As per design principles, ASSIST-IoT has adopted K8s as the container orchestration framework that coordinates such execution. Thus, directly derived from this decision, an ASSIST-IoT node is the (virtualised) hardware that takes over the computation of an enabler's logic. Considering that one single hardware could comprise several clusters (e.g., a powerful node in a high tier of the topology, see Deployment View – Section 3.4.23.4), it is more precise to refer to "nodes" in ASSIST-IoT as these logical units rather than to the complete hardware as a single piece. Nodes might vary on geographical location (close to the source, same building, same city, remote…), on the topological spot (FAR edge, different edge tiers, etc.) and on processing capabilities (datasheets, restrictions, technical specifications). In terms of integration within a given architecture realisation, every ASSIST-IoT node must be prepared to execute K8s applications as requested, as well as to be addable to a cluster. This explanation is graphically illustrated in Figure 10.

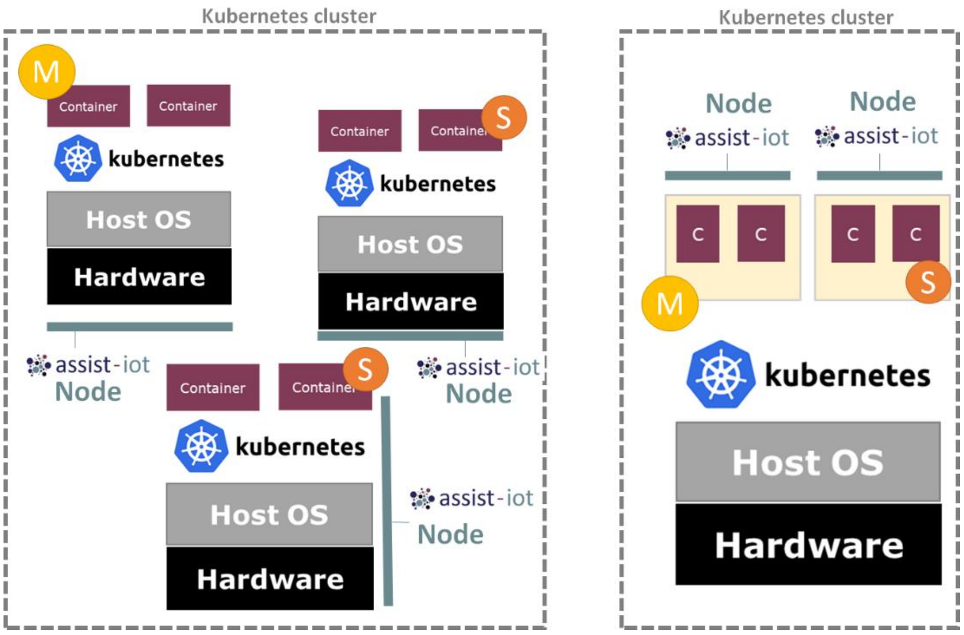


Figure 10. ASSIST-IoT Node concept and spots

The most relevant properties of a node are its capacity to connect with other elements of a deployment (network connections, IP address, names, etc.), its accelerators (number and type of GPUs, TPUs, FPGAs, etc.), computing capabilities (CPUs, cores) and its storage capacity.

According to the previous, the following pre-requisites are expected by any **hardware** (*host*) to potentially become an ASSIST-IoT node:

- **Linux Operative System** (recommended Ubuntu 16 and beyond). Other distributions are also accepted as long as they are able to run the next requisites. Here, a relevant comment applies: in the IoT field, not all the computation-capable hardware rely on Linux OS[12]. Hence, if wishing to move ASSIST-IoT nodes as closest as possible to the data source, the reader must be aware that such boundaries will be marked by the "smallest Linux OS reachable hardware" in terms of the computing continuum. Other processing elements before such points will need a direct connection with an ASSIST-IoT node compatible hardware.
- **Virtualisation technology**. In principle, any container runtime compatible with K8s can be considered (e.g., Docker, containerd, CRI-O, Mirantis). In any case, Docker is recommended due to its maturity and community support.
- **Container orchestration framework**. A compatible distribution of Kubernetes. Open source distributions such as **kubeadm**, **K3s**[13], **microk8s**[14] and **k0s**[15] have been tested (in different hardware, e.g., RaspberryPis, low-resources servers…) and have been determined as **valid** to be used as baseline K8s technologies for ASSIST-IoT nodes. This decision depends largely on the computation capabilities of the device (e.g., K3s and k0s are more suitable for constraint devices while kubeadm is recommended for nodes with higher computing capabilities). Commercial distributions can also be considered.

However, it is not enough only with the existence of pure virtualised hardware and the appropriate K8s distribution. There is a need to include pre-installed software in a host candidate to exert ASSIST-IoT nodes. The RA suggests considering add-ons or plugins for bringing the following features:

- **Packaging and deployment manager**. Needed to package enablers in a self-contained format with all the required K8s manifests needed to work and also to deploy them. **Note**: Helm[16] has been the selected technology in the technical developments, although other alternatives exist (see Section 3.4.3). The Smart orchestrator should be developed in accordance.
- **Container Network Interface (CNI) plugin**. These plugins (which can be combined) expand the basic networking features provided by K8s. These can be used for e.g. easing the connection of services, enabling multi-cluster communication and introducing network policies over the K8s network. **Note**: Cilium[17] has been used in the technical packages, allowing smooth communication among nodes that live in different clusters based on the assignment of identification names at kernel level to each of those nodes and a DNS-like system. Apart from networking capabilities, it also includes layer 7 features like encryption and observability. Other plugins[18] could be considered or combined.
- **Container storage technology**. Needed to extend the basic storage possibilities provided by K8s, e.g., define virtual volumes, manage replicas in one cluster or multi-cluster environments, etc. Note: The project has considered OpenEBS[19] for the technical developments, but alternatives such as Rook[20] or Portworx[21] can be considered.

If complying with all the previous, any K8s node to be devised over such hardware is considered compliant with ASSIST-IoT architecture.

[12] https://www.intuz.com/top-iot-operating-systems-for-iot-devices
[13] https://k3s.io/
[14] https://microk8s.io/
[15] https://k0sproject.io/
[16] https://helm.sh/
[17] https://cilium.io/
[18] https://kubernetes.io/docs/concepts/cluster-administration/networking/
[19] https://openebs.io/
[20] https://rook.io/
[21] https://portworx.com/

3.2.1. Processing architectures considerations

This subsection presents two particular issues related to processing that need to be carefully considered for deploying workloads in an NGIoT system: (i) hosts with different CPU processing architectures and (ii) use of hardware acceleration for workloads requiring intensive computation.

The majority of operating servers on the market, additionally to operating on Linux OS, contain CPUs with x86 or x64 set of instructions (like Intel and AMD architectures). However, especially in constrained devices (e.g., RaspberryPis and pico-clusters), other sets like ARM, that root on better power-efficiency, are used. In summary, these instructions translate high-level code into actual machine-understandable CPU instructions. Accordingly, services are normally developed and built to be run under a specific processor architecture. Then, if a service is built for an x64 architecture, it will not function properly in an ARM processor.

Moving to actual enablers realisation, the pragmatic truth is that every software becoming an enabler is designed and built making use of baseline Docker images. Sometimes, these must be very raw (e.g., python, alpine) or even custom, but on most occasions the enablers root on inherited images that, at the same time were specifically devoted and built for a specific processor architecture. It may happen that, in an IoT deployment, that enabler should live on a node equipped with a different processor (e.g., x64 vs ARM). In order to cope with this variability, the architecture considers two lines of action:

1. In cases where enablers will definitely only work either on powerful servers or in constrained, ARM-based devices (or other architecture for this kind of devices), the containers could be built bearing in mind only the target instruction set.
2. In cases where enablers could be hosted in different spots within the computing continuum, their **images should be built considering, at least, the most common CPU instruction sets** (e.g., amd64, arm64v8, arm32v7).

The second approach is the most followed one in typical open source solutions available in the market, and is the course of action suggested by the RA for realising the enablers' images. In particular cases such as the Smart orchestrator or the LTSE that will always run on computers with certain processing capabilities, the first line of action is accepted, but it is discouraged for the rest.

When utilising Docker as the underlying container runtime installed in the devices, container images can be built considering multi-architectures, so the needed ones are compiled in the same image. To that end, Docker provides a tool named *buildx*[22] to facilitate this effort. A drawback here is the increased compilation times, but the trade-off clearly favours the architecture; incorporating multi-architecture compatibility automates the CI/CD pipeline's process. Finally, when these images are run on a given device, the container runtime will select the right architecture.

Moving to the issue related to acceleration capabilities, containers cannot access this kind of hardware without doing some installations and configurations over the underlying host space. In the particular case of **GPUs, providing access for the containers to them** requires that the drivers are provisioned and working on the host (meaning that it could run GPU operations within the OS user space) and also a set of libraries so the container runtime is aware of this resource and can make it accessible to the deployed workloads. For instance, specific steps need to be followed at host OS level so K8s can leverage NVIDIA GPUs[23]. Similar provisioning actions are also required when considering hardware acceleration via **FPGAs**[24]. These actions are usually specific to the brand of the acceleration model used, and hence there is not a unified way to prepare the hosts and instructions have to be found in manufacturers' documentation.

3.3. Development view

Grouping functionalities under enablers has clear advantages for development and maintenance. It allows having a conceptually more compacted view of the features and services available in the system, similarly to what happens in a SOA architecture, while having the benefits of working with virtualisation, like in the case of microservices. Also, no language or communication interface is mandated for internal enabler designs,

[22] https://www.docker.com/blog/how-to-rapidly-build-multi-architecture-images-with-buildx/
[23] https://docs.nvidia.com/datacenter/cloud-native/kubernetes/install-k8s.html
[24] https://www.accelize.com/how-to-containerize-an-fpga-application/

allowing flexible implementations that still preserve interoperability when it comes to services exposed outside of the enabler.

This version of the ASSIST-IoT architecture introduces the Development view, which aims at providing some **high-level recommendations and guidelines for facilitating the process of design and development of enablers**. The architecture has defined a set of enablers that belong to different Planes and Verticals and that have been considered highly useful (and, in some cases, essential) for NGIoT systems. Also, the architecture is accompanied by implementations of the enablers, developed under the scope of the technical work packages.

Still, it is possible (and likely) that some future ASSIST-IoT systems will need their own particular enablers, not envisioned in the functional view nor in the Verticals of the architecture. This can happen if an additional feature is needed (e.g., some Big Data-related enabler at the Data Management plane) or a novel, breakthrough technology has appeared. It should be mentioned that this view does not substitute the DevSecOps methodology presented in D6.1 [13], but rather complements it and aids developers in “thinking in enablers”.

Besides, enablers come with a set of design principles, constraints and common conventions that need to be followed (see Sections 2.3, 0 and 5.3) and that should be kept in mind during the whole design and development process, mainly:

- **Virtualisation**: Each enabler is delivered as a set of containers that run the components, along with automated deployment scripts. This ensures lightweight, portability, and the fact that enablers can be (re)deployed or removed as needed and exhibit a level of independence from one another. In practice, enablers may rely on each other to deliver functionalities, such as system-wide security. However, each enabler should be independently deployable and deliver its core functionality on its own.
- **Encapsulation principle**: enablers can communicate with each other and with other clients only via explicitly exposed interfaces (e.g., REST API, gRPC) that are jointly called “**enabler interface**”. Enabler components cannot be interacted with directly from outside of the enabler’s internal scope. Informally, the encapsulation presents an enabler as a black box whose state cannot be directly influenced and that can only be communicated with through specific channels. In essence, components of an enabler must be placed in a secure environment with a network perimeter (like in a DMZ network). Being run in dedicated containers, components are separated from the underlying host system. This design paradigm allows flexibility in internal component communication and implementation without compromising the standardisation, documentation, and security of consumer-facing communication channels (e.g. REST APIs, streams, etc.).
- **Manageability**: Enablers should expose a set of basic endpoints and logs. Even though the architecture itself is open in this regard, ASSIST-IoT proposes here to follow the *de facto* conventions. While cluster and container metrics (i.e., hardware and network resource usage) can usually be gathered by the container orchestration framework administrator, applications must expose their metrics individually. According to the first proposed design, enabler metrics are exposed via a dedicated endpoint, following Prometheus-compatible format[25], whereas logs are sent out via *stderr* and *stdout* interfaces. Besides, the general state of an application in a container will be accessible also following current standards, exposing endpoints that report health. Instantiations of the RA that do not follow ASSIST-IoT design, may opt for different methods of exposing metrics and logs.

As explained before, it is important to note that enablers are not microservices, but rather “**micro-applications**”. This name refers to similarities with microservices and not to the perceived size or functionality limitation of an enabler. Just like microservices, enablers can be orchestrated, monitored, and have their internals hidden from an external user. Enablers are, however, not limited to a single function, they place no requirements on state management and can implement their components following any design pattern. For example, all its components may, but are not required to, be microservices. In such an enabler, all microservices that should be available externally just become part of the enabler interface, while others are not exposed and are used for internal communication only.

[25] https://prometheus.io/docs/concepts/data_model/

After familiarising themselves with the foundations of encapsulated enablers (outlined above) and concepts summarised in the previous paragraphs (and within other sections of the architecture), developers are advised to follow the Development view, as presented here. The main steps of the view are summarised in Figure 11.

1. As with any hardware and software development, the design of an enabler starts with its **definition and elicitation of requirements**. As with the RA itself, it should gather the actual needs, expected features and technological constraints of its users and, likely, their use within the context of a particular business scenario (they could also be designed to be generic, applicable to different Verticals). Specifically, it is needed to:
    - Depict the main objective of the envisioned enabler with a set of key, foreseen characteristics.
    - Describe user and business-related, functional and non-functional requirements.
    - Analyse the general constraints of the (hardware and software) environment in which it is expected to run.
    - Outline use cases and scenarios in which it will apply. Knowing what stakeholders and end users expect from it can help consolidate its definition.

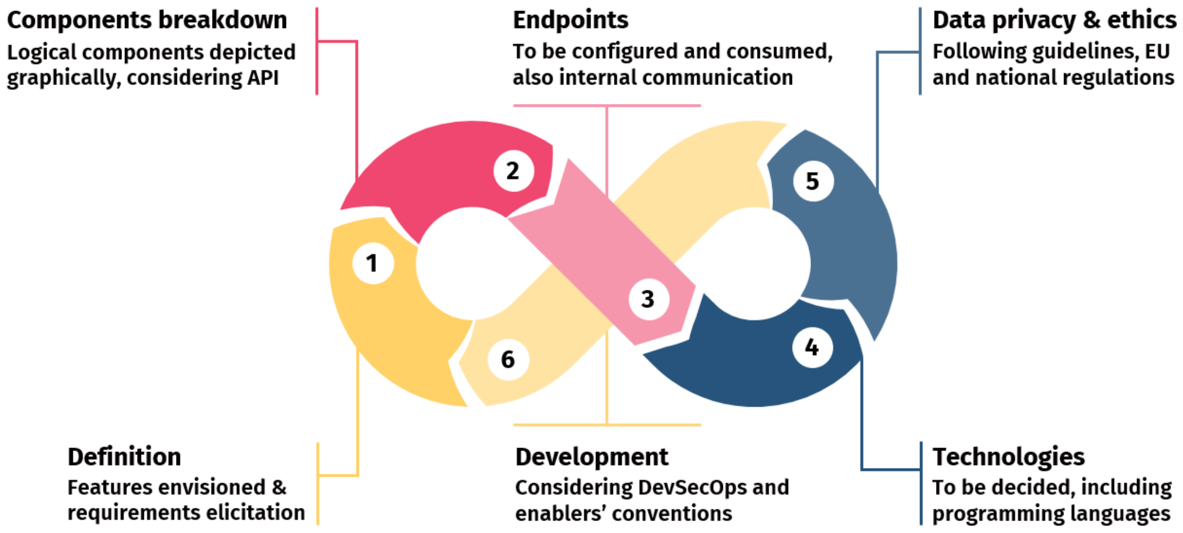

Components breakdown
Logical components depicted graphically, considering API
Endpoints
To be configured and consumed, also internal communication
Data privacy & ethics
Following guidelines, EU and national regulations
1
2
3
4
5
6
Definition
Features envisioned & requirements elicitation
Development
Considering DevSecOps and enablers' conventions
Technologies
To be decided, including programming languages

Figure 11. Continuous enablers' development process

2. Once features are ready, a **breakdown of the internal, logical components** should be depicted. Ideally, each internal component would be in charge of providing an internal feature of the enabler (even though during implementation, some features are grouped and executed by a single container).
    - It is recommended to be depicted graphically, considering an **API** as the main interface to be configured and consumed (additional interfaces, e.g., pub/sub, can be defined if needed).
    - Also, describing the lifecycle of an enabler and addressing its use cases in terms of (high-level) interaction of the internal components eases further design steps.
3. A first mock-up of the communication **endpoints** and methods that will be exposed with the API should be outlined.
    - Apart from inherent configuration and functional endpoints, ASSIST-IoT strongly recommends the provision of a set of common endpoints that should be present at all enablers (see Section 0) and using standards, such as REST.
    - Also, the internal communication among components has to be addressed (e.g., via gRPC, REST API, etc.).
4. Baseline **programming languages** and existing **technological solutions** should be chosen. In some cases, some internal features might be already provided by existing software and, if they fit the edge-oriented ecosystem, can be adapted to the project's needs. If custom components are needed, they should be developed considering resource optimisation (hardware resources are scarce at this kind of deployments) and decentralisation in mind.

5. If **data is involved**, considerations declared in D2.4 [12] should be followed to avoid issues with privacy regulations and ethical aspects. Besides, particular clauses for specific sectors and/or specific national-level regulations might apply, which must be studied and considered.
6. Once developments start, **DevSecOps methodology** [13] should be followed (i.e., DevOps embedded with security controls providing continuous security assurance [14]). As some high-level recommendations related to virtualisation and shifting towards Cloud-Native approach:
    - Containers should be based on well-known, tested images (either baseline ones over which insert custom code, e.g., alpine, python, node, etc., or application-ready ones, e.g., MongoDB, Kibana, rabbitmq, etc.; provided license compatibility). Verified Docker images from *dockerhub*[26] provided by consolidated vendors are recommended.
    - For very early building and testing processes, the initial integration of internal components can be based on Docker Compose files before moving towards K8s.
    - Enablers' packaging should follow a set of conventions to be deployed in K8s clusters managed by ASSIST-IoT. To that end, specific packaging guidelines and a Helm chart package generator are provided (see D6.4 [15]).
    - A CI/CD pipeline for automating part of the operations should be set up. Unit testing, code and security testing tools, packaging tools and repositories should be provisioned and incorporated into the DevSecOps framework (e.g., GitLab, GitHub), adapted to the working environment and the expertise of the development, security and operations teams.

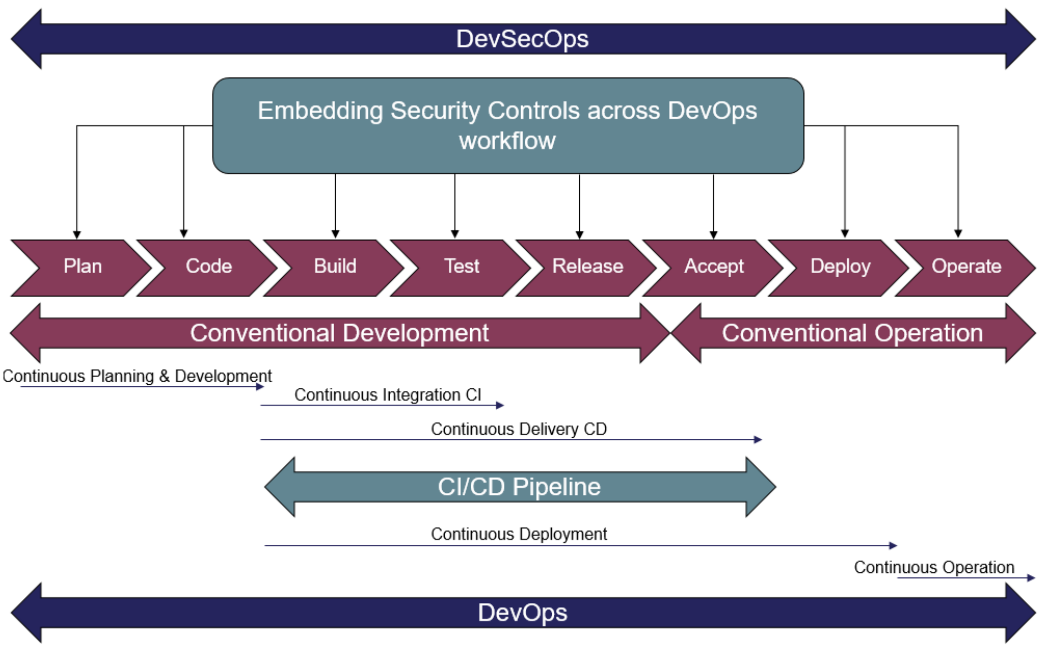


Figure 12. DevSecOps embedded security control [14]

If DevSecOps principles are followed (see D6.1 [13]), the final result should be secure by design, prepared for edge-oriented environments, and ready to be deployed with the Smart orchestrator. Also, in the same way as DevSecOps is an iterative process, steps 1 to 5 are not static and can (and should) be iteratively evaluated during the process with the insights gained during the design, implementation and testing phases.

3.4. Deployment view

The Deployment view provides guidelines and recommendations to facilitate an NGIoT system realisation in a given environment, in order to address the specific use cases of a given business scenario. This involves different aspects, from (i) the network provisioning and (ii) the K8s setup to (iii) the deployment of enablers and (iv) their combination. A dedicated subsection covers each of these aspects. This view can be of utility for stakeholders (or system owners), ASSIST-IoT system administrators and developers to gain insights about:

[26] https://hub.docker.com/

- The hardware elements (computation nodes, networking elements) present within a deployment, or needed for a system realisation. It can also be used to evaluate if the hardware is underused or if there is a need for hardware resources to support the current (or foreseen) use cases.
- The configuration of the network and the K8s infrastructure to support the workloads to be deployed; how many clusters are deployed, with how many workers, how are they interconnected, etc.
- The enablers deployed within the continuum (in a specific location – far edge, edge, cloud; in all the clusters or nodes) to address a use case, or a business scenario.

3.4.1. Network considerations

As one can observe in Figure 13, an ASSIST-IoT system can **consist of** different **computation nodes**, distributed within **various computation tiers** across the edge-cloud continuum. Tiers may extend from end devices (with low computation capabilities, e.g., IoT sensors/actuators, MR/AR glasses, IP cameras) to edge (with one or multiple levels, being FAR edge the closer to the end devices) and, if needed, cloud – all this without precluding possible scenarios solved considering just a single tier.

As introduced in the section devoted to design principles (Section 2.3), Kubernetes is strongly recommended as the container orchestrator platform for the computing nodes to deploy the needed workloads. The main network requirement posed by this technology is that the underlying **network connecting** the physical **nodes** must be **based on IP**, must have a valid IP configuration and should be reachable among them. The fact of being an SDN-enabled network or not should be transparent from the point of view of K8s, as the actions that the SDN controller applies over the network equipment (with OpenFlow capabilities) would affect only at layer 2 of the OSI model.

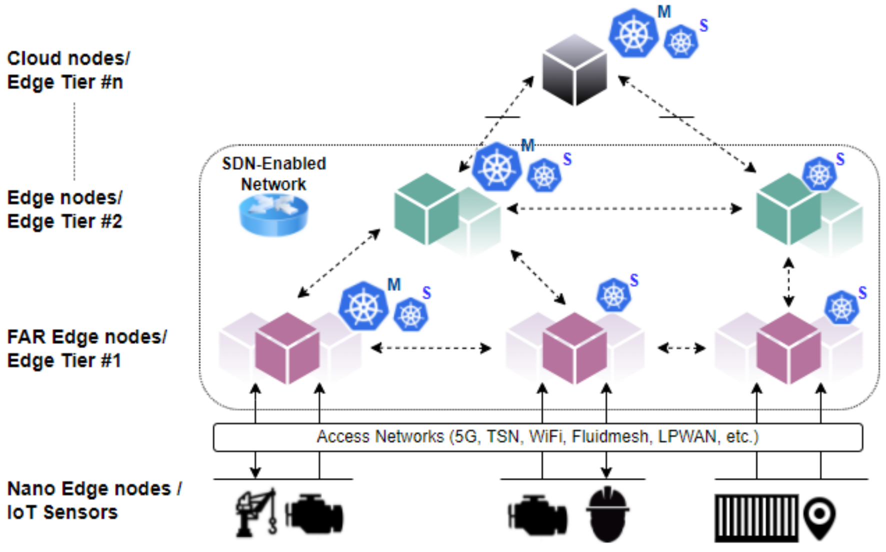

Cloud nodes/ Edge Tier #n
Edge nodes/ Edge Tier #2
FAR Edge nodes/ Edge Tier #1
Nano Edge nodes / IoT Sensors
SDN-Enabled Network
M
S
Access Networks (5G, TSN, WiFi, Fluidmesh, LPWAN, etc.)

Figure 13. Generic topology of an NGIoT system

Having said that, the topology of a given business scenario might involve additional networking implications. Specifically, it could be needed to have connectivity over two (or more) geographically-separated sites or require giving connectivity to devices belonging to an external network. The required connections should always be managed following the principles of security and anonymisation, especially if traffic is going to be forwarded through public networks, or otherwise, data can be compromised. Hence, tunnelling technologies are highly suggested. These situations are contemplated in the reference architecture, having two enablers of the Smart network and control plane (see Section 3.1.2) to tackle them.

Lastly, the connection between the nodes of the (FAR) edge tier and the end devices can be realised with several access technologies, depending on the requirements demanded by the addressed use cases (wired/wireless, low latency or not, deterministic or not, high wireless coverage or not, IP-based or not, etc.). The ASSIST-IoT RA has been envisioned as **access network agnostic**, still, nodes and end devices must have the dedicated physical interfaces and firmware ready, as well as supporting network equipment present in case it is needed (e.g., in case of 5G connectivity, a nearby cell station should be present, and the involved nodes and devices should have dedicated antennas, chipset, SIM cards and firmware ready).

3.4.2. K8s-related considerations

As mandated by the Node view, the computing nodes should have a K8s distribution installed, considering the recommendations given in that view for its selection. Nodes are grouped in clusters, one of them acting as *master* (in charge of managing control plane aspects, such as scheduling resources) and the rest, workers or *slaves* (which actually execute workflows). If high availability strategies are applied, there could be multiple master nodes, so in case one fails the others can take over. Here are introduced some **aspects that should be considered** for a proper implementation:

- A K8s master ("M", in Figure 13) should control slaves ("S") with similar performance (e.g., a high-performance cloud node and a constrained node from a FAR edge tier should not be part of the same cluster).
- A deployment logically divided into different tiers suggest having at least a dedicated master controlling the slaves available in each of them, as this separation is likely related to different hardware resources.
- All the nodes of a cluster must belong to the same K8s distribution (kubeadm, k0s, K3s, etc.); Otherwise, a master must exist for each of them, even if they belong to the same tier.
- If new nodes are to be added to a deployment, it is preferable to include them as workers of an existing cluster, since masters dedicate more resources to control aspects.
- The architecture does not cover K8s-related security aspects, as it is out of the scope of the architecture. In any case, best practices for K8s security are strongly suggested[27].
- K8s distributions and required plugins or add-ons are supposed to be already installed on each node. In case that cluster configuration (mainly, to add nodes to an existing cluster) is not done automatically via dedicated tools such as Ansible, these operations must be done manually by an administrator.

3.4.3. Deployment options

Once the network and the K8s infrastructure is ready, workloads can be deployed. An architecture compliant with the ASSIST-IoT RA provides its plane and vertical features in the form of enablers, which should follow the principles described in Section 2.3 and also consider some additional aspects depicted in Section 5. The **Smart orchestrator** of ASSIST-IoT is the main tool for deploying the rest of enablers (**Note**: currently, this enabler can be installed using a dedicated script). Before deploying enablers with it, **clusters must be registered on it** by submitting their respective Kubernetes' configuration files.

The Smart orchestrator aims at easing the work of ASSIST-IoT administrators. For it to work, enablers should follow a **specific packaging format** (or accept different types), where K8s resources are defined and can be tailored before being deployed in a specific environment. This is strongly recommended as enablers can consist of several components, each of them with multiple K8s resources assigned (declared with K8s manifests). This format could be custom (based on typical K8s manifests or *Kustomize*[28] configurations) or leveraging existing ones, like *Helm charts*[29] or *Juju bundles*[30]. The **orchestrator should be developed** considering the selected format/s to deploy enablers **accordingly**. Alternatively, deployments could be done using K8s' native *kubectl* command, Helm or Juju tools (if their formats are used and they are installed within the clusters), although it would be less user-friendly and some features added by the orchestrator would be lost. Also, in early development stages, Docker and Docker Compose tools could be used to deploy containerised workloads, but this is highly discouraged in the integration and production stages.

As a realisation example, the Smart orchestrator developed within the framework of the technical work packages works with enablers packaged as charts, which contain all the manifests needed to work in a K8s environment – with some particularities posed by this packaging format (see D6.4 [15]). With the orchestrator, the user can select manually the cluster (or node) within the continuum to deploy the enablers, or let it decide based on some

[27] https://kubernetes.io/docs/concepts/security/
[28] https://kustomize.io/
[29] https://helm.sh/docs/topics/charts/
[30] https://juju.is/docs/olm/bundles

policies. As an alternative, charts could be deployed with Helm, as long as it is installed in the targeted clusters (**Note**: some benefits brought by the orchestrator, like the provisioning of K8s' networking rules, will be lost).

3.4.4. Enablers integration

Enablers are independent artifacts that bring a specific feature to the system. They are always deployed in a standalone fashion, without needing other enablers to be running. Still, in functional terms, they might be fully functional by themselves (e.g., Smart orchestrator), or might need the combination of more than one to provide actual value (e.g., FL enablers cannot work if they are not deployed altogether, see Section 4.3).

ASSIST-IoT RA does not force (nor expect) that all enablers are fully integrated; there are many artifacts, data models depending on specific use cases, technologies with their own constraints, etc., and putting too many integration efforts might limit the extent of the provided features. It is also encouraged that enablers can be leveraged in different sectors or environments, even outside NGIoT environments, so too many integration efforts might not be worth in the long term.

There are enablers that will be integrated by default as they are part of a common stack or they will likely be used in combination (e.g., Federated Learning-related enablers, BKPI and LTSE enablers, cybersecurity enablers, etc.), and then the only configuration needs to be done at deployment, usually related to server address (or more likely, service name, which in general will be the default one). Still, for those that require data exchange without being integrated, dedicated mechanisms are needed. In ASSIST-IoT, a general-purpose manageability tool for deploying integration agents within the managed ecosystem is envisioned (see Section 4.5), aiming at:

- Translating among transport protocols. The Edge data broker is the main enabler for moving data because of scalability performance, still, many enablers might not work as MQTT node but work with API calls (or Kafka topics, or other mechanisms).
- Providing basic data formatting capabilities. Enablers might return data following their own format, which might not be the same as the one receiving it.
- Wrapping these agents following enablers' conventions and deploying them in the right place of the computing continuum.

3.5. Data view

The Data View presents a high-level perspective on data collection, processing, and consumption. It is meant to be a viewpoint on the flow of data, considering the specific actions that a set of enablers, or other data processing elements, perform upon them. This view allows to understand the system without superfluous technical details (usually required to implement designed software, such as the description of every message in an authentication scheme), proposing an overview meant to expose a high-level view of the flow of data within the system.

As a highly decentralised architecture, ASSIST-IoT does not mandate any particular flow of data. Instead, it allows connecting data processing functions, spread across the system, into concrete flows that answer particular requirements and needs of concrete deployments. A design abstraction introduced by ASSIST-IoT to describe high-level data flows is called ***data pipeline***.

Data pipelines are descriptions with visual and textual components of how data flow in distinct parts of ASSIST-IoT deployments. Being more abstract than typical, dedicated UML diagrams (e.g., activity diagram, sequence diagram, communication diagram), they are not meant to replace any diagramming standard but rather complement them. Pipelines may abstract away details from other layers (e.g., network topology, physical location, protocol details), while exposing elements critical for data processing and integration – information most relevant for data engineers and system integrators and designers. Priority is placed on visualising the flow of data and exposing necessary connections and integrations between elements.

In essence, a data pipeline is a **linear sequence** of data processing steps where data is transmitted (sent and received) between data processing elements. It may include joins or splits, but always has at least one beginning and end.

ASSIST-IoT introduces specific terminology to describe these sequences. A pipeline describes a process in which data is generated at a **source**, **processed by** some processing elements **(enablers)**, and output at a **sink**, from the point of view of a message travelling through a "virtual *stream*" (technically there may be more than one source or sink). Because data pipelines are a design tool and not a software specification standard, there is no requirement to use streaming technologies or asynchronous processing. Data flowing within pipelines is divided into **messages**, which are distinguishable pieces of said data that travel through different **data paths**. Each message has a specific **shape**, which includes its format and contents. It is important to note that in this context, a message must contain concrete data that is directly relevant to the described pipeline. Items, such as performance logs, even though technically may also be named messages, do not need to be included if they do not affect the understanding of the flow. Again, a data pipeline is a high-level abstraction, so gratuitous details are to be avoided.

Every pipeline includes a number of inputs, which are data-generating elements (also called sources). Those are services, endpoints, devices, sensors, etc., at which the data originates, and is usually assumed to be produced constantly, periodically, or as a result of a data-generating event measured or perceived at some sensor. Data is "created" at sources in the form of messages containing information about events, entities, measurements, and any other relevant artifact. Messages travel along directional data paths between enablers, that process the data, i.e., decide where to forward it, change or verify its contents, split the message into two or more, or remove parts of it before sending it further. The kind of processing done depends on the enabler. Eventually, any data path is terminated in a data sink, where the data is finally consumed, i.e., stored in a database, displayed in a dashboard, added to a log, or simply deleted if it is no longer needed. In complicated systems, the consumption of data may result in starting another process (i.e. lead to another pipeline). In such cases, it is important to clearly separate concerns and functions described in data pipelines, so that independent data processes are described in separate pipelines. Even though, technically, data sinks of one pipeline may be visually (and logically) connected to sources of another pipeline, it is recommended to use simpler and detached diagrams.

In order to document the data pipelines under ASSIST-IoT architecture, the concepts outlined above have materialised in a set of icons and visual instructions that will help illustrate them. The simplest scheme of a pipeline is presented in Figure 14. It visualises a mock pipeline with a single source and sink, where data goes through a single (unnamed) enabler. The data path has only 2 hops. Notice, that the purpose of this pipeline is not clear and the diagram itself is not enough to provide a useful explanation. Thus, visual description of data pipelines should always be complemented by a written description to provide additional explanation.

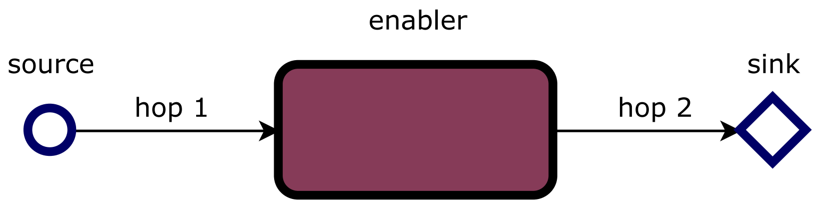


Figure 14. Basic data pipeline elements

In practice, pipelines contain more elements and more meaningful labels than in the mock example. A more comprehensive example (see Figure 15) includes more meaningful labels, for protocols and message shapes, multiple sources and sinks, as well as multiple inputs and outputs from a single enabler.

In the following example, a mock set of sensors feed information to an Edge Data Broker Enabler, which forwards some data to two storage services (one offered by the Long-Term Storage Enabler), and generates alerts for a log service. Notice, that the meaning or conditions that are necessary at the points of multiple inputs or outputs are not obvious from the image itself. Whether the input messages are joined (i.e., the enabler must always wait for a pair of messages to be available before processing), or entirely independent, should be explained in the written description of the pipeline. Similarly, the point at which the alert messages are created (marked in the image as hop "1.") should be accompanied with concrete textual explanations, that refer to the hop label. The message split (hop "2.") signifies that exactly the same message is sent at the same time to two receivers. In practice, this is unlikely to happen if the receivers are of different types (i.e., not replicas of the same element), because of differences in protocols and APIs. Including a message split in a diagram signifies that the message shape on this part of the path is exactly the same for all receivers.

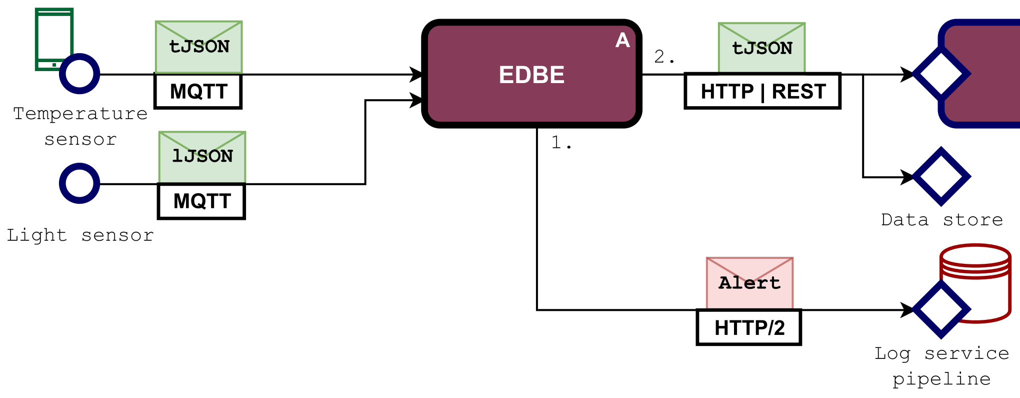


Figure 15. Data pipeline example

A well-designed pipeline should answer more questions than it raises. Simplicity of the visual language, proper labelling and explanation of message shapes, separation of complex systems into task-specific pipelines are very important. One of the key principles of pipelines is the unidirectional data flow. On a diagram, any loops or two-directional paths obfuscate information. It is not immediately clear what the shape of the messages travelling through a path is. In general, judging only by the visual language, the message may be the same (which raises the question, "why is the same message being bounced back and forth?"), or different ("at what point in the loop is it modified?"). This directly leads to the problem of not being able to point at an element in the diagram and read from it, what is the shape of the message at that point in the pipeline. Keeping to the standard of unidirectional data flow, such problems are avoided, and the pipeline description becomes more direct, accurate and readable.

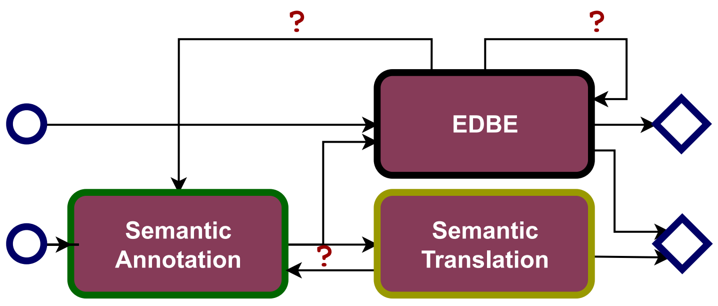


Figure 16. Wrong pipeline example

Expanded description of the visual elements of data pipelines, including ones not presented in this section, can be found in Appendix A.

3.6. Relation among views

One of the aspects that complete an architecture, from what concerns its Views, is the relation among those particular viewpoints. While the previous sections aimed at explaining the (final, formalised status of the) Views of ASSIST-IoT, these paragraphs reflect on the relation held among them.

As ASSIST-IoT intends to be a blueprint, Reference Architecture for Next Generation IoT deployments, it aims at following well-known (or *de facto*) standards for architecture specification creation. In the case of Views relation, the team has selected the 4+1 model published by Kruchten [16] as a valid reflection of its intention. In this reference, a structure of four views is exposed while an additional one (representing the "scenario" or use-case validation) is included where all of them converge. Figure 17 represents how the 4+1 approach fits ASSIST-IoT architecture, which the team of T3.5 decided to extend to a custom coined "4½+1" by: (i) tailoring it to the project' solution, (ii) adding relevant agents involved in them and specifying heritage relations, and (iii) dividing the original "Development view" by Kruchten's model into two different views (Development and Deployment). See the following figure for further clarification. Similar approaches approach has also been followed in previous works [17].

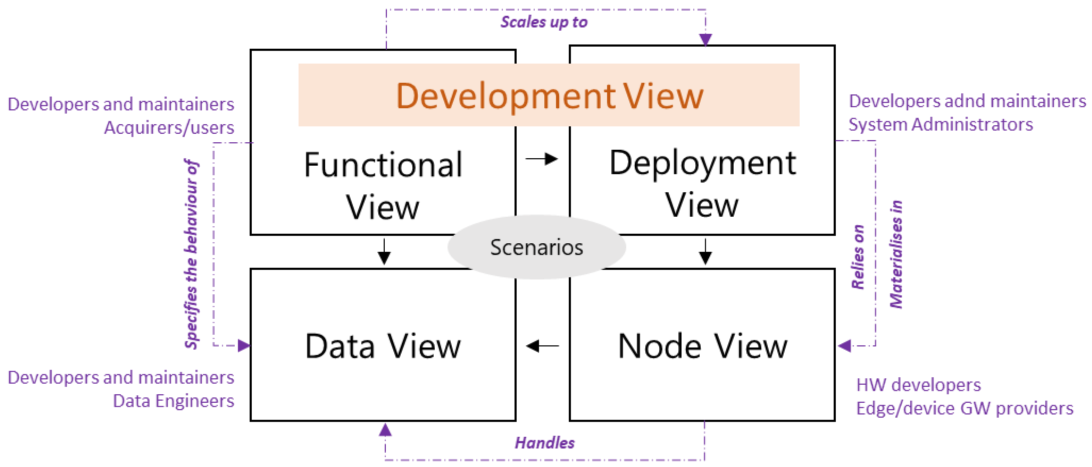
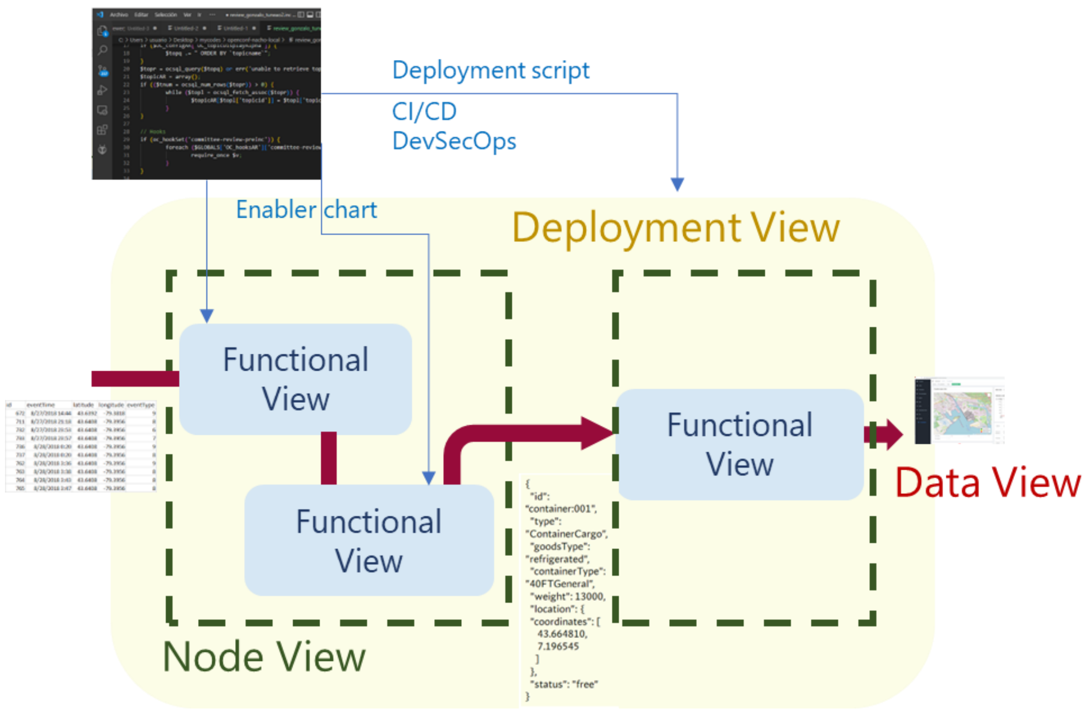


Figure 17. Custom 4½+1 model of relation among Views in ASSIST-IoT RA

Comparing to canonical Kruchten's division, the Logical view corresponds to ASSIST-IoT's Functional view, aiming at describing the functionality of the system. As per the recent incorporation of the Development view, it is necessary to distinguish between both. While the former (Development) is focused on providing guidance and orientation to enabler developers, the latter (Deployment) addresses the concerns of installers and maintainers stakeholders. However, under Kruchten's model, both occupy quite similar spots in the diagram. The development falls partially under the Deployment space (setting up scripts, charts, flows, etc.) and partially under the Functional view, as enablers to be developed target a functionality/NGIoT characteristic to be provided over the architecture. The project's Node view matches with Kruchten's Physical view, which is useful for system administrators relating to the computation infrastructure. Finally, ASSIST-IoT's Data view maps to the Process view, trying to represent the dynamic, flowing aspects of the system in terms of processing.

Notwithstanding the previous, and for the sake of clarity, Figure 18 represents a more natural way of expressing their relation. The Data view is about the changes over the data processed by the system (sensors, apps data, services data, metadata); The Functional view expresses the actual functionalities, while the Development view accomplishes the realisation of software to provide such functionalities, following DevSecOps defined methodology. Node view represents physical scopes where computation takes place; and the Deployment view joins everything together in an all-encompassing, global, topological, composition approach.



```
        $topq .= " ORDER BY `topicname`";
}
$topr = ocsql_query($topq) or err('unable to retrieve top
$topicAR = array();
if (($tnum = ocsql_num_rows($topr)) > 0) {
        while ($topl = ocsql_fetch_assoc($topr)) {
                $topicAR[$topl['topicid']] = $topl['topic
        }
}

// Hooks
if (oc_hookSet("committee-review-preinc")) {
        foreach ($GLOBALS['OC_hooksAR']['committee-review
                require_once $v;
        }
}
```

id	eventTime	latitude	longitude	eventType
672	8/27/2018 14:44	43.6392	-79.3818	9
713	8/27/2018 23:18	43.6408	-79.3956	8
742	8/27/2018 23:53	43.6408	-79.3956	6
744	8/27/2018 23:57	43.6408	-79.3956	7
746	8/28/2018 0:20	43.6408	-79.3956	9
747	8/28/2018 0:20	43.6408	-79.3956	8
762	8/28/2018 3:36	43.6408	-79.3956	9
763	8/28/2018 3:38	43.6408	-79.3956	8
764	8/28/2018 3:43	43.6408	-79.3956	8
765	8/28/2018 3:47	43.6408	-79.3956	8

```
{
  "id":
"container:001",
  "type":
"ContainerCargo",
  "goodsType":
"refrigerated",
  "containerType":
"40FTGeneral",
  "weight": 13000,
  "location": {
  "coordinates": [
    43.664810,
    7.196545
    ]
  },
  "status": "free"
}
```

Figure 18. Alternative representation of the relation among ASSIST-IoT architecture Views

4. Verticals

The ASSIST-IoT architecture is structured following a multidimensional approach composed of horizontal Planes and transversal Verticals. Specifically, Verticals target NGIoT properties that exist on different planes, either independently or requiring the cooperation of elements from different Planes. Verticals in ASSIST-IoT include: self-*, interoperability, scalability, manageability, and security, privacy and trust. As discussed above, the main building block in ASSIST-IoT architecture is an enabler – an abstraction term that represents a collection of components, running on nodes, that work together to deliver a particular functionality to the system. Enablers can support capabilities from one or more Verticals. Within the scope of the technical work packages, specific transversal enablers are designed, developed/adapted and integrated to facilitate the realisation of NGIoT systems. In the following subsections, for each vertical, methods for supporting given transversal capabilities are outlined. They include either dedicated enablers or technological and design choices. Note that each vertical can possibly contain more enablers.

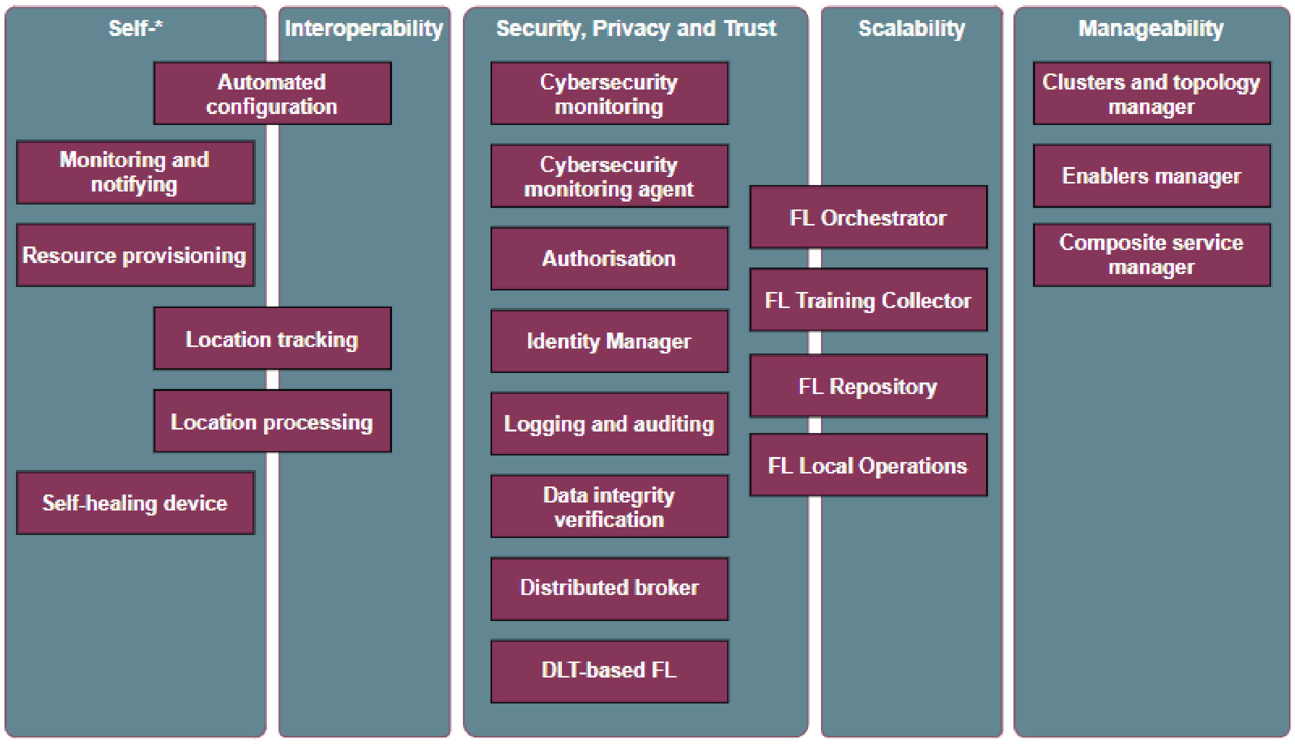


Figure 19. Enablers addressing vertical capabilities

4.1. Self-*

With the rapidly growing numbers of IT systems, their large-scale applications, and the vast amounts of processed data, the need of designing new architectural approaches based on the **implementation of certain system autonomy capabilities** arises. Autonomic computing is a computer's ability to manage itself automatically through adaptive technologies, which lead to (among others) reduced cost of ownership and maximised availability. The role of humans is then to manage the policies that drive their behaviour rather than manually acting over the involved mechanisms. Making intelligent decisions and following self-management procedures based on data (including contextual) available within the ecosystem allows the deployment and execution of contextual applications. Self-awareness and semi-autonomy (the extent of which is decided by a human) allow the system to be prepared for threats before they happen (e.g., a specific machine starts to fail) and to react faster and with more accuracy than possible in non-self-aware systems. IBM formulated eight conditions that a fully self-autonomous system should have [18]:

1) The system must know itself in terms of what resources it has access to, what its capabilities and limitations are and how and why it is connected to other systems.

2) The system must be able to automatically configure and reconfigure itself depending on the changing computing environment.
3) The system must be able to optimise its performance to ensure the most efficient computing process.
4) The system must be able to work around encountered problems by either repairing itself or routing functions away from the trouble.
5) The system must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.
6) The system must be able to adapt to its environment as it changes, interacting with neighbouring systems and establishing communication protocols.
7) The system must rely on open standards and cannot exist in a proprietary environment.
8) The system must anticipate the demand on its resources while being transparent to users.

In particular, five self-* mechanisms have been envisioned and defined for NGIoT (without precluding the possibility of defining additional ones, nor not needing some of them at all deployments). **Self-healing enabler** aims at providing the nodes or IoT devices with the capabilities of actively attempting to recover themselves from abnormal states based on a pre-established routine schedule (thus, it combines **self-awareness** with **self-diagnosis** and **self-healing** capabilities). This functionality is provided by the combination of three components: (i) a component that collects data from the device (such as battery usage, memory access, network connection, or CPU usage); (ii) a service that determines based on the gathered data if the device is in a healthy state or not, by applying ML models; and (iii) a remediation logic component, which based on the diagnostic, determines from a set of remediation processes which one should be used for proper healing.

The **Self-resource provisioning enabler** performs adaptative operations related to the resources assigned to the deployed software based on actual and forecast demand. It is able to increase or decrease CPU and RAM pre-emptively based on collected data and trends' pattern recognition. Particularly, it can predict demand peaks to prevent performance issues and also free resources when these are not expected to be needed (hence, it includes **self-awareness** and **self-organisation** capabilities). Models and anomaly detection can be done for the whole system, per selected enablers, or per selected components of a set of enablers.

The goal of the **Location processing enabler** is to provide highly configurable and flexible geofencing capabilities based on location data. This enabler endows the IoT system with the ability to **self-inspect** with focus on location data. The enabler consists of two main functions: (i) a tailored database that stores geolocation data, collected either from input streams or from HTTP requests, and (ii) an application whose purpose is to provide intelligence based on the received configuration (obtaining raw location data from queries run against the database).

The **Monitoring and notifying enabler** is designed and developed to ensure the uninterrupted functionality of IoT and edge devices, including gateways. Its monitoring module is designed to respond to malfunction incidents in the form of notifications. It is important to notice that a feature of a self-* enabler might be difficult to classify within a single self-* property, and might belong to more than one (this statement applies to all enablers). The self-* capabilities of this enabler are conceptualised around **self-configuration, self-diagnose** and **self-protection**. First, self-configuration is achieved by adding an automated backup and recovery system; access control of the enabler, in the form of proactive identification and protection from arbitrary attacks, covers the self-protection part; lastly, it is also considered to embed pre-failure warning to the software system, in the form of ML prediction, hence covering self-diagnose to further improve quality of service.

Finally, the **Automated configuration enabler** keeps heterogeneous devices and services synchronised with their configurations. Based on flexible "contextual" rules that describe the reaction to system-generated events, the enabler is able to maintain the desired system state. To make the structure of the enabler generic and application-agnostic, it utilises "connectors" as communication interfaces with its clients. These connectors allow the enabler to abstract away unimportant system details and concentrate on essential, configuration-related events. Both the configuration handling rules and types of accepted events can be expressed in a simple but expressive configuration data format. Through its functionality, the enabler realises a **Self-configuration** ability of Self-* systems.

4.2. Interoperability

Interoperability is the **ability of equipment from different manufacturers or different systems/devices/ applications to communicate together on the same infrastructure** (same system), or on another while roaming. It helps to achieve higher efficiency and a holistic view of information. The ASSIST-IoT RA should be applicable in vertical sectors with heterogeneous settings in terms of applications and data. Considering this, interoperability plays an especially important role for a fruitful NGIoT architecture realisation, regardless of the systems used in each separate case. This vertical is not addressed by means of dedicated enablers, but rather it is present at different levels. Interoperability is undertaken at three levels:

- **Technical interoperability**: it means the ability of two or more information and communication technology applications to accept data from each other and perform a given task in an appropriate and satisfactory manner without the need for extra operator intervention.
- **Syntactic (structural) interoperability**: it allows two or more systems to communicate and exchange data using compatible formats and protocols in case the interface and programming languages are different (e.g. by using a standardisation of the communication between a software client and a server).
- **Semantic interoperability**: this is the highest level of interoperability which denotes the ability of different applications/artefacts/systems to accurately understand (interpret) exchanged data in a similar way, implying a precise and unambiguous meaning of the exchanged information.

Interoperability is one of the key challenges of IoT deployments in which heterogeneous artifacts need to effectively communicate, share data and perform together in order to achieve a shared outcome. Without interoperability, applications would grow in costs and complexity, making the novel technologies' adaptation such as digital twins or federated machine learning not possible. In ASSIST-IoT RA, interoperability is included by design. It is one of the features that guide the choice of technologies and design of selected enablers from different Planes and Verticals, justifying it to be recognised as an independent vertical. Hereinafter, some examples of interoperability included in different Planes and Verticals are depicted.

Interoperability is addressed in terms of scalability, security, privacy and heterogeneity of data sources. ASSIST-IoT supports data interoperability by proposing a semantic data governance toolset and offering data sharing, privacy, security and trust enablers. Enablers from the Data management plane (Semantic Annotation, Semantic Translation and Semantic Repository, see Section 3.1.3) provide capabilities to process, translate and store data in different semantics, enabling effective communication and cooperation between IoT artifacts.

Another area that supports interoperability in ASSIST-IoT is the adoption of DLT, with benefits coming specifically from the utilisation of smart contracts. DLT-based enablers allow storing critical logs providing immutability so as to ensure non-repudiation, verifying data integrity, and providing data source metadata management. DLT, by its nature, is decentralised and scalable, which makes it a technology fit to be evaluated in NGIoT environments.

With respect to self-* mechanisms, examples of how interoperability is supported appear in localisation tracking and processing, in which various geospatial data sources, such as UWB location tags, GPS, and information about the layout of the site (such as the Building Information Model, BIM), are handled. Also, automated configuration proposes an abstract and flexible service, providing a representation of system state, configuration flows and fallback strategies that can be applied to deployments with heterogeneous artifacts. Here, the design includes connectors that allow the same mechanisms to be applied across the system.

Finally, the solution proposed for Federated Learning (based on four enablers) is also prepared to support interoperability. FL Local Operations enabler instances can be run on heterogeneous clients due to the principles behind FL. Moreover, each client may host data in a different format. Here, the FL Data Transformation component has been foreseen to prepare the data according to training job requirements.

To sum up, there are no enablers dedicated strictly to providing interoperability in ASSIST-IoT; However, interoperability is a feature that needs to and is supported on different levels and considered in various ASSIST-IoT related applications.

4.3. Security, privacy and trust

The aspects involved in this vertical should be considered meticulously when designing an NGIoT system or, otherwise, it is destined to fail. According to [19], more than 70% of deployed systems related with IoT have severe vulnerabilities because of a lack of encryption mechanisms, ineffective software protection, insecure web environments and insufficient authorisation. This is problematic as the managed data and the system itself can be compromised, which ultimately will affect its adoption. In the RA, this vertical is addressed considering dedicated enablers, but enablers belonging to other Planes and Verticals should be developed with good practices and security and privacy principles in mind to be present as inherent properties of the system.

Security

Broadly accepted security objectives associated with information assets include: confidentiality, integrity, availability, authentication, authorisation and auditability. More specifically, this property includes both **good programming, deployment and sharing practices to develop/implement secure code, infrastructure and access by design**, as well as enablers that provide **identification and access control** and **cybersecurity monitoring and response mechanisms**, which aims at offering protection against threats associated to ASSIST-IoT infrastructure that may conclude on an alteration of its characteristics to carry out activities not intended by owners, designers, or users. These are:

- **Identity manager enabler (IdM)**: it relates closely to the authentication phase as the access control process requires the identity of users, devices, systems, and other entities to later evaluate the access policies and grant or deny access.
- **Authorisation enabler**: this enabler is responsible for the authorisation phase in the access control process. Authorisation is a process of granting, or automatically verifying, permission to an entity to access resources after this has been authenticated. This enabler's operation, in the context of a decentralised environment, is depicted in Section 4.3.1.
- **Cybersecurity monitoring enabler**: this enabler consolidates the necessary information for cyber threat detection over the deployed system, including cybersecurity awareness, visibility and response against detected threats.
- **Cybersecurity monitoring agent enabler**: it reports to the cybersecurity monitoring server, enabling the execution of processes on the system target under study to provide relevant information to detect potential cybersecurity breaches, and also performing response actions if required by the monitoring enabler.

Privacy

It has the objective of **protecting the information of individuals (or private data) from exposure** in NGIoT environments. This aspect is not addressed with a specific enabler but must be included inherently in the all the developments as it is one of the most critical adoption barriers of an IoT system. The project has developed its own set of **rules for preserving privacy** during development, information that can be found in D2.4 [12] (apart from them, European and national regulations must be complied with). In summary, these rules identify security and privacy conditions applied for collection, sharing, storage and usage of personal data and personal data protection in ASSIST-IoT project activities (which could be applicable to other NGIoT deployments).

Regarding machine learning training of privacy-sensitive assets, ASSIST-IoT proposes a **Federated Learning (FL) strategy** and associated enablers. FL is a machine learning technique that trains an algorithm across multiple decentralised edge devices/servers holding local data samples without exchanging them. This framework is very suitable for NGIoT environments because of its decentralised nature, and consists of the following enablers:

- **FL orchestrator**: it is responsible for specifying FL workflow(s)/pipeline(s) details, including lifecycle management, job scheduling and error handling, among others.
- **FL repository**: provides storage for FL-related data, including ML algorithms, already-trained ML models suitable for specific data sets and formats, averaging approaches and auxiliary repositories.

- **FL local operations**: embedded in each involved party performing local FL training, it is in charge of data format compatibility verification and potential transformations, local model training, sending encrypted recommendations to the training collector, and inferencing the model over new arrival data.
- **FL training collector**: responsible for aggregating local updates of the ML model prepared by independent parties as part of a model enhancement process, as well as its delivery to the FL repository in order to be retrieved back by the FL orchestrator to the FL Local operations in further training rounds.

Trust

In an IoT system, devices generate/receive data which is sent among them or towards a system, where it can be processed in different ways. When the size of a given system increases, trustworthiness (i.e., the level of confidence that can be granted to a device, data, a system or an entity within the ecosystem [20]) becomes a great challenge. Apart from good programming and integration practices (exhaustive guidelines and recommendations can be found in trusted architectures like the one produced by the TIoTA alliance [21]), the ASSIST-IoT RA defines a set of enablers that leverage Distributed Ledger Technologies (DLT) for easing the realisation of a decentralised trust ecosystems, namely:

- **Logging and auditing enabler**: it logs critical actions that happen during the data exchange between stakeholders to enable transparency, auditing, non-repudiation and accountability of actions during data exchanges. It also logs resource requests and identifies security events to help provide digital evidence and resolve conflicts between stakeholders, when applicable.
- **Data integrity verification enabler**: this enabler is responsible for providing data integrity verification mechanisms that allow data consumers to verify the integrity of any data in question. Network peers host smart contracts (chaincode) which include the data integrity business logic; the enabler can compare the hashed data of the queries made by clients with the stored ones to verify their integrity.
- **Distributed broker enabler**: it provides mechanisms to facilitate data sharing between different heterogeneous IoT devices belonging to various edge domains and/or among different enablers of the RA. It deals with data source metadata management and provides trustable, findable, and retrievable metadata for the data sources (e.g., from IoT devices, enablers), which can be indexed and queried.
- **DLT-based FL enabler**: it fosters the use of DLT-related components to exchange the local, on-device models (or model gradients) in a decentralised way. The DLT can act as an auxiliary component to manage AI contextual information and prevent any alteration to the data, which is a major threat to the FL approaches (see privacy property above).

4.3.1. Security strategy in decentralised environments

When building a technology solution, there can be different network architecture options: centralised, distributed, and decentralised. In particular, decentralisation refers to the transfer of control and decision-making from a centralised entity to a distributed network.

In ASSIST-IoT, the envisioned strategy for decentralised security is the following. To begin with, the authorisation server shall include the possibility of sending security policies from one entity to another, by enabling exporting and importing policy files. This entails that the PAP (Policy Administration Point) can be present in just one authorisation server, where the required policies are defined and managed, and then these can be exported to remote authorisation servers that import them into to a local repository, referred to as PRP (Policy Repository Point). In Figure 20, two instances of the authorisation server (*Authz server*) can be seen located in different computing elements, one with the role of defining and exporting policies, and the other with the role of importing it to a local repository.

In the following figure, PEP is the Policy Enforcement Point, where the access control decisions are required, allowing or accessing the access to the managed resource; PIP is the Policy Information Point, which obtains from available sources the environment information missing in the original request to complete the request that has to be analysed; and PDP is the Policy Decision Point, where the decision is taken by crossing the request with the policies to obtain the effect to be applied (both PAP and PRP have been defined in the previous paragraph).

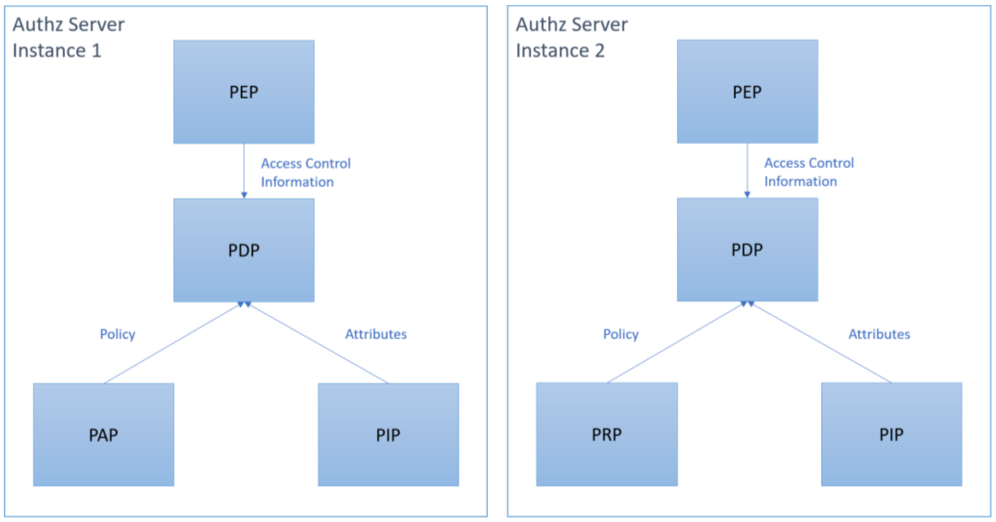


Figure 20. Policy exchange functionality between Authz server instances

This policy exchange feature among authorisation servers offers the possibility also to separate the decision-making feature from a central point to decentralised endpoints by exporting to these systems the corresponding security policy, so that it can be applied there. The following figure shows an example of decentralisation of authorisation features.

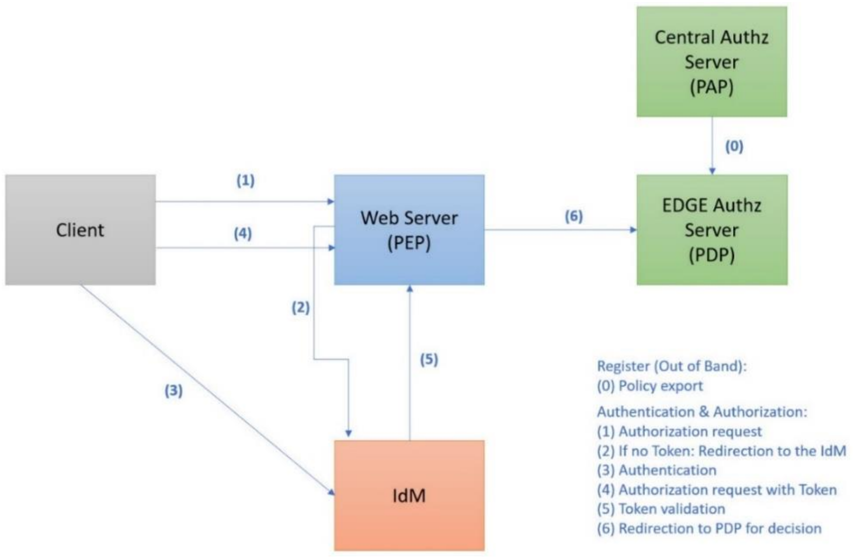

Central Authz Server (PAP)
(0)
Client
(1)
(4)
Web Server (PEP)
(6)
EDGE Authz Server (PDP)
(2)
(3)
(5)
IdM
Register (Out of Band):
(0) Policy export
Authentication & Authorization:
(1) Authorization request
(2) If no Token: Redirection to the IdM
(3) Authentication
(4) Authorization request with Token
(5) Token validation
(6) Redirection to PDP for decision

Figure 21. Decentralised authorisation process example

The steps are the following:

Registration (Out of Band process):

0) The central Authz server defines a policy and exports it to the remote one, which imports and stores it in a PRP. The policy is applied here. This step is done previously offline.

Authentication and authorisation:

1) A client makes an authorisation request to the Web server (PEP) requesting the needed service.
2) If the request has no authentication token, it is redirected so that the client is authenticated first.
3) The client is authenticated with the IdM (Identity Manager) enabler through Oauth2 protocol, and the corresponding token is provided.
4) The client makes a second request to the Web server with the authentication token already included.
5) The Web server verifies this authentication information with the IdM.
6) The Web server then redirects the authorisation request to the remote Authz Sever, where the policy has been previously imported and stored, and there the decision is made.

In the scope of a use case, a central Authz server (with the PAP) manages the policy administration, while the remote Authz server (with the PDP) may be deployed in edge-oriented deployments to promote decentralisation.

4.3.1.1. Third-party access

Application Programming Interfaces (APIs) serve as a fundamental part of modern software development across industries. ASSIST-IoT proposes that, when required, **internal services are exposed to third parties through an OpenAPI** (software) gateway as the primary mechanism.

By definition, APIs may expose application logic and sensitive data and, because of this, they have increasingly become a target for attackers. Therefore, APIs pose an increasing security concern because they are the gateway to data and to systems. API security provides solutions to understand and mitigate the vulnerabilities and security risks of APIs. In this way, the **number of attack surfaces is reduced** as this acts as the first security point and also, first-level access policies can be applied on there, without preventing that additional checking points are implemented within the platform. In any case, as a reference for ensuring that the developed and consumed APIs are secure, the list of most critical vulnerabilities released by the OWASP organisation can be consulted[31]. Briefly, they include:

1. **Broken object-level authorisation**: Object-level authorisation is an access control mechanism that should be considered at code level to validate that a user can access only objects that they should have access to, as APIs tend to expose endpoints that handle object identifiers.
2. **Broken user authentication**: a compromise of authentication tokens or implementation flaws might cause that malignant actors could assume other user’s identities temporarily or even permanently, compromising API security overall.
3. **Excessive data exposure**: developers tend to return all object properties without considering their individual sensitivity, relying on clients to perform needed filtering. Attackers could sniff the traffic to analyse the API responses, looking for sensitive data that should not be sent.
4. **Lack of resources and rate limiting**: APIs should implement quotas to users or external services, to prevent Denial of Service (DoS) attacks, brute force attacks for gaining authentication credentials, and excessive use of resources (RAM, CPU, etc.).
5. **Broken function-level authorisation**: overly complex access control policies with different hierarchies, groups, and roles can lead to implementation flaws, so legitimate calls to API endpoints might be accepted when they should not have access.
6. **Mass assignment**: Modern frameworks encourage developers to use functions that automatically bind input from the client into code variables and internal objects. This may imply binding client provided data (e.g., JSON) to data models, without proper properties filtering based on an allow list. Since APIs expose the underlying implementation along with property names, attackers (guessing properties, exploring other endpoints, reading the documentation, or providing additional object properties in request payloads) can use this methodology to overwrite sensitive properties.
7. **Security misconfiguration**: It can happen at any level of the stack, result of unsecure default/ad-hoc/ incomplete configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods, permissive Cross-Origin Resource Sharing (CORS), and verbose error messages containing sensitive information. Attackers may attempt to find unpatched flaws, common endpoints, or unprotected files and directories to gain unauthorised access or knowledge of the system.
8. **Injection**: Injection flaws, such as SQL, NoSQL, command injection, etc., occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s malicious data can trick the interpreter into executing unintended commands or accessing data without proper authorisation, which may lead to information disclosure, data loss, DoS attacks or complete host takeover.
9. **Improper assets management**: APIs tend to expose more endpoints than traditional web applications, making suitable and updated documentation highly important. Proper hosts and deployed API versions inventory also play an important role to mitigate issues such as deprecated API versions and exposed debug endpoints. Old API versions are usually unpatched, so attackers can gain access through them.

[31] https://www.synopsys.com/blogs/software-security/owasp-api-security-top-10/

10. **Insufficient logging and monitoring**, which coupled with missing/poor incident response strategies, allows attackers to further affect systems once compromised, as they might have large periods of time to perform malicious activities without being noticed.

Based on this list, it can be concluded that authentication and authorisation are the key to API security. These failures are the most common means by which API systems are hacked. API authentication capability is crucial because the API relies on the user's authentication to decide which authorisation privileges to grant.

4.3.1.2. MQTT security considerations

There might be cases in which the network is not considered secure or trusted enough thus pub/sub communication based on MQTT needs to be secured. To that end, a set of security mechanisms need to be considered at different levels to prevent attacks of different nature[32] (**Note**: MQTT specifically, as it is the main pub/sub protocol considered for data routing within the RA, but similar concepts should be applied if other mechanisms are considered):

- **Network level**: One way to provide a secure, trustworthy connection at the network level is to provision VPN connections for the communication between clients and brokers, so traffic is tunnelled. This is especially needed when sensitive data is to be shared through public or untrusted channels.
- **Transport level**: On this level, communication should be encrypted, and identities authenticated. TLS/SSL is commonly used for transport encryption, providing confidentiality. This method prevents data from being read during transmission and provides client-certificate authentication to verify the identity of both sides. The feasibility of TLS on constrained devices is a key point to consider.
- **Application Level**: The MQTT protocol provides a client identifier and username/password credentials to authenticate devices on the application level. These properties are provided by the protocol itself, while fine-grained control of what devices are allowed to do is specified in the broker configuration.

Hence, MQTT authentication and authorisation is a crucial aspect when considering security aspects in this protocol. ASSIST-IoT RA recommends to setup MQTT authentication and authorisation so **external nodes and applications can only publish and subscribe to their own MQTT topics**. There are multiple ways to setup authentication and authorisation in MQTT, such as username-password stored in dedicated databases (or files) plus **ACL** (Access Control List) files to manage access rights.

In addition to **username and password**, MQTT clients provide other information that can be used for authentication [22]. One option is the client identifier, so all of them should provide its unique ID to the broker in the CONNECT message alongside its username and password. Another authentication method is to use the **X.509 client certificate**. The client presents it to the broker during the TLS handshake. After a successful handshake process, some brokers permit the use of certificate information for application layer authentication. The latter, jointly with ACL or an analogue mechanism to fine-tune the accessibility to the exposed broker topics, is the recommended stack for granting access to third-party systems in case MQTT protocol is used.

4.3.2. DLT strategy in decentralised environments

4.3.2.1. Data pre-screening

The deployment of DLT-related enablers in ASSIST-IoT will handle privacy, trust, and decentralisation. DLT is well-known for: (i) decentralising procedures for IoT networks, (ii) offering a stable state for IoT devices, (iii) being resistant to change, (iv) offering anonymity, and (v) establishing data immutability.

The RA proposes a **permissioned** blockchain, which operates among a group of known, identifiable, and verified participants under a governance mechanism that produces a certain level of trust. The permissioned network allows for flexibility in the creation of the network, as the business case can dictate the number of participants guiding the adoption of a **private** blockchain with one organisation or a **consortium** blockchain with more members. Every participant can have different types of access rights, so only authorised users can post data to the blockchain. Also, in such a permissioned setting, the possibility of a participant purposefully adding harmful code via a smart contract is reduced. The reduction of possible harmful events is achieved,

[32] https://www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals/

firstly, because the participants are acquainted with one another, and all transactions in the DLT are recorded on the blockchain in accordance with an endorsement policy established for the network and relevant transaction type. Rather than being entirely anonymous, the perpetrator may be easily identified, and the issue addressed in accordance with the governance model's criteria. Furthermore, data will be posted in the blockchain via transactions, so all actions are recorded in the ledger. Lastly, data need to have a specific format defined by the smart contracts or otherwise they will be rejected.

4.3.2.2. Critical information assessment

DLT-based technologies grant immutability of the data kept on the ledger. Data stored in DLT is data which changes have to be acknowledged, either in case they have been tampered with or changed at any time. Transactions cannot be tampered once recorded on the ledger.

The most critical data will be stored in a central system/database (e.g., LTSE) but also in DLT for extra security. The user will thus be able, whenever needed, to pull these data from the central database and from the DLT and confirm that no alteration has been made. The RA will root on a permissioned blockchain that works with consensus. Regardless of the fact that every transaction is recorded in the ledger, there is also the consensus mechanism, which provides the extra security since a consensus must be reached in order to submit a transaction.

The inclusion of solely critical events is essential for two main reasons:

- **Scalability**. The CAP theorem [23] states that distributed systems cannot have the following three features simultaneously: consistency, availability, and partition tolerance. Blockchain achieves eventual consistency through the replication of the ledge on the nodes. In detail, consistency is not concurrent with partition tolerance and availability, as it is the result of the validation of the state from multiple nodes. In ASSIST-IoT, flexibility should be permitted for achieving architecture's pivotal properties of scalability and future use. For this reason, only the critical events should be documented.
- **Data privacy**. In the future, new stakeholders can join the network. Data are visible to the network's participants, and business data should not be stored on the blockchain. In essence, blockchain is used as a trust layer with minimisation for the data to achieve privacy. There is no need for the actual data to be stored on the chain but rather only for representation. The abstraction complements the first point of scalability and the data volume decrease.

Some guidelines regarding the decision flow of the critical events can be found in the following items, that are structured based on questions that an administrator of the system should make to themselves:

- **Who should make the decision of whether an event should be considered critical or not?** The decision on the critical events is to be made by stakeholders (always with the aid of technical personnel). This is due to the requirements and applied technologies which can be complex and radically different for providing an elegant solution for general us.
- **How to decide which events are critical?**
    - In order for the enabler's responsible to decide whether their data are critical, they must consider the impact of events. In detail, the probability of an event occurring and the impact of the consequences are factors to guide the decision to define critical data.
    - Multiple parties can have a vested interest in accessing the exact dataset. DLT can facilitate the need to propagate data to different parties.
    - If they have data that cannot be tampered with and have to be at all times retrievable.
    - If they want to verify that their data are actually correct and has not been tampered with.
    - If they have events that are related to healthcare, sensitive or personal data.
- **Should these critical events be "static" or "dynamic"**, i.e., could new events be considered critical and could current ones be modified dynamically? If so, how? The majority of the cases will be fixed due to the complexity and differences in each use case. There is room for a degree of flexibility in providing a degree of dynamic data storage. For example, the definition of XACML in the proposed policy-based management system could be complemented by critical event rules that could be inserted via frontend. This functionality would allow admin users to specify relevant fields. It is expected that the selection of these events would be tightly coupled together with business use cases.

- Examples related to which events can be considered **critical**:
  - The fall of a worker has a high impact on the organisation. Third parties can be involved in the process and required to access critical data like employee groups or government authorities.
  - If a worker from a construction site is in a place where they should not be (dangerous zone where only specific personnel can work), the location data are considered critical according to the company policy.
  - Monitoring and notifying enabler can monitor data from an edge thermometer. If the temperature surpasses a threshold and is dangerous, these data are critical and would be stored in the DLT.
- Example of events that are **not critical**:
  - Monitoring and notifying enabler can monitor data from a thermometer at an edge location. If the thermometer has a normal temperature it does not need saving on DLT.
  - An error from the Semantic translation enabler caused by a bad input format.

4.3.2.3. Decision flow and enablers integration

DLT enablers are mostly independent from the rest of the architecture, and they are deployed on the cloud or at the edge, depending on privacy aspects and resource capabilities. Privacy issues arise from the need of keeping the data on each of these enablers. IoT devices' capabilities are another feature to consider when deploying a DLT solution, as they may have limited capabilities. As presented below, the DLT logging and auditing enabler (see D5.3 for more information) is cloud-based.

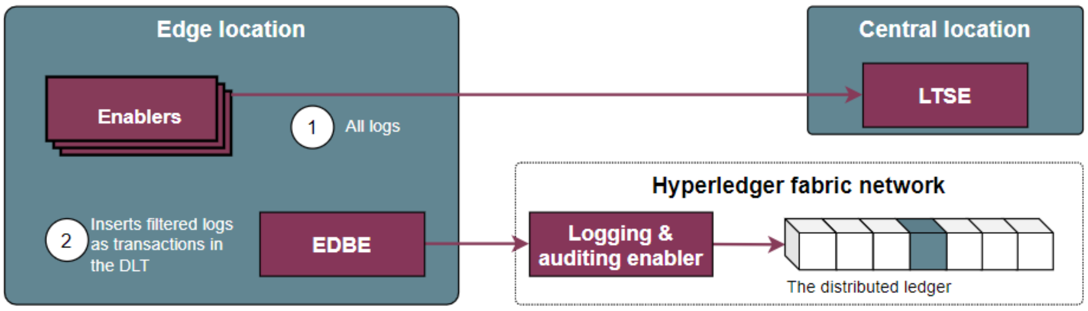

Edge location
Enablers
1 All logs
2 Inserts filtered logs as transactions in the DLT
EDBE
Central location
LTSE
Hyperledger fabric network
Logging & auditing enabler
The distributed ledger

Figure 22. Pipeline of Logging and auditing enabler

However, the FL DLT enabler will have DLT nodes placed at the edge to prevent FL model metadata from leaking.

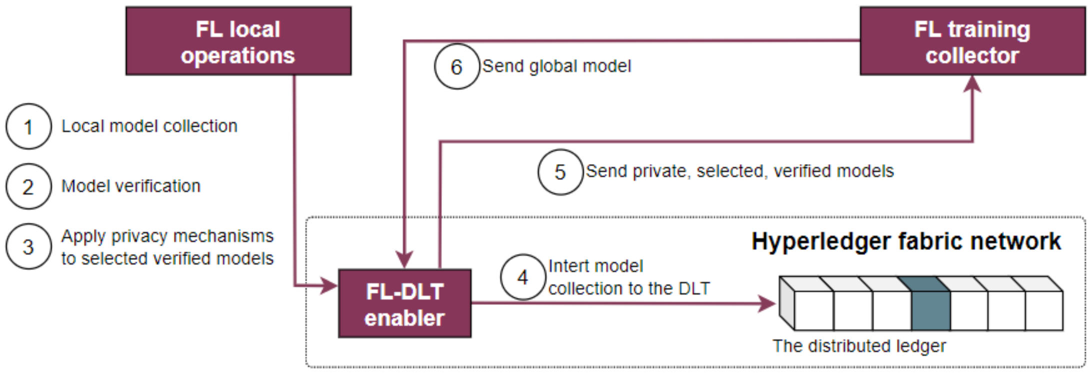

FL local operations
1 Local model collection
2 Model verification
3 Apply privacy mechanisms to selected verified models
6 Send global model
FL training collector
5 Send private, selected, verified models
Hyperledger fabric network
FL-DLT enabler
4 Intert model collection to the DLT
The distributed ledger

Figure 23. Pipeline of FL-DLT enabler

Due to blockchain's relatively limited storage scalability and privacy issues, it is recommended that only the most essential events be stored on DLT and blockchain as defined above. DLT will store only critical logs for the DLT logging and auditing enabler in ASSIST-IoT. At the same time, the DLT will accommodate other data depending on the rest of the DLT enablers defined in D5.3 [24], such as transactions integrity verification hashes, data source registries to enable interoperability, FL model metadata, and XACML policies. An example of a workflow for integrity verification enabler is displayed below.

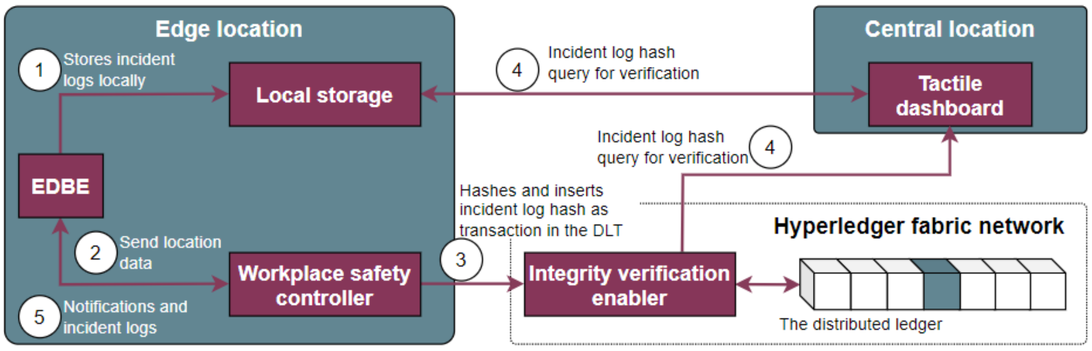

Edge location
1 Stores incident logs locally
Local storage
EDBE
2 Send location data
Workplace safety controller
5 Notifications and incident logs
4 Incident log hash query for verification
Central location
Tactile dashboard
Incident log hash query for verification 4
Hashes and inserts incident log hash as transaction in the DLT
3
Integrity verification enabler
Hyperledger fabric network
The distributed ledger

Figure 24. Pipeline of Integrity verification enabler

The implementation of smart contracts is vital for encoding logical functions to be executed on the DLT's nodes. "A smart contract defines the rules between different *organisations* in executable code. Applications invoke a smart contract to generate transactions that are recorded on the ledger[33]". ASSIST-IoT's DLT implementation uses the native Chaincode for deploying smart contracts. The goal is to execute decisions on chain for decentralising some tasks. Once smart contracts are deployed on the chain, the data are immutable, building trust between the parties. Any changes in smart contracts require the deployment of a new contract.

In cases where blockchain nodes are executed at the edge (FL combined with DLT, and XACML combined with DLT), the consortium blockchain network will be completely decentralised, whereas in cases where blockchain nodes are deployed on the cloud (logging and auditing, integrity verification, and broker), the consortium blockchain network will be dependent on networking capabilities of the cloud. The consortium blockchain is essentially a private network backed by Hyperledger Fabric for identity management.

4.4. Scalability

The Scalability vertical addresses the dynamic technical and business needs behind NGIoT deployments, in general, and ASSIST-IoT, in particular. Because of the variability of IoT-edge-cloud continuum scenarios for the NGIoT, the RA envisions to enable elastic scaling deployments ranging from modest barely local operations, up to large heterogeneous deployments based on demand features and functionalities. This scalability is essential for adapting to different workloads, performance, costs, and other business needs.

Scalability vertical in ASSIST-IoT is a **property** of the system that is **present due to (i) the design principles, (ii) the container orchestration technologies leveraged, and (iii) the functionalities covered by the Planes and other Verticals of the architecture**. This means that no specific enablers are provided to guarantee this property, but rather comes implicitly from all the former, still, enablers should be designed with scalability in mind considering their particular characteristics. Hence, ASSIST-IoT scalability property involves not only software scalability, but also hardware scalability.

4.4.1. Scalable hardware

It involves the ability to add, and modify the configuration, of the internal elements (either sensors, actuators, or computation nodes) of an NGIoT deployment. In addition, it also covers (i) computation scalability, (ii) storage scalability, (iii) interfaces scalability and (iv) system scalability:

- *Computation scalability*: from single core CPUs on PLCs, to specialised GPUs with thousands of cores required for AI model training.
- *Storage scalability:* from simple flash memory chips to large arrays of cluster disks.
- *Network interfaces scalability*: from a single wireless (or wired) interface to large arrays of wireless (or wired) interfaces with aggregate capacities of high throughput.
- *System scalability*: wraps the former hardware scalability aspects, so the architecture can support small-size to large-size, decentralised topologies, considering the possibility of having thousands or tens of thousands of clusters with their respective managed devices without adding too much complexity in management aspects.

[33] https://hyperledger-fabric.readthedocs.io/en/release-2.2/smartcontract/smartcontract.html

This scalability dimension is available throughout the ASSIST-IoT deployments due to the scalable-by-design edge nodes, i.e., GWEN. Its layout schematic includes two M.2 interfaces (to enable the use of WiFi6 and 5G networks in parallel), and two SD card connectors (to separate firmware from data storage). In addition, although several modules are mounted by default, other interfaces might be implemented as expansion modules, according to users' needs. For further details, see Section 3.1.1.

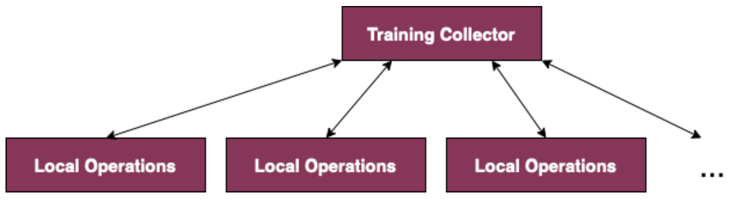
4.4.2. Scalable software

The software scalability is of paramount importance for NGIoT deployments, as it will not only include applications, but also their optimal management of assigned resources. In detail, the cluster management of deployments must be adaptable and scalable in order to enable the efficient operation of tens/hundreds of computing/processing nodes in support of thousands of smart and connected things. To do so, a **scalable orchestration that manages the partitioning, balance, and allocation of resources across an architecture realisation is needed**. Furthermore, as business data analytics algorithms will handle data of several orders of magnitude, it will also have a particularly aggressive scalability target. To do so, composability and modularity are key aspects of software scalability. To cope with this **scalability dimension, Kubernetes and related technologies for container orchestration at the edge** are leveraged to schedule and monitor ASSIST-IoT encapsulated enablers. Consequently, developers can focus only on running in the most reliable and safe manner their applications. The use of container orchestration systems is not the only way that addresses software scalability in ASSIST-IoT. Both **microservices architecture and containerisation design principles** contribute to it, as the addition of extra features is eased by the decoupled nature of microservices and the low overhead added by containers (in comparison to other virtualisation technologies, as explained in Section 2.3).

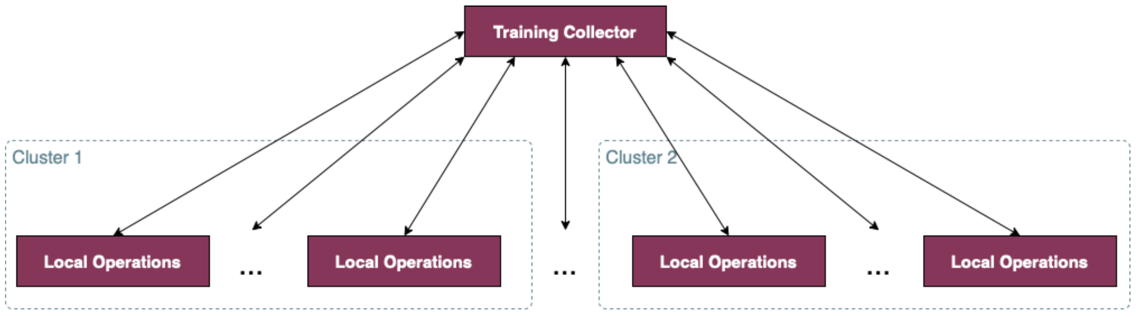
In detail, as explained in D6.5, as well as in Section 3.4, an NGIoT deployment can be composed of one or more tiers, which in turn, comprised of one or more nodes, each of them running a K8s distribution. Whereas top-tier nodes could be deployed on premises or in the cloud, following high availability strategies, nodes from low-level tiers requirements greatly depends on the use cases to be addressed. Also, in large-scale deployments where data is transmitted to upper tiers, a careful data aggregation and filtering strategy should be applied to avoid network bottlenecks and overuse of storage resources.

One last remark with regards to scalability on ASSIST-IoT is the one associated with the enablers themselves. The design principles aid, but the involved **software has to be designed with this principle in mind**. Examples for security and DLT strategies in decentralised environments have been already depicted (Section 0). A practical example related to the topology vision for the FL system, also introduced in Section 0, can be given. In particular, as support for different topologies has been shown to be important in large-scale real-life systems, the possibility of easily implementing them is crucial. Also, the ASSIST-IoT FL architecture is expected to be resistant to sudden user dropout, network connection interruptions, or uneven grouping of clients, by performing slight modifications to the basic centralised architecture. In particular, as it can be seen in the diagrams of igure 25, four topologies have been envisioned, thanks of the modularity approach of the FL enablers of the project.

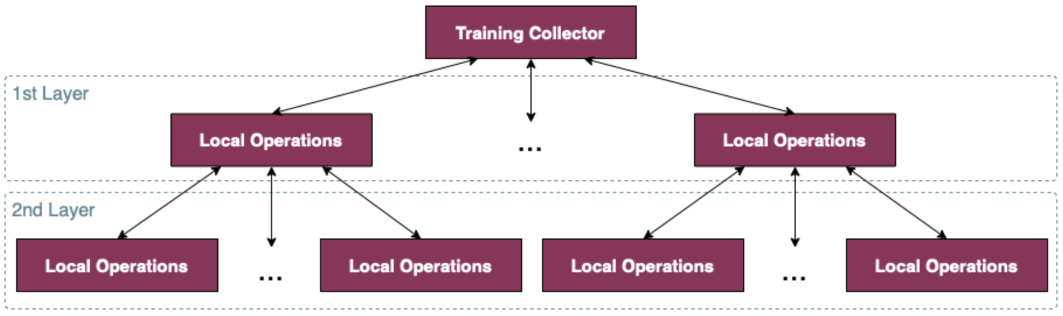
a)



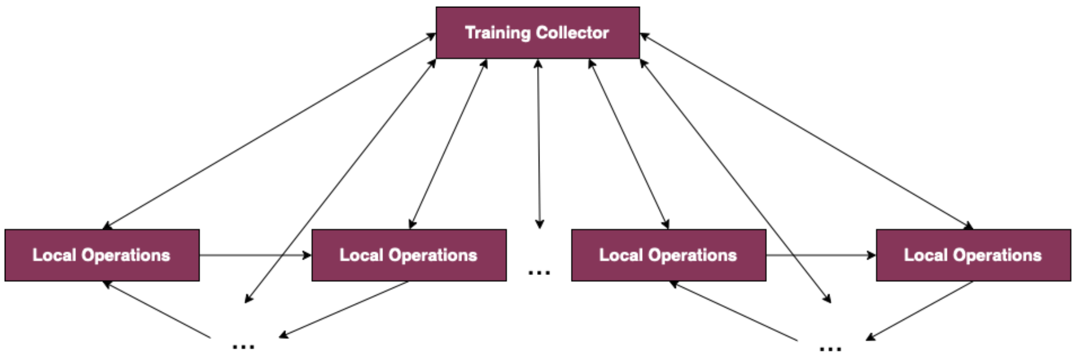
b)



c)



d)



igure 25. ASSIST-IoT FL alternative architectures for scalable IoT environments: (a) centralised architecture, (b) clustered architecture, (c) hierarchical architecture, (d) star architecture with ring-based groups

4.5. Manageability

This vertical fulfils the need for high-level **management of the overall system and the enablers** from other Planes and Verticals. Enablers are expected to be quite functionally-independent among them (even though in some cases, a set of them might be tightly-related to provide a complete feature), still, a certain control is needed. Moving towards a decentralised ecosystem, with workloads that can be deployed in different places of the computing continuum, claims to have a set of manageability mechanisms adapted to the idiosyncrasy of these environments, to aid platform administrators in their daily operations. It is important to stress that, apart from dedicated tools that belong inherently to this vertical, manageability should also be a cross-cutting property of the system, so enablers belonging to other Planes or Verticals could contain mechanisms that aid in its realisation.

This vertical is inspired by the FCAPS model (Fault, Configuration, Accounting, Performance, Security), transversal to the rest functional and vertical solutions. Particularly, it should (i) integrate mechanisms to **detect and inform** about **faults** in the system, involving either clusters or enablers themselves; (ii) provide user-friendly ways to **configure the clusters** to be managed **and the enablers required**, so the latter are deployed where needed and with the proper **configuration parameters** (also, for some of them working together to address a given use case, e.g., for realising a data pipeline – see Section 3.5); (iii) specify processes for storing and presenting operational information (i.e., **logs**) of the managed enablers; and (iv) ensure mechanisms to allow **capturing and presentation of metrics** to have the possibility of assessing the performance of the managed enablers, (v) everything in a **secure manner** so the integrity of the communications and the managed data is not threaten. The ASSIST-IoT architecture proposes the next enablers and tools for easing the management of a Cloud-Native, NGIoT ecosystem:

- **Clusters and topology manager**: The purpose of this enabler is, mainly, to register a cluster to later on deploy the enablers from the rest of Planes/Verticals. It ensures that a registered cluster is installed and working correctly, and also deploys a set of graphical interfaces for a user-friendly control of the orchestrator from the Smart network and control plane (Section 3.1.2). Finally, it provides information of the current topology and the enablers deployed over each of the managed clusters.

- **Enablers manager**: This enabler facilitates the management of enablers, mainly by facilitating the initial configuration and deployment of enablers in a graphical, user-friendly way (realising strategies for deploying them in all clusters, specific nodes, etc.) and also by listing the ones deployed on the system so far. It also allows getting the enablers status, terminating them and showing their logs.
- **Composite services manager**: The last manageability enabler aims at facilitating the integration of enablers that are not natively integrated (see Section 3.4.4). Enablers usually expose an API (based on HTTP) to be consumed, though some of them might move data considering other protocols (e.g., MQTT). In some cases, an auxiliary agent might be needed to move data between enablers, translate among different communication protocols and/or prepare data considering specific formats (bringing interoperability). This enabler provides a graphical tool to design and launch these agents, which will then be wrapped as enablers and deployed in the right place of the managed computing continuum.
- **Other tools for supporting manageability vertical**: Apart from dedicated enablers, this vertical also suggests a set of mechanisms (endpoints) that all enablers should have to ease the overall management. These endpoints, detailed in Section 0, are not mandatory but rather recommended. In summary, they are related to health monitoring, metrics provisioning, API publication for third-party access, and versioning. Lastly, logs (as well as metrics) from all the deployed enablers should be collected and stored so in case of failure they can be consulted and presented by the enablers' manager (see Section 5.3 for detailed considerations followed within the technical work packages).

5. Considerations for architecture realisation

This chapter encompasses some **complementary information for realising an ASSIST-IoT system**, enriching the guidelines, recommendations and best practices provided in the architecture Views. Specifically, these considerations start by (i) introducing the **essential enablers** that should be present in all (or at least, most) NGIoT deployments, with the (ii) set of **common endpoints** that every enabler (essential or not) should incorporate for providing typical information about its status and performance, also (iii) presenting the decentralised **approaches** for **monitoring and logging** from these enablers, leveraging already-defined enablers and exposed endpoints, and (iv) explaining the possibility that some features might not be provided following the RA design principles or the enabler abstraction specifications, thus introducing the **encapsulation exceptions**. These considerations go beyond the information that the views of the RA should provide, as they might be too specific and biased by the actual realisation from technical work packages, and thus they are presented separately from the previous sections rather than incorporated into them.

5.1. Essential enablers

The architecture at hand has specified the enablers that will provide functionalities to an ASSIST-IoT system. Depending on the characteristics and goals of such an IoT deployment, some of them might be strictly necessary (**essential**) while others might not be. After a thorough analysis of the whole list of enablers, it has been defined that **11** (**out of the 41)** should be considered essential (**26.82%** of them). It is believed that this reduced list will help teams prioritise the deployment in real scenarios.

Some of the essential enablers will be **pre-installed** in any ASSIST-IoT deployment. This means that, at the moment that a user will start being able to deploy enablers, the previous (pre-installed) will already exist, as they are the ones that facilitate all the process. Those (pre-installed) enablers do not follow the encapsulation principle, neither are subject to be managed by the Manageability enablers. Therefore, those (pre-installed) can be considered the "pre-requisites" for an ASSIST-IoT deployment to be usable. The essential enablers are depicted in the figure below (highlighted with a green square), and the explanation of their necessity to be considered as such (together with other indications) can be found at Table 4.

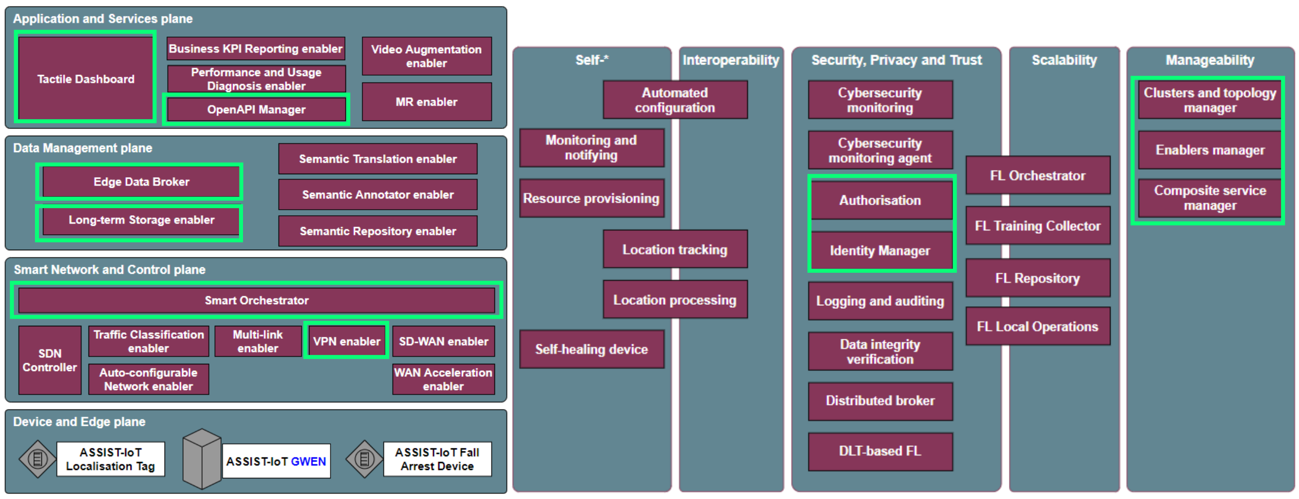


Figure 26. Highlighted ASSIST-IoT essential enablers

Table 4. Rationale behind "essential" consideration

Enabler	Rationale	PI*
Smart Orchestrator	It is the "brain" and "engine" of the whole solution of a deployment, allowing the installation of the rest of the enablers via the *kubectl* proxy tool.	X
VPN enabler	This enabler is in charge of facilitating 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 one (edge and far-edge nodes are expected to live in remote networks, geographically and logically, thus this is needed.	
Edge Data Broker	This enabler is the facilitator of data moving across enablers in the deployment. Especially, relevant where data from IoT devices must be distributed.	
LTSE	LTSE has been designed to cover any kind of storage requested by the enablers in the deployment, added to keeping enablers status information for potential recovery and supporting roles and access related information.	
OpenAPI Manager	The element that allows external parties (not owners of the deployment) to access enablers' data and functionalities.	
Tactile Dashboard	Centralised user interface for the configuration of the deployment. Main interface and access gate for humans to a whole ASSIST-IoT deployment.	X
Security enablers	Must-have security features, such as identity management, authentication, authorisation and access and policy rules specification.	
Manageability enablers	It allows the interaction with the orchestrator, oversees the enablers and nodes status and allows the combination of enablers towards data transfer in the deployment.	X

***** PI stands for **P**re-**I**nstalled, indicating those that will be natively existing (not encapsulated nor managed by the orchestrator) in an ASSIST-IoT deployment.

5.2. Enablers' common endpoints

A first convention that should be followed by all the enablers is API versioning. This practice is required if (typically major) changes may break consumer services or applications that make use of the API. Versioning is realised by adding /v1/, /v2/, etc. at the start of the API paths. Other practices such as using nouns instead of verbs in endpoint paths, handling errors with status codes and informative messages, enabling filtering and pagination, and caching are encouraged although not required. Besides, four minimum endpoints have been defined to be incorporated in all the enablers.

- **/health** (GET method): In K8s deployments, it is considered a good practice to incorporate three kinds of probes[34], namely: (i) *startup* probes, used to confirm that containers are up and ready, (ii) *liveness* probes, which detect if the application process has crashed; and (iii) *readiness* probes, which ensure that the application is ready to handle requests. A probe can be of type HTTP, which outcome depends on the response code (most common); container command; or TCP probe, where a TCP connection on a specified port is tried to be established. Thanks to them, containers can be re-deployed automatically by the container orchestrator system, and in case of failure, get some data for later debugging processes.

  In the scope of ASSIST-IoT, as a minimum requirement, this endpoint is responsible for collecting and providing information about the liveness status of the internal of the components, without the need of the platform administrator to interact with any K8s console. It is suggested to make use of status codes, providing in case of error a response indicating the unready or failed component.

- **/api-export**: this endpoint should provide a list of methods and endpoints that can be served by the enabler, in accordance with the OpenAPI specifications[35]. The format will be in typical JSON or YAML format. Since this endpoint satisfies the objectives to be used both as documentation, by providing knowledge about the usage of each of the available resources, as they are accompanied by examples, as well as they can be used in API managers or API frameworks (e.g., Swagger) to publish and access API resources, so users with proper rights can make use of them. It will incorporate two main sub-endpoints:
    - **/openapi** (GET method): it should return the OpenAPI specification in JSON format. Users will be able to download the API specification.
    - /**docs** (GET method): is should return an OpenAPI documentation provided by Swagger UI. Swagger UI lets users visualise and interact with the APIs' resources, which simplifies backend implementation and client-side consumption.
- **/metrics** (GET method): this endpoint provides performance metrics relevant to the particular enabler to be scrapped by the PUD diagnosis. They should expose raw metrics reflecting the current status of the overall enabler in a format that is compatible with the PUD and the monitoring stack realisation. More specifically, the RA recommends the use of Prometheus as central technology for monitoring (see next section), and therefore metrics should be served as plaintext following this technology's conventions. More detailed info related to its construction can be found in Section 5.3.1.
- **/version** (GET method): This endpoint returns the current version and patch of the enabler. The versioning format should follow the Semantic Versioning Specification (SemVer[36]), and should be carefully considered within CI/CD pipelines.

5.3. Monitoring and logging approaches

5.3.1. Monitoring considerations

Successful data-driven decision-making processes of systems with a large variety of devices and services require a careful gathering of data. In particular, metrics about the performance and usage of the resources involved in the lifecycle management of the various services and devices must be captured. A monitoring system is called upon to resolve this critical task. As various services and devices function on multiple layers, the traditional method in which system administrators, network administrators and application developers manually collect metrics on a regular basis is not an acceptable solution. The use of novel technologies can provide a more automated solution to this problem.

In the context of a scalable, trustworthy, domain agnostic and interoperable NGIoT ecosystem, created with several building blocks and enablers proposed by the reference architecture; the requirement for a monitoring service is intensified, since the need to predict failures is considered critical in such deployments. In the RA, the Performance and Usage Diagnosis enabler –more information can be found in deliverable D4.2 [25]– is

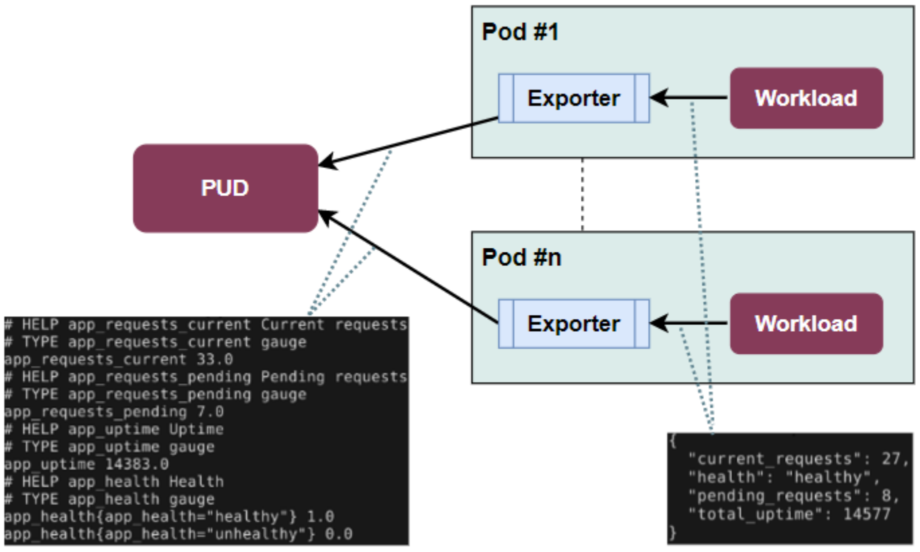
[34] https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
[35] https://swagger.io/specification/#:~:text=Introduction,or%20through%20network%20traffic%20inspection.
[36] https://semver.org

filling the role of the monitoring service, collecting data and metrics and making it available to platform administrators as well as to other enablers.

Monitoring architecture

The PUD enabler is a service capable of gathering data and metrics from endpoints made available by other enablers within the ASSIST-IoT ecosystem. The RA does not force any particular technology or monitoring stack, and different approaches (based on, e.g., Metricbeat[37], cAdvisor[38], K8s' kube-state-metrics[39], Prometheus[40], or even a combination of them) could be followed. Specifically, the PUD has been developed based on Prometheus since the latter is the *de facto* standard for metrics gathering in Cloud-native environments.

To be integrated with it, all deployed enablers need to expose a metrics endpoint that provides their relevant metrics data (see Section 0). PUD follows an HTTP pull model: it scrapes performance metrics from endpoints routinely. Typically, the abstraction layer between the application and PUD is an **exporter**, which takes application-formatted metrics and converts them to Prometheus metrics for consumption (see Figure 27). There are a number of available exporters[41] that are maintained as part of the official Prometheus GitHub. One such exporter used in PUD enabler by default is kube-state-metrics which is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual enablers' components, but rather on the health of the various objects inside, such as deployments, nodes and pods.



```
# HELP app_requests_current Current requests
# TYPE app_requests_current gauge
app_requests_current 33.0
# HELP app_requests_pending Pending requests
# TYPE app_requests_pending gauge
app_requests_pending 7.0
# HELP app_uptime Uptime
# TYPE app_uptime gauge
app_uptime 14383.0
# HELP app_health Health
# TYPE app_health gauge
app_health{app_health="healthy"} 1.0
app_health{app_health="unhealthy"} 0.0
```

```
{
  "current_requests": 27,
  "health": "healthy",
  "pending_requests": 8,
  "total_uptime": 14577
}
```

Figure 27. Typical metrics export process

The PUD enabler then forwards the pulled data to the LTSE, where it is stored on a time series database that implements a highly dimensional data model, where the time series entry is identified by a metric name and a set of key-value pairs. The PUD enabler also provides the capability of creating a series of dashboards in order to display the gathered metrics in a more configurable, consistent and user-friendly manner utilising Grafana for this cause. The architectural design of the PUD enabler is presented in Figure 28.

[37] https://www.elastic.co/es/beats/metricbeat
[38] https://github.com/google/cadvisor
[39] https://github.com/kubernetes/kube-state-metrics
[40] https://prometheus.io/
[41] https://prometheus.io/docs/instrumenting/exporters/

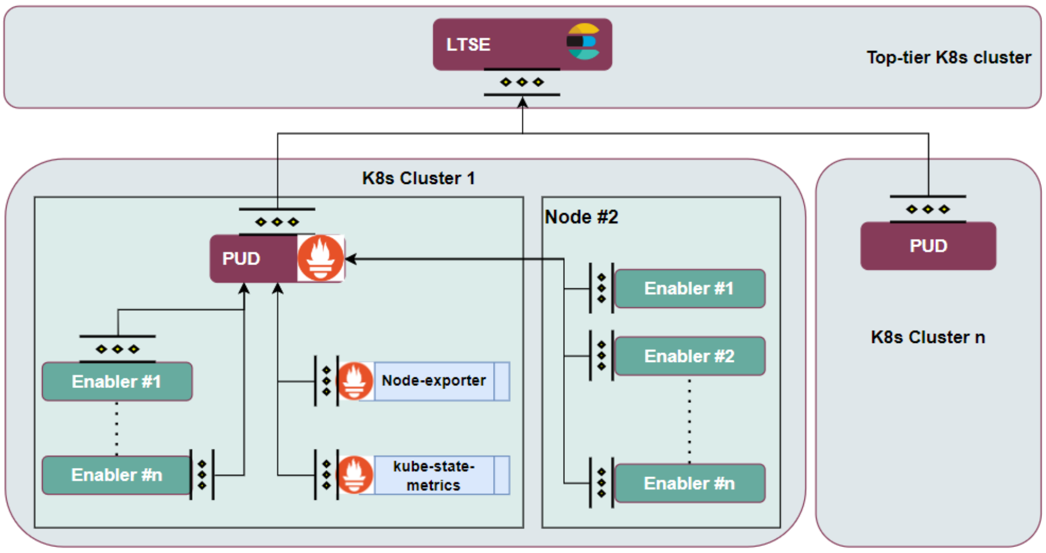


Figure 28. Example of realisation of a monitoring stack

In the case that a 3rd party application or custom-written software lacks an established exporter, a custom health metrics procedure can be established and be developed using common languages and client libraries, such as Python, Go, Java, etc. For example, in the case of MR enabler, where the K8s cannot be deployed, a custom health metrics add-on was developed, following the NGIoT principles. The methodology that should be followed is the following:

Custom-based software should be developed in order to store values and provide necessary endpoints to the connected devices (such as Microsoft HoloLens 2). On the other hand, the connected devices, with the use of API calls, provide their metrics to the custom-based software, which summaries them, before publishing them to the PUD. In order for the PUD to get the data and metrics acquired before, a Python exporter add-on should be developed, which formats the data in a meaningful way for PUD to retrieve them. The formatted data are then available in a defined endpoint, which can be accessed by any client.

In the case of Microsoft HoloLens 2, a set of device-related metrics was gathered that are related with the a) performance (CPU, GPU, memory, network), b) battery health and c) state, as follows:

1. Returns the system performance statistics "api/resourcemanager/systemperf"
2. Gets the current battery state "/api/power/battery"
3. Checks if the system is in a low power state: "/api/power/state"

5.3.2. Logging gathering considerations

The logging stack is in charge of gathering and storing the logs generated by all the enablers' components deployed in a given architecture realisation. This stack is key as it provides a centralised analysis of the system's information, easing debugging processes (technicians do not need to contact with K8s/platform administrator to get access to K8s' node logs nor manually navigate to component's logs one by one), achieving observability and accountability while avoiding issues related to local storage corruption, node crash or network failure.

Logging architecture

The high-level stack is depicted in Figure 29. In each node, a log gathering service (also referred to as "shipper") will be responsible for collecting all the logs coming from the standard output (*stdout*) and standard error (*stderr*) stream interfaces of the deployed components, and then it will send these logs along with relevant metadata (yet to be formalised, e.g., node id, cluster id, container id, enabler name, component name, timestamp, etc.) to a central location, where they will be stored within a persistence storage system.

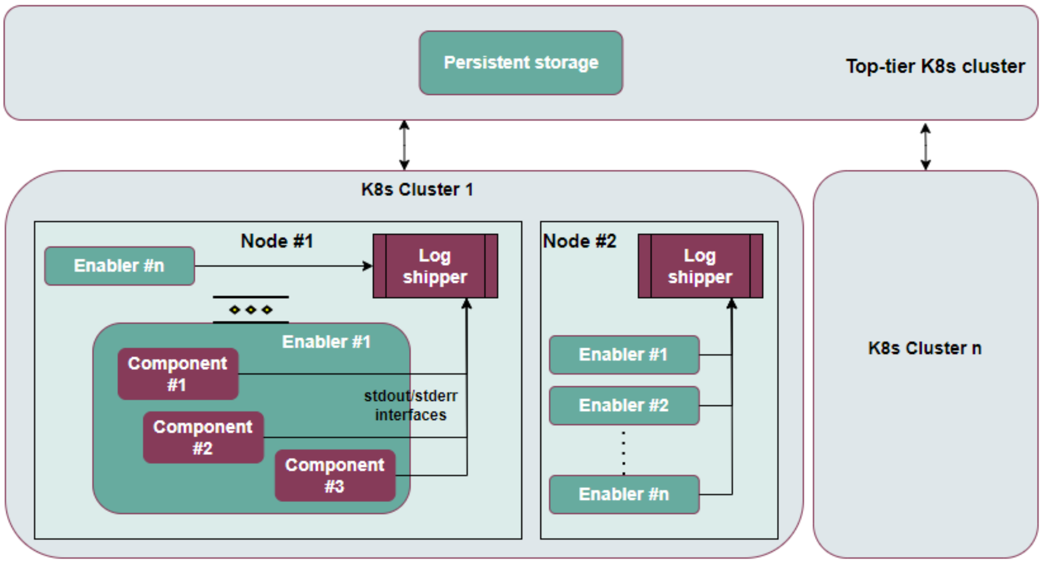


Figure 29. Logging stack concept

The RA does not force any particular logging stack. In a K8s-governed environment, the shipper can (and should) be deployed as a DaemonSet in all the clusters, i.e., as a workload that is executed in all the nodes of a cluster. There are different technologies for realising it, being Filebeat[42], Logstash[43] and fluentd[44] (and its lightweight version, fluentbit[45]) the ones standing out. Depending on the considered solution, logs can be forwarded into different alternatives; while fluentd is a technology that could send logs to several options (MongoDB, MySQL, Elasticsearch, as well as other services not devoted to storage), Filebeat is integrated just with Elasticsearch and Logstash (the three of them belong to the same stack, ELK). Logstash is not recommended as a decentralised shipper as it is relatively heavy to be run in all the nodes of a system.

After analysing the expertise of the project partners, Filebeat will be the option to be implemented in technical work packages, as it natively works with the LTSE to have the logs stored, having Logstash as a complementary tool (it can be deployed jointly with the LTSE in a central location if more advanced processing is needed, to transform, filter and/or aggregate them before ingesting it to the storage service). A possible realisation of the logging stack can be seen in Figure 30.

[42] https://www.elastic.co/es/beats/filebeat
[43] https://www.elastic.co/es/logstash/
[44] https://www.fluentd.org/
[45] https://fluentbit.io/

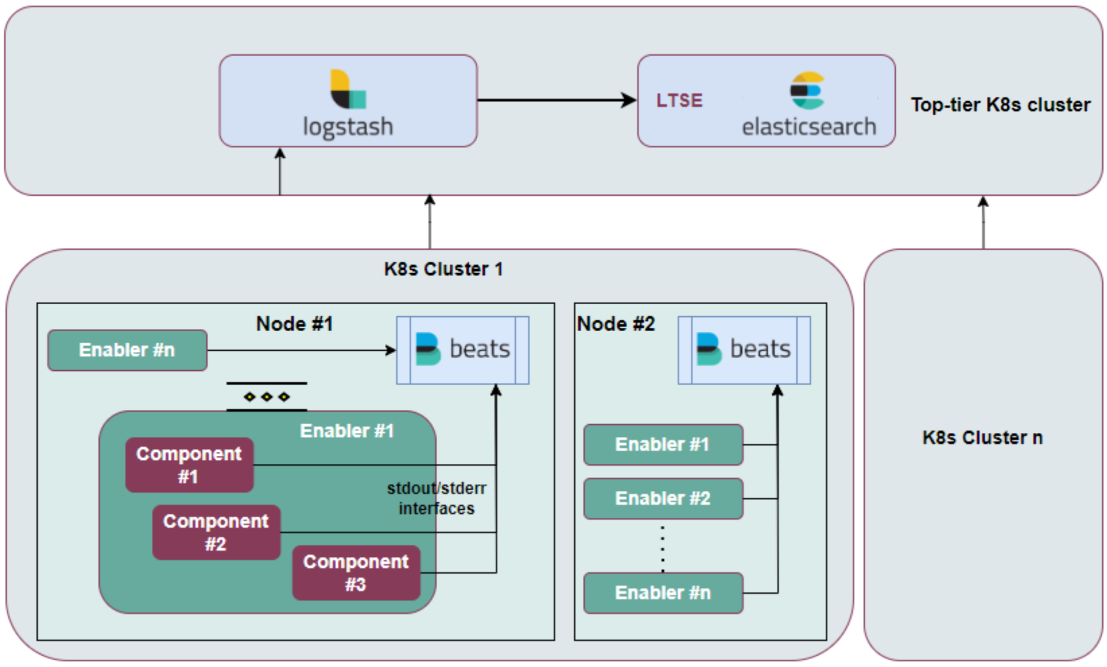


Figure 30. Example of realisation of a logging stack

5.4. Encapsulation exceptions

5.4.1. Hardware-specific exceptions

Any application that requires specific hardware (with restricted resources, without virtualisation possibilities, or vendor-locked to the used software, e.g., by licensing) can face difficulties in encapsulation. As the RA design principles aims at developing software that works in a virtual environment managed by a K8s distribution, it puts additional requirements that may be difficult to meet on a given hardware. One example of such issue has arisen when developing the Mixed Reality (MR) enabler in the scope of the technical work packages.

Novel interfaces, that are used in the MR enabler, offer human-centric interaction through better cooperation of end-users with the IoT environment. Through the MR enabler, human effort and decision can be introduced in the loop of every critical action, if and when needed. They can receive and provide tactile, real-time and visual feedback as well as data capable of identifying critical improvements, preventions and triggers in long-, short-term, or real-time.

In particular, the ASSIST-IoT MR enabler, through its interface (the Microsoft HoloLens 2[46] was chosen for the pilot implementation), consumes its processing power to visualise CAD-based models and interact with real-time data through wireless services. Also, in order to operate properly, any AR application requires direct access to device's hardware such as cameras, sensors (Head Tracking, Eye tracking, Depth and IMU) and specific features such as 6DoF tracking[47] and Spatial Mapping. Considering the above-mentioned energy-demanding processes, it is not feasible to install a fully-fledged virtual environment (such as K8s), as the orchestrator for deploying containerised services as it would reduce the efficiency and at the same time, running multiple replicas on one device will cause a variety of parallelisation-related problems. Apart from that, the HoloLens built-in software does not currently fit the encapsulation principles, as there is, at this point, no known (official or otherwise) K8s support for the Windows Holographic Operating System used by the HoloLens Devices. Summarising, the Microsoft HoloLens 2 is a low-resource node and, at the same time, an AR application already drains most of those resources. Containerising a service, even if it would technically be possible, it is not supported and would aggravate the mentioned resource usage problems.

[46] https://www.microsoft.com/es-es/hololens/hardware?SilentAuth=1&wa=wsignin1.0
[47] https://www.ar.rocks/glossary/6dof-tracking

In order to overcome the aforementioned obstacles, a potential workaround would be to use a low-consuming power that will provide a layer of data containing information on how the device will connect and communicate with the rest of the platform along with providing the required metrics about the status of the device. In such design, **the enabler itself would consist of a virtualised service to manage the MR device running non-encapsulated software**. In other words, the hardware itself would be just a component, and not the whole enabler. Such a solution fits, in general, any enabler with specific hardware requirements or constraints. However, this potentiality introduces a complication in the form of additional software (and some hardware node to run it), potential network and computing delays.

5.4.2. Virtualisation unfeasibility

Encapsulated environments, like containerised solutions, have multiple benefits, such as portability, agility, efficiency, flexibility, or fast delivery. But depending on the service to encapsulate, it may entail some issues also. For instance, containerisation has an ephemeral behaviour, that is, they are designed to exist only when needed. Therefore, stateful application activities are difficult to move to this scheme due to the fact that any data produced within the container disappears when the container ceases to exist. A clear example for this is the case of server-agent architectures where the agent requires a registering process in the server for completing the deployment. This registration would be lost if the agent container disappears.

Also, container runtimes usually work by isolating containers from the underlying host, and therefore some OS features such as cgroups, seccomp filters are restricted. In this way, the overall security is increased, among the containers that might be running on top of the host and the host itself. Containers with full access to the host features go against the nature of this virtualisation technology, so this can pose some issues for specific services.

5.4.2.1. Cybersecurity agents

Cybersecurity monitoring solutions based on server and agent architecture are broadly accepted as ideal solutions to collect, process, and analyse information and to provide an accurate and precise cybersecurity awareness over the deployed infrastructure. Agent monitoring solutions require to be linked to a server component as well as a registering process to this component. Due to the ephemeral behaviour of containerised solutions, the register of the agents to the server will be lost if the agent container disappears. Hence, it could be considered as an exception to run agents in an encapsulated environment, like a containerised solution. Also, security agent enabler might need to monitor host services and interfaces that might not reachable if deployed as a container.

The proposed solution for overcoming this exception is to deploy the cybersecurity monitoring agent as a service in the underlying host system running the containerised solution, and then evaluating the way to redirect the information that the agent needs to collect from the monitored components running encapsulated.

5.4.2.2. Underlying K8s self-healing

As described above, one of the exceptions for not embracing the encapsulation paradigm is the principle that container runtimes usually work by isolating containers from the underlying host, and therefore some, OS features such as cgroups, seccomp filters are restricted. This is the specific goal of the self-healing enabler, which aims at providing to IoT devices self-healing capabilities (i.e., actively attempting to recover themselves from abnormal states) by performing periodical monitoring status of the most relevant resource of its host OS (such as battery usage, memory access, network connection, or CPU usage). Therefore, since the self-healing enabler functionalities go completely against containerisation pillars, i.e., containers with full access to the host features are contrary to virtualisation technology nature, the self-healing enabler has been forced not to be encapsulated.

6. Conclusions

This document presents the final definition of ASSIST-IoT architecture, being the last of a series of three iterations, completing the Task 3.5 which is responsible for the technical and functional design of the ASSIST-IoT Reference Architecture. The presented architecture responds to the perspectives and objectives identified for the Next Generation IoT and it is based on the expertise of the technical partners of the project as well as on iterative learning process that started from the preparation of its initial version, defined in D3.5. Apart from sharing the Reference Architecture with the rest of the research community, this document also sets the ground for the concurrent and following actions of the project's execution; it feeds the technical developments being produced within the technical work packages (WP4 and WP5), outcomes that will be further tested and integrated (WP6), deployed in pilot-premises (WP7) and then evaluated (WP8) and disseminated (WP9).

After evolving and refining the design principles presented in D3.5 and updated in D3.6, D3.7 outlines the following main advances formalised over previous versions of the architecture:

- **Development view:** A new view has been included, aiming at providing guidelines and recommendations for designing enablers that follow ASSIST-IoT design guidelines.
- **Deployment view:** Aspects that should be considered during the development of enablers, including network, K8s-related, deployment as well as integration considerations have been refined.
- **Node view:** Specifications and best practices for preparing a node for Cloud-Native NGIoT ecosystems are depicted, presenting some implementation options considered in the project – but not mandated.
- **Security, Privacy and Trust:** In this version, some detailed considerations about security (in terms of authentication and access rights) in decentralised ecosystems, including OpenAPI and MQTT aspects, are given. Some considerations with Privacy, in terms of regulation aspects, and Trust, in the context of DLT, are also provided.
- **Scalability:** Design decision, that ensures the scalability of the application and developments not only in terms software but also hardware aspects, providing control and modification mechanisms to add, remove, and scale them across the layers.
- **Manageability:** The finalisation of the high-level management system of the overall ASSIST-IoT architecture that manages horizontally the enablers, assisting the human operators via tactile dashboards to detect and inform about faults, to configure the clusters and enablers parameters among other.
- **Essential Enablers:** The final selection of enablers that should be used in each Next Generation IoT application. Specifically, 1 out of 4 developed ASSIST-IoT enablers are essential in real scenarios and part of them will be **pre-installed** in any ASSIST-IoT deployment.
- **Monitoring and logging:** Strategies that should be used to monitor, collect and process monitoring and functional metrics of the platform as well as of other enablers have been refined, specifying the technological approaches followed in the development activities.
- **Encapsulation exceptions:** New cases where the encapsulation is not feasible due to virtualisation unfeasibility (cybersecurity and self-healing needs) have been reasoned.

This deliverable is the last one of the WP3 (M1-M21), whose main goal was to deliver a Next Generation IoT Reference Architecture with a degree of freedom in technological choices and deployment strategy based on market needs. During the execution of the WP3, state of the art and market analysis (T3.1) was performed to identify the most critical and upcoming technologies, that further were used to formulate the initial version of the architecture (D3.5). In the second iteration of the architecture (D3.6), the legal and regulatory requirements (T3.4) were also taken into consideration (reported in D3.4), highlighting technologies such Federated Learning that guarantee data privacy in decentralised, multi-company environments. Furthermore, the formulation of the use cases (T3.2) and pilot-specific or common requirements (reported in D3.2 and fine-tuned) were also used to improve the adaptability of the architecture, generating unique encapsulation constrains such as the blockage deployment of Kubernetes in special devices like MR devices. The final version of the ASSIST-IoT architecture (D3.7) was refined not only based on the final requirements and use cases (D3.3), but also on the continuous developers' guide and outputs that followed in the developments of WP4 and WP5. As a result, a reliable, scalable and adaptable Reference Architecture has been produced, that will be firstly evaluated in port, construction and automotive sectors and other sectors based on open-call proposals.

References

[1] ISO/IEC JTC, “Internet of Things (IoT) - Preliminary Report,” 2015.

[2] NGIoT Project, “D3.1. IoT research, innovation and deployment priorities in the EU,” 2020.

[3] M. Weyrich and C. Ebert, “Reference architectures for the internet of things,” *IEEE Softw.*, vol. 33, no. 1, pp. 112–116, Jan. 2016.

[4] CREATE-IoT Project, “D6.3. Assessment of convergence and interoperability in LSP platforms,” 2020.

[5] OpenFog Consortium, “OpenFog Reference Architecture for Fog Computing,” 2017.

[6] AIOTI WG Standardisation, “High Level Architecture (HLA) Release 5.0,” 2020.

[7] ASSIST-IoT Project, “D3.5 - ASSIST-IoT Architecture Definition – Initial,” 2020.

[8] C. M. Mackenzie, F. Mccabe, P. F. Brown, P. Net, R. Metz, and A. Hamilton, “Reference Model for Service Oriented Architecture 1.0,” 2006.

[9] ISO/IEC/IEEE 42010, “ISO/IEC/IEEE 42010 - Systems and software engineering - Architecture description,” 2011.

[10] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison Wesley, 2011.

[11] ETSI, “GS NFV-MAN 001 Network Functions Virtualisation (NFV); Management and Orchestration,” 2014.

[12] ASSIST-IoT Project, “D2.4 - Ethics and Privacy manual v2,” 2022.

[13] ASSIST-IoT Project, “D6.1 - ASSIST-IoT DevSecOps Methodology and tools,” 2021.

[14] R. Kumar and R. Goyal, “Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC),” *Comput. Secur.*, vol. 97, p. 101967, Oct. 2020.

[15] ASSIST-IoT Project, “D6.4 - Release and Distribution Plan - Initial,” 2022.

[16] P. B. Kruchten, “The 4+1 View Model of Architecture,” *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995.

[17] Q. Tu and M. W. Godfrey, “The build-time software architecture view,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 398–407, 2001.

[18] IBM, “An architectural blueprint for autonomic computing,” 2005.

[19] Hewlett Packard, “Internet of Things Research Study,” 2014.

[20] NIST, “Internet of Things (IoT) Trust Concerns,” 2018.

[21] TIoTA Alliance, “Trusted IoT Alliance Reference Architecture,” 2019.

[22] OASIS, “MQTT Version 3.1.1,” 2014.

[23] Alan Fekete, “CAP Theorem,” in *Encyclopedia of Database Systems*, Springer, New York, NY, 2018, pp. 395–396.

[24] ASSIST-IoT Project, “D5.3 - Transversal Enablers Development - Intermediate Version,” 2020.

[25] ASSIST-IoT Project, “D4.2 - Core Enablers Specification and Implementation,” 2022.

A. Visual instructions for documenting data pipelines

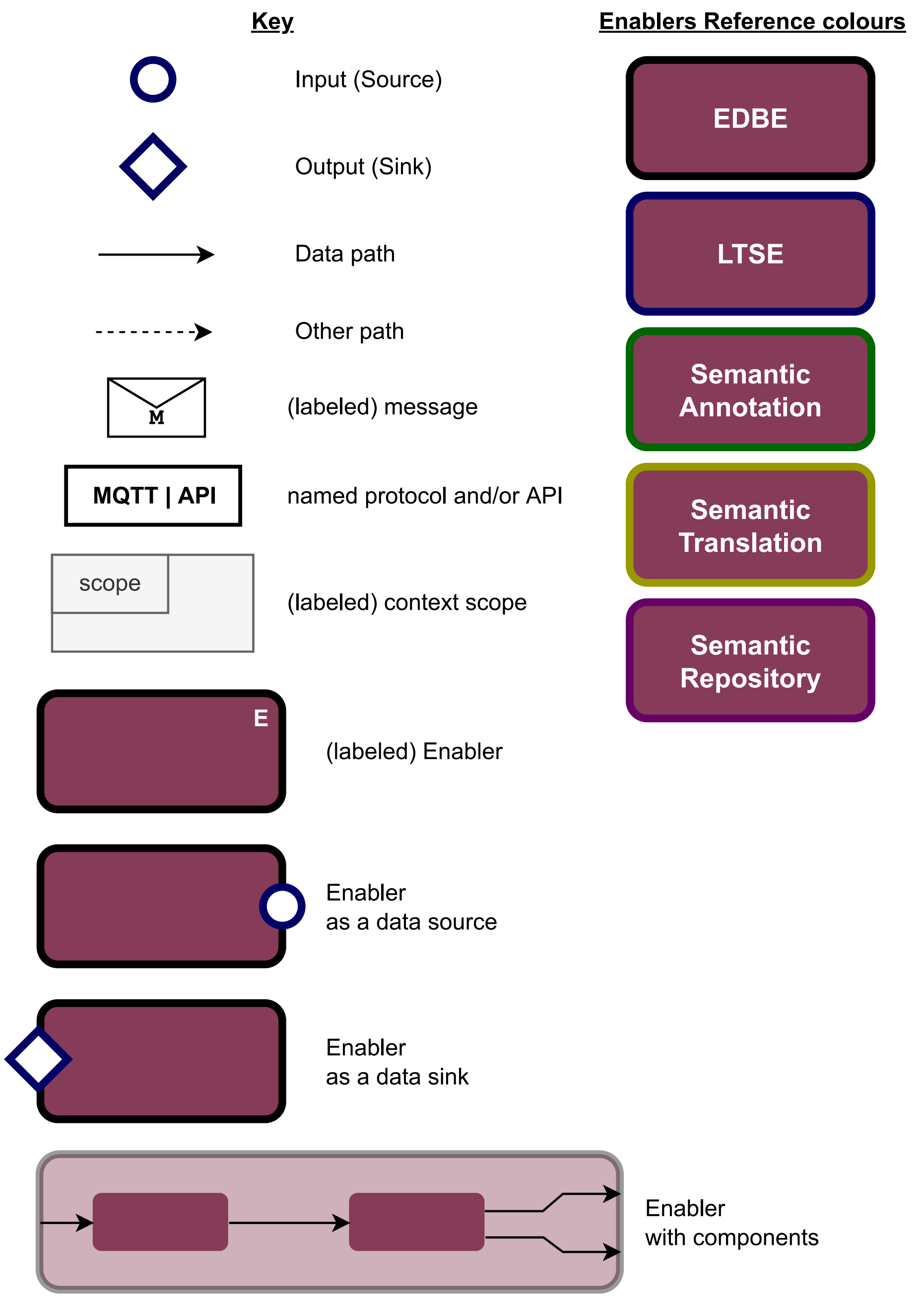


Figure 31. Data pipelines visual language key

- Note, that **colours** are optional, and serve only to visually differentiate and group diagram elements. If different colours are chosen, they should be distinguishable when converted to black-and-white. In general, importance is placed on shapes and relative position of the diagram elements, not on the colours.
- **Other paths** in a pipeline describe data that is important for operation of a pipeline, but is not generated at a source, or consumed at a sink. It is usually a piece of configuration, or additional information, that is required to process the data most relevant to the pipeline.
- Even though the visual language includes diagrams of **enablers with components**, those should be avoided and used only, if enabler internals are necessary to explain the flow of data.
- **Context scopes** are a simple way to group elements visually and have otherwise no usage restrictions or recommendations. An example is presented in Figure 32.

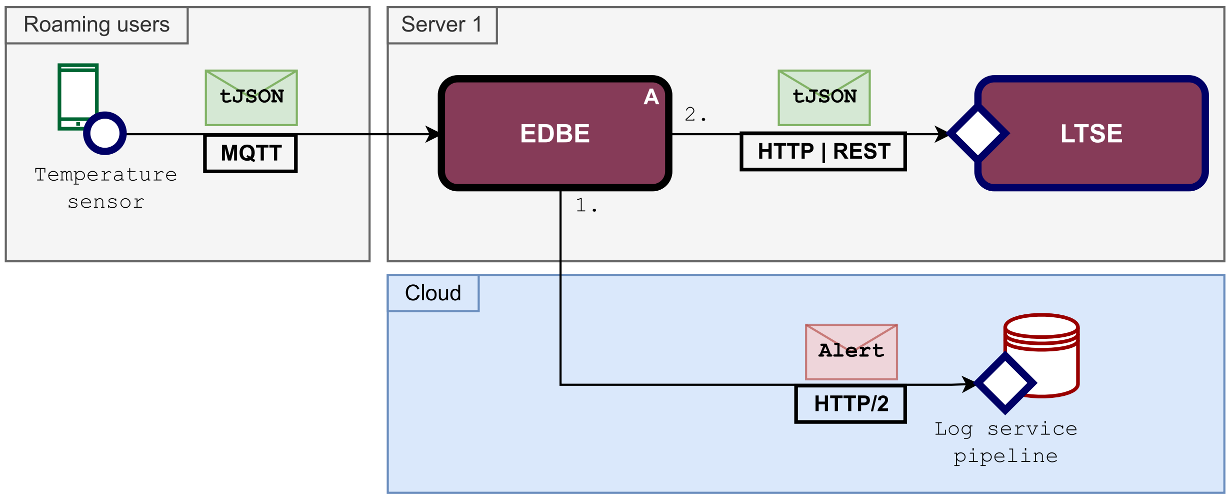


Figure 32. Data pipeline example with context scopes

B. ASSIST-IoT Reference Architecture summary

With IoT consolidated in several application domains, the Next Generation IoT (NGIoT) is evolving to address more ambitious and complex use cases. According to [2], the following technologies have been identified as key enablers for this evolution of IoT: Edge Computing, 5G (including NFV features), Artificial Intelligence and analytics, Augmented Reality and Tactile Internet, Digital Twins and Distributed Ledgers. As already happened with IoT, there is not a single architecture that can suit all the technical and non-technical needs of all the NGIoT realisations: there are simply too many topologies, access networks, protocols, devices, data types and technologies involved. Still, a set of requirements, guidelines and recommendations are necessary to guide its design, development and implementation, being that one of the main objectives of ASSIST-IoT: to envision a Reference Architecture for NGIoT, where a Cloud-Native paradigm adapted to the edge-cloud continuum will be considered as a baseline for its conception.

B.1. Conceptual approach

The conceptual, multidimensional and decentralised architecture of ASSIST-IoT, is separated in horizontal Planes and Verticals (see Figure 33). Planes depict functionalities that can be logically grouped together, whereas Verticals represent properties, cross-cutting concerns that either needs cooperation among elements of more than one plane or that can exist in different planes, in an independent manner. Four planes are part of the architecture, namely:

- **Device and edge:** it comprises the physical elements that support an architecture realisation, from servers and edge nodes to IoT devices, sensors, and network hardware. It also encompasses those functionalities required to either perform local intelligent analysis/actions, or to pre-process and lift data to services from upper Planes.
- **Smart network and control:** The key functions handled on this plane are encompassed by technologies that deliver software-related and virtualised networks (e.g., SD-WAN, NFV, MANO). Features, such as dynamic network configuration, tunnelling, routing and load-balancing are provided by this plane.
- **Data management:** groups functionalities related to data, from acquisition to routing, fusion, aggregation, transformation, and storage.
- **Application and services:** related to functions to be consumed by end-users, administrators and/or external systems. It makes use of functions from lower Planes (and also Verticals) to offer high-value applications for stakeholders.

Regarding Verticals that play supportive role on the functionalities of the horizontal planes, they consider both inherent properties of the system and enablers to include capabilities belonging to them:

- **Self-*:** property of a system related to its autonomy or semi-autonomy, comprising specific operations that do not require human intervention (self-healing, self-configuration, self-awareness, self-organisation, etc.).
- **Interoperability:** property that ensures that, at hardware level, equipment from different manufactures can communicate within a deployment, and at software level, services can share data thanks to the use of common formats or protocols, or the use of dedicated tools.
- **Security, privacy and trust:** Set of properties of the architecture related to integrity and access restriction of data, as well as guarding against malicious threats, among others.
- **Scalability:** trait to ensure proper system performance and dedication of resources in case of change of operational or business conditions or requirements. It comprises not just software, but also hardware and communication dimensions.
- **Manageability:** related to the control of the lifecycle of the functions of other Planes and Verticals, from their instantiation and configuration to termination. It also comprises the management of devices and coordination of workflows.

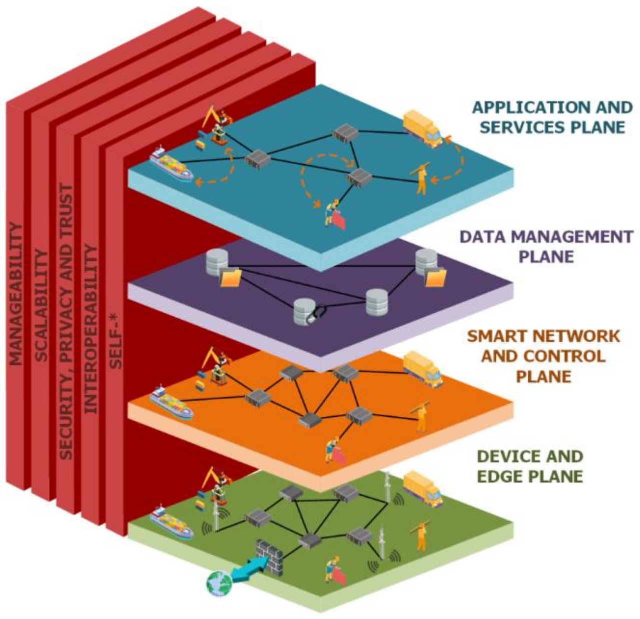

APPLICATION AND SERVICES PLANE
DATA MANAGEMENT PLANE
SMART NETWORK AND CONTROL PLANE
DEVICE AND EDGE PLANE
MANAGEABILITY
SCALABILITY
SECURITY, PRIVACY AND TRUST
INTEROPERABILITY
SELF-*

Figure 33. ASSIST-IoT Conceptual Architecture

B.2. Key design principles

The design principles of the ASSIST-IoT RA have been considered following the pillars of (i) decentralisation, (ii) scalability, (iii) software lightness and modularity, (iv) adoption potential and (v) production-readiness. IoT and edge ecosystems are not like cloud; computing resources are lower, less stationary (they might be increased, decreased or reallocated) and might be geographically scattered. Hence, it is mandatory that the selected principles take into account the possibility of having different types of system topologies and are flexible enough ahead of modifications in terms of hardware resources and usage.

ASSIST-IoT proposes a reference architecture governed by the following key principles and design decisions: (i) a **microservices-based** software architecture. Given the large number of technologies and features that can be used, keeping software in independent modules that can be later interconnected facilitates their maintenance and their use only when necessary; (ii) instantiation of these service in **containers**, hence ensuring lightness and flexibility; (iii) introduction of the abstract concept of **enabler**, which is a collection of software components running on computation nodes that work together to deliver a specific functionality of a system; and (iv) **Kubernetes** as the suggested underlying technology for orchestrating enablers, bringing a set of production-readiness advantages. Although this last kind of imposition is not usually mentioned in RAs, it enables providing more useful guidelines and recommendations for actual realisation, reducing ambiguity and facilitating its use.

B.2.1. Enablers

In this RA, the term **enabler** is a conceptual abstraction that represents a collection of interconnected micro-applications (that can be microservices), deployed as components, that jointly provide a functionality for the system, and hence should fall under one of the specified Planes or Verticals. The rationale behind bringing this concept is mostly related to modularity, which entails that only those enablers required for addressing a specific scenario need to be considered. They also allow having a conceptually more compacted view of the features available in the system, similarly to what happens in a SOA architecture, while having the benefits of working similarly to as with microservices. In addition, realising functionalities as enablers has some advantages for development and maintenance.

Enablers are not atomic but presented as a set of interconnected components. An enabler component is a software artifact (which might consider specific hardware components) that can be viewed as an internal part of an enabler, and that performs some action necessary to deliver the functionality of an enabler as a whole. Each enabler must define and control communication between its components, and any language, technology or communication interface can be considered for internal enabler designs, allowing flexible implementations. Each enabler provides a single point of entry (interface) to communicate with it, without exposing the internal communication mechanisms between its components, thus having an *encapsulation* of components.

B.3. Enablers from the functional view and Verticals

The architecture is composed of a **set of views** that address different aspects from the perspective of specific system concerns; here the Functional view (Figure 34) and the Verticals are presented as they specify the core and transversal features provided (or to that should be provided) by the architecture in the form of enablers, still additional information about computing nodes, enablers' development, system and enablers' deployment and data can be found in the respective views in the core report, documented in the project's wiki[48].

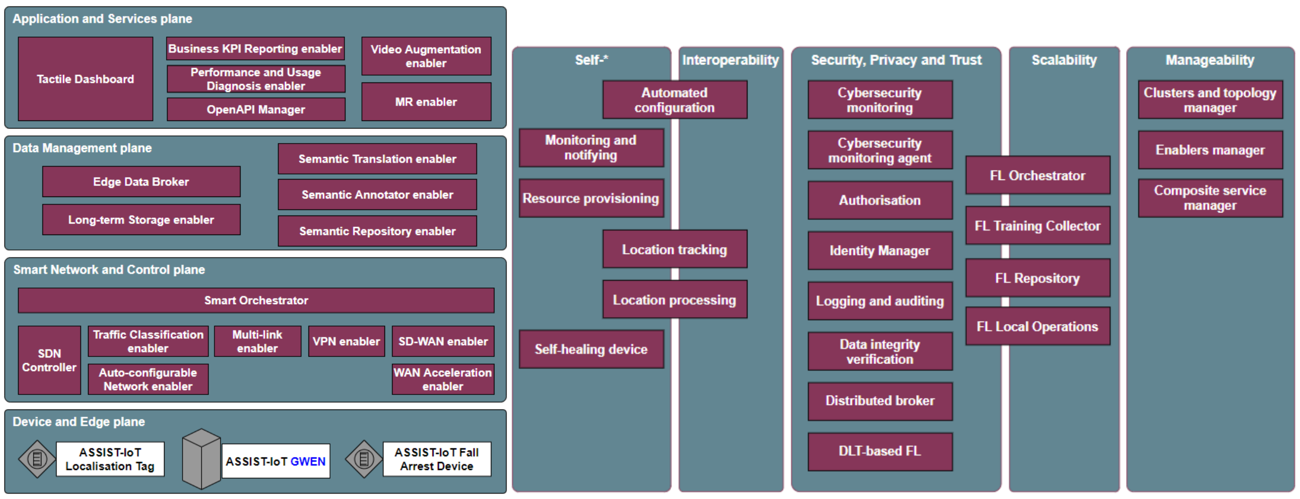


Figure 34. Functional view and Verticals - summary of enablers

B.4. Integration Perspective

This section describes the different principles that should be followed in order to either develop new enablers, or integrate your system with the ASSIST-IoT architecture.

B.4.1. Enabler development for ASSIST-IoT (encapsulated software)

To start building a new software based on enablers' concepts, the principles described in the previous sections should be followed. This means that your enabler should act as part of the ASSIST-IoT architecture and use the other enablers in order to retrieve/store/use/transmit data through the Data management plane (or other planes, depending on your needs). Also, as applications will be controlled by a platform interface, then your enabler should support two types of endpoints (which might be realised as a single exposed API): endpoints responsible of interacting with the other enablers, and endpoints that allow interaction with end-users and/or admins so they can use your enabler in a user-friendly way, without hardcoding. Thus, the following steps should be considered:

1. Design your enabler, considering the main principles of the architecture: decomposing it as a set of components; to be implemented as containers; defining their internal communication; exposing one (or various) interfaces, generally APIs, to be integrated with other enablers or consumed by users; assess data privacy/ethical regulation constraints; decide technologies. Detailed information given in the Development view of the core architecture document.
2. Implement the basic functionalities of your components (in case they cannot be taken from existing artifacts available in the research community).
3. Integrate internally the different components in order to build a service.
4. Implement the interactions with mandatory and optional enablers, if needed. Your enabler should define the features exposed so they can be accessed through its main interface.
5. Build the required Docker images to deploy all the above. During early stages, they can be integrated considering (e.g.) Docker Compose before moving to a Kubernetes environment.
6. Create a Helm chart to package the enabler as a whole so it can be deployed in Kubernetes clusters with the Smart orchestrator. A chart generator is provided in the project to facilitate this process.

[48] https://assist-iot-enablers-documentation.readthedocs.io/en/latest/index.html

B.4.2. Software to be executed on ASSIST-IoT architecture

In case that your enabler is deployed in an ASSIST-IoT platform managed by Kubernetes, four endpoints should be part of the exposed API:

- **/health**: this endpoint is responsible for collecting and providing information about the liveness status of the internal of the components, without the need of the platform administrator to interact with any K8s console. It is suggested to make use of status codes, providing in case of error a response indicating the unready or failed component.
- **/metrics**: this endpoint provides performance metrics relevant to the particular enabler reflecting its current status. It will be scrapped by a Prometheus server, so data should be presented in a compatible format.
- **/api-export**: It should provide a list of methods and endpoints that can be served by the enabler, in accordance with the OpenAPI specifications[49]. The format will be in typical JSON or YAML format. Documentation and examples should be included. Two sub-methods are expected:
  - **/openapi**: it should return the OpenAPI specification in JSON format. Users will be able to download the API specification.
  - /**docs**: is should return an OpenAPI documentation provided by Swagger UI. It lets users visualise and interact with the APIs' resources, which simplifies backend implementation and client-side consumption.
- **/version** (GET method): This endpoint returns the current version and patch of the enabler. The versioning format should follow the Semantic Versioning Specification (SemVer[50]).

In the case your software is expected to **interact with services that are external to the ASSIST-IoT deployment** (either because they need specific data from them, or it is sharing its data with them), its exposed interface should be registered in the Open API enabler, and hence the related endpoint must be implemented. The OpenAPI is considered an essential enabler, meaning that it will be present in any ASSIST-IoT deployment. It consists in an API manager that exposes a Swagger UI through which external software can interface with the ASSIST-IoT internal enablers to be queried/managed, when needed (in Swagger-JSON format).

To use this API, the user must subscribe to the API subscription (via Swagger GUI) as "ASSIST-IoT External users". Thereafter, every enabler should decide which methods are exposed to external users. The OpenAPI is also thought to be a door for accessing documentation, tutorials, sample code, software development kits, etc. It also allows to manage subscription keys and obtain support from the API provider, if needed. The internal diagram of the OpenAPI enabler of ASSIST-IoT (just for applicants' information) is as follows:

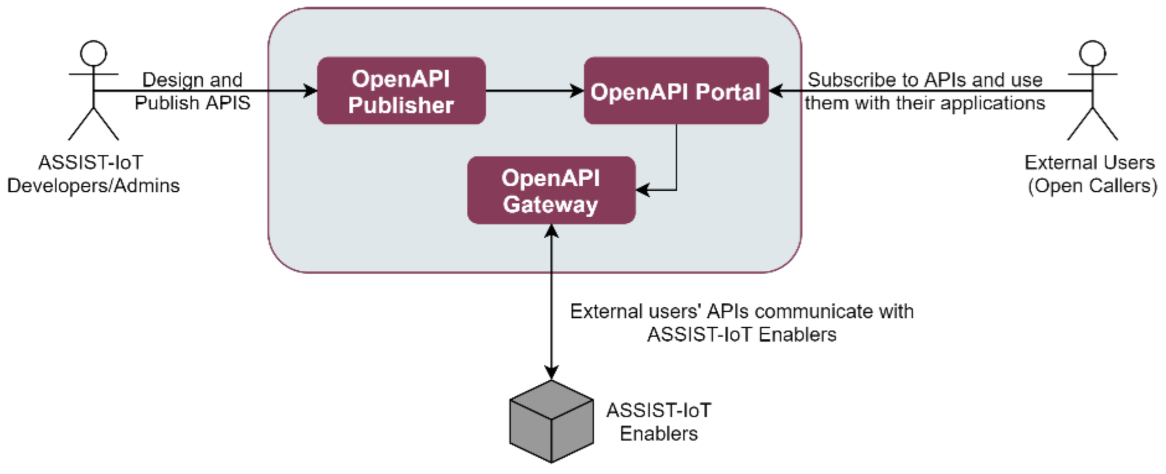

Design and Publish APIS
ASSIST-IoT Developers/Admins
OpenAPI Publisher
OpenAPI Portal
OpenAPI Gateway
Subscribe to APIs and use them with their applications
External Users (Open Callers)
External users' APIs communicate with ASSIST-IoT Enablers
ASSIST-IoT Enablers

Figure 35. OpenAPI enabler diagram

[49] https://swagger.io/specification/#:~:text=Introduction,or%20through%20network%20traffic%20inspection.
[50] https://semver.org

B.4.3. Integration examples

Integrate external software component

An external mobile-based application is used to visualise data produced by ASSIST-IoT devices installed in (e.g.) the port pilot. In this scenario, the ASSIST-IoT device controls a crane within the port, which means the data is transmitted to the Long-Term Storage enabler (LTSE) for further usage. So, the external software should be interfaced with the LTSE via the OpenAPI Manager enabler, in order to receive the data in their infrastructures.

Deploy IoT sensors

In case that an application is needed to deploy new IoT sensors and associated equipment into the infrastructure of one pilot, it is encouraged that they support MQTT protocol in order to transmit the data to the Edge Data Broker, which is deployed on ASSIT-IoT gateways. For this scenario, (i) MQTT security considerations should be consulted in order to grant access to the ASSIST-IoT architecture, especially if the environment is not considered secure or trusted, and (ii) the VPN enabler should be used in case the device belong to an external network.

Deploy computing nodes

In case that a node (including gateways) is selected to operate in a deployment, it should be prepared to act as a master/node in a Kubernetes cluster. In addition, they should incorporate all the needed services at host OS level in order to transmit the data to the system through Smart network plane. Specifically, the host must have installed:

- Linux Operative System (recommended Ubuntu 16 and beyond). Other distributions are also accepted as long as they are able to run the next requisites.
- Docker as virtualisation technology.
- Kubernetes as container orchestration framework (kubeadm and K3s strongly recommended, as they have been validated).
- Helm3 as packaging and deployment manager.
- Cilium as Container Network Interface (CNI) plugin.
- OpenEBS as container storage technology.

A script has been prepared for installing kubeadm distribution and the aforementioned engines and plugins on top of a Linux device (K3s version, better for devices with constraint resources, still pending at this moment).

Develop AI/ML services

In the case that you want to deploy Machine Learning models under the auspicious of Federated Learning or develop new mechanism for AI training, then you should also leverage the OpenAPI enabler to consume the Federated Learning enablers or interconnect your AI-based service with ASSIST-IoT architecture. In case that you develop new enablers, then the principles described in section B.4.1 apply.

Concerning to data for training, the main scenario involves using data gathered via ASSIST-IoT devices and stored in the LTSE. Otherwise, you can store your own dataset to the LTSE in order to train them using ASSIST-IoT enablers. In both cases, the data will be received or transmitted to the ASSIST-IoT storage system through the OpenAPI enabler.