



D4.1 CPS-IoT Resource Management System Specification

Deliverable No.	D4.1	Due Date	28/02/2022
Description	Analysis of medical and technical requirements and on the specification of software modules embedded into the ODIN CPS-IoT RMS		
Type	Report	Dissemination Level	PU
Work Package No.	WP4	Work Package Title	CPS-IoT Resource Management System Specification
Version	1.0	Status	Final Version



Authors

Name and surname	Partner name	e-mail
Pablo Lombillo	MYS	plombillo@mysphera.com
Pilar Sala	MYS	psala@mysphera.com
Alejandro Medrano	UPM	amedrano@lst.tfo.upm.es
Eugenio Gaeta	UPM	Eugenio.gaeta@upm.es
Luis Carrascal	INETUM	luis.carrascal@inetum.com
Antonio Gamito	INETUM	antonio-jesus.gamito@inetum.com
Ilias Kalamaras	CERTH	Ilias.kalamar@iti.gr

History

Date	Version	Change
30/07/2021	0.1	Initial TOC
09/09/2021	0.2	TOC after Alejandro Medrano (UPM) comments Added Requirement Management & tools section Added Architectures and solutions for more components and categories Added glossary section
21/09/2021	0.3	Added new requirements sections and content Added sections regarding KPI architectures and solutions
10/11/2021	0.4	Added content to Enterprise Architecture Patterns
14/11/2021	0.5	Added Event Driven Architecture content
19/11/2021	0.6	Added Enterprise Service Bus content. Added IoT Architecture content. Included requirements for ODIN, Security, Resource Manager. Removed section 6.6.1 Web Of things Architecture.
09/02/2022	0.7	Removed Section 5.2 Use case requirement as most of the requirements are out of the scope of WP4 work. The requirements will come from the layers and the platform itself.

		Updated requirements Added content to section 6.6 Added content to section 7 ODIN's CPS Architecture
	0.8	Merged content for DLT requirements and DLT Architecture section Added more test examples for the architecture to section 7 Added more content to requirements sections
17/02/2022	0.9	Added resource requirements
28/02/2022	0.10	Added missing content Prepared for peer review
13/04/2022	0.11	First review changes Homogenization of requirements
27/04/2022	0.12	Peer review modifications
02/05/2022	1.0	Final version

Key data

Keywords	IoT, resources, robot, gateway, integration, platform, layer, API, messaging, web service, architecture, components, ...
Lead Editor	Pablo Lombillo (MYS)
Internal Reviewer(s)	Alejandro Medrano (UPM), Daphne Plati (FORTH)

Abstract

D4.1 discusses medical and technical requirements to meet ODIN objectives and vision, which is integrating a set key enabling resources into one homogeneous platform to deliver services to solve hospitals problems. Before diving into the requirements, the methodology used to manage the requirements is explained. Requirement section leads to an enumeration to the analysis of the best software architecture and components that must be used in the Platform solution to fulfil those requirements. After this analysis, several proposals of the architecture to be used in CPS-IoT is presented in an iterative way. Finally, a validation of the architecture is performed using several uses cases where the architecture must succeed.

Statement of originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation, or both.

Table of contents

TABLE OF CONTENTS	5
LIST OF FIGURES.....	7
LIST OF TABLES	8
1 ABOUT THIS DELIVERABLE	9
1.1 DELIVERABLE CONTEXT	9
2 ODIN'S VISION AND WP3 ARCHITECTURE.....	11
2.1 WP3 ODIN'S ARCHITECTURE.....	13
3 REQUIREMENT MANAGEMENT & TOOLS	14
3.1 REQUIREMENT GATHERING	14
3.2 REQUIREMENT DOCUMENTATION	14
3.2.1 <i>Volere requirement template</i>	14
3.2.2 <i>ODIN's Volere criterion</i>	15
3.3 REQUIREMENT VALIDATION.....	16
4 REQUIREMENTS & FUNCTIONALITIES	17
4.1 ODIN CPS-IOT LAYER REQUIREMENTS.....	17
4.2 RESOURCE LAYER REQUIREMENTS.....	21
4.3 EXTENSIBLE SPECIALIZED AI AND HIGH AI REQUIREMENTS	22
4.4 HOSPITAL KNOWLEDGE, COGNITION & PERCEPTION CYCLES REQUIREMENTS	23
4.5 PRIVACY, SECURITY & TRUST REQUIREMENTS	27
4.6 RESOURCE GATEWAY REQUIREMENTS	27
4.7 DLT RESOURCE FEDERATION REQUIREMENTS	31
4.8 RESOURCE CHOREOGRAPHER REQUIREMENTS	41
4.9 KPI COLLECTION REQUIREMENTS	45
5 TECHNOLOGY AND ARCHITECTURE CONCEPTS.....	51
5.1 SYSTEM AND SOFTWARE ARCHITECTURE DEFINITION.....	51
5.2 ENTERPRISE INTEGRATION PATTERNS.....	51
5.2.1 <i>Integration styles</i>	52
5.2.2 <i>The Messaging concepts</i>	53
5.2.3 <i>Channel</i>	53
5.2.4 <i>Message</i>	54
5.2.5 <i>Router</i>	55
5.2.6 <i>Transformation</i>	56
5.2.7 <i>Messaging Endpoint</i>	56
5.2.8 <i>System Management</i>	57
5.2.9 <i>A simple method to apply patterns</i>	58
5.3 MICRO-SERVICE ARCHITECTURE	61
5.3.1 <i>Architecture components</i>	61

5.3.2	<i>How it works</i>	62
5.3.3	<i>Pros and Cons</i>	63
5.4	ENTERPRISE SERVICE BUS ARCHITECTURE.....	64
5.4.1	<i>Architecture components</i>	65
5.4.2	<i>How it works</i>	66
5.4.3	<i>Pros and Cons</i>	68
5.5	RESOURCE GATEWAY AND RESOURCE ARCHITECTURES	69
5.5.1	<i>Architecture components</i>	71
5.5.2	<i>How it works?</i>	72
5.5.3	<i>Pros and Cons</i>	73
5.6	DISTRIBUTED LEDGER TECHNOLOGIES ARCHITECTURES	74
5.6.1	<i>Architecture components</i>	74
5.6.2	<i>How it works</i>	75
5.6.3	<i>Pros and Cons</i>	76
5.7	SOFTWARE CHOREOGRAPHY ARCHITECTURES.....	77
5.7.1	<i>Architecture components</i>	77
5.7.2	<i>How it works?</i>	78
5.7.3	<i>Pros and Cons</i>	79
5.8	KPI COLLECTION SYSTEM ARCHITECTURE.....	80
5.8.1	<i>Architecture components</i>	80
5.8.2	<i>How it works?</i>	81
5.8.3	<i>Pros and Cons</i>	81
5.9	FINAL DECISIONS	82
6	ODIN'S CPS ARCHITECTURE	83
7	ARCHITECTURE VALIDATION	85
7.1.1	<i>Example 1: SERMAS use case</i>	85
7.1.2	<i>Example 2: AI data exploitation</i>	87
7.1.3	<i>Example 3 Resource provisioning</i>	88
8	CONCLUSIONS AND NEXT STEPS	89
APPENDIX A	GLOSSARY	90
APPENDIX B	USE CASES	92

List of Figures

FIGURE 1: ODIN'S ARCHITECTURE INITIAL CONCEPT	11
FIGURE 2 ODIN DATA INFORMATION FLOWS	12
FIGURE 3 ODIN'S FINAL ARCHITECTURE	13
FIGURE 4 VOLERE TEMPLATE.....	15
FIGURE 5 CHANNEL EXAMPLE.....	53
FIGURE 6 HUB AND SPOKE.....	65
FIGURE 7 ESB ARCHITECTURE COMPONENTS.....	65
FIGURE 8 ESB WORKING EXAMPLE	67
FIGURE 9 IOT LAYERED ARCHITECTURES	70
FIGURE 10 IOT TOPOLOGIES	71
FIGURE 11 IOT ARCHITECTURE EXAMPLE.....	72
FIGURE 12: POSITIONING OF DLT RESOURCE FEDERATION WITHIN THE ODIN PLATFORM ARCHITECTURE.	75
FIGURE 13: ORGANIZATIONAL CONSENT MANAGEMENT.....	76
FIGURE 14: RESOURCE SHARING AND TRANSACTION AUDITING.....	76
FIGURE 15 ORCHESTRATION	77
FIGURE 16 BPEL EXAMPLE DIAGRAM AND CODE.....	78
FIGURE 17 KPI COLLECTION SYSTEM ARCHITECTURE	80
FIGURE 18 PART OF ODIN'S ARCHITECTURE WITH FOCUS ON WP4 COMPONENTS	84
FIGURE 19 ARCHITECTURE TEST	86
FIGURE 20 AI ARCHITECTURE TEST	87
FIGURE 21 RESOURCE PROVISION TEST	88

List of tables

TABLE 1 ROUTER TYPES	55
TABLE 2 TRANSFORM OPERATORS	56
TABLE 3 MESSAGING ENDPOINT TYPES.....	57
TABLE 4 SYSTEM MANAGEMENT COMPONENTS	58
TABLE 5 APPLYING PATTERNS METHOD.....	60

1 About this deliverable

This deliverable studies and analyses in depth the medical and technical requirements gathered mainly from the objectives of Document of Action (DoA), the work performed in “WP2 DEMAND AND SUPPLY COCREATION: Reference UC and Catalogue Specifications”, and from “WP5 Extensible Specialized AI: AI-Robotics cognition, connecting resource and tools”, “WP6 - High Level Ecosystem for AI Operations”, “WP7 ODIN Pilots Design, Deployment, Evaluation and Validation”, to define the best software architecture and components to deploy ODIN’s platform. More concretely, defines the Resource Management System which connects the sources of data with the rest of the ODIN’s Platform layers.

The document first reviews the vision of the project and follows with the methodology to gather and work the requirements.

Then, the sources of requirements and the requirements is listed, with a mapping into the different components of the CPS.

Following the requirements, a light review of the technology and architectures available is performed to discuss the best matching to fulfil the requirements.

Finally, the solutions selected, and the designed architecture is presented.

1.1 Deliverable context

The following table sets the context of this deliverable

PROJECT ITEM	RELATIONSHIP
Objectives	<p>The deliverable is relevant to ODIN's Objective 1, as it describes and defines the software architecture of the ODIN platform to cover medical and technological requirements.</p> <p>The WP4 objectives are:</p> <ul style="list-style-type: none"> • Specification of the CPS-IoT RMS requirements based on input from WP2 • Definition and development of the Resource Descriptor module • Specification and development of the Resource Gateway module • Specification and development of DLT Resource Federation and management framework • Specification and development of Resource Choreographer module • Specification of KPI and metrics collection framework
Exploitable results	<p>The analysis of the solutions to fulfil requirements will be a very valuable know how. Moreover, the proposed architecture is an exploitable result that will be used in the whole platform. This architecture could conform a new trending way to work at the hospital system integration.</p>
Workplan	<p>D4.1 is attributed to the tasks of WP4, WP4 - CPS-IoT Resource Management System [Months: 3-42] MYS, CERTH, UPM, IECISA. The task T4.1 CPS-IoT Resource Management System specification (MYS) [M3-M12] is the main responsible of this deliverable.</p>

Milestones	The D4.1 is a must for the milestones PREPARATION (MS1) and IMPLEMENTATION (MS3) phases of the project.		
Deliverables	D4.1	1CPS-IoT Resource Management System Specification	
	D4.2 to D4.4	Implementation of Local CPS-IoT RSM Features v1 to v3	Regarding Resource descriptor, Resource Gateway and measurement components
	D4.5 to D4-7	Implementation of Advanced CPS-IoT RSM Features v1 to v3	Regarding functionalities of DLT and Resource Choreographer
	D3.2 to D3.3	Hospital Knowledge Base and ODIN semantic ontology v	Odin ontology
	D3.4 to D3.6	Privacy Security and Trust report	
	D7.2 – D7.7	KPI Evolution Report (I to IX)	Regarding the collection of KPIs about DevOps activities.
Risks	D7.9	Pilot Studies Evaluation Results and sustainability	Regarding component evaluation results of unit/integration testing.
	<p>The guidelines provided in this deliverable can help in minimizing the following risks identified in the Grant Agreement:</p> <ul style="list-style-type: none"> • Technologies not available in time • Technical problems during component/module development • Complexity of unification procedure 		

2 ODIN's vision and WP3 Architecture

Inside ODIN's Platform it will exist several heterogeneous systems with its own purposes and needs but all must push together to work in a decoupled but organized way to achieve ODIN's and Pilot's objectives.

ODIN's concept architecture handles several layers that encapsulate and publish the right information and functionality for the rest of the system.

Moreover, WP4 is aligned with WP3 mission, regarding the coordination of horizontal activities as well as implementation of vertical components in the platform. Within this scope, WP4 provides the means for integration of all ODIN's components.

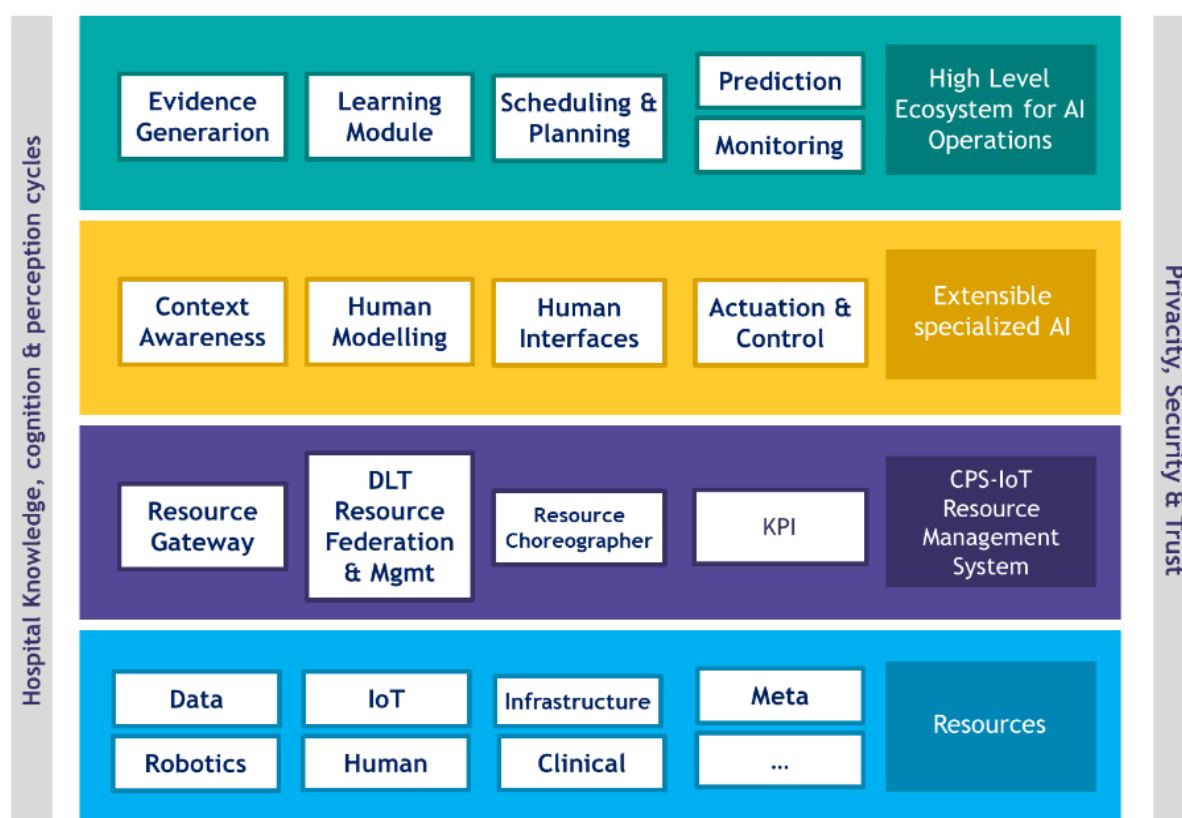


Figure 1 ODIN's architecture initial concept

As a summary, the “Resource layer” on the bottom of the previous figure, conforms all the resource data, robotic, human, clinical and infrastructure data so it's available for the upper layer in order to take decisions. This layer also supports functions to be taken autonomously or indicated by the upper layers.

The “Resource management System” hides Resource layer complexity and exposes it to the upper layers in a secure and efficient way. Moreover, it handles federation of data and resources, and controls most important KPI that ODIN Platform must handle at this level.

The “Extensible specialized AI” is the first application layer with business logic intelligence, providing the brain infrastructure to the Platform to understand the resources situation such as location, status and so on in terms of Context Awareness. It also handles the tasks the resources must perform under the Actuation and Control module and understands Human status and provides Human interfaces to interact with the system.

In the top level, “High Level Ecosystem for AI operations” introduce the components for providing learning capabilities, data management and data sharing, data-model creation, data-model creation and High-level AI support to the ODIN project, based on the Extensible specialized AI layer.

The supporting layers are the Hospital Knowledge (HK), cognition & perception cycles, and the Privacy, Security & Trust Layer (PST). The HK provides the common data model for the platform from the clinical and organizational point of view, through HL7/FIHR standards, while the PST manage identity and authentication services for the whole platform and ensure the cybersecurity of the platform through monitorization.

Beyond the layers, the platform looks for enabling the resources to be able to use any other resource available, horizontally, or vertically inside the platform.

In the following picture it can be reviewed how the data can flow between the different layers and modules, retrofitting where needed. A little example will ease the figure reading.

At the resource layer, a Location sensor can send the position of another resource, a robot for example, to the CPS-IoT Resource MGMT system, where the Location sensor's information is traduced to the platform language. The services form Extensible AI or High-level AI interested in the robot's position, can read the information through the Resource Gateway. For example, the Context Awareness is one of the services that could use the information to trace the best rout the robot must follow to find a patient. Thus, once the Context Awareness computes the route, it can publish it to the Resource Gateway, so the robot gets the information on time to start the route.

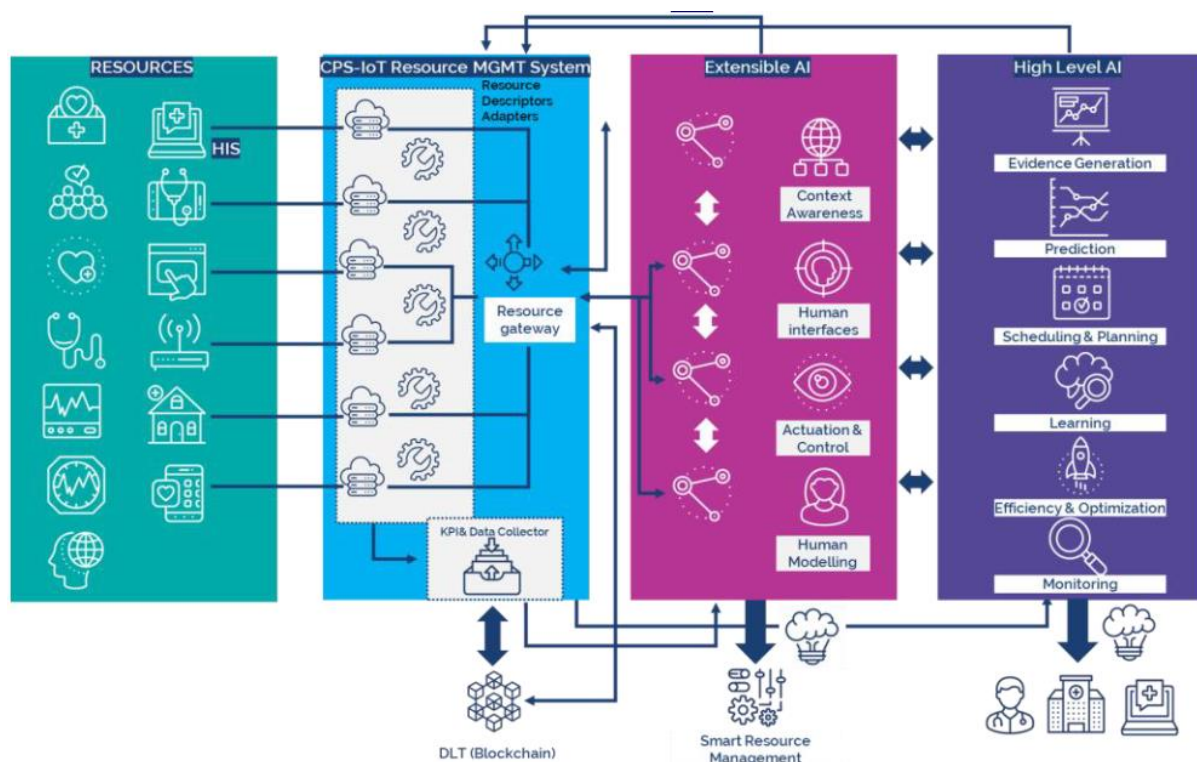


Figure 2 ODIN data information flows

To finalize the Vision, the goal of sharing the achievements among the different hospitals is inherent to the platform, such as datasets, models, or direct results.

2.1 WP3 ODIN's Architecture

From WP3 D3.10 ODIN's platform we reference ODIN's Architecture that has been worked also from WP4.

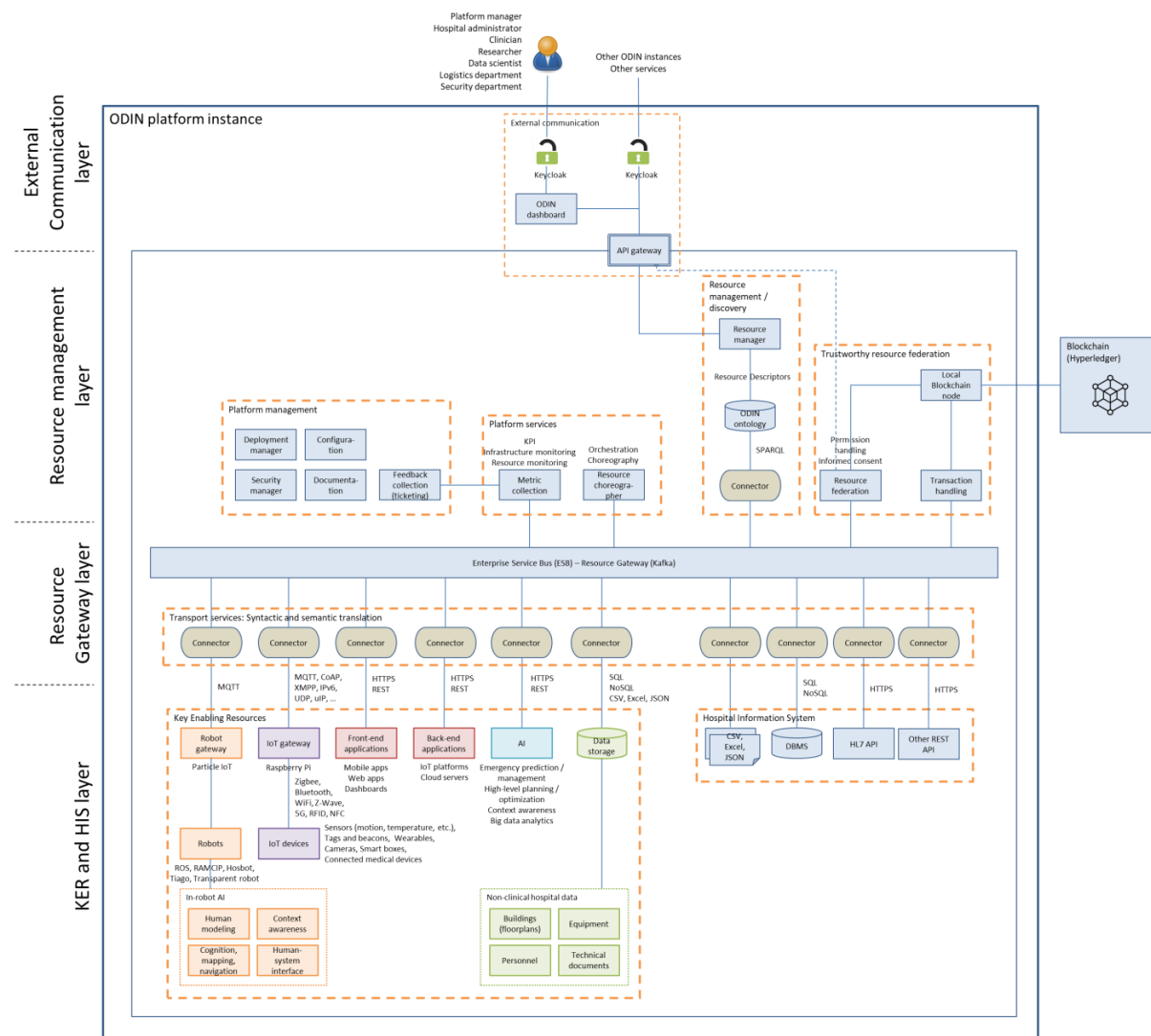


Figure 3 ODIN's final Architecture

Main WP4 contribution goes to the Enterprise Service Bus architecture used for the whole platform, the Resource Gateway layer and part of the Resource management layer.

In D4.1 most of the components and architecture will be discussed.

3 Requirement Management & tools

ODIN project was thought as a traditional waterfall project where the typical phases (Requirement gathering, Analysis, Design, Implementation, Tests, etc) were going to be carried, but in the roots was considered the fact that an agile iterative maturation process was inside the whole project, so several rounds of phases are going to be performed and materialised through the different deliverables that will follow D4.1 and related ones.

Several tools could have been used to track the requirements but at the moment of the creation of this deliverable there was not a clear tool to be used nor deployed.

3.1 Requirement gathering

Requirements gathering or eliciting is the task of searching for the sources of requirements and performing the tasks to get the base requirements.

WP2 has gathered requirements from pilots and users, performing interviews, brainstorming, document review and finally everything has been included into D2.1 ODIN co-creation workshop and end-user requirements, and D2.2 Hospital requirements report.

For the purpose of this deliverable, more meetings have been performed with WP3, WP5, WP2 and WP7 and other documentation such as the DoA. The types of requirements that will be managed in this deliverable are Functional and Non-Functional as followed by WP2.

3.2 Requirement documentation

After reviewing several techniques, the Volere template has been selected to document the requirements.

Volere template takes care of description, priority and traceability, so it covers most of WP4 needs to document the requirements.

3.2.1 Volere requirement template

The Volere template^{1,2}, comes from the need to have atomic requirements at the lowest level. They are atomic because contain all the information to keep track of the requirement, the stakeholders, and also are measurable, testable, traceable and detailed enough to define all aspects of a need without further breakdown.

To document such amount of information and keep traceability, Volere uses a set of attributes and document template in the form of a Snow card.

¹ http://www.inf.ed.ac.uk/teaching/courses/seoc/2008_2009/resources/volere-template.pdf

² <https://www.volere.org/wp-content/uploads/2018/12/06-Atomic-Requirements.pdf>,

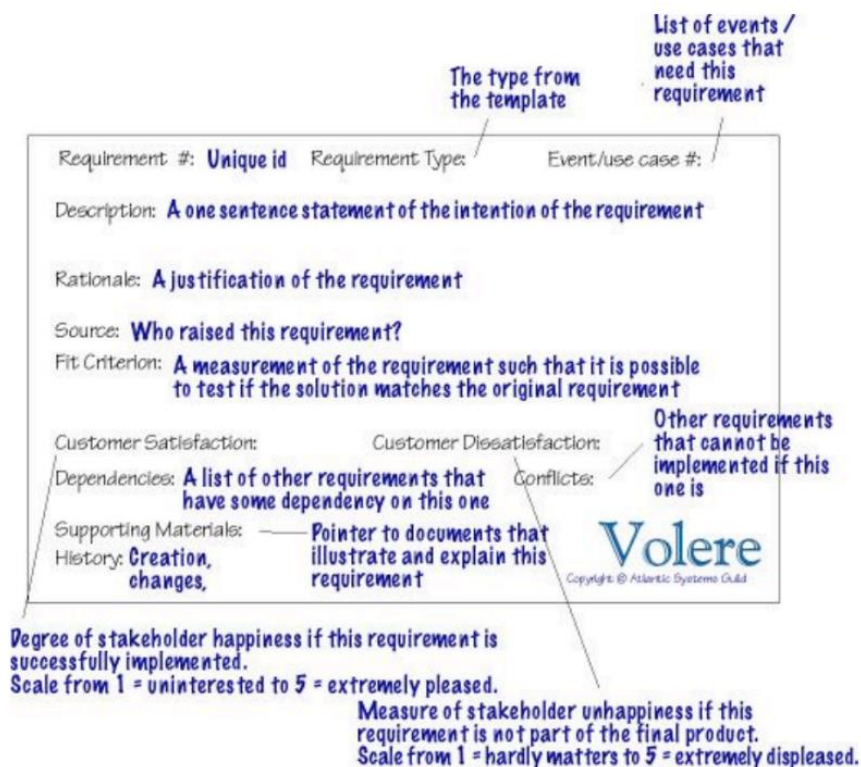


Figure 4 Volere template

The figure displays each attribute description, and what is more important, this template can be used to trace information from enterprise cases, business cases to product cases or events, just pointing them in the “Event/Case” field.

In addition, the template keeps more traceability using “History”, “Supporting Material”, “Conflicts” and “Dependencies” attributes.

3.2.2 ODIN's Volere criterion

The following clarifications are listed to understand how to fill the Volere card's fields in ODIN's context.

- **Requirement #:** The requirement ID must follow the template “ReqX_Component_Number”. Where X is the WP number, so in this case is a “4” after Req. The component is CP for CPS-IoT layer, RM for resource manager, RF for Resource Federation component, etc. This way, everyone can know where a requirement comes from, knowing the WP and component.
- **Requirement type:** can be Functional or Non-functional
- **Event/use case:** Will be “All” unless exist one or more use cases from D3.10 use cases list, or you identify any use case not present in the list.
- **Description** must be a statement describing what the requirement wants
- **Rationale** must be present as it is the reason that support creating the requirement
- **Source:** will be Platform Owners most of the times, but in case it is identified a use case related to it and the type of stakeholder that requested the requirement, we will use the types of Pilots, Third party providers or the type of stakeholder.

- Fit criterion: must a testable statement.
- Customer satisfactions and dissatisfaction must range from 1 to 5, and it must be expressed from Source's point of view and interests.
- Dependencies: For each requirement written, it will be added any other requirement that has dependencies on the latter.
- Conflicts: most of the times it will be left empty. The method to include a conflict is the same that dependencies.
- Supporting materials: reference to any document that may include relevant information about this requirement.
- History: Format must follow "V.Major Version Date of modification" (day/month/year).

3.3 Requirement validation

During requirement management, a requirement validation process must be performed in order to ensure that requirements are complete, consistent and aligned with customer's needs.

The first validation must be performed after the requirements are created and analysed to set up the requirement baseline. As soon as the developments are finished, more concrete checks must be performed to validate the developments.

During the project, changes to the requirements can take place, so a new analysis must be performed and after accepting the change, a new validation must be performed.

The baseline requirement validation can be performed using customer/stakeholder review of the requirements to check completeness, consistency, feasibility and dispel ambiguities.

On the other side, development checks can be accomplished by "Acceptance criteria" checks and Test Case validation.

4 Requirements & Functionalities

4.1 ODIN CPS-IoT layer Requirements

The DoA and WP2 work define a set of requirements and needs that must be considered for the platform development. These high-level requirements must be transformed into requirements in the domain of the WP4 and set up the long-term vision of the ODIN's platform.

Requirement #: Req4_CP_01	Requirement type: Non-functional	Event/ use case #: All
Description: The CPS must support multi-domain exchange of information by defining the corresponding data structures		
Rationale: Without a common data model and canonical messaging protocol is near impossible to try to integrate heterogeneous devices, resources and components into a single platform.		
Source: Platform owners		
Fit criterion: The platform will define the data and messages to support interoperability		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: Req4_CP_05, Req4_CP_02, Req4_CP_03, Req4_AI_01, Req4_RC_1	Conflicts:	
Supporting Materials: DoA, D3.2, D4.2, D2.2		
History: V.0 19/10/2021		

Requirement #: Req4_CP_02	Requirement type: Non-functional	Event/ use case #: All
Description: Deliver the necessary data services for the hospital environment		
Rationale: Beyond the needs around internal interoperability, the platform must offer enough data services to cover hospital needs		
Source: Platform owners		
Fit criterion: The platform must offer at least the necessary data services to cover the current use cases		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D2.2		
History: V.0 19/10/2021		

Requirement #: Req4_CP_03	Requirement type: Non-functional	Event/ use case #: All
Description: The CPS follows an Enterprise Service Bus (ESB) approach		
Rationale: Connecting multiple services using an ESB has lots of benefits in terms of integration, scalability, adaptability and decoupling, but also maintainability.		
Source: Platform Owners		
Fit criterion: The CPS will use a messaging bus to interconnect all the services and resources.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: Req4_CP_08, Req4_RC_1	Conflicts:	
Supporting Materials: DoA, D2.2, D3.10		
History: V.0 19/10/2021		

Requirement #: Req4_CP_04	Requirement type: Non-functional	Event/ use case #: All
Description: CPS-IoT RMS and has an open and flexible architecture, based on microservices and Enterprise Integration Patterns (EIP)		
Rationale: Using EIP and Microservices allows to select best solutions for each problem pattern, using state of the art knowledge. A microservice approach reduces coupling and fosters a 1 responsibility per service, easing the development.		
Source: Platform owners		
Fit criterion: services and solutions will follow EIP patterns.		
Customer satisfaction: 5	Customer Dissatisfaction: 4	
Dependencies: All Req4_RM_X, Req4_CP_07, Req4_CP_08, Req4_RC_1	Conflicts:	
Supporting Materials: DoA, D4.1, D3.10, D2.2		
History: V.0 19/10/2021		

Requirement #: Req4_CP_05	Requirement type: Non-functional	Event/ use case #: All
Description: CPS-IoTRMS incorporates standards and data to dynamically integrate all the resources into a single point of access through a common interface		
Rationale: Resources can be integrated manually and dynamically in automated form. First option is not operative as soon as services and resources start to grow.		
Source: Platform owners		
Fit criterion: Resources will be able to be discovered automatically		
Customer satisfaction: 5	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA,		
History:V.0 19/10/2021		

Requirement #: Req4_CP_06	Requirement type: Non-functional	Event/ use case #: All
Description: CPS-IoT not only offers the centralized access but also an access which is secure and semantically interoperable.		
Rationale: Healthcare is very sensitive domain by itself but also due to GDPR laws.		
Source: Platform owner		
Fit criterion: All the modules are authenticated and authorized to connect the platform. All the users are authenticated and authorized to connect the platform.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: All Req4_PS_X, Req4_RC_1	Conflicts:	
Supporting Materials: D3.3, D3.2		
History: V.0 19/10/2021		

Requirement #: Req4_CP_07	Requirement type: Non-functional	Event/ use case #:All
Description: Defines an easy way to expand open and flexible common architecture of service modules that integrate available open-source middleware components for CPS-IoT and Smart Cities (i.e.; Apache Camel, FIWARE) and other relevant components to provide the needed functionality to the upper layers of ODIN platform		
Rationale: The construction of the platform will be iterative and new use cases can arise which cannot be covered without platform evolution. The platform must be adaptable and open. Creating a platform which takes advantage of open-source projects is not an option, is the right strategy.		
Source: DoA		
Fit criterion: Opensource components for CPS-IoT can connect to the platform through a standardized mechanism.		
Customer satisfaction: Satisfied	Customer Dissatisfaction: Unsatisfied	
Dependencies: Req4_CP_03, Req4_CP_04, Req4_RC_1	Conflicts:	
Supporting Materials: D2.4, D3.10, D4.1		
History: V.0 19/10/2021		

Requirement #: Req4_CP_08	Requirement type: Functional	Event/ use case #: All
Description: The CPS-IoT support interconnection of available resources to achieve different goals such as: enable analytics and AI, support Context Awareness modules, enable Service Orchestration and any upper layer service.		
Rationale: Without resource integration and data collection, all of the components and functionalities are impossible.		
Source: Platform Owners		
Fit criterion: The platform integrates all the required resources to support upper layer objectives		
Customer satisfaction: 4	Customer Dissatisfaction: 5	
Dependencies: Req4_CP_03, Req4_CP_04, Req4_RC_1	Conflicts:	
Supporting Materials: D2.4, D3.10, D4.1		
History: V.0 19/10/2021		

4.2 Resource layer requirements

The Resource Management System puts available to the upper layers of ODIN Platform a set of resources that need to be consumed or exploited to accomplish Pilot's use cases and must be integrated in a seamless way to allow future integration of more resources. In the following lines a list of the derived resources from the WP2 requirements and the boundary layers requirements to be considered in ODIN's Platform.

Requirement #: Req4_RL_01	Requirement type: Non-functional	Event/ use case #: All
Description: The platform should support the communication with most important IoT, AI and Robot resources.		
Rationale: The CPS-IoT is the entry point of the resources to the platform.		
Source: Platform Owners		
Fit criterion: In IoT, the protocols used will be CoAp and MQTT, the data will be structured under the FHIR and HL7 health messaging standard. For the Robots the ROS framework will be used and for the infrastructures we will use the Modbus protocol. SQL and NoSQL databases will be used.		
Customer satisfaction: 5	Customer Dissatisfaction: 1	
Dependencies: Req4_RM_01,	Conflicts:	
Supporting Materials: DoA, D2.2, D2.4, D5.1, D5.1, D5.7		
History: V.0 15/02/2022		

Requirement #: Req4_RL_02	Requirement type: Non-functional	Event/ use case #:
Description: The platform should support the ontologies described by Wp3 T2.3		
Rationale: The CPS-IoT must have a common language and semantics.		
Source: Platform Owners		
Fit criterion: The ontology to be used will be BOT – Building Topology Ontology, Organization Ontology, NCIT – National Cancer Institute Thesaurus Ontology, SNOMED – Systemized Nomenclature of Medicine Ontology, ICD9CM – International Classification of Diseases, CORA – Core Ontology for Automation and Robotics, AI – Artificial Intelligence Ontology -- Generic: REST, NoSQL, WoT – Web Of Thing Ontology, FIHR/HL7 (health messaging)		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: Req4_RM_01,	Conflicts:	
Supporting Materials: DoA, D3.2		
History: V.0 15/02/2022		

4.3 Extensible specialized AI and High AI requirements

The Resource Management System interacts directly with the Extensible specialized AI and High AI layers, so we dig in this section the requirements the Platform must have to derive part of the components and architecture.

Requirement #: Req4_AI_01	Requirement type: Non-functional	Event/ use case #: All pilot UC
Description: The platform defines a data model that allows learning capabilities, data management and data sharing, data-model creation, data-model creation and High-level AI support to the ODIN project		
Rationale: Without the right data model, upper layers hardly can perform its tasks.		
Source: Platform owners		
Fit criterion: The data model supports all the needs from WP6 AI layer.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: D3.10, D4.2		
History: V.0 14/02/2022		

Requirement #: Req4_AI_02	Requirement type: Non-functional	Event/ use case #: All pilot UC
Description: CPS-IoT allows data storing for batch AI processing		
Rationale: Data may be available to be analysed after perception at any time		
Source: Platform owners		
Fit criterion: SLQ, Non-sql databases and other media databases are available for data processing.		
Customer satisfaction: 4	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.10, D4.1		
History: V.0 14/02/2022		

4.4 Hospital Knowledge, cognition & perception cycles requirements

Requirements for ontology, data models and hospital knowledge are included in this section.

Requirement #: Req4_HK_01	Requirement type: Non-functional	Event/ use case #: All
Description: Use the BOT – Building Topology Ontology		
<p>Rationale: BOT originated from the need for the implementation of web-based applications to enhance the Building Information Modelling (BIM) methods.</p> <p>The Building Topology Ontology is a simple ontology defining the core concepts of a building. It is a simple, easy to extend ontology for the construction industry to document and exchange building data on the web</p>		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_02	Requirement type: Non-functional	Event/ use case #: All
Description: Use the NCIT - National Cancer Institute Thesaurus Ontology		
Rationale: Developed by the National Cancer Institute’s Centre for Bioinformatics ²⁶ and Office of Cancer Communications ²⁷ with the main objectives of providing a base terminology for cancer.		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_03	Requirement type: Non-functional	Event/ use case #: All
Description: Use the SNOMED - Systemized Nomenclature of Medicine Ontology		
Rationale: SNOMED-CT Systemized Nomenclature of Medicine Clinical Terms Ontology is a clinical healthcare terminology system used for electronic healthcare records		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_04	Requirement type: Non-functional	Event/ use case #: All
Description: Use the ICD 11 - International Classification of Diseases 11		
Rationale: The International Classification of Diseases (ICD) is a classification system that organizes diseases and injuries into groups based on defined criteria. International Classification of Diseases 11th revision Clinical Modification		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_05	Requirement type: Non-functional	Event/ use case #: All
Description: Use the CORA - Core Ontology for Automation and Robotics		
Rationale: CORA-Core Ontology for Robotics and Automation is defined by the IEEE 1872-2015 standard from the IEEE Robotics and Automation Society		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_06	Requirement type: Non-functional	Event/ use case #: All
Description: Use the AI – Artificial Intelligence Ontology		
Rationale: AI-Artificial Intelligence ontology was chosen to construct the ODIN ontology. The Artificial Intelligence ontology is a comprehensive model of AI activities and applications		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_07	Requirement type: Non-functional	Event/ use case #: All
Description: Use the NANDA NIC NOC – Nursing Ontology		
Rationale: Nurses have performed their nursing practice according to the standard guidelines such as NANDA, NIC, and NOC, and recorded the information on nursing process into EMR system.		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials:		
History: v1_19/02/2022		

Requirement #: Req4_HK_08	Requirement type: Non-functional	Event/ use case #: All
Description: Use the WoT – Web of Thing Ontology		
Rationale: Web of Thing-WoT Ontology is an RDF axiomatization of the Web of Thing. The main aim of the ontology is to provide a semantic representation of knowledge in the domain of Web of Thing.		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

Requirement #: Req4_HK_09	Requirement type: Non-functional	Event/ use case #: All
Description: Use the Organization Ontology		
Rationale: Organization Ontology is a core ontology for generically representing organizational architectures and roles across a multitude of domains and designed to allow domain-specific extensions to add classification in the core ontology to provide a further level of specification.		
Source: Platform Owners		
Fit criterion: Ontology of the hospital environment necessary for the use cases. Selected for development of ODIN Ontology.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D3.2		
History: v1_19/02/2022		

4.5 Privacy, Security & Trust requirements

Due to the nature of security requirements and the existence of WP3 which is focused on Privacy, Security and Trust, D3.4 and D3.5 will handle those requirements.

4.6 Resource Gateway requirements

Resource Gateway requirements include:

Requirement #: Req4_RM_01	Requirement type: Non-functional	Event/ use case #: All
Description: The Resource Gateway is in charge of managing communication to the ODIN upper layers, functioning as the only entry point and coordinating calls to the other modules.		
Rationale: Having an only one entry approach to access the modules is a good approach to avoid infinite point to point connections, that increase complexity, dependencies and maintenance time.		
Source: Platform owners		
Fit criterion: The only way to access the rest of the modules is through the Resource Gateway. Beware that that does not mean it has to be a hub and spoke component		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D4.1, D4.2		
History: v.0 01/11/2021		

Requirement #: Req4_RM_02	Requirement type: Non-functional	Event/ use case #:
Description: The Resource Gateway will be implemented following ESB approach, as a framework for message-oriented middleware with a rule-based routing and mediation engine that provides an implementation of the Enterprise Integration Patterns to configure routing and mediation rules and being able to be extended to incorporate semantic capabilities.		
Rationale: The RM must be the entry point to ODIN modules, but a hub and spoke approach is not scalable in terms of adaptability of the platform. An ESB allows to achieve adaptability, maintainability and scalability.		
Source: Platform owners		
Fit criterion:		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: Req4_RM_1, Req4_RC_1	Conflicts:	
Supporting Materials: DoA, D4.1, D2.2, D3.10		
History: V.0 01/11/2021		

Functional Requirements:

Requirement #: Req4_RM_03	Requirement type: Functional	Event/ use case #: UC_CP_5, UC_CP_6, UC_CP_7 UC_CP_8 UC_CP_9, UC_CP_10
Description: As an Architect I want the RG to be the main channel for communication among components.		
Rationale: The Resource Gateway is in charge of managing communication to the ODIN upper layers, functioning as the only entry point and coordinating calls to the other modules.		
Source: Platform owners		
Fit criterion: Any communication among components and upper layers will be done through the RG		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D4.1, D2.2, D3.10		
History: V.0 01/11/2021		

Requirement #: Req4_RM_04	Requirement type: Functional	Event/ use case #: UC_CP_4, UC_CP_13
Description: As an Administrator I want to add resources to the platform so they can be used		
Rationale: ODIN's platform must be open to add resources, so it must offer a method to add resources.		
Source: Platform owners		
Fit criterion: There will be at least an API call to add resources to the platform.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D2.2, D3.10		
History: V.0 01/11/2021		

Requirement #: Req4_RM_05	Requirement type: Functional	Event/ use case #: UC_CP_4, UC_CP_13
Description: As an Administrator I want to list the resources of the platform so I can review what resources the platform has.		
Rationale: Managing the resources means an administrator has a clear overview of the current resources.		
Source: Platform Owner		
Fit criterion: There will be at least an API call to get the platform resources, so a list of the resources, with the type and status is shown.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D2.2, D3.10		
History: V.0 01/11/2021		

Requirement #: Req4_RM_06	Requirement type: Functional	Event/ use case #: UC_CP_4
Description: As an Administrator I want to remove resources of the platform so I can stop using one resource.		
Rationale: Managing the resources means an administrator has the power to remove resources that are not useful anymore.		
Source: Platform owners		
Fit criterion: There will be at least an API call to get the platform resources, so a list of the resources, with the type and status is shown.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D2.2, D3.10		
History: V.0 01/11/2021		

Requirement #: Req4_RM_07	Requirement type: Functional	Event/ use case #: UC_CP_4 to 6
Description: As an Administrator I want the API Gateway to have dynamic publishing of resource API services.		
Rationale: Manual tasks are hard and error prone. ODIN platform must be capable of exposing resources in an automated fashion.		
Source: Platform owners		
Fit criterion: When a new resource is added, the platform must publish the new resource and in case is not already available, expose the API service to the platform it terms of messages to be used on the ESB.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: DoA, D2.2, D3.10		
History: V.0 01/11/2021		

4.7 DLT Resource Federation requirements

Resource federation is one of the core aspects of the ODIN platform, allowing resources of one ODIN instance to be available for use in other ODIN instances, making the overall ODIN system a decentralized system. Resource federation needs to be performed in a trustworthy manner, so that one hospital can be sure that the use of its resources by another hospital is according to given permission and that any potential misuse can be tracked and accounted for. Trustworthy resource federation will be implemented based on Digital Ledger Technologies (DLT), which offers the desired properties of auditability, non-repudiation and accountability.

The Resource Federation requirements include the following.

Functional requirements:

Requirement #: Req4_RF_01	Requirement type: Functional	Event/ use case #: Resource federation
Description: Allow controlled and secure direct share of resources between different ODIN platform instances (e.g., hospitals). Resources include: data (e.g. Electronic Medical Records), AI/ML models, services, KPIs, IoT and robotic platforms, knowledge.		
Rationale: To enable decentralization of resources, facilitate interoperability across hospital environments and ensure privacy, security and trust across organizations.		
Source: Platform Owners, hospital administrator		
Fit criterion: An organization with one ODIN instance can use resources of another remote organization with another ODIN instance.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_02	Requirement type: Functional	Event/ use case #: Resource federation
Description: Allow each organization (e.g. hospital) to store sharable data and resources in private local or cloud spaces.		
Rationale: Hospitals usually have strict rules regarding data and resource accessibility, so they should be allowed to have full control of ownership and how to share.		
Source: Platform Owners, hospital administrator		
Fit criterion: Organizations have data and resources stored/deployed in their private local or cloud spaces.		
Customer satisfaction: 4	Customer Dissatisfaction: 5	
Dependencies: Req4_RF_01	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_03	Requirement type: Functional	Event/ use case #: Resource federation
----------------------------------	-------------------------------------	--

Description: Use state-of-the-art standards, such as Decentralized Ledger Technologies (DLT) and blockchain to access, manage and federate resources.	
Rationale: Decentralized Ledger Technologies (DLT) and blockchain allow secure and trusted information exchange, through their auditability, traceability and immutability characteristics.	
Source: Platform Owners	
Fit criterion: DLT is used to support resource federation.	
Customer satisfaction: 4	Customer Dissatisfaction: 4
Dependencies: Req4_RF_01, Req4_RF_06, Req4_RF_07, Req4_RF_08, Req4_RF_15	Conflicts: None
Supporting Materials: DoA, D3.10, D4.5	
History: V.0 20/4/2022	

Requirement #: Req4_RF_04	Requirement type: Functional	Event/ use case #: Resource federation
Description: Develop ODIN's own private/permissioned blockchain network to manage assets, audit transactions and ensure secure data exchange. Only authorized nodes should be allowed to write data on the blockchain.		
Rationale: The blockchain network should not be made public due to privacy and security concerns.		
Source: Platform Owners		
Fit criterion: The developed blockchain network is private/permissioned.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_05	Requirement type: Functional	Event/ use case #: Resource federation
Description: Use open-source platforms for the blockchain.		
Rationale: Open-source platforms do not impose cost restrictions, while allowing the implementation of private blockchain networks.		
Source: Platform Owners		
Fit criterion: The blockchain network is developed using open-source platforms.		
Customer satisfaction: 3	Customer Dissatisfaction: 3	
Dependencies: Req4_RF_04	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_06	Requirement type: Functional	Event/ use case #: Resource federation
Description: Every procedure that transfers health sensitive data and resources from one ODIN instance to another must be logged in the blockchain.		
Rationale: Logging to the blockchain ensures the immutability of the transaction, enhancing trust between organizations.		
Source: Platform Owners, hospital administrator		
Fit criterion: All sensitive transactions are logged in the blockchain.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_07	Requirement type: Functional	Event/ use case #: Resource federation
Description: Informed consent regarding data/resource transfer across organizations should be stored in the blockchain.		
Rationale: Storing informed consents in the blockchain ensures the immutability of data/resource authorizations.		
Source: Platform Owners, hospital administrator		
Fit criterion: Informed consent for resource sharing is stored in the blockchain.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_08	Requirement type: Functional	Event/ use case #: Resource federation
Description: Smart contracts should be used for auditing, resource sharing, permission management and informed consent.		
Rationale: Smart contracts can automate the execution of resource sharing agreements and workflows, so that all involved parties are immediately aware of the outcomes.		
Source: Platform Owners		
Fit criterion: Smart contracts are used for auditing and resource sharing.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_09	Requirement type: Functional	Event/ use case #: Resource federation
Description: The blockchain auditing mechanism should include a timestamp in any transaction.		
Rationale: Timestamps facilitate traceability and liability support.		
Source: Platform Owners		
Fit criterion: Timestamps are included by the blockchain in every transaction.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_10	Requirement type: Functional	Event/ use case #: Resource federation
Description: The blockchain should allow several transactions to be committed in a single batch.		
Rationale: Transaction batching provides performance benefits.		
Source: Platform Owners		
Fit criterion: Transaction batching is allowed by the blockchain.		
Customer satisfaction: 3	Customer Dissatisfaction: 3	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_11	Requirement type: Functional	Event/ use case #: Resource federation
Description: Allow old data/transactions to be purged/obfuscated after a certain amount of time, if required by relevant privacy regulations.		
Rationale: Certain user privacy requirements imposed by specific organization/regional regulations may require the deletion of old data.		
Source: Platform Owners		
Fit criterion: Old data/transaction obfuscation is allowed if needed, e.g. by modifying the encryption keys.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_12	Requirement type: Functional	Event/ use case #: Resource federation
Description: Allow queries to the blockchain through a REST API.		
Rationale: A REST API allows the querying of the blockchain both by users and by external systems.		
Source: Platform Owners		
Fit criterion: The blockchain can be queried through a REST API.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: Req4_RF_13	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_13	Requirement type: Functional	Event/ use case #: Resource federation
Description: Provide a graphical user interface to allow users to query the blockchain, obtain an overview of its status and historical transactions, and visualize the blockchain status and KPIs.		
Rationale: A graphical user interfaces facilitates the use and monitoring of the blockchain by the end users (e.g. hospital administration).		
Source: Platform Owners, hospital administrator		
Fit criterion: A graphical user interface is provided for querying, monitoring and visualization the blockchain status, transactions and KPIs.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Non-functional requirements:

Requirement #: Req4_RF_14	Requirement type: Non-functional (security)	Event/ use case #: Resource federation
Description: Log files and informed consents should be encrypted in the blockchain.		
Rationale: Encryption enables secure data/resource sharing, increasing the level of trust by the organizations and facilitating their transition from existing centralized models.		
Source: Platform Owners		
Fit criterion: Log files and informed consents are encrypted in the blockchain.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_15	Requirement type: Non-functional (security)	Event/ use case #: Resource federation
Description: All resource transactions across organizations should be auditable, traceable and immutable. Accountability and non-repudiation of the involved parties should be ensured.		
Rationale: Accountability and non-repudiation increase trust among the interconnected organizations.		
Source: Platform Owners, hospital administrator		
Fit criterion: Usage of blockchain technologies for logging all resource sharing transactions, which ensures auditability, immutability and non-repudiation.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_16	Requirement type: Non-functional (integrity)	Event/ use case #: Resource federation
Description: Hash keys of shared files should be stored in the blockchain.		
Rationale: Hash keys can ensure data integrity and non-repudiation.		
Source: Platform Owners		
Fit criterion: Hash keys of shared data files are stored in the blockchain.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_17	Requirement type: Non-functional (availability)	Event/ use case #: Resource federation
Description: Ensure continuous and uninterruptable operation of resource sharing. The blockchain auditing mechanism should work without any interruption at least for 5 consecutive years.		
Rationale: Continuous availability of resource federation is important for supporting the ODIN decentralized platform vision and use cases.		
Source: Platform Owners, hospital administrator		
Fit criterion: Resource federation is available continuously for at least 5 years.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_18	Requirement type: Non-functional (scalability)	Event/ use case #: Resource federation
Description: Resource federation should be scalable to large datasets (large storage per block) and large ODIN instance federations. The blockchain should allow the addition of new nodes on the fly.		
Rationale: Scalability measures ensure that the solution will be operational even in large federations, and dynamic as federations change in size.		
Source: Platform Owners		
Fit criterion: The on-the-fly addition of new blockchain nodes is supported.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

Requirement #: Req4_RF_19	Requirement type: Non-functional (performance)	Event/ use case #: Resource federation
Description: Once authorized, resource federation should operate in real-time, resembling local resource usage in performance.		
Rationale: Real-time performance is important for supporting the ODIN decentralized platform vision and use cases.		
Source: Platform Owners, hospital administrator, clinical staff, supporting staff		
Fit criterion: The usage of remote resources is performed without noticeable latencies due to resource federation mechanisms.		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies: None	Conflicts: None	
Supporting Materials: DoA, D3.10, D4.5		
History: V.0 20/4/2022		

4.8 Resource choreographer requirements

A list of Resource Choreographer requirements is included in this section.

Requirement #: Req4_RC_01	Requirement type: Non-functional	Event/ use case #: UC_CP_8
Description: The Resource Choreographer should follow an orchestration approach, but Choreography should be used when possible		
Rationale: Orchestration is easier to implement and maintain, but choreography can be used when the actions taken are controlled by the domain of a service. For example, to get the data to compute a forecast, an AI Service does not need the orchestrator, but the event that triggers the AI service to start the forecast computing, is issued by the orchestrator.		
Source: Platform owners		
Fit criterion: Big interactions should be driven by orchestration, while small interactions could be choreographed		
Customer satisfaction: 4	Customer Dissatisfaction: 4	
Dependencies:	Conflicts:	
Supporting Materials: D4.1, D2.2, D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_02	Requirement type: Non-functional	Event/ use case #: UC_CP_8
Description: The Resource Choreographer will use BPEL or other workflow notations to express the processes as long as are compatible with most common tools.		
Rationale:		
Source: Platform Owners		
Fit criterion: The Resource Choreographer can work with a BPEL file as long as is well formed and all the resources involved are available.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_03	Requirement type: Non-functional	Event/ use case #: UC_CP_8
Description: The Resource Choreographer will have a workflow engine to execute the business processes		
Rationale: Workflow engines are more powerful than rule engines		
Source: Platform Owners		
Fit criterion: At least, the big processes where more than 3 services interact beyond the scope of its domain functions, will be controlled by the RC		
Customer satisfaction:	Customer Dissatisfaction:	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_04	Requirement type: Functional	Event/ use case #:
Description: As an Admin I want a single component (the RC) to control service orchestration		
Rationale: The Resource Choreographer module will enable the coordination, in the form of business choreography services of the available resources accessible from the RG and upper layers of ODIN platform		
Source: Platform Owners		
Fit criterion: Business services will be formed by the collaboration of the resources and service through the orchestration of a single component.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_05	Requirement type: Functional	Event/ use case #: UC_CP_8
Description: As a Business manager I want to setup the processes available in the RC loading a workflow configuration		
Rationale: Managing Business use cases require to setup the resources and services to collaborate. That means a method to load the workflow dictating the actions and events must be available.		
Source:		
Fit criterion: It must exist one method to load the workflows into the RC and the method shall verify any resource or service used in the workflow is available on the platform.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_06	Requirement type: Functional	Event/ use case #: UC_CP_14
Description: As a Business manager I want to list the workflows loaded in the platform		
Rationale: Managing Business requires to know what active or inactive.		
Source: Platform Owners		
Fit criterion: It must exist one method retrieve the workflows loaded in the platform, with its description and its status, active or inactive.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_07	Requirement type: Functional	Event/ use case #:
Description: As a Business manager I want the Resource Choreographer to perform the dynamic generation of messages to trigger changes in the rest of the services, resources or components		
Rationale: Events and status changes must be kept aligned through the platform so processes work correctly.		
Source: DoA		
Fit criterion: Any event or status that must be communicated must have its own message to be used on the ESB.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

Requirement #: Req4_RC_08	Requirement type: Functional	Event/ use case #:
Description: The Resource Choreographer allows the dynamic generation of states and tasks to automate orchestration, self-initiate communication among resources intervening in the process		
Rationale: Tasks and state changes must be kept aligned through the platform so processes work correctly.		
Source: Platform Owners		
Fit criterion: Any task or status that must be communicated must have its own message to be used on the ESB.		
Customer satisfaction: 5	Customer Dissatisfaction: 5	
Dependencies:	Conflicts:	
Supporting Materials: D4.5		
History: V.0 12/01/2022		

4.9 KPI collection requirements

The KPI collection is a feature ideated in the platform which is intended for global monitoring of the platform itself, as well as all the KER connected to it, and their interactions. Thus, the KPI collection component should be able to monitor technological metrics, monitor these metrics and be able to react when metrics are out of operational parameters. Because the workflow and framework for KPI collection is the same as for automatic collection of non-technical KPI, i.e. monitoring Reference Use Case KPI and pilot KPIs (T7.2) which can be determined from the data the platform is able to access; the KPI collection tool will also extend its functionality to this broader understanding of KPI.

The KPI collection requirements are:

Requirement #: Req4_KPIC_01	Requirement type: Functional	Event/ use case #: UC_CP_9
Description: The system must be able to collect different types of metrics each providing a value, in the form of a defined data type (e.g. integer, float, Boolean, enumeration, date, text), and should include a name of the metric, a unit of measurement as well as a timestamp referencing the time at which the measurement was taken.		
Rationale: The system must collect metrics in a standard structured way so these metrics can be properly stored, aggregated, displayed, processed, and importantly compared against metrics collected at other time for the same system or with other systems.		
Source: Platform Owners, Administrators		
Fit criterion: a metric is reported (pushed), or collected (pulled) to/by the KPI collection component in a defined structure		
Customer satisfaction: 5	Customer Dissatisfaction: 3	
Dependencies: Req4_KPIC_02, Req4_KPIC_03, Req4_KPIC_09	Conflicts:	
Supporting Materials: DoA, D3.10, D4.2		
History: V.1 21/04/2022		

Requirement #: Req4_KPIC_02	Requirement type: Functional	Event/ use case #: UC_CP_9
Description: The KPI collection system must be able to aggregate, store, process, and export metrics from different sources and targets.		
Rationale: The system must be able to process the metrics being collected from all the metric sources aggregated so they can be stored centrally (for historic processing and display), further processing or to offer to consumers such as display, monitor alert, or support provision systems.		
Source: Platform Owners, Administrators		
Fit criterion: All the collected metrics (independently from their origin) can be displayed with their historical values.		
Customer satisfaction: 5	Customer Dissatisfaction: 3	
Dependencies: Req4_KPIC_04, Req4_KPIC_05, Req4_KPIC_08, Req4_KPIC_09	Conflicts:	
Supporting Materials: DoA, D3.10, D4.2		
History: V.1 21/04/2022		

Requirement #: Req4_KPIC_03	Requirement type: Functional	Event/ use case #: UC_CP_9
Description: The KPI collection system must be able to react to changing metrics according to configurable parameters.		
Rationale: In order to notify other entities (whether human or other systems) about abnormal metrics being reported. Notification systems may include email, push notifications on a screen, other type of human communication systems (e.g. voice alert) or IPC such a message in the EBS or a webhook.		
Source: Platform Owners, Administrators		
Fit criterion: when properly configured, the system will send an email when the specified metric reports a value over the specified alert value.		
Customer satisfaction: 4	Customer Dissatisfaction: 2	
Dependencies: Req4_KPIC_08	Conflicts:	
Supporting Materials: DoA, D3.10, D4.2		
History: V.1 21/04/2022		

Requirement Req4_KPIC_04	#:	Requirement type: Functional	Event/ use case #: UC_CP_9
Description: The system must dynamically manage KPI sources			
Rationale: by being able to list all KPI sources, it can be determined if a registered metric source is not producing metrics (comparing it to the metrics it is reporting). The system must allow the add or removal of metric sources (always in the common data model) so that other subsystems (or evaluators) can report their own metrics.			
Source: Platform Owners, Administrators			
Fit criterion: The KPI collection system can list and modify current list of metric sources			
Customer satisfaction: 4		Customer Dissatisfaction: 2	
Dependencies: Req4_KPIC_05, Req4_KPIC_07, Req4_KPIC_08, Req4_KPIC_09		Conflicts:	
Supporting Materials: DoA, D3.10, D4.2			
History: V.1 21/04/2022			

Requirement Req4_KPIC_05	#:	Requirement type: Functional	Event/ use case #: UC_CP_9
Description: An audit of interactions with the platform should be available.			
Rationale: The system must automatically register metrics pertaining to the interaction through the platform (Like send message, (un)registering, receiving message, user login, session establishment, etc...), in this way the different components should not be required to manage these, and every component is minimally monitored through these metrics.			
Source: Platform Owners, Administrators			
Fit criterion: All interactions through the platform can be visualized in the form of metrics, even if the different components (outside the core and KPI collection components) are not explicitly reporting these metrics.			
Customer satisfaction: 5		Customer Dissatisfaction: 4	
Dependencies: Req4_KPIC_06		Conflicts:	
Supporting Materials: DoA, D3.10, D4.2			
History: V.1 21/04/2022			

Requirement #:	Requirement type:	Functional	Event/ use case #:
Req4_KPIC_06			UC_CP_9
Description: The system must automatically collect logs and aggregate them.			
Rationale: Logs are an important aspect of system monitoring, even if it may not be considered a metric in itself, standard logs (typically standard output from containerized systems in the format of line per event) should be managed in the same way as metrics (structured, source management, aggregation, storage, export, reaction, display, and automatic registration of standard elements).			
Source: Platform Owners, Administrators			
Fit criterion: logs produced by different components can be visualized (optionally API available) through the metric collection system.			
Customer satisfaction: 5		Customer Dissatisfaction: 3	
Dependencies: Req4_KPIC_08		Conflicts:	
Supporting Materials: DoA, D3.10, D4.2			
History: V.1 21/04/2022			

Requirement Req4_KPIC_07	#:	Requirement type: Non-functional	Event/ use case #: UC_CP_9
Description: The system should support questionnaires.			
Rationale: Many RUC KPIs are collected in the form of questionnaires: PROMs, PREMs, user satisfaction, etc., thus the system should support collection and transformation of questionnaire answers into metrics. The system may include a module for questionnaire creation, management and execution.			
Source: Platform Owners, HP, Evaluators			
Fit criterion: Standard and non-standard questionnaires can be collected as metrics and displayed in the KPI dashboard.			
Customer satisfaction: 4		Customer Dissatisfaction: 2	
Dependencies:		Conflicts:	
Supporting Materials: DoA, D3.10, D4.2			
History: V.1 21/04/2022			

Requirement Req4_KPIC_08	#:	Requirement type: Non-functional	Event/ use case #: UC_CP_9
Description: The system should include a customizable dashboard displaying collected metrics, enabling end users to customize the metrics displayed, as well as how they are displayed and grouped.			
Rationale: The system manages a dynamic number of metric streams; thus, the display must be able to adapt to these. Users will not like everchanging dashboards, and because each user is interested in different metrics, they should be able to customize the metrics displayed in their dashboard, as well as how they are displayed in order for them to be more effective in the interpretation of the monitoring of the system.			
Source: Platform Owners, Administrators, Evaluators, HP			
Fit criterion: User A can customize the metrics displayed in their dashboard independently from the customization User B has performed. User A and B may share each other's dashboards if they choose to (e.g. through exporting-importing function, or multiple dashboard function).			
Customer satisfaction: 4		Customer Dissatisfaction: 2	
Dependencies:		Conflicts:	
Supporting Materials: DoA, D3.10, D4.2			
History: V.1 21/04/2022			

Requirement #:	Requirement type:	Event/ use case #:
Req4_KPIC_09	Non-functional	UC_CP_9
Description: The system should enable easy custom metric processing and alert management		
Rationale: Many RUC KPIs require specific processing from variables in the system, and the system won't have all of them predefined. Users may also use this system to define custom KPI processing. For the same reasons, the alert system needs to be configurable.		
Source: Platform Owners, Administrators, Evaluators		
Fit criterion: A new metric can be defined from other metrics and/or variables, through a formulaic expression. When an alert is triggered for a metric, the user can change the trigger condition and/or action, so the alert is no longer triggered.		
Customer satisfaction: 4	Customer Dissatisfaction: 2	
Dependencies: Req4_KPIC_07	Conflicts:	
Supporting Materials: DoA, D3.10, D4.2		
History: V.1 21/04/2022		

5 Technology and architecture concepts

5.1 System and Software Architecture definition

We can find several definitions that mostly converge for the term Software³ and System architecture.

From the book Documenting Software Architectures: Views and Beyond (2nd Edition), Clements et al, Addison Wesley, 2010: The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

From ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software Intensive Systems Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

In the following sections we dive into several technologies and architectures related to the CPS and its components. As a result, a more in depth understanding of the best options will be achieved and this knowledge will be used in the following sections and deliverables.

5.2 Enterprise Integration patterns

Enterprise Integration Patterns are solution templates created to solve common problems that arises when connecting several enterprise systems.

A non-complete, but most common, taxonomy of problem types solved through integration pointed by Gregor Hohpe and Bobby Woolf in Enterprise Integration patterns book ⁴are:

- Information Portals: A place to aggregate information from several systems
- Data Replication: many systems that composed by subsystems that stores same information such as customer names or other details, may need a replication of data changes if the data is changed in one of the subsystems
- Shared Business Functions: when 2 systems use a common functionality, it is better to extract the functionality to service, and use it from the 2 systems.
- Service-Oriented Architectures: this is an evolution of the shared business function, were all the functionalities are separated into single services, and each layer or service can discover and use the ones it needs through a well-defined interface.
- Distributed Business Processes: in this case, a business process need is segmented through several system (not services) and a business process management component are introduced to control the process.
- Business-to-Business Integration: while previous problem types usually happen inside a business or enterprise boundary, the Business-to-Business integration problem tackles 2 or more business or enterprises. This way the systems may interact through defined

³ https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf

⁴Gregor Hohpe, Bobby Woolf. SBN 0321200683 Addison-Wesley, 2004, Enterprise Integration Patterns. Page 34

interfaces across the Internet, which imposes additional requirements regarding security, authorization and auditing in most cases.

ODIN with some blurred lines, can be seen as a mix of Information Portal, Service Oriented Architecture and Business-To-Business integration.

The Information Portal style will be used in the following way. As states in the DoA and the meetings performed in WP3 mainly and WP4, the platform will have at least a Dashboard for managing the platform, so information from many systems will be aggregated in an organized way to be presented for each user depending on its rol. Other dashboards may exist also to integrate the information for the developed use cases, the Resource Choreographer (check Req_RC_04 to Req_RC_06) and the KPI metric system.

Service oriented Architecture pattern is in the DoA roots, and it has been a top priority in WP3, so the platform is going to be architected as the collaboration of many services or micro services with clear separated functionalities that work together to fulfil hospital needs. Through this document is reviewed the components related to the CPS-IoT and section 7 dives into those concrete services and final components and architecture can be checked on deliverable D3.10 ODIN platform. Some of the platform's components are: the Messaging Bus, the Resources, the Connectors that bridge the resources and the Messaging Bus, the Resource Directory or the Resource Choreographer.

Finally, the Business to Business Integration pattern can be observed due to the objective of resource and data federation. It must be taken into account that the platform is not only a hospital deployment, the platform will be the whole collaboration of the hospital instances, sharing resources and services, plus a Cloud instance as it can be checked on section 5.7 for Distributed Ledger technologies and more in depth in deliverable D3.10 ODIN platform. So, those instances can be seen as different Business units collaborating through the Internet, thus extra communication interfaces and security requirements must be considered.

5.2.1 Integration styles

There exist several kinds of integration styles, depending on the needs and requirements. Sometimes a mix of them have to be used to achieve an integration level.

Examples of integration styles are:

- File Transfer — Have each application produce files of shared data for others to consume and consume files that others have produced.
- Shared Database — Have the applications store the data they wish to share in a common database.
- Remote Procedure Invocation — Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to run behaviour and exchange data.
- Messaging — Have each application connect to a common messaging system, and exchange data and invoke behaviour using messages.

As stated in ODIN's Grant Agreement, Messaging is the preferred communication pattern as stated by the Req_CP_03 so we will focus on describing the most important patterns used in a Messaging integration style from a problem-solving point of view. On the other hand, other kind

of integration styles may co-exist among some of the platform components that will fit better in some cases, for example front-end communication to the platform may work better with direct client-server API calls, while video streaming may be available from some resources to deliver the video to higher platform levels such as AI monitoring what happens in a room. Some hospitals have expressed their wish to keep their systems disconnected from Odin and load information from files, so FTP or other ways to integrate data from files will be used.

5.2.2 The Messaging concepts

A messaging system based on messages, has several important concepts to manage.

Channel: an application sends data using Channels, which is a logical pipe that connects senders and receivers. The pipes must be designed following the requirements of the applications.

Message: the data sent through a channel must follow a template and is atomic for the message system. Once an application or component receives a message, can extract the information needed to process it.

Multi-step delivery: few times a message can be sent directly from a sender to a receiver, usually more steps must be performed with the message before reaches the final destination.

Routing: messages may go through several components and channels before reaching the destination. This is very common in large systems so the messaging system needs to handle the routes the message must take. Usually this happens using “Message router” components and “filters” along the pipes of information.

Transformation: integrating heterogeneous applications usually means to handle data sets in different formats, so the messaging system must be able to handle the diversity to deliver or receive the messages in the right format for each application. “Message Translator” is the way to achieve this.

Endpoint: an application usually does not have by default the right functionality to connect to the messaging system. So, an adapter or “Message Endpoint” must be introduced to enable the application to send and receive messages.

5.2.3 Channel

As explained, channels are the roads where the data flows from a sender application to the receiver applications. A service bus, which is ODIN’s objective, is a set of channels where the services connect to communicate the rest of the components.



Figure 5 Channel example

There exist 2 main types of channels:

- Point-to point channel: used to send a message that ought to be consumed just by one receiver.

- Publish-Subscribe: used to send one message to many receivers.

Under those main channel types there exist other subtypes such as:

- Datatype channel: it should be used to forward messages that must be recognized by the receiver to process them, depending on the type of the information sent.
- Invalid message channel: it is used to forward messages that a receiver detects have no sense, due to the rules of the channel, the format of the message and many other cases that should discard a message.
- Dead letter channel: It is a channel to manage messages that the messaging system cannot process to deliver it, so it is forwarded to this type of channel to allow the messaging system act over it in case it wants to, for example logging the message or deleting it.
- Guaranteed delivery channel: this type of channel makes sure the message has been received whatever happens, system crash included. So, the messages are stored until reception by receiver is confirmed.
- Channel Adapter: This channel allows to adapt a system with an API, for example, to interact with the messaging system. Usually are bidirectional but can also be just unidirectional. This way, if anything changes in one application, the changes can be adapted in the Channel Adapter without affecting the rest of the system. It can be used in several layers, for example interfaces, business logic, database layer, etc. Usually, the Channel Adapter is combined with a Message Translator, that converts the data into the canonical data model of the messaging system.
- Messaging Bridge Channel: this is a special grouping of channel adapters, used to adapt more than one messaging system, so it is ensured that every or at least necessary messages are available in all messaging systems. For example, due to integration of previous messaging systems in a company acquisition, or in a B2B client that bridges many provider applications.
- Message Bus: this is a group of channels, that enable a decoupled architecture so the applications/services can be added or removed without affecting the others. It is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces.

5.2.4 Message

The Message is the atomic data packaging used to send information through the messaging system.

A Message usually has 2 parts, the header and the body. The header helps the messaging system processing it, with a sender, receiver and whatever is needed to work in the messaging system. The body contains the data that actually wants to be sent from the sender to the receiver.

Depending on the intention of the message, there exist 3 types of messages:

- Command Message: usually sent from the sender to the receiver to perform a given action or method.
- Document Message: when a sender wants the receiver to get some information, it uses a document message, but it does not specify what to do with it.

- Event Message: this kind of message is used to notify the receiver that something happened.

Special messages are used for example to send a message and receive a response, so the sender's message must indicate where the response message should be forwarded and usually a correlation identifier, so the sender know what the reply is related to.

Other messages may imply a lot of information and a Message Sequence must be used, so the data is separated in chunks.

A set of messages will compose the canonical messaging system used to communicate all the services through ODIN's service bus.

5.2.5 Router

The routers allow dispatching messages to different channels depending on the content or other rules. This way a solution can have just one channel for an application that has to connect to several receivers, and let the router dispatch the message to the right receiver.

Main router types are listed in the following table

Router Type	Problem solved
Content based router	Dispatches messages from one channel to many based on the content of the message
Filter	Accepts or discards a message based on the content
Dynamic router	A router that can change the routing rules upon special messages from the participating destinations
Recipient List	Dispatch is ruled by a whitelist of receivers
Splitter	Divides a message into several ones and delivers each one by different channels
Aggregator	Composes a message from several ones and sends it through a channel
Scatter-Gather	Is the combination of a splitter and aggregator used when a message needs to be sent to several receivers that may also answer
Process manager	Routing messages through multiple processing steps that may not be known at design time and that are not sequential uses this approach. This is a composed router type that has to manage a state for each message
Message broker	This router decouples the destination of a message from the sender and maintains central control over the flow of the messages. The sender does not know who the receiver is

Table 1 Router types

ODIN can use several types of routers. For example, the bus connecting all the components will be a special type of Message broker with a publish/subscribe approach.

5.2.6 Transformation

The messaging system need to ensure that messages are compatible in data format (header/body) or information type among senders and receivers, and the messaging system itself, so most of the times a transformation of the message must be performed.

Transform component	Problem solved
Envelope Wrapper	Hides application information inside an envelope that is compliant with the messaging infrastructure. Once arrives to the destination, it is unwrapped
Content enricher	When part of the information needed to process a message is not available, it must be added to the message, so a content enricher is used to look for the information and added to a new message
Content filter	Helps simplifying a message by deleting superfluous information that can be removed to increase efficiency, or for example for security reasons.
Normalizer	Allows to process messages that are semantically equivalent in different systems connected to a shared channel but that arrive in heterogeneous formats, so the receiver can process them in a common way. Useful if the receiver cannot control the format used by senders, as in B2B.
Canonical data model	Helps minimizing the dependencies when integrating applications that use different data formats, so a common model is used, and senders must adhere to the format. This way all the applications just to implement only one message transformation. When adding one more application, it just needs to adapt to the canonical data model

Table 2 Transform operators

ODIN's Connectors, which will transform physical and logical protocols, are indeed transformation components of type Canonical Data model and Content enricher.

5.2.7 Messaging Endpoint

To connect an application to the messaging system, a piece of code shall be included on each application connecting.

There are several strategies to do so, and we are listing the most important to be considered in future designs.

Endpoint type	Problem solved
Messaging gateway	Is the simplest way to encapsulate messaging specific code access to the messaging system, minimizing code dependencies in the application. Very useful for testing purposes
Messaging Mapper	Usually, business objects data from an application cannot be handled in the messaging system without keeping both independent. Messaging mapper allows just that.

Transactional client	Allows to handle transactional behaviour in a client application or among several applications while using a messaging system
Polling consumer	Gives the control of consuming a message to the client application, so the client polls for new messages to the channel. This can lead to application block in case no messages are received because reception is synchronous
Event-Driven consumer	In this case the client application consumes the messages as they become available in the channel, in an asynchronous fashion
Competing consumer	This pattern is used when we want to increase the rate of message processing in a channel, so several clients process messages as long as they arrive. This pattern is bad for transactional clients. Usually needs a Messaging dispatcher
Messaging dispatcher	When several consumers want to process messages on a channel and avoid multiple processing of the same message, it can be used a messaging dispatcher to distribute the messages
Selective consumer	In channels with multiple consumers, filters usually cannot be used as would filter the messages for all the consumers, so, an individual filter must be used inside each client. This way the consumer gets a selective behaviour.
Durable consumer	For publish-subscribe channels which need to deliver the message to each subscriber at any scenario (crash, disconnect situations) the consumer shall implement this pattern so it will store messages while in is disconnected from the channel

Table 3 Messaging Endpoint types

Services connecting to ODIN's service bus use several types of endpoint component, usually a combination of them.

5.2.8 System Management

Up to now, the patterns listed are used to deliver messages for business logic, but patterns may also be used to manage and control a logical infrastructure. In the following lines a list of patterns used to control and monitor the messaging system and the components.

Component Type	Problem solved
Control Bus	When multiple components must be controlled in a distributed system across multiple geographies and systems, to control Configuration of components, Heartbeat to control metrics and keep alive behaviour, Test messages, Exceptions, collect Statistics or show a Dashboard, use a control bus that connects specific channels to each component, separated from the business channels.
Detour	In cases where there is a need to route or modify messages to perform testing, debugging or validation, we may need detour the messages to treat them. A detour is a composed component that uses a context-

	based router controlled by a Control bus, so it can detour messages in a controlled way through the normal business logic or through the debug/testing logic.
Wire Trap	In a Point-to-Point channel only one listener gets the messages, so to allow monitoring, testing or troubleshooting, an inspection must be performed, but it cannot be added another listener. Thus, a Wire Trap is used. A wire Trap is a Recipient List with one input from a point-to-point channel and 2 output, one for the usual path and other for the testing/debugging path. A wire trap cannot modify a message.
Message History	This technique helps analysing and debugging the flow of messages in a loosely coupled system. This involves attaching to the message header each Component ID the service traversed in an ordered way.
Message Store	In case reporting shall be done, a Message History is useful but not enough as the message may not be available when it is needed. So, capturing messages in a Store in a central location keeps them available for late reporting or analysis.
Smart Proxy	When track of Request-Response messages must be performed is hard to achieve as the sender A and receiver B put the addresses of each destination. So, to hack the system, a smart proxy replaces the reply destination address B with its own address C. When the receiver B replies, sends it to the smart proxy as reply indicates. Once the smart proxy receives the reply it replaces its address and reroutes the message to the original sender A address.
Test message	Controlling the status of the messaging system cannot be covered by heartbeat message with metrics reported, as the messages delivered by a component could be broken due to internal fault, while the heartbeat shows a good health of the component. This is because the control does not operate at the application-level message format. Thus, to check the wellness of the system at application level, the control layer must issue Test messages that must be processed by the component and assure the health of the component. To issue Test messages must be used several components along the infrastructure.
Channel Purger	When testing, so many times a messaging system can be wake up or down, so it is usual that old messages are waiting to be delivered next time the system gets loaded. In testing situations, this can produce tricky behaviours, thus, messages should be removed from channels before.

Table 4 System Management components

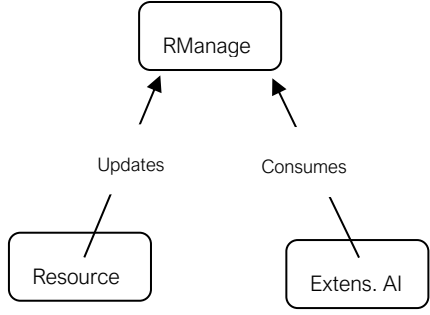
Some patterns from the System Management section could be used to help implementing the service bus, but the technologies under review, such as Kafka, could include some of them while others not.

5.2.9 A simple method to apply patterns

Working with patterns it is not an easy task and experience is a plus to do the task of recognizing the problems and apply the solutions.

Here, we recover the method described by Hohpe⁵ to work over a problem.

The method is not exhaustive but can be used as a first approach.

Task	Description	Example
Draw the main systems	To contextualize the systems to be integrated draw them in a high-level diagram with arrows representing the high level flow of information. Subsystems will flourish in the following steps	
Describe the data flow	To detect the logic of the process and actions, we write in a high level the data flow process and actions performed with the data by each system.	The sensors detect the temperature, location and CO2 and the Ext. AI modules consumes it as arrives to be aware of the situation. Temperature is sent each second but actually Ai does not need it so frequently, but location and CO2 must be measured and consumed quite fast. The resource manager must handle all types of sensors. Sensors can be connected or disconnected sometimes. New sensors can be setup at any time plug-play, or disappear
Describe the requirements	The high-level requirements may impose constrains that drives the messaging system boundaries	Sensor, temperature, location, CO2, Ext. AI modules, situation, temperature not measured or consumed frequently, CO2 and location must be ready fast. Plug-play, disconnected, etc.
Analise or detect verbs, nouns, adjectives, and entities	The verbs will be the actions, nouns and entities will probably become subsystems, and adjectives may become attributes or constrains that help deciding what to use	The data coming from sensors goes to the different AI modules with a low latency when the AI needs it, so and RPC style is not suitable as many AI modules would require too many 1to1 connections, and File sharing is also a bad idea as we require low latency. Messaging and Pub-Sub style could fit very well.

⁵ Gregor Hohpe, Bobby Woolf. SBN 0321200683 Addison-Wesley, 2004, Enterprise Integration Patterns. Page 34

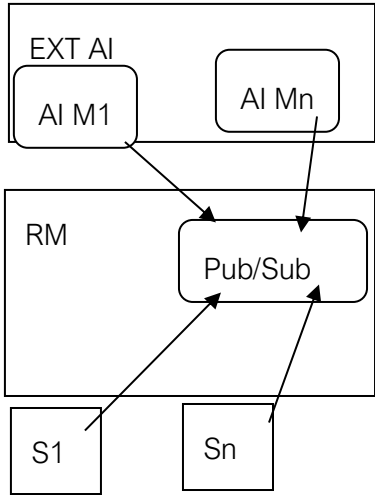
Let subsystems appear in the drawing.	Take the systems and draw the subsystems, taking into account patterns from the requirements analysed	
Think about the information you have to detect patterns	Read the information again, inspect the drawings and dataflows, take into account constraints and attributes, what is missing to achieve your objective?	<p>The sensors send the information with one format each one.</p> <p>We can use a Canonical Model in the RM (and the rest of the platform), so all the AI modules share the same model. We need to use translators for each Sensor and a definition of the canonical model</p>
Try to find problems with the patterns inserted and refine	Some patterns may introduce problems and iterating over the design can help discovering problems and fixing them	<p>Temperature sends too many updates and could overflow and waste resources</p> <p>As the temperature is consumed so unfrequently, it can be used a Filter after the temperature sensor to drop Temperature updates that come so close in time. This way, the system can modify the reading frequency.</p>

Table 5 Applying patterns method

Within this section we have described a light way method to find problems so we can map a pattern to solve it.

This method can be applied for each flow of data among different entities to refine the patterns needed.

5.3 Micro-Service architecture

5.3.1 Architecture components

Microservices are an architectural approach to building applications. As an architectural framework, microservices are distributed and loosely coupled, so team changes don't corrupt the entire application. The benefit of using microservices is that development teams can create new application components to accommodate ever-changing business needs.

What distinguishes the microservices-based architecture from traditional monolithic approaches is the breakdown of the app into its basic functions. Each function, called a service, can be compiled and deployed independently. Therefore, individual services may or may not work without compromising the others. This accommodates the technological aspect of DevOps and makes iteration and constant distribution (CI / CD) easier and feasible.

This section will focus two types of advanced micro-service architectures that are important for ODIN. First one is the serverless architecture, and the second one is mesh service.

5.3.1.1 Serverless

Serverless computing is a cloud-native development model that allows developers to build and run applications without managing servers that are using resources only when executing (scale to zero).

While servers are used in this model anyway, they are abstracted from app development. The routine tasks for provisioning, maintaining, and scaling the server infrastructure are handled by a cloud service provider. Developers simply need to package code inside containers for deployment.

After deployment, serverless apps respond to requests and automatically adapt to different scalability needs. The utilization of serverless solutions offered by public cloud providers is usually measured on-demand through an event-based execution model, so serverless functions cost nothing when not in use.

The serverless architecture is perfect for asynchronous stateless applications that can be started instantly. The serverless model is also optimal for usage scenarios that involve sudden and unpredictable spikes in demand.

Examples include batch processing of incoming image files, which may be required infrequently but must be available when a large batch of images arrives to be processed at the same time, or monitoring changes to a database, which require the application of a set of functions, such as checks to verify that they meet quality standards or their automatic conversion.

Serverless apps are also suitable for use scenarios involving inbound data streams, chatbots, scheduled tasks, or business logic.

Other common use scenarios for the model include backend APIs and web apps, business process automation, serverless websites, and cross-system integration.

5.3.1.2 Service mesh

A service mesh allows you to control how different components of an application share data. Unlike other systems for managing communication between services, a service mesh constitutes an infrastructural layer integrated directly into the app. This visible level can document the quality of the interaction between its various components, optimizing communication and avoiding downtime due to the expansion of the app itself.

Modern applications are often broken down into a network of services, each of which performs a specific function. Each part or "service" of an application relies on other services to give users what they want. To perform its function, a service may need to request data from several other services. But what if some services are overloaded with requests for data from other services, such as the reseller's inventory database? To manage this overload, a service mesh is used, which redirects requests from one service to another, balancing the overall load of all elements.

A microservices architecture allows developers to make changes to an app's services without having to redistribute it in its entirety. Unlike app development in other types of architectures, individual microservices are built by small teams who can choose which tools and which coding languages to use. The microservices while communicating with each other are independent. This means that the failure of one of them does not lead to the failure of the entire application.

Service to service communication is essential for microservices to function. The logic behind the communication can be coded into each service without the need for a service mesh. However, a service mesh becomes useful as communication starts to get more complex. For cloud-native applications integrated into a microservices architecture, a service mesh allows you to include a large number of "small" services in a functional application.

A service mesh does not introduce new functionality into an application's runtime environment applications of any type of architecture have always needed rules that specify how requests move from one point to another. A service mesh, however, extrapolates the logic that governs service-to-service communication from individual services and abstracts it into an infrastructure layer.

This is why a service mesh is integrated into an app as a set of network proxies. Proxies are a familiar concept to enterprise IT.

In a service mesh, requests are routed between microservices through proxies present in the infrastructure layer. This is why the individual proxies that make up a service mesh are sometimes called "sidecars": they move alongside each service, rather than within them. All these "sidecar" proxies, decoupled from each service, together form a mesh network.

Without a service mesh, each microservice must be coded using logic that governs service-to-service communication. This prevents developers from not only focusing on business goals but also diagnosing any communication problems, as the logic that governs communication between services is hidden within each service.

With a service mesh, developers can focus on increasing business value rather than connecting services.

5.3.2 How it works

A microservice is a basic function of an application, which runs independently of the other services. However, the microservices-based architecture not only involves the low coupling between the basic functions of an app but proposes a restructuring of the development teams and the communication framework between the services. This approach offers the ability to manage unavoidable criticalities, supports dynamic scalability, and facilitates the integration of new features.

Microservices can communicate with each other, typically stateless, allowing apps to be built with greater fault tolerance and less dependent on a single ESB. In addition, they communicate via language-independent application programming interfaces (APIs), allowing development teams to choose their tools.

Considering the evolution of SOA, microservices are not an absolute novelty, but lately, they have become more attractive thanks to advances in containerization technologies. Today, Linux containers allow you to run multiple parts of an app independently, on the same hardware, with much greater control over individual components and lifecycles. Combined with the DevOps APIs and teams, containerized microservices form the foundation of cloud-native applications.

5.3.3 Pros and Cons

Some advantages of platforms based on a distributed architecture, microservices allow for more efficient development and routines. The ability to develop multiple microservices at the same time allows multiple developers to work on the same app simultaneously, reducing development time.

- **Faster time-to-market:** By enabling shortened development cycles, a microservices-based architecture supports more agile deployment and updates.
- **More Scalability:** As demand for certain services increases, microservices can be deployed across multiple servers and infrastructures, depending on business needs.
- **Resilience:** Each service, when built correctly, is independent and does not affect other services in the infrastructure. Consequently, any component failure does not cause the entire app to crash, as happens with the monolithic model.
- **Easy deployment:** Because microservice-based apps are smaller and more modular than traditional monolithic applications, all the problems associated with such deployments are automatically eliminated. While this approach requires superior coordination, for which a service mesh layer can be useful, the benefits that come with it are critical.
- **Accessibility:** As larger apps are broken down into smaller parts, it is much easier for developers to understand, update and improve those components, which accelerates development cycles, especially in combination with agile development methodologies.

On the other hand, there exist also drawbacks. To move to an architecture based on microservices, it is essential to be able to change the structure of communication and collaboration between teams, not just that of the apps. Changing the corporate culture can be difficult, in part because each team follows its deployment cadence and is responsible for a unique service, for a particular group of customers. While not strictly a developer issue, overcoming it becomes essential to the success of the microservices-based architecture.

In addition to these, a microservices-based architecture has two main challenges: complexity and productivity. About these two criticalities, eight categories of problems have been identified:

- **Compilation:** Spend time identifying dependencies between services. Because of these dependencies, when you run one build, you may need to run many more as well. It is also essential to consider the effects produced by microservices on data.
- **Testing:** Integration testing, as well as end-to-end testing, can become very difficult, but more important than ever. Remember that a failure in one part of the architecture could cause a failure in a nearby component, depending on how the services are structured to support each other.
- **Version management -** Upgrading to a new version may affect backwards compatibility. The problem can be handled through conditional logic, but in a short time it can become difficult to handle. Having multiple live versions for different clients can be an alternative, but maintenance and management can be much more labour-intensive.
- **Deployment:** During initial setup, this is also a critical step. To simplify deployment, you must first invest heavily in automation solutions. The complexity of microservices would make manual deployment extremely difficult. Think about how and in which order to roll out services.
- **Logging:** in distributed systems, centralized registers are required to reconnect all the various components, without which managing them in a scalable way would be impossible.

- Monitoring: It is essential for operations teams to have centralized visibility of the system in order to identify the source of problems.
- Debugging: Remote debugging via the local integrated development environment (IDE) is not a viable choice with dozens or hundreds of services. There is currently no single solution related to debugging.
- Connectivity: it is necessary to evaluate which method of discovery of the services to choose, if centralized or integrated and how to connect them together.

5.4 Enterprise Service Bus architecture

The Enterprise Service Bus is an architecture that recovers the best practices from B2B, SOA, EAI, web services and many other architecture trends to address the problem of adapting the systems technical solutions to the business needs, as long as they evolve.

Loosely coupling, secure, scalable, configurable, manageable, auditable, reliable and integrable are only a few attributes an ESB must follow with the aim of integrating a heterogeneous field of applications. This integration is not only interesting among intra enterprise applications but can be extended to applications out of the boundaries of an enterprise, creating the concept of the extended enterprise.

An ESB architecture is far beyond of a couple of web services adding connectivity to a messaging broker or EAI solution. A good introduction to application integration evolution toward ESB, is presented in “Enterprise Service Bus” by David A Chappell⁶. As a summary of his ideas, initial integration approaches looked for a hub and spoke schema, where a central entity, the hub, oversaw routing all the data among the applications connected to it and took all the decisions. This was an easy schema but faced some drawbacks on scalability when the number of applications connected increased; and most important, business functions had no clear separation in the hub entity. Examples of this kind of integration were application servers, with a monolithic approach where the integration logic and the application logic were coupled.

An evolution to get a more decoupled applications of the routing logic was the EAI approach, which still was a hub and spoke model. EAI was better than the previous but still had scalation and business boundary problems.

Other solutions went through an enhancement of the distributed integration approach, such as messaging middleware that offer a very easy integration strategy with very little code, but in the early days, messaging middleware lacked routing capabilities, so depending on the

⁶ Enterprise Service Bus” By David A Chappell . Oreilly

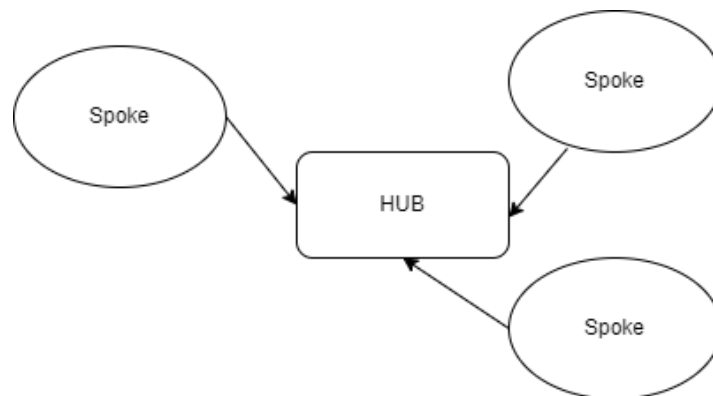


Figure 6 Hub and spoke

implementation, a lot of routing logic code could get coupled with the applications and services integrated.

ESB follows an approach where the integrated applications and the logic that glues the integration is separated, and the integration follows a distributed strategy. ESB took a new paradigm, where using a messaging middleware architecture, services could be configured rather than programmed, so information flow could be managed, distributing the integration, and decoupling the routing logic from the applications and services.

5.4.1 Architecture components

Enterprise Service Bus architecture can include a vast number of components, a lightweight approach to an ESB can be summarized from the following picture:

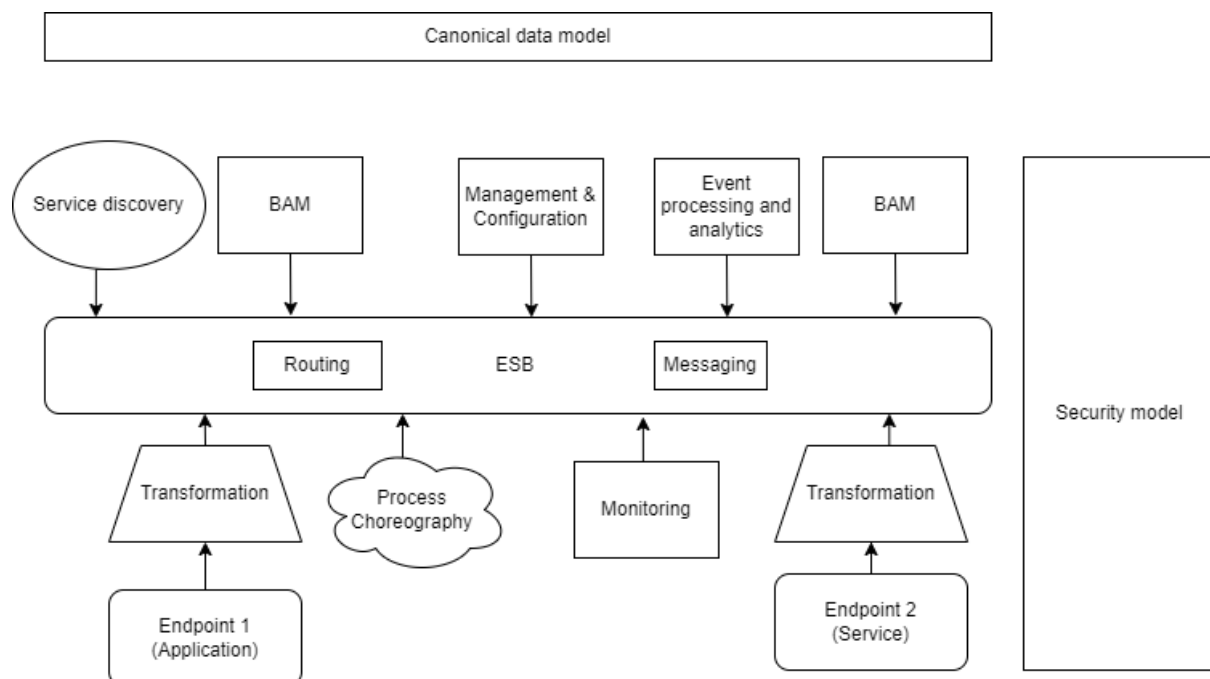


Figure 7 ESB Architecture components

Support components for transport protocols. Those components allow to connect external applications or systems to the ESB working with different transport protocols such as JMS, HTTP, FIX, FTP, Mail and many others.

A common language and canonical model to integrate all datatypes and logic. This is the root of the integration, as it allows to bring common semantics from all the systems to be integrated to the core of the ESB. Usually, XML or JSON are implementations of the language used, and the tool to create a canonical model.

A transformation layer to adapt and enrich external data to comply with internal canonical model and vice versa, plus a mechanism to route and forward the information to the right destination or filter it. Routing can be based on static or dynamic rules, event triggered, content triggered and many other options.

A Message Oriented Middleware to connect all the systems. Integrated systems must remain decoupled but connected and reliable. Moreover, the systems integrated must work without interruptions and scaling well under heavy loads, which usually it is achieved using asynchronous interactions with distributed services. Thus, using a messaging broker is a cool way to achieve all those objectives. For instance, pub/sub approaches work very well, using topics, messages, producers, and consumers. Although ESB promotes asynchronous interactions, that does not mean it does not support synchronous interactions.

A service-oriented architecture through endpoints that encapsulate application or services. This means, systems are connected to the ESB to support clear decoupled functions or services.

A registry to allow service and application discovery. This component helps publishing service addresses so other services or the EDB services can use them and is related to the routing ESB capabilities.

In order to manage, configure and monitor the ESB, several components are available to control the ESB. Management & Configuration plus Monitor components take care of this.

Another important component of an ESB is the complex event and processing analytics. This component takes care of events happening in the ESB, looking for patterns and triggering other events and alerts. It is quite related to the monitor component.

Beyond ESB monitoring there exists the Business Application Monitoring (BAM) component, which take care of business performance, offering KPI and other monitoring functions related to business. Section 6.9 dives into this component more in depth.

A component to implement complex business logic, that is Process Choreography. The ESB provides the means to define logic that can be achieved using the services available in the connected systems. The objective is to decouple logic from services. Usually, business process languages and definitions can be used to define workflows to work with this component.

Finally, a Security model keeps track of the facts around audits, access control, data protection, confidentiality and authenticity.

In fact, it can be seen that the ESB is what exactly ODIN's platform is looking for as the requirement section states, and the components shown in the example architecture are exactly a possible realization of ODIN's architecture as D3.10 finally has presented.

5.4.2 How it works

It is quite challenging to explain how an ESB works as it is a full ecosystem of several services and applications. Due to this, an example will be explained, where several applications interact and use the ESB services.

The use case can be summarized as follows:

One hospital wants to control medical devices position to track them and study how they are used and forecast probable situations to be taken into account to avoid running out of devices. For example, detect when the hospital must buy more devices with time ahead, or just move them from one service to another.

The system which provides real time location of medical devices such as pumps or echographers is an IoT system with tags sending Bluetooth signals to gateways that read signal power and other means, information that must be used by the location engine to compute the real time location of a device. The location engine uses a simple data format, with an identifier of the locator and the position.

The identifier is meaningless without knowing the real device, and this is important to be understood by the medical staff.

The position of the devices will be analysed using an AI algorithm, which read information about medical device usage, number of patients in the hospitals and historical data for seasonal behaviours.

The information will be presented to the medical staff using nice boards.

All the system must work with a common language to foster and ease future integrations.

Taking those requirements and using an ESB architecture, let see how this could be achieved.

The architecture could be like the following, using a microservice approach, connecting the key services to the ESB.

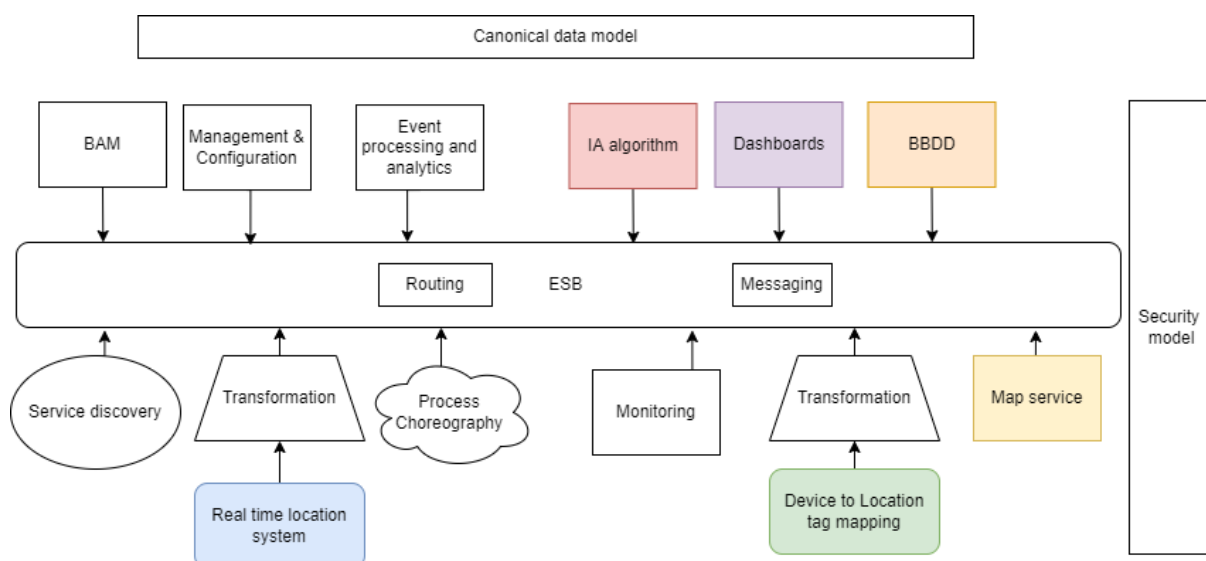


Figure 8 ESB working example

The RTLS systems connected to the ESB through a Transformation component, which will enrich and traduce the simple location format to a standard format of the ESB with information about the real device. The information about the real device can be extracted from the “Device to Location tag Mapping” system, which has information about the location tags and the devices itself. This decouples the configuration system from the RTLS itself, thus in case the hospital needs to change tags from one device to another, it can be handled in a separated service without mixing implementation details of the device information with the RTLS logic.

Once the information about the device position is transformed into the canonical model (standard message) can enter to the ESB to be used by other services. The first place will be the Map

Service, to compute the actual room location inside the hospital, because positions are not useful to get insights from the AI algorithm, as are not meaningful to the medical staff.

Once the rooms are computed, the results are put available to the ESB so other services can use it. For example, the BBDD will store those room locations with the device identifications so they can be used by the AI algorithm.

The Process Choreography component is the actual director of the process, and at some point, it will signal the AI Algorithm to start analysing the device usage and forecast the needs for each service, recording the results in the BBDD service.

Once the results are available, the Process Choreography component will send an e-mail to the medical staff.

The medical staff will read the e-mail, and using the Dashboard Service, will review the results. The Dashboard Service will read the information from the BBDD service and present it to the medical staff, which will take some decisions using the insights.

This smart example gives an overview of how things can work using an ESB, but more detail can be reviewed about other components that have not been already mentioned.

For example, the Process Choreographer and all the services, can reach and discover the services available in the ESB thanks to the Service Discovery, that keeps track of the services running.

The Monitor component can gather information about the systems and applications and store it to be processed in other services, such as the BAM module. This module processes the data about business application, for example the timing of the reports generated by the IA and in case something is not working ok, warn the IT staff from the hospital.

The event processing and Analytics could monitor real time data to control the status of the devices and signal warnings in case some of them stop sending the position.

In case the medical staff wants a new functionality, to display actual device position, using the Management and Configuration module the ESB could be set up to accommodate new services, such as a new dashboard and new routing functionality, so the Dashboard can access the data generated by the Map Service directly. Then the Device Position dashboard could display device position in real time, if data from the Map Service is generated.

With those examples this section is closed, expecting to have explained how ESB work and how it can be used.

5.4.3 Pros and Cons

A list of top important ESB pro's follows:

- Pervasiveness. An ESB is one of the most complex solutions to span an integration strategy inside an enterprise, becoming the root of integration.
- Highly distributed, event-driven SOA. The ESB architecture fosters loosely coupled integration components, and due to its capabilities can be widely distributed along different geographically locations, accessing shared services from anywhere on the bus.
- Scalability. More services and ESB components can be added as needed, due to new functionality integration or clustering needs to support more working loads.
- Security and reliability. The ESB supports reliable messaging with transactional integrity, and secure authenticated communications.

- Orchestration and process flow. An ESB allows data to flow across any applications and services that are plugged into the bus, whether local or remote, and what is more important, uses configuration over coding integration.
- Autonomous departments with managed environment. An ESB allows local autonomy at functional/business unit level but also allows to integrate in a larger managed integration environment.
- Incremental adoption. An ESB can be used for small projects. The integration process can be managed in iterations, where new systems are added.
- Real-time insight. An ESB provides real time information about the business, something that was not possible when batch integrations using daily FTP data transfers.
- Deployment options. Modern ESB can be deployed on-premises, cloud, hosted, and hybrid cloud, adopting the best solution for each scenario.

Some of the top con's a ESB has are:

- Single point of failure. Depending on the implementation, an ESB can be a point of failure that could end on isolated services, stopping all the functionality created across the ESB. This can be minimized using redundancy techniques.
- ESB can be overkilling. For some types of integrations, an ESB can be overkilling. As a rule of thumb, it is quite accepted that integrating less than 3 services through an ESB is a waste of time, moreover if the system is not going to grow with more services or need to scale in the future.
- ESB are complex. Adopting an ESB can be challenging, not only by its complexity but the fact ESB are complex to program or setup, so people developing a solution using a ESB usually need 3 to 5 years training. Anyway, modern ESB have evolved and include visual tools, known as no code tools, to manage them without programming and learning curve has been reduced.

5.5 Resource Gateway and Resource architectures

This section tackles some of the architectures that could be used into the ODIN's Resource Gateway, which is a component quite abstract, but its main duties can be summarize into: connect all the resources to the platform, transport and transform the data around the platform, and allow other applications/services to access the resources and the data. Actually, the Resource Gateway could be a network gateway plus some components more such as the ESB, but Task4.1 wanted to study whether it could be used an IoT platform to solve several problems and implement more components at once. Those requirements look like to the IoT platform 3-Layer Architecture of Figure 9 IoT layered architectures, which includes Perception, Network and Application layers, described in the work driven by WP2 on D2.2 Hospital Requirements report. Thus, the Resource Gateway can be seen as a part of an IoT like platform with extended functionalities, as other means of sources beyond IoT and Robots.

To ease the reading of this document, here it is included the layered approaches for an IoT platform. Check D2.2 Hospital Requirement Report, Section 3.1.1 Main architectures & Data Processing for more information.

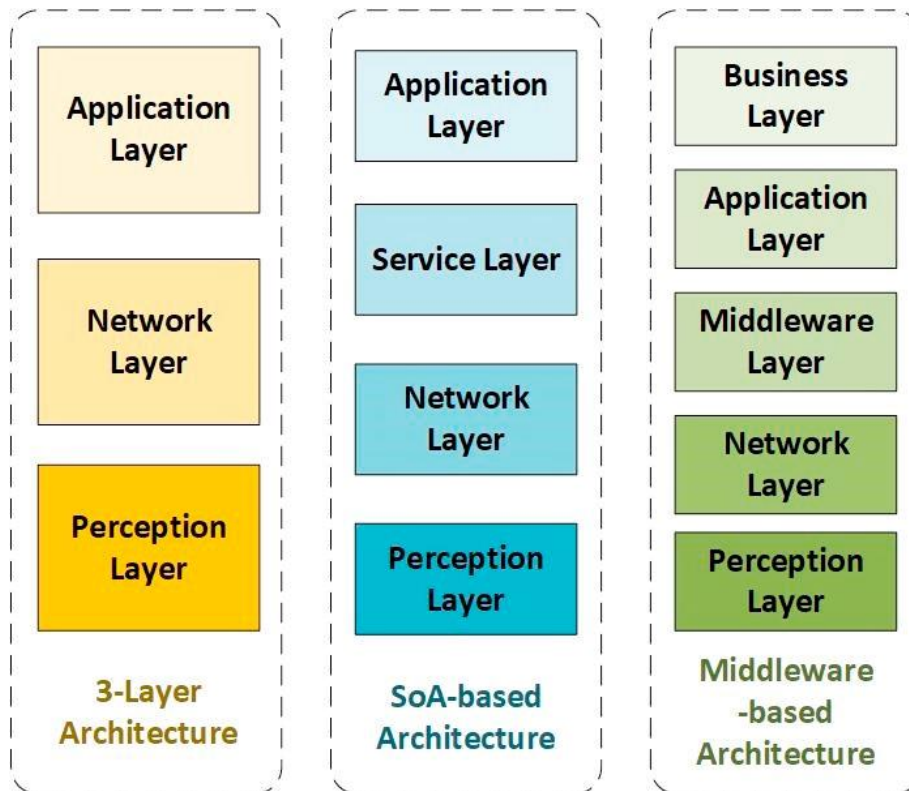


Figure 9 IoT layered architectures

The first topology has been introduced in the previous paragraph, and to understand the other two, the SoA (Service oriented Architecture) includes the Service Layer that is responsible of servicing the Application Layer, managing service discovery, management and orchestration to compose complex services from the smaller ones.

Finally, the Middleware-based Architecture replaces the Service Layer by the Middleware Layer, taking its responsibilities and managing infrastructure and easing Application creation. The Application Layer is complemented with a Business Layer, that adds value to the concrete enterprise where the IoT platform is used, providing tools to understand how it is going, learn and act.

ODIN Platform can be seen as a Middleware based architecture, where the CPS-IoT could be the first bottom 3 layers (Middleware, Network and Perception), and the Resource Gateway would be the first two, from the bottom, and part of the Middleware.

Once it has been pointed out the Resource Gateway can be seen as an IoT or part of an IoT architecture, let's review what it is an IoT architecture.

An IoT architecture is the one that helps connecting things to other things or applications in order to enhance operation costs, automate information handling and analysis, improve resource use and enhance people security, to mention a few, without human intervention.

When hundreds or thousands of devices have to be managed, in person manual handling is a waste of time, error prone and insecure. Oil and gas pipes systems are a good example where security is enhanced, and operating costs are saved, as for example reading in a remote fashion all the counters to bill gas customers or control valves in the distribution system are more efficient in term of time and cost and secure using IoT, as no one has to move to perform the tasks and are less error prone.

Measuring all the information to take decisions for agriculture management, involves lots of sources, so automating the information from sensors, weather forecasts, market prices and other sources, becomes a must to take advantage over competitors.

From Tech Target⁷: The internet of things, or IoT, is a system of interrelated computing devices, mechanical and digital machines, objects, animals, or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction

So, to summarize, IoT helps connecting things to share information, process it and perform tasks without human intervention, but it can be seen it handles more problems than just a Resource Gateway

5.5.1 Architecture components

The following image, from the EdgeX documentation⁸ (an Edge IoT ecosystem enabler), covers most of the IoT available topologies with the base components. The first example is a simple and elegant Cloud IoT where information goes directly to the cloud, and the rest are more complex architectures toward Edge IoT platforms that deliver computing power to the Edge or in other words, close to where the devices are located. Edge is also known as Fog in some literacy.

For advantages of one type or another, check D2.2 Hospital Requirement Report, Section 3.1.1 Main architectures & Data Processing.

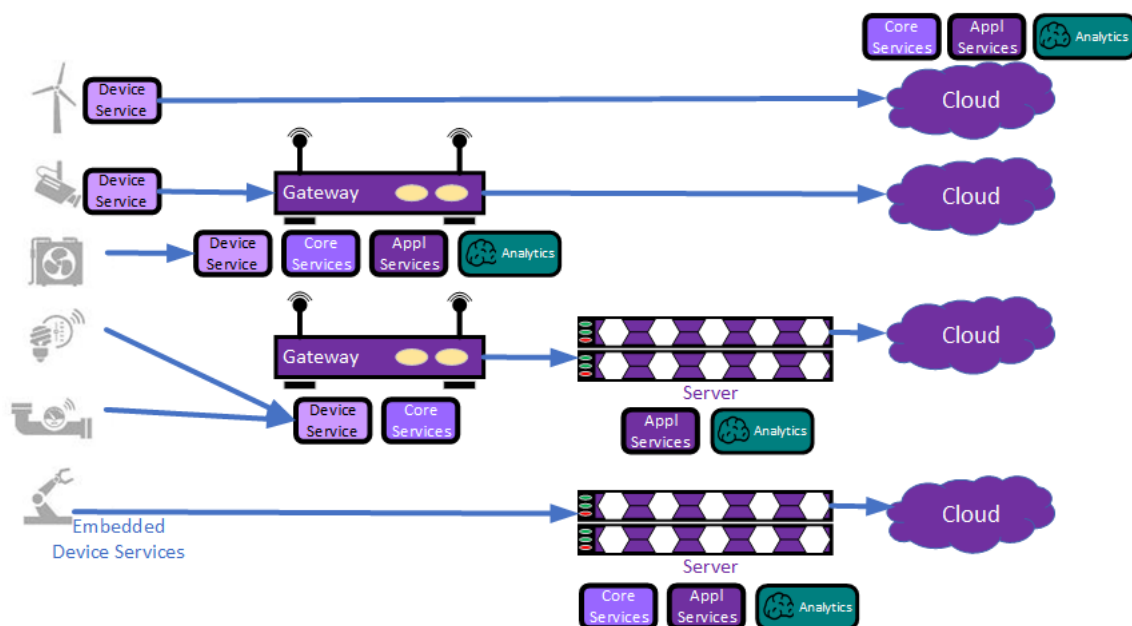


Figure 10 IoT topologies

So, depending on the IoT architecture selected and mapping to ODIN's needs, ODIN's Resource Gateway could be the Cloud, the Gateway/Broker or the Server, but it must be pointed out that

⁷ <https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT>

⁸ https://docs.edgexfoundry.org/2.2/general/EdgeX_deployments.png

actually, ODIN's Resource Gateway needs are closer to a plain network gateway which internal services just control access to the resources, and no other special AI services run on it.

Reviewing the components, the Devices are the most important components, they can measure things and perform actions, and are connected to an IoT ecosystem. Without them IoT is meaningless. There exist Devices with different complexities, with more or less computing power and services, and more or less standardized connectivity. Also, the Device Service which is very close and related to the device, the Device Service is the component that allows the connectivity of a Device to a concrete platform, so the rest of the platform understands the data and tasks that the Device can perform and act over it. Device Service could be seen as the Connectors presented in D3.10 ODIN Platform, serving as transport and semantic services.

Between the device and the platform, there is the Broker. The Devices need to connect to the IoT platform through a broker, which depending on the topology and other facts, it can be the Cloud directly, a central Server or a Gateway in case it is a very close-range network topology. For ODIN's purpose, a Resource Gateway using third architecture presented, could be enough.

Finally, the Application components. Core Services, Application Services and Analytics, should be reviewed as Applications as a whole, although Core applications perform tasks for the IoT Platform, such as routing, orchestration, management and so on. Services aim to deliver value for external applications, out of the IoT platform. Finally Analytic applications deliver learning and monitoring capabilities to the platform or other services. Depending on the grade of Edge oriented platform, that is, more or less computing on the edge of the platform, the Applications and services may be in the Cloud, the Server or the Gateways, from less to more edge degree. This part of the architecture could resemble Core services, ODIN AI services and so on.

5.5.2 How it works?

Depending on the implementations, the behaviour may change a little but in essence, and taking the following architecture which could be the closest to ODIN, an example is developed.

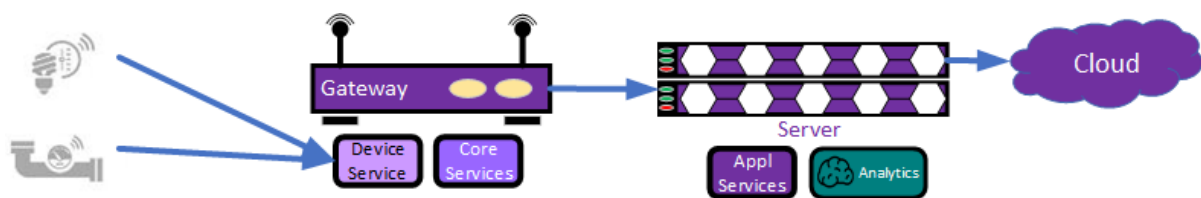


Figure 11 IoT architecture example

The example considers two sensor system types for water waste and light consumption for an hydroelectric plant.

The architecture adopted is Edge like, with gateway (i.e Resource Gateway and ESB) and Edge Server (Odin AI and Core services, etc). It can be considered that cloud is used also for some application services and some data storage for business cases.

The systems are somewhat standard, as they can send data metered using Modbus TCP via an Ethernet network. Modbus is a transport standard for industrial communication.

The sensor system is connected using Ethernet to the Gateway, but the data is not compatible with the IoT platform data model. Thus, the Devices Service component receives sensor data and transforms it to the platform data model.

Once the data received is transformed, the Core Services module inspects the data and decides the additional data transformations and routing that must be performed. Because there are no Analytics or Application services at the Gateway level, the Core Services stores locally the data

for 24 hours. This will allow to recover from disasters in case the data cannot travel to the Edge Server or the Cloud, as there are intermittent communication shortcuts among the Gateway and the Edge Server, which are connected using WiMax. Because the data must be analysed, the Gateway sends the data to the Edge Server.

The Edge server receives the data and the Analytics Module controls that all the rules defined by the engineers are ok. Once the data is processed, some results are sent to the Cloud, where the information is stored and put available for the Business Units of the company, that operate in several countries.

Some results are strange to the analytic process and violate some rules at the Edge Server, so an alarm is triggered, and the engineers receive the alert on the dashboards they have available from the Edge Server Application Services module. The engineers start to check the results, but they are very relaxed as the Edge Server has an automated AI application that will perform a correction in the water flow pipe, sending commands to the water pipe system, using the Control Application Service from the Edge Server and monitor the electric consumption until the alarm stops.

At the end of the month, a report is generated in the Cloud and sent to all the Business Units to be reviewed to take actions. The profit has dropped 3 points the last 3 months and that is inadmissible.

This is a very simplified example of the things that can be monitored, processed and commanded using an IoT platform, but it is quite clear that using an IoT platform it can be covered a platform like ODIN, but is beyond what a Resource Gateway needs

5.5.3 Pros and Cons

A brief list of advantages to use an IoT platform are:

- Holistic perception through interoperability. Is the basic added value, as it can connect a wide range of devices to sense the environment, and what is more important, provides interoperability among sensors and services.
- Device management and security. IoT platforms usually includes facilities to manage sensors and devices in a centralized way using secure mechanism, which eases operations and increases security. For example, a rogue device will not be able to connect to the system, or a hijacked device, can be disconnected form the platform.
- A 360° degree automation of control and tasks come to reality of cognition with IoT platforms. Detecting information and processing it at the right point gives access to decision making at the right moment. If AI or rule-based automation is added, the IoT platform can control the resources to correct problems, recommend actions, etc. This enables a dynamic platform that can respond to the environment.
- Information distillation and information management. Using and Edge approach, each level of the platform can deal with the right amount of data. Sensor layers manage a lot of data but unless processed may not be useful for upper layers or applications. This creates a funnel of data that is converted to information with the minimal latency where is needed.
- Scalability and modularity. IoT platforms provide scalability for operational and business objectives. The solution provides means to expand the number of devices, data and services to be managed, providing the ability to integrate more modules as long as they are needed.

On the other side, some disadvantages are:

- Integration. Usually, to add a device which has not been deployed before in the platform or that is not standard, may need some work to integrate it.
- Size matters and one size does not fit all. IoT platform solutions cover a wide number of scenarios, but usually are focused on a type of architecture or mode. For example, Microsoft, Amazon and Google do have IoT Cloud platforms, which are of for deployments where latency, and general costs are not a problem, but in some scenarios are not the best approach, such as hospitals for example. Thus, a careful planning of the needs may be carried on before a solution is selected.
- Security. Security is increased as a whole, but must be handled in the right way, because it can turn into insecurity. When a system does not go beyond the premisses of a fabric, is more secure than one that opens to external connections. In that sense, a careful approach to security must be taken.
- ODIN can be seen as an IoT platform but using an IoT platform from scratch may not fit ODIN's needs.
- Resource Gateway is not exactly an IoT Gateway

As a result, an IoT architecture may not suit ODIN platform and it gets clear a Resource Gateway can be implemented with less effort that using an IoT platform.

5.6 Distributed Ledger Technologies architectures

Distributed Ledger Technologies (DLT) are employed in ODIN to accomplish two major functionalities: provide immutable auditing of sensitive data transactions and handle permissions and consents to enable secure resource federation.

Immutable auditing is related to the requirement that the functionalities of ODIN must ensure privacy, security and trust (PST). The organizations involved in ODIN, primarily hospitals, handle sensitive patient data that must be secured against unauthorized access, while ensuring the accountability of any party that has access to them. While unauthorized access is handled through user authentication and encryption mechanisms, DLT addresses the accountability requirement, by logging every transaction involving sensitive data (e.g. reading data from a medical database, or writing data to it) in an *immutable* manner, i.e. in a way that cannot be altered after the transaction has been made without all involved parties being aware of it. This is important for the involved organizations to trust each other, or the ODIN components running within the organization.

Resource federation is one of the major goals of the ODIN project, which envisions implementing a decentralized platform, where resources (data, IoT, robots, AI, services) are provided by edge ODIN instances that run in different hospitals or cloud environments, and are shared with each other. The role of DLT in this respect is to ensure that resource sharing between two parties is allowed by both of them, unquestionably, and that the party that shares the resources has provided consent to the other party to use the resources. The immutability and traceability features provided by DLT ensure that resource sharing is performed in a trustworthy manner.

5.6.1 Architecture components

Architecturally, the DLT forms a separate component that is connected to the main Enterprise Service Bus (ESB), through which it can communicate with the other components, as seen in Figure 12. Through the ESB, the DLT component can intervene in transactions of data between the Hospital Information System and other components of the same ODIN instance, in order to provide auditability of all transactions.

Resource federation between one ODIN instance and other ODIN instances (e.g. other hospitals) is achieved through the ESB, with the DLT intervening to address permission and consent handling. When another ODIN instance needs to use a local resource of the current ODIN instance, the request first passes through an associated security module that authorizes the remote ODIN instance. Then the request is handled by the DLT module, which makes sure that the remote instance has permission to use the specific local resources. The DLT module authorizes the resource sharing, and then the remote party can use the local resources, by exchanging direct messages through the bus, while the DLT audits the transactions made.

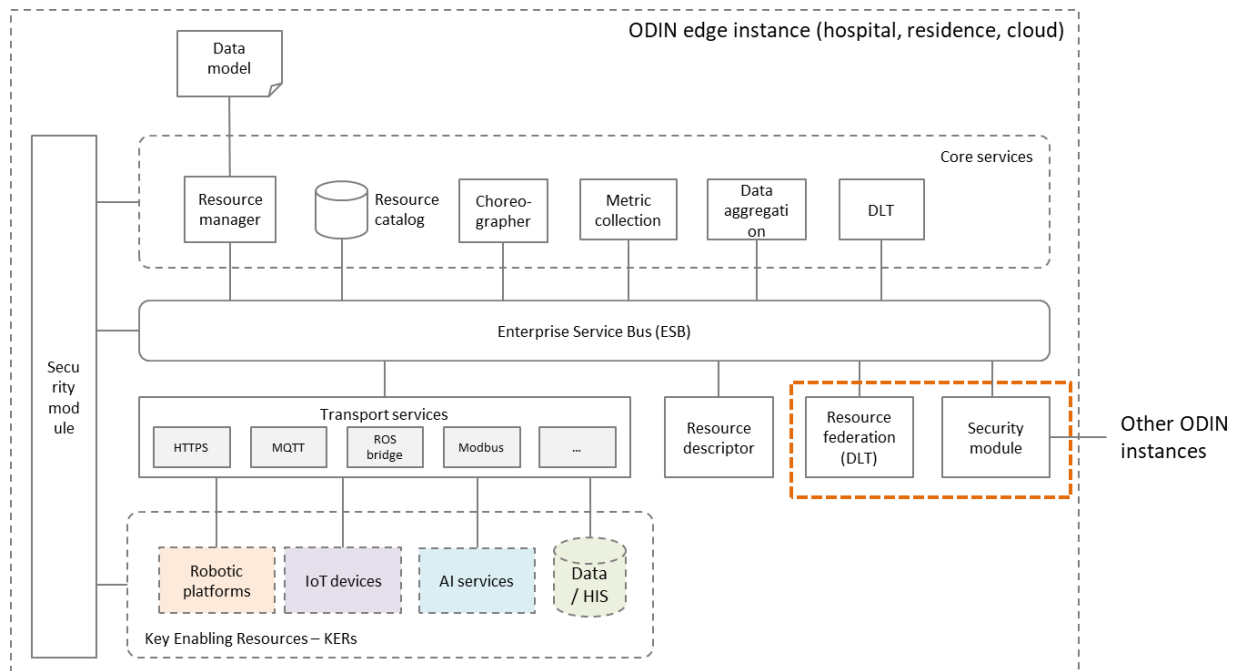


Figure 12: Positioning of DLT resource federation within the ODIN platform architecture.

The DLT is implemented as a blockchain, primarily hosted in the ODIN cloud infrastructure, which supports the necessary chain nodes. If an organization wishes to participate as a node in the blockchain, and if they have the required processing resources, they can be connected as a node in the blockchain, obtaining a local copy of the distributed transaction ledger and enhancing the immutability and traceability functionalities of the DLT.

5.6.2 How it works

Secure resource sharing is split into two main actions: a) management of organizational consent, and b) resource sharing and auditing.

Whenever a remote ODIN instance wishes to use local resources, a request for permission is made, which is handled by the local resource federation (RF) component. The RF component communicates with the resource catalogue to query about the characteristics of the requested resources (e.g. if they are public or not). Through the component's user interface, the manager of the local organization can grant or refuse permission to the requested resources. If permission is granted, a notification is sent to the requester, while the permission metadata (requester, timestamp, involved resources, etc.) are logged in the blockchain. This process is depicted in Figure 13.

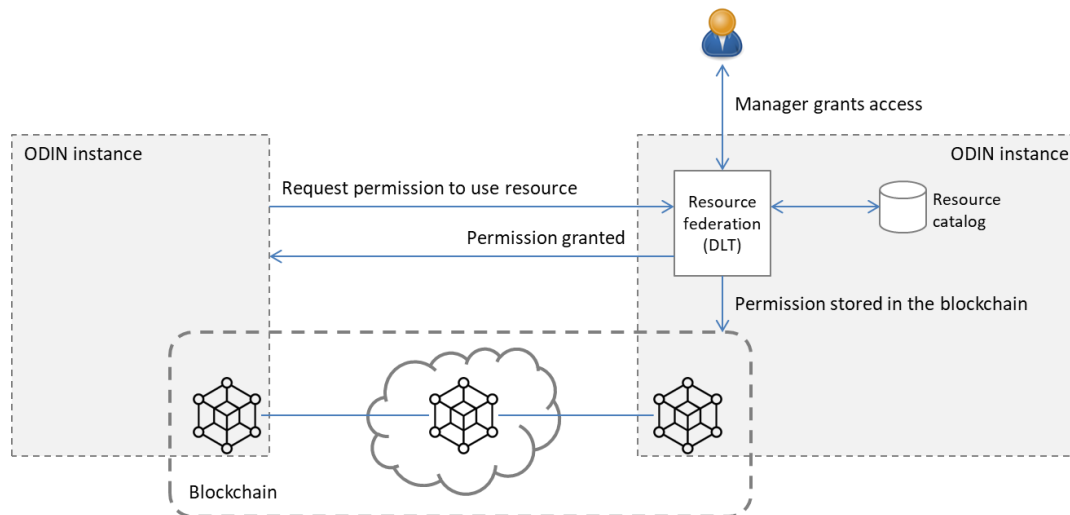


Figure 13: Organizational consent management.

Once permission is granted, the remote ODIN instance can use the local resources. A connection between the remote instance and the local ESB is made, so that the remote instance can use the local shared resources as if they were local in its premises. Throughout resource sharing, the RF component verifies the resource transactions and logs them in the blockchain.

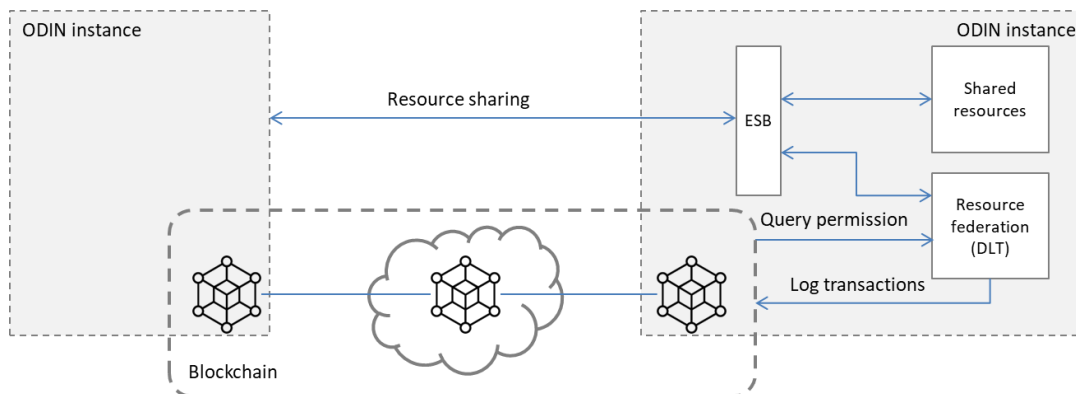


Figure 14: Resource sharing and transaction auditing.

5.6.3 Pros and Cons

The advantages of using the DLT mechanism for resource federation are the following:

- Auditability of transactions involving sensitive data
- Traceability of resource sharing requests
- Immutability of the transactions to unauthorized changes
- Permission and informed consent handling
- Accountability of the involved parties
- Non-repudiation of the involved parties
- Trustworthy federation and interoperability across organizations

The disadvantages of the DLT mechanism for resource federation are the following:

- Increased overhead in transaction making

- High computational power required for cloud or local blockchain nodes

5.7 Software choreography architectures

In the ecosystem of a Service Oriented Architecture, when a process must be implemented, several services must collaborate to achieve the final process's objective. This is due to the essence of SOA, where the architecture functions are separated in services, very decoupled and without dependencies. This way, services focus on specific responsibilities, but can't accomplish higher level business objectives by themselves, thus, several services must participate in an end-to-end sequence of interactions. The way the interaction is performed can be from two approaches. First one is software or service Choreography, and the other one is Orchestration.

The Choreography is inspired by dancing choreographies: the business process is understood by each service that know what to do at each moment, as it happens with dancers, that follow the music.

The Orchestration solution, is more like an orchestra group, guided by the director, who controls each of the musicians. This way a service orchestrator oversees managing every step that must be performed to achieve the business objective.

5.7.1 Architecture components

In a SOA architecture or an ESB, we can have several services communicating each other.

The communication among services can be 1 to 1, hub and spoke or through a messaging bus, but while this has several performances and scalation issues, all the cases still can benefit from using a Choreography or Orchestration approach.

Recalling the ESB architecture it can be described the scenario mentioned for an orchestrated approach.

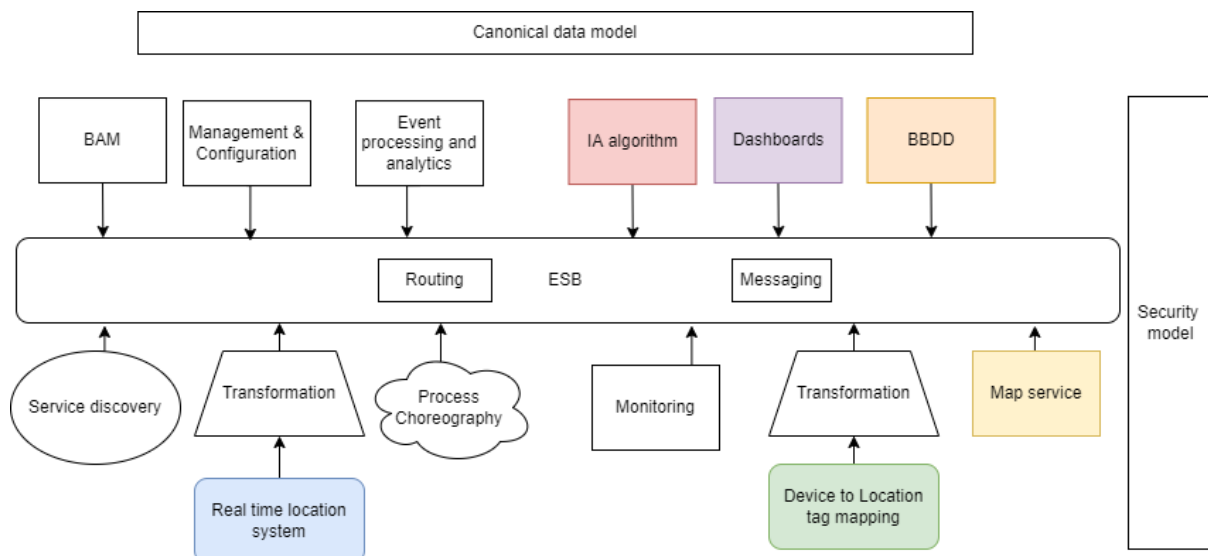


Figure 15 Orchestration

In the figure, it can be seen several services connected to the ESB, and we also see a Process Choreography module as a cloud shape. This module is the director of the business processes that must be accomplished through the architecture and knows what to do and what service must do each task at the right moment.

The Choreography approach just deletes the Process Choreography from the architecture, and every service must know how to act to follow the process or processes that must be carried.

5.7.2 How it works?

The Orchestration approach is very straightforward, as processes are handled only by the Process Choreography service.

To understand how it works, just imagine that the Process Choreography has a template of each process the whole system must support. Usually, in most evolved situations, this involves using a language to express the processes, the tasks, the events that must be fired and the actors or services that must act. An example of this implementation are the workflows, and the language can be BPEL (Business Process Execution Language). In addition to the process template, the Process Choreography has a rule or workflow engine, which is the component in charge of interpreting the BPEL template and performing the calls for each service, generating events, commands, or task. There exist also hardcoded implementations of the workflows, but those are not adaptable and require lots of efforts each time a change must be performed, thus parametrizable tools are the most recommended.

In the following figure, a representation of BPEL description and a diagram

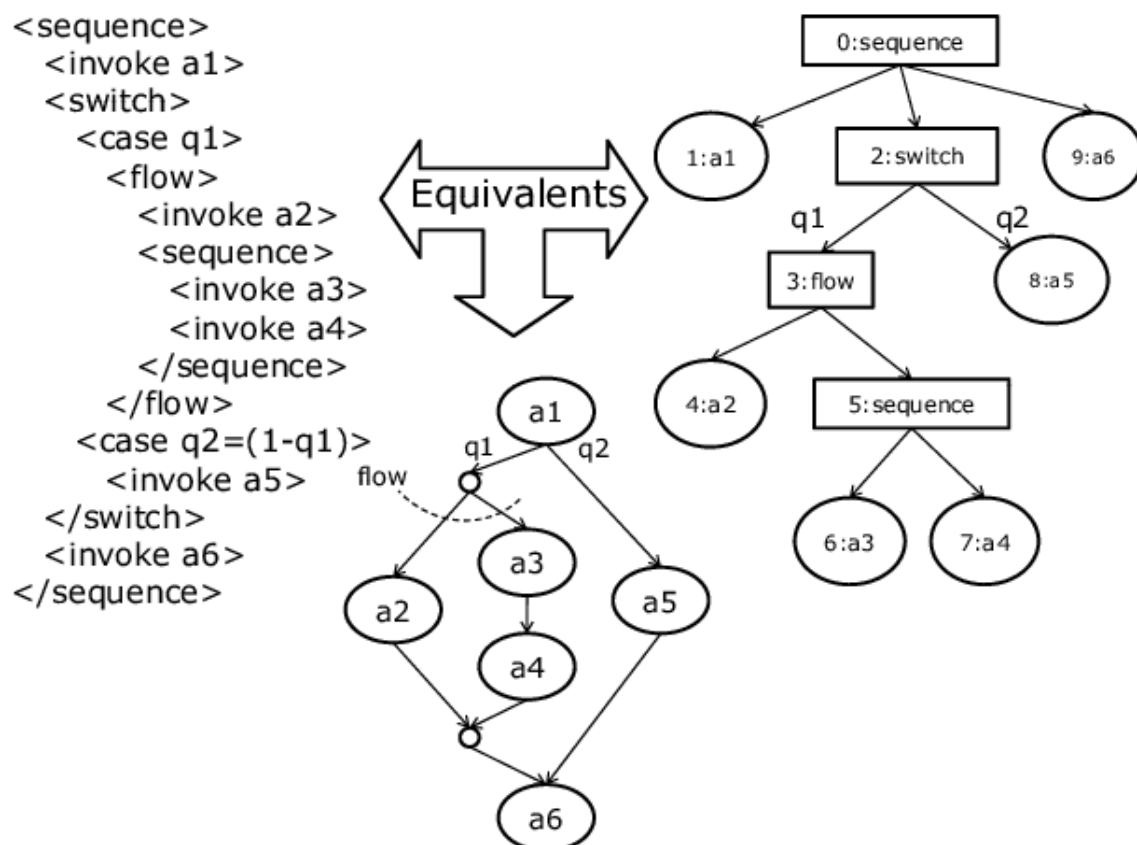


Figure 16 BPEL Example diagram and code

Under a message bus architecture, the Process Choreographer will send the right messages to each service, which will require to have a rich message base.

In the Choreography case, there are several ways to achieve the whole process objective:

Include a Process choreographer in each service. This way each service knows what happens and what to do but increases each service complexity. On the other hand, it reduces messages

in the bus. Usually this can be done using a sidecar for the service as presented in section 5.3 with Service Mesh architecture, so the logic code is not mixed with the service.

Increase service simplicity and reduce their responsibilities. This way, each service will listen only for a few messages and perform their simple task over the messages processed. When a message of interest is received, the service processes and then returns the result to the bus. In case it has been processed successfully, a new service will pick the new message and process it until all the process steps are completed.

5.7.3 Pros and Cons

Both approaches have pros and cons.

For the Orchestration case, the pros are:

- Fast adaptation to changes
- Single point of control
- Clear overview of the whole process
- Easy testing
- More suitable for synchronous behaviour, but it may be introduced asynchronous modes

While the disadvantages of Orchestration are:

- Single point of failure
- High overload of the Process Choreographer module and the bus with messages

The Choreography advantages are:

- No single point of failure
- Natural choice of choreography in case serverless systems are used or services can be created and killed on runtime.
- Easy to adapt in case lots of service changes are planned.
- More suitable for asynchronous behaviour using micro services

Main Choreography disadvantages are:

- If there exists lots of services, it may be hard to manage.
- Unclear overview of the whole process
- Hard testing
- In case a service fails processing a message, a mechanism must be introduced to process the failure and perform undo actions if needed.

To finalize it will be mentioned that in some scenarios is quite straightforward to follow a mixed approach, where some of the processes can be driven by choreography, and other more complex may follow an orchestration approach, making profit of both worlds.

5.8 KPI collection system architecture

The KPI collection system is effectively the BAM component of the ESB (see 5.4), intended for monitoring of the platform itself (the performance of the ESB and core modules), as well as all the KER connected to it, and their interactions. Because the system includes automatic metric collection, and has the ability for post processing, it can also be used to collect higher level KPI, i.e. monitoring Reference Use Case KPI and pilot KPIs (T7.2). Overall, the system is composed of 3 blocks: collection, processing, and reaction the overall effect being that the metrics are being monitored.

5.8.1 Architecture components

For the first block, collection (left side of Figure 17), the architecture contemplates 2 methods of metric collection. Pushing metrics, is employed when the module actively initiates the process, contacts the aggregator component (the central component of the KPI collection system) using its interface to report at this interaction the metric. On the other hand, pulling metrics method is initiated by the system itself (through the polling module), it contacts the module at the module's predefined interface (which has been previously registered), periodically polling the module for the latest batch of collected metrics, which, if any, are then reported to the aggregator. There are certainly differences between pushing and pulling metric methods, but neither can be discarded as it will pose an integration issue later.

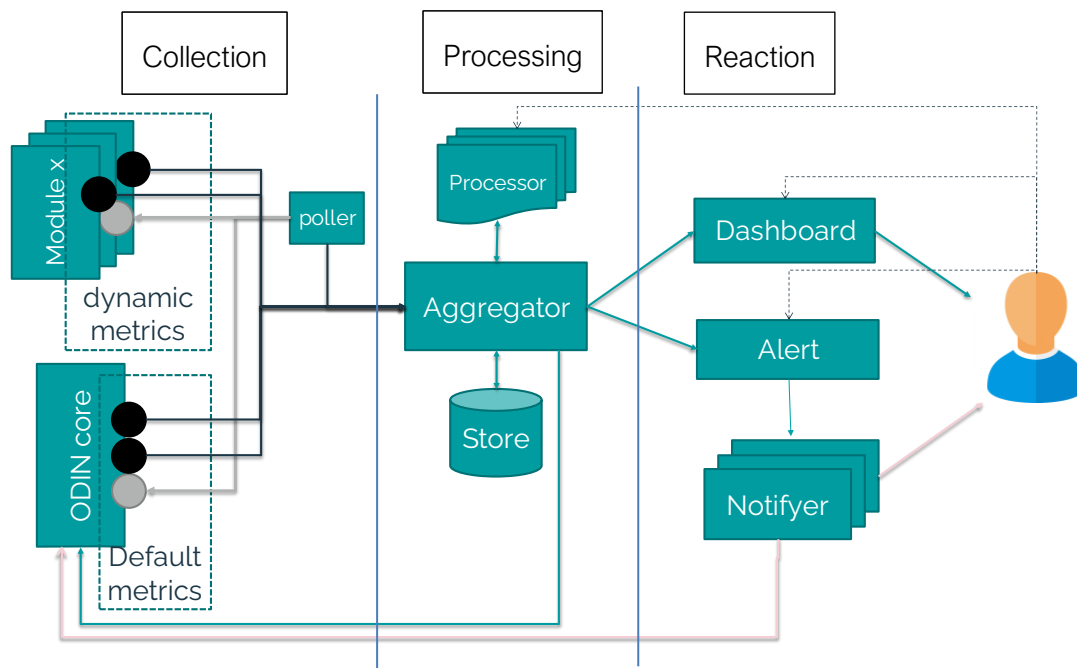


Figure 17 KPI collection system architecture

The second block, the processing stage (middle of Figure 17) is when the metrics are collected in a central component: the Aggregator. This component receives all the metrics being produced in the system, stores them and then it routes them. Depending on the configuration, i.e. the registered subscribers, each KPI may be sent to custom processing (e.g. calculating mean value per week out of individual measurements) which may produce more metrics, to the dashboard, an alerting system, or to other subscribers as part of the ODIN core functionalities.

The final block, the reaction stage (right side of Figure 17), is composed of 3 main components:

- The dashboard is used for reporting real-time, and historic, metrics directly to the user, so they can then take decisions.
- The alert component evaluates the received values with a configurable set of rules, which describe conditions which would trigger alerts, when an alert is triggered then the alert component uses one of the configured notifiers to report the alert. Typically, the rules are as simple as checking if a specific metric is over, or below, a specified value.
- The pluggable notifiers are components that report alerts through a specific channel. There may be different notifier for sending reports through email, push notifications in the dashboard, or more complex channels as the ODIN ESB.

As many hospitals driven KPIs involve the use of questionnaires as a KPI collection tool, the KPI collection system will include an open-source online questionnaire management system (not shown in Figure 17, but would be located a one of the KPI source modules), preconfigured and connected to the aggregator.

5.8.2 How it works?

The KPI collection system is a system which integrates potentially all other ODIN systems and applications through IPC or API as well as providing important human interaction.

Any system which generates metrics, can report them, using the common data model. To do so they either push the metrics directly, or they register their endpoint for the puller to periodically collect them. In either case the KPI collection system can keep track of all the sources of metrics. System administrators can manually add and remove additional sources of metrics. A default set of metrics, for those identified KPIs within the ODIN core components, will be set up automatically on system deployment, this way at least a minimal set of KPIs will be monitored through the system.

Platform administrators will be able to add custom processor modules; these can be fully customizable containers; standard components (such as a component which calculates hourly, daily, weekly, monthly or annual means configurable for specific), or programable modules (e.g. server-less functions); in any case they will use the aggregator interfaces to retrieve the metrics they need and push the new calculated ones.

In a similar way administrators will be able to add new notifiers, some standard notifiers will be bundled with the KPI collection system. The alert system will also be configurable through API or through an administration panel, to input the trigger rules.

5.8.3 Pros and Cons

A Brief list of advantages of the KPI Collection system architecture:

- It Works from the get-go, thanks to the default metric
- It is expandable, new modules, not just from the ODIN platform, can be added to the system
- It covers all types of metrics, from simple hardware resource consumption to very highly conceptual experiment KPIs based on automatable parameters.
- It is extensible, standard, or custom processors and notification channels can be added
- It is customizable, the way the system reacts to, processes and displays the metrics
- Components and configurations can be packaged, thematic components and configurations can be exported and then imported where needed, the KPI packages could even be bundled with ODIN applications

On the other side, some disadvantages are:

- It may be complex, managing an increasing number of metrics can be daunting, maybe metrics should be categorizable
- It may produce overhead, for application developers to think and connect metrics that later users may never use
- There may be incompatibility issues, especially when dealing with very heterogeneous metrics, metric collection methods, and protocols which all need to be adapted to the common model
- It may be inconvenient, even though the alerting system is highly configurable, users may only understand basic configurations of it, and without the ability to fine-tune alert conditions they may turn important notifications all together
- It can consume a considerable number of resources if installed on premises that has a limited hardware.

5.9 Final decisions

Section 5 has reviewed most of the architectures and patterns that the CPS-IoT and its component should follow to meet section 4 requirements and platform needs.

As a summary:

The integration among the components will be done using an Enterprise service bus, with a rich messaging language to cover the use case's needs. This integration type allows a platform that can evolve and be adapted, decoupling the components and systems integrated.

Whenever it is possible to identify a problem pattern for which exists a pattern solution, an Enterprise Integration Pattern or a combination of them, will be used to solve the problem, as ODIN must be created using reliable solutions, and not reinvent the wheel.

Following the ESB approach, ODIN will use not only services to decouple functionalities, but will go beyond using micro-services and in case the available solutions allow it and do not impose restrictions to its interests, a mesh service and serverless pattern will be used when the conditions allow it.

The Resource Gateway will be implemented using a straightforward approach, using API Gateways with reverse-proxy capabilities, and combining the functionalities of a Resource Manager in charge of holding all the resource information.

The federation of resources and auditability tracking of transactions will be created using blockchain technologies that will control the relations between hospitals and track important changes.

The service management in terms of business application creation will be performed mainly using orchestration techniques, where a central Resource Choreographer will control the rest of the services, but whenever is possible and efficient, choreography approaches will be introduced so a service or several services can be more autonomous to reduce central overhead.

The KPI component is the BAM module of an ESB architecture, and it will be based on a collector-aggregator/processor scheme, where the data will feed dashboards and an event system that will react upon important situations. The side-car pattern may be used in the collection layer and also push or pull strategies as it will depend on the technology and integration capabilities available.

6 ODIN's CPS Architecture

The CPS-IoT architecture has lots of requirements to be fully compliant with the use cases.

During work performed in WP3 and WP4 meetings and individual partner work, reviewing the requirements, several ideas and proposals have raised and been confronted to the requirements and the use cases, using a process similar to the one escribed by Hofmeister⁹ where analysis of the most important requirements induces a design task and an evaluation of the designs against the requirements and needs. To summarize it will be presented only part of the final proposal of the whole platform presented in WP3 D3.10 ODIN Platform. CPS-IoT proposal contributed to several patterns and components to the WP3 architecture.

The most important ideas proposed from WP4 are:

- Only one bus to rule all the services.
- The architecture could be the same at any level. Cloud, hospital, edge.
- A Rich messaging protocol must be created to support the communication through the ESB.
- The Resource Manager manages all the resources available in the platform
- To fulfil the security and authorization requirements a Security Manager service has been included
- There exist an API Gateway which holds the control and access from external services and applications
- The choreographer controls business logic mostly in the form of orchestration but choreography may also exist.

The following image is only part of ODIN platform, highlighting WP4 components.

⁹ C. Hofmeister et alters. Generalizing a Model of Software Architecture Design from Five Industrial Approaches. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5). Pittsburgh, PA, November 6-9, 2005. Named one of the five best papers of the conference.

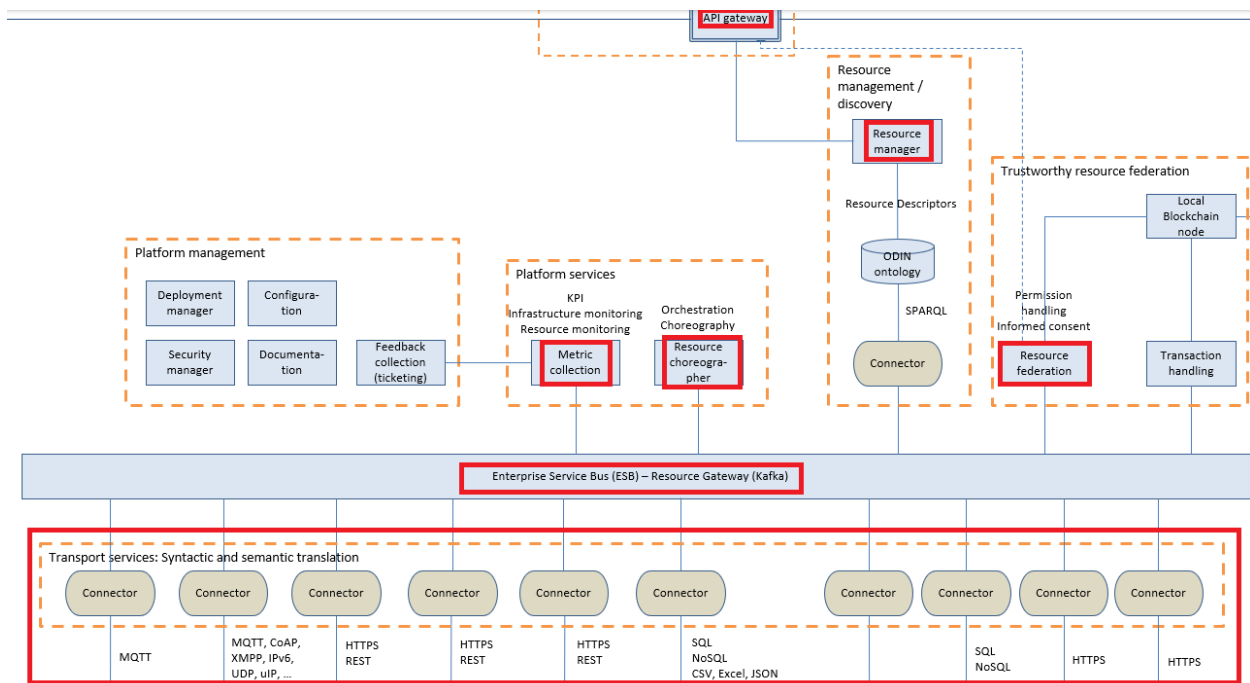


Figure 18 Part of ODIN's architecture with focus on WP4 components

The final architecture uses Connectors as Transport services to adapt physical connections and semantic protocols. This way resources and services can be integrated into the Enterprise Service Bus reviewed in section 5.4.

Other services also coming from the conceptual architecture form ODIN's vision are integrated into the architecture, such as Metric Collection service, Resource Choreographer and the Resource Federation service, in charge of sharing resources among hospitals and the cloud part of ODIN.

As part of the Resource Gateway, several components were extracted such as the API Gateway and the Resource manager.

7 Architecture Validation

A good way to validate the architecture is to confront it to a use case, to try to define the fine-grained details of the communications and interaction of the modules and resources of the architecture.

The summary of components involved in each example are as follows:

Example	Components
SERMAS logistic support	Choreographer, ESB, Resource Manager, AI (Context Awareness, Control and actuation), Robots, RTLS, Connector
AI data exploitation	Choreographer, ESB, AI, Datastore, Connector
Resource provisioning	Web, API Gateway, Choreographer, ESB, Resource Manager, RTLS

7.1.1 Example 1: SERMAS use case

An example of a use case to restore catheters from a department has been developed in the following sequence diagram to test the architecture. Some steps are avoided to simplify the diagram.

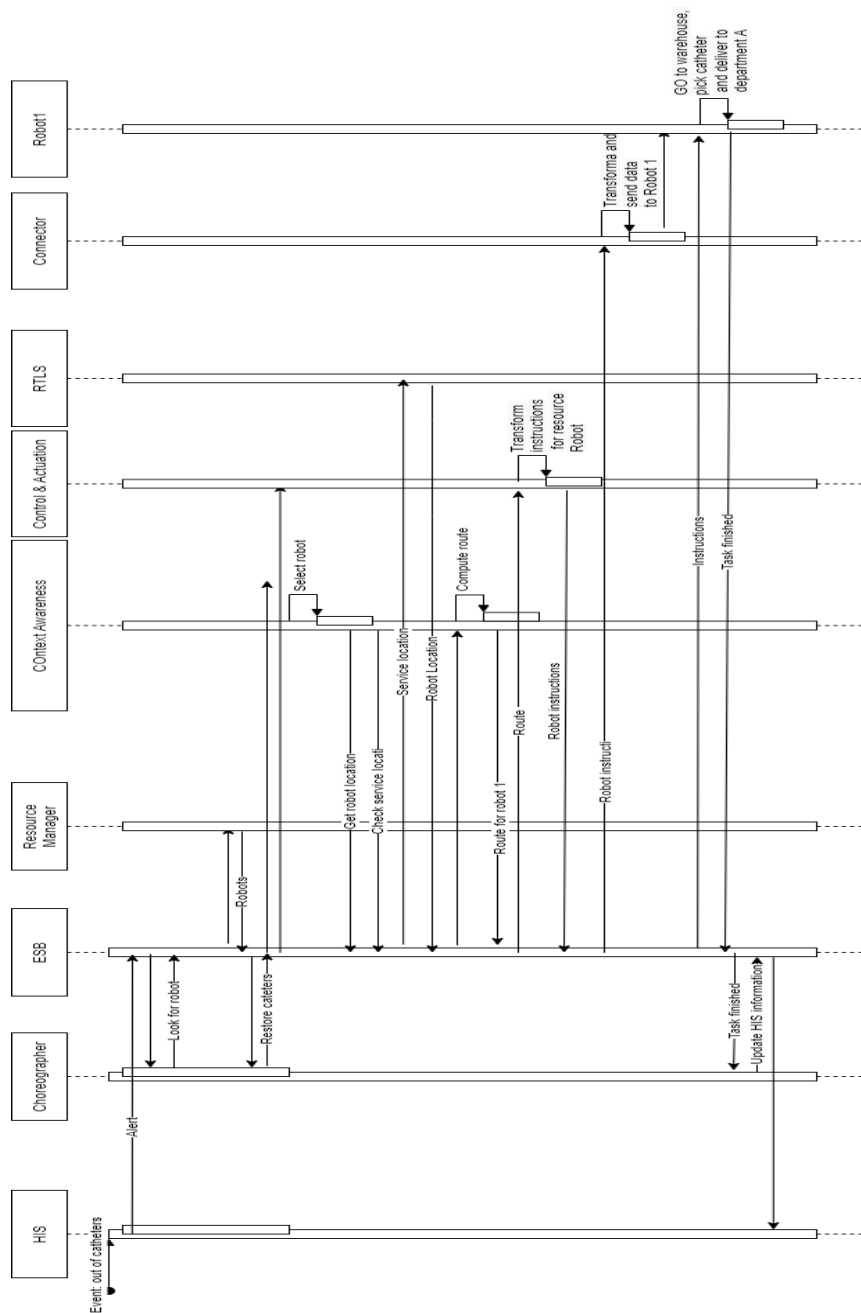


Figure 19 Architecture test

The example starts with the event that the hospital's HIS registers because some service has run out of catheters.

The event is listened by the Choreographer and starts orchestrating the tasks. First recovers the available robots and then fires a task to restore the catheters. So, the context awareness gets information about the location of the robot, the warehouse and the service. Then computes the routing to the service. The Actuation and control module composes the tasks that must be performed by the robot. Those instructions must be transformed into the Robot domain language so the Connector for the robot does the work, and the robot gets the instructions.

Finally, the robot executes the task assigned and fires an event when finishes, so the choreographer understands what happened and request the HIS to update catheter information.

The architecture has the power to handle the situation in several ways, and the presented one is just an option. Final implementation may differ, but the architecture supports the objectives of the use case proposed.

7.1.2 Example 2: AI data exploitation

In this example, an AI service in charge of learning from the available data, is used to forecast hospital needs.

The AI service is trained to forecast fungible wasting, so needs historical data and current data to be used with a trained algorithm.

The forecast is scheduled to be computed using a rule from the Choreographer, so it calls the AI service to start the forecast once per week. This decouples functions and allows parametrizing in a centralized way when a forecast must be done.

Upon the forecast compute signal, the AI service recalls the data from the Datastore, where the valuable data has been stored each week. Once the forecasting finishes, it publishes the results to the bus, so each service with interest in the results, can use them. The Datastore stores the results, so deviations can be checked later, the HIS interface publishes into the HIS the results, so Business and Service staff can use the information, and the Choreographer fires new tasks. For example, it could call another AI service to compute which are the best service providers to cover the forecasted needs and trigger the procurement of the fungible items.

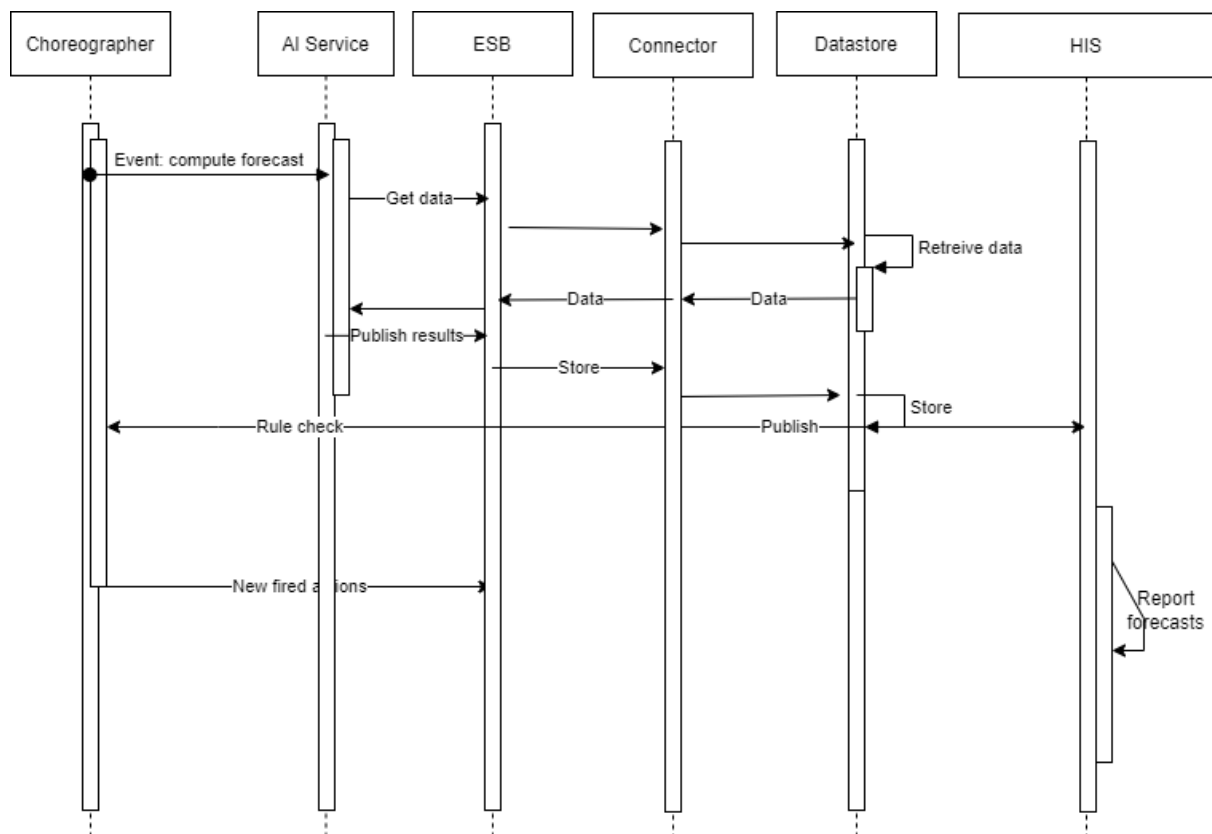


Figure 20 AI Architecture test

7.1.3 Example 3 Resource provisioning

In this case, a new resource is going to be added to the platform, so several steps have to be performed.

We will focus on a manual resource provisioning such a robot. The robot has several satellite devices which are also resources for the platform, such as the RTLS tag and a dock station to reload the battery to name a few.

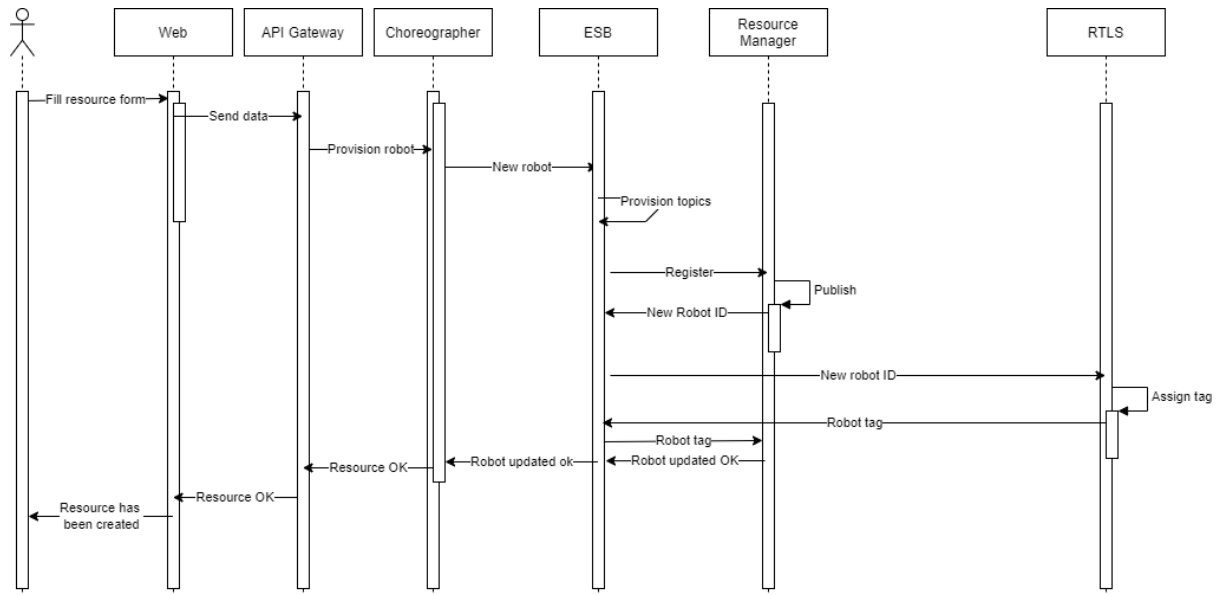


Figure 21 Resource provision test

The process to provision a new robot needs to fill a web form provided by the Resource Choreographer, which is published through the API Gateway. The form includes the name of the robot, the type of the robot, the RTLS tag reference and the dock station name and position.

Upon receiving this form, the resource choreographer fires a message with the robot information, so the Resource Manager includes the robot information, the messaging bus provisions new messaging endpoints (topics) for the robot in case it is needed, the RTLS adds a new Tag to monitor the robot position, and other services may also listen for the new robot resource, such as the security service, which provisions new credentials for the robot so it is allowed to interact with the platform.

Finally, the user receives an update with the result of the resource provision.

8 Conclusions and next steps

D4.1 has reviewed most important requirements from each resource of needs, DoA, WP2, ODIN's layers and discussions among partners.

From the initial blueprint of the DoA and D2.2 proposal, a review of the different architectures that can be used to solve each problem and component has been carried out.

With this approach WP4 has selected several solutions to conform the first draft for the CPS-IoT's architecture that is aligned with ODIN's project vision and needs and has contributed to WP3 architecture presented in D3.10. This work must be developed more in depth and refined through the work that is being performed through the multi-disciplinary teams reviewing the different Reference Use Cases defined by WP7.

Finally, D4.1 has setup the guidelines to study each of the components in the deliverables D4.2 and D4.5 where it has been reviewed concrete solutions and designs for the CPS-IoT components. In future versions of those deliverables and also in D3.10, several decisions will be taken to get a more concrete architecture with a more grained level of detail, solving questions about streaming communications and a closer review of resources connected to the CPS-IoT.

Appendix A Glossary

Term	Definition
AI	Artificial Intelligence. It is a set of concepts and methods for creating algorithms that can learn and decide on their own.
API	Application Programming Interface. It is the set of functions of a system that are exposed to other systems to allow its use.
BOT	Building Topology Ontology
CORA	Core Ontology for Automation and Robotics
DLT	Digital Ledger Technologies. This is a set of technologies (based on Blockchain) that allow the secure and immutable storage of information, leveraging a decentralized security scheme.
Edge computing	Is a technique where the computing power or services is moved close to the customer's premises. In IoT, Edge means that part of processing of sensor's information is performed close to the sensor. This eases and reduces the amount of data handling
EIP	Enterprise Integration Pattern. It is a compendium of well-known solution approaches to solve commonly known problems related to integration of services and systems.
ESB	Enterprise service bus is an integration solution where several independent services and components communicate each other through a common bus. This solution usually is used in enterprises that have to integrate several platforms.
ICD 11	International Classification of Diseases 11
IoT	Internet of Things. This is a broad term referring to creating a network of things and devices, by assigning an ID to each thing and enabling communication or information exchange between them.
KPI	Key Performance Indicator. This is a measurable indicator for assessing the performance of a system regarding a particular aspect.
ML	Machine Learning. This is a subset of Artificial Intelligence methods, focusing on algorithms that can learn based on presented examples.
NCIT	National Cancer Institute Thesaurus Ontology
Pub-sub	Is an integration technique where producers of messages and consumers of messages achieve the communication of the information sending messages without knowing the destination actually, decoupling the components.

REST	Representational State Transfer. This is a set of guidelines for designing web applications, regarding how they call each other and how they pass parameters.
RTLS	Real Time Location System
SNOMED	Systemized Nomenclature of Medicine Ontology
WoT	Web Of Thing Ontology

Appendix B Use cases

Appendix B covers the list of use cases in ODIN's scope.

The list has been extracted from deliverable D3.10 ODIN Platform.

ID	Use case
UC_CP_1	User authentication
UC_CP_2	User creation
UC_CP_3	Self-user registration
UC_CP_4	Resource registration
UC_CP_5	Resource communication
UC_CP_6	Communication between front-end and back-end resources
UC_CP_7	Communication with HIS
UC_CP_8	High level application creation
UC_CP_9	Resource federation
UC_CP_10	Metric collection
UC_CP_11	Feedback collection
UC_CP_12	Delete resource
UC_CP_13	List resources
UC_CP_14	List workflows
UC1	Aided logistics support
UC2	Clinical engineering, medical locations, real-time management
UC4	AI-based support system for diagnosis
UC5	Automation of clinical workflows
UC6	Inpatient remote rehabilitation
UC7	Disaster preparedness