This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 101017331

D4.3 Implementation of Local CPS-IoT RSM Features v2

Deliverable No.	D4.3	Due Date	28/02/2023
Description	D4.3. Second version presenting the final technologies for the Resource Descriptors, the technologies for the Resource Gateway and the measurement collection software components.		
Type	Report	Dissemination Level	PU
Work Package No.	WP4	Work Package Title	CPS-IoT Resource Management System
Version	1.0	Status	Final

Authors

Name and surname	Partner name	e-mail
Sofia Granda	INETUM	sofia.granda@inetum.com
Antonio Gamito	INETUM	antonio-jesus.gamito@inetum.com
Luis Carrascal	INETUM	luis.carrascal@inetum.com
Alejandro Barnadas	INETUM	alex.barnadas@inetum.com
Pablo Lombillo	MYS	plombillo@mysphera.com
Eugenio Gaeta	UPM	eugenio.gaeta@lst.tfo.upm.es
Alejandro Medrano	UPM	amedrano@lst.tfo.upm.es

History

Date	Version	Change
27/07/2022	0.1	Initial ToC
07/11/2022	0.2	Final ToC
19/12/2022	0.3	First Draft
09/01/2023	0.4	First Draft review
16/01/2023	0.5	Final version first draft
31/01/2023	0.6	Peer review document
15/02/2023	0.8	Completed version
20/02/2023	0.9	Version after peer-review
24/02/2023	1.0	Version ready for submission

Key data

Keywords	IoT, resources, robot, gateway, integration, platform, layer, API, messaging, web service, architecture, components
Lead Editor	Sofia Granda (INETUM)
Internal Reviewer(s)	PEN, UMCU

Abstract

Deliverable D4.3 “Implementation of Local CPS-IoT RSM Features v2” is the second iteration of deliverable D4.2 with the same name. It describes the fundamental features of the CPS-IoT Resource Management System, the ODIN platform layer that supports the interconnection of available resources more in depth. The Resource Manager, and within it the Resource Descriptor is the key component that defines and manages the data collection infrastructure. Along with the Resource Directory that acts has storage point. The Resource Gateway manages communication to the ODIN upper layers. The Measurement Collection Software Components are used to register and collect performance indicators. In this second version, we will select the technologies to apply for each component and how to implement them.

Statement of originality

Table of contents

List of tables

List of figures

1 Introduction

The following deliverable is the second version of deliverable 4.2, which began by outlaying the main features of the CPS-IoT Resource Management System and the possible technologies to implement it. In this second version, we will review the proposals set out in D4.2 and select the most appropriates for each element that was presented: the Resource Descriptor, the Resource Gateway, and the Measurement Collection Software Components. We will also present the possible integration schema of all the elements and evaluate the different alternatives to implement it in the local ODIN instances.

1.1 Deliverable context

Table 1. Deliverable context

PROJECT ITEM IN THE DOA	RELATIONSHIP
Project Objectives	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. The WP4 objectives are: • Specification of the CPS-IoT RMS requirements based on input from WP2 • Specification of KPI and metrics collection framework
Exploitable results	The final definition of the software architecture components implementation and their usage to tackle the ODIN platform's needs and goals. It will also give a interconnection model for all the technologies used, to interact with each other and work as one to the end user. The docker images necessary to deploy the technologies used.
Workplan	D4.3 (M24) was preceded by D4.2 (M12) and will be followed by D4.4 (M36). They belong to WP4 Task T4.2 CPS-IoT Resource Descriptor Module lead by INETUM.
Milestones	It is relevant for milestone MS2, that defines de technologies.

Deliverables	• D3.3: Report on the Data model, ODIN semantic ontology, datasets harmonization plan. • D4.1: CPS-IoT Resource Management System Specification. • D4.2: Implementation of Local CPS – IoT RSM Features v1. • D4.6: Implementation of Advanced CPS – IoT RSM Features v2. • D7.3: KPI Evolution Report • D7.9: Pilot Studies Evaluation Results and Sustainability.
Risks	This deliverable tackles the following risks: • Technical problems due to a poor previous analysis and unsuitable technologies for the proposes that want to be achieved. • Delays in implementations and pilots • Incompatibilities between technologies

1.2 Platform Architecture review

Figure 1 shows the ODIN platform architecture scheme. The components that will be explained in the deliverable are highlighted with a red square. This way we can see their place within the architecture.

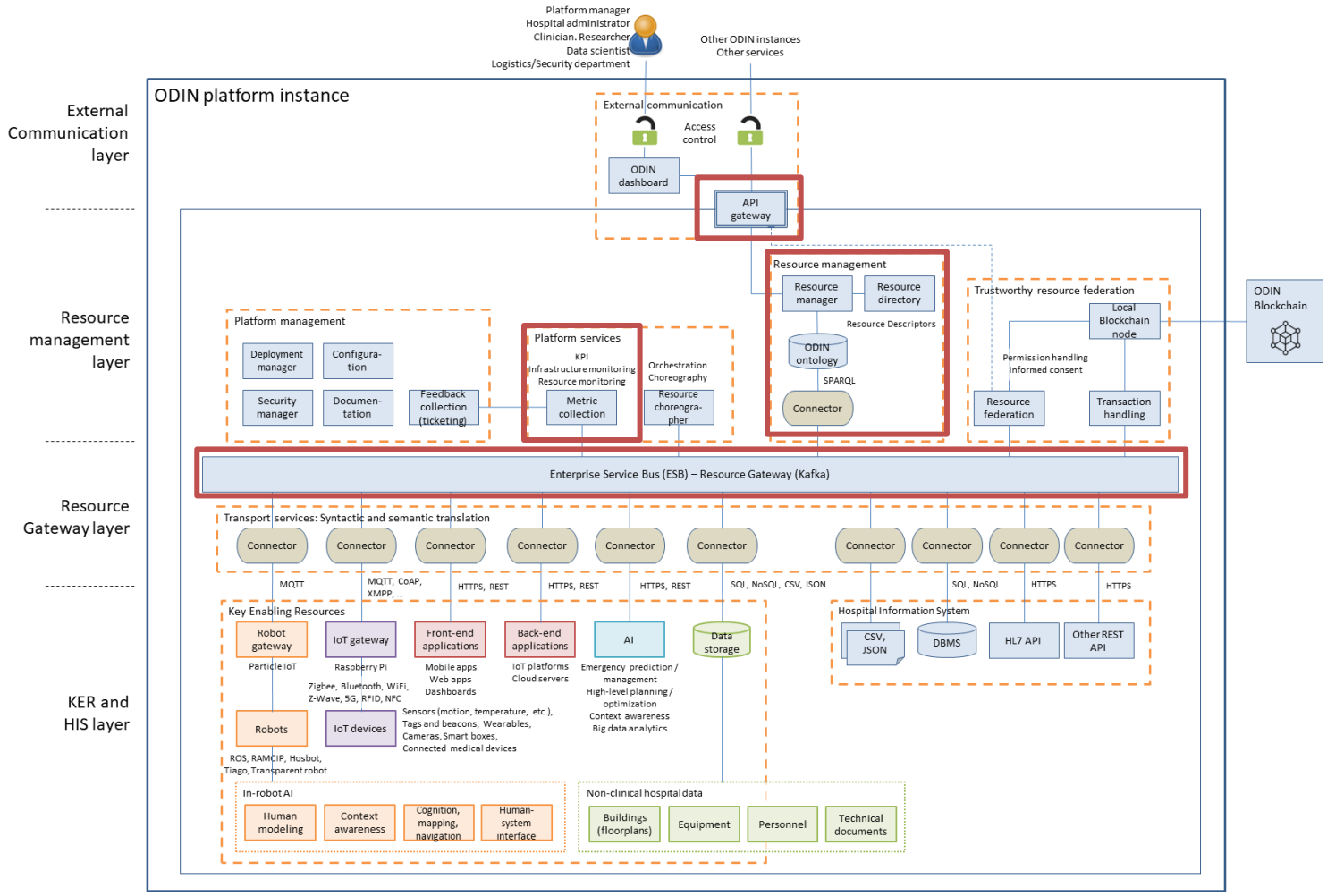

Figure 1: ODIN Platform Architecture

2 Resource Management

2.1 Resource Manager

The Resource Manager (RM) is the hub for resource registration, querying and update[1]. At the same time, it is the connection to the ODIN platform and the middle component between the ontology and the Resource Directory. It will offer a set of services regarding resources, and their management.

The main functionalities of the Resource Manager are:

- Ensuring that the resources been registered follow the structure defined by the Resource Descriptor
- Ensuring that the information contained follows the ontology.
- Additionally, it is the focal point to consult about the available resources and extract the information stored in the Directory.

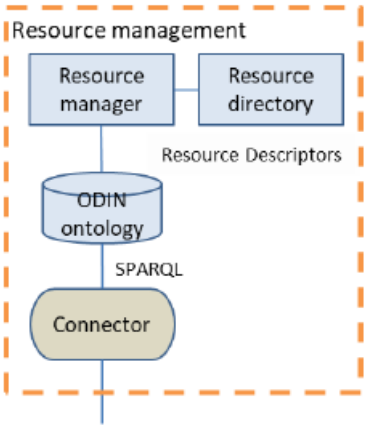

Figure 2: Resource Manager in the ODIN platform architecture detail

2.2 Resource Descriptor

2.2.1 Introduction

The Resource Descriptor is the element that homogenises data description. It provides a common scheme to describe everything that integrates the ODIN platform in terms of the supported data types, formats, protocols, and structures. It describes how the data is organized (it will help to fed T3.2 modelling the ontology for internal representation of the KERs in the system) and the information that there is available. This way, we can have a structured digital directory to look for the ODIN platform's components[1] available (KERs, AI algorithms, HIS…), the information they

[1] D3.10 ODIN platform v1

provide and the mechanism necessary to interact with them, not only in terms of data collection, but also to send back instructions and control them.

2.2.2 Resource Descriptor in D4.2

In the previous version of this deliverable[2], we already presented the Resource Descriptor's requirements to establish the information this component would handle. Those requirements were:

- Semantic Resource Description
- Resource Services
- Resource Federation
- Resource Privacy, Security, and Trust
- Resource Metric Reporting
- Resource Health
- Resource UIs
- Resource Administration
- Resource Documentation
- Resource Deployment
- Resource Communication

We could see that it covers every aspect that integrates a technological solution, regardless if it is software or a physical device, going from security to documentation.

Additionally in D4.2, three different solutions were analysed and compared:

1. Web of Things: a W3C initiative for IoT devices. It proposes to standardize device's description in a way that is understandable by machines. This would overcome integration and interoperability problems between different systems.
2. OpenAPI: A standardized format for describing REST Application Programming Interfaces (APIs), resources and services. It contains information regarding resources, endpoints, operations, parameters, and authentication, to be able to make use of the described services adequately.
3. FHIR: This standard is focus only on healthcare information and it defines how it can be exchanged between different computer systems regardless of how it is stored.

2.2.2.1 Final Approach

From the analysis, it was shown that the proposed solutions are heterogenous and tackle different types of needs and that the information they handle is different. That is why, the final approach is to integrate all of them and end up with a Resource Descriptor were all technologies complement each other. Depending on the component of the ODIN platform, we will make use of the

[2] D4.2 Implementation of Local CPS-IoT RSM v1

technology that best suits its functionality and adapts to the information been stored. We could even use more than one technology if necessary. In the end, we will be able to properly describe all the elements of the ODIN platform in a structured manner without losing any information due to standards constraints.

To sum up, the final solution will use the technology that bests suits the functionality of the element been registered and its needs.

This concept aligns with the channels approach presented in D3.11[3]. This approach is based on event-based communication within the ESB. Were each KER opens a channel making use of specific topics to publish and consume data. The Thing Descriptor of each KER will have to include the information relative to that channel such as the data model and the topics used. Additionally, there can be channels for peer-to-peer communication using http services. In this case the Thing Descriptor would include the endpoints description based on the standards defined (OpenAPI, WoT or FHIR). Finally, if there would be streaming channels, the URL of those should also be included in the Thing Descriptor.

2.2.3 Selected Technologies

In the timeframe between the first version and the second version of this deliverable, another possible technology for the Resource Descriptor (additionally to WoT, OpenAPI and FHIR) raised up:

2.2.3.1 AsyncAPI

AsyncAPI[4] is an open-source standard based on OpenAPI but focused on Event-Driven Architectures (EDA) and, therefore, in message-based systems. Unlike OpenAPI, that is focused on REST APIs. At the same time, one of the advantages is that both standards are fully compatible with each other.

The main difference between these two systems is what their own names tell us. AsyncAPI is made to describe asynchronous message systems, instead of the typical REST API where you make a request and wait for a response from the server (like OpenAPI). It is made for systems where a response is not expected, so called “fire and forget”. The devices send messages to the server but do not expect any messages back. A good example of this could be a temperature sensor. It just needs to report the temperature to the server so the server can analyse the data and take the adequate measures, but the device does not need to receive any feedback from the server to perform its activities.

The core concepts of AsyncAPI are:

- Message broker: It is the central point to which all messages arrive. It takes care of receiving and delivering the messages. For example: Apache Kafka.
- Publisher/Subscriber: The application that publish or consume the messages that arrive to the broker.

[3] D3.11 ODIN platform v2

[4] AsyncAPI, https://www.asyncapi.com/docs

- Message: The piece of information been transmitted.
- Channels: The routes that the messages follow. Normally, brokers have different topics that categorize the information arriving to the brokers. For example: we can have a temperature topic to which arrives all temperature data and another called humidity to which arrives all humidity data.

As mentioned before, AsyncAPI originates from OpenAPI and it adapts many of the structures to the asynchronous world[5]. In the next image we can see the structure of both for better comparison.

[5] AsyncAPI vs OpenAPI, https://www.asyncapi.com/blog/openapi-vs-asyncapi-burning-questions

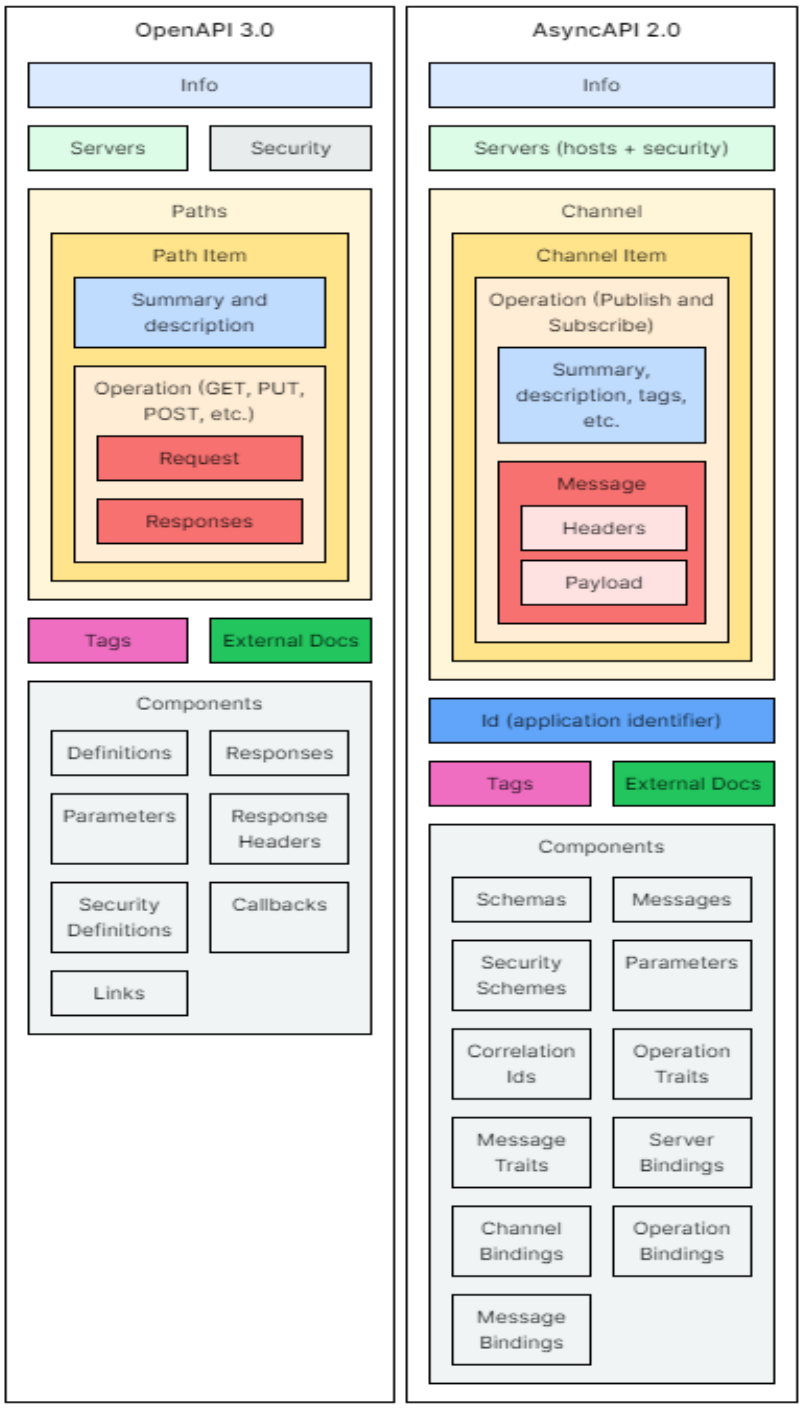

Figure 3: OpenAPI vs AsyncAPI structure

In the following image we can see a hello-goodbye message broker example. It shows how information is organized following the standard and according to the core concepts we just defined.

```yaml
asyncapi: '2.5.0'
info:
  title: Hello world application
  version: '0.1.0'
servers:
  production:
    url: broker.mycompany.com
    protocol: amqp
    description: This is "My Company" broker.
    security:
      - user-password: []
channels:
  hello:
    publish:
      message:
        $ref: '#/components/messages/hello-msg'
  goodbye:
    publish:
      message:
        $ref: '#/components/messages/goodbye-msg'
components:
  messages:
    hello-msg:
      payload:
        type: object
        properties:
          name:
            type: string
          sentAt:
            $ref: '#/components/schemas/sent-at'
    goodbye-msg:
      payload:
        type: object
        properties:
          sentAt:
            $ref: '#/components/schemas/sent-at'
  schemas:
    sent-at:
      type: string
      description: The date and time a message was sent.
      format: datetime
  securitySchemes:
    user-password:
      type: userPassword
```

Figure 4: AsyncAPI Example

In Figure 4[6], above, we see:

a) General information regarding the AsynAPI's version and the application's name and version. (Lines 1-4).

b) Server's (broker) configuration. It specifies the address, the protocol (here it's amqp but other common options are mqtt, kafka, ws or http) and the security of it. (Lines 5-11).

[6] https://www.asyncapi.com/docs/tutorials/getting-started/security

c) The channels. The possible routes the messages can follow depending on their content. (Lines 12-20).

d) At the same time, the channels are related to the components field (lines 16 and 20). That defines the contents of the messages. We can see that both, hello and goodbye messages, have the same data types, an Object for the main information and a String for the date and time stamp. (Lines 21-44).

In this case the description language is YAML, but JSON is also supported.

COMPARISON BETWEEN WEB OF THINGS, OPENAPI AND ASYNCAPI

The following table was already present in D4.2 to compare WoT and OpenAPI, now we will complete it with AsyncAPI to compare the three technologies.

Table 2: Comparison between Web of Things, OpenAPI and AsyncAPI

WOT	OPENAPI	ASYNCAPI
ADVANTAGES: • WoT Thing Descriptor can be enhanced with a context field for converting the JSON format to JSON-LD. • It can handle many protocols such as CoAP, MQTT, WebSocket. • WoT description uses events to represent state transitions (simpler). • WoT is specific for IoT and it applies to any IoT application domain, from consumer electronics to heavy industries	ADVANTAGES: • Enriched with text that can be understood by humans providing both, human and machine-readable descriptions of Web services. • Defines services in a way that eliminates ambiguities and provides Web Thing service descriptions which are uniquely defined and discoverable. • Meets the HATEOAS requirement of REST architectural style. • It is possible to convert an OpenAPI description to an ontology • OpenAPI is supported by a complete tool pallet (e.g., editors, description validators and client SDK generators)	ADVANTAGES: • It supports many protocols such as: AQMP, Kafka, MQTT, WebSockets… • Great variety of tools to edit, validate or develop contents under this standard. • Easily understandable by humans and small learning curve to use it. • For asynchronous interactions. One to one and one to many. • OpenSource.

DISADVANTAGES: • Description is a much shorter document. • Ambiguities: The same property may appear with different names. • Does not support HATEOAS requirement of REST architectural style.	DISADVANTAGES: • Does not support JSON-LD. • Only supports HTTP(S) and webhooks. • Subscription to property changes is more complex. • Simpler security scheme than WoT.	DISADVANTAGES: • Does not support JSON-LD. • Does not support HATEOAS requirement of REST architectural style. • Simpler security scheme than WoT.

At this point, we can see that there are similarities between WoT and AsynAPI with respect to the type of information handled and the problems they tackle. Still, their differences make unnecessary to make a choice between them. One of the purposes of the ODIN platform is to be able to integrate all kinds of elements that might be in the hospital and not been limited to one or another standard to add new devices to the platform. Therefore, AsyncAPI will be included to the technologies used by the Resource Descriptor along with WoT, OpenAPI and FHIR. This way, several standards will be supported to describe the different resource that will interact with the ODIN platform according to their functionalities, as it was explained in section 2.2.2.1.

The usage of AsyncAPI would be mainly for devices that send asynchronous messages and for whom Web of Things is not suitable because they cannot be considered environment or interactive sensors. Within the ODIN catalogue, we can find the RTLS (Real Time Location System) from Mysphera that is under this scenario. In Figure 5, below, we provide with an example of how the implementation of AsynAPI to describe this device would be used.

```yaml
asyncapi: '2.5.0'
info:
  title: MYSPHERA RTLS resource
  version: 1.0.0
  description: |
   The RTLS resource exposes the location of MYSPHERA RTLS smart tags to track patients, medical staff and high value assets.
   ### Features:
   * Publish real-time information about smart tag position 📈
servers:
  test:
    url: test.odinesb.org:8092
    protocol: kafka-secure
    description: ODIN ESB

defaultContentType: application/json
channels:
  odin.platform.iot.rtls.{tagID}:
    publish:
      message:
        $ref: '#/components/messages/positionUpdated'
components:
  messages:
    positionUpdated:
      payload:
        type: object
        properties:
          version:
            type: string
            description: Message version
          message:
            type: string
            format: string
            description: Type of position
          timestamp:
            type: string
            format: string
            description: WHen the position was updated. Java format of time.
          id-tag:
            type: string
            format: string
            description: ID for smart tag
```

Figure 5: AsyncAPI example for RTLS MYS

The example above shows the specification for RTLS system publishing instant position of ODIN assets and tracked people. The first section, “info”, describes the API. Then, there is the “servers” section, that would define the kafka broker address to connect to. Afterwards, the “channels” that indicate the topic where the messages will be published. And finally, the structure And content of the messages.

The tool available at studio.asyncapi.com also allows to edit the RTLS API definition and create the documentation at once. In the following image the documentation generates from the RTLS definition.

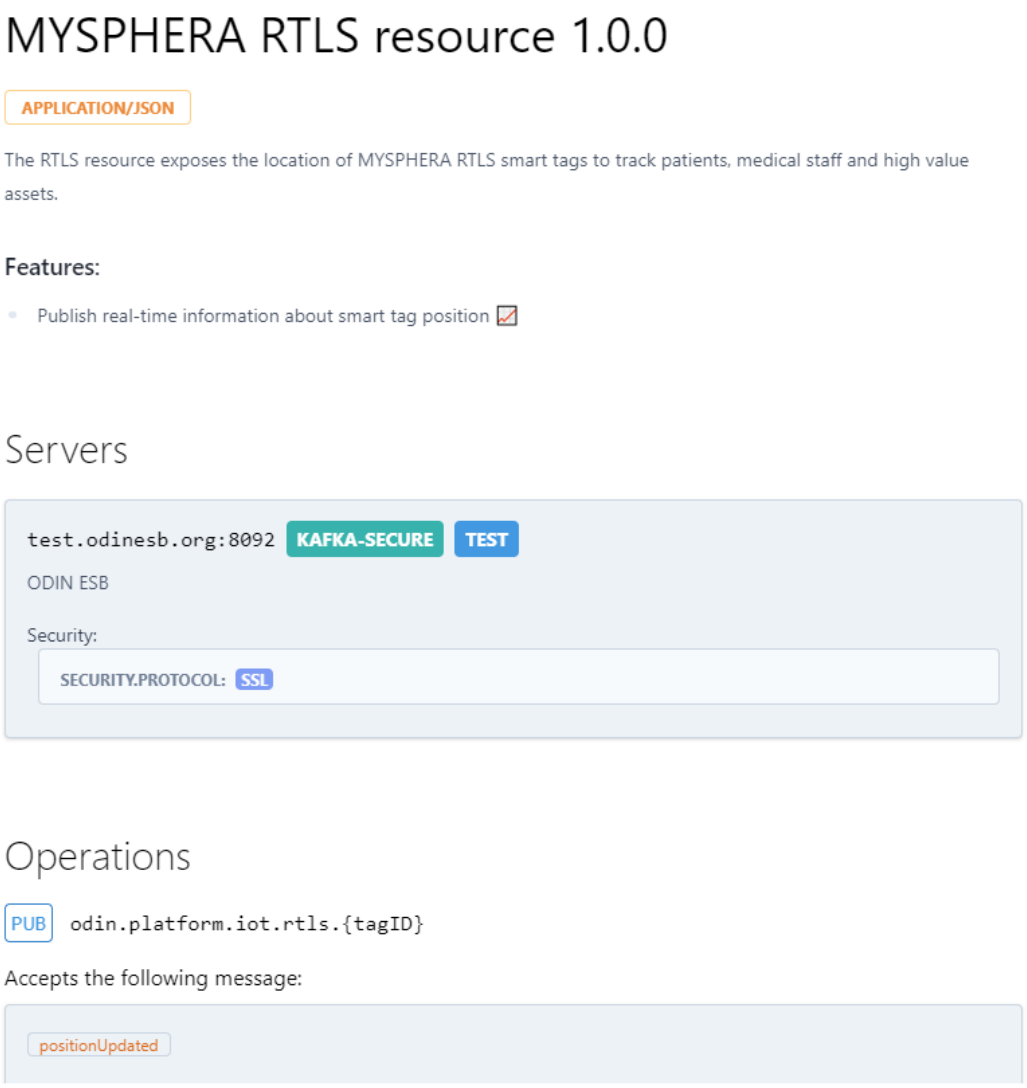

Figure 6: Documentation from RTLS AsyncAPI exmple

2.2.3.2 Web of Things

Web Of Things[7] is intended to be used for IoT devices, especially sensors in smart homes to enable control over lights, doors, alarms, humidity, etc. Some of these sensors can be also

7 WebThings, https://webthings.io/framework/

suitable for smart hospitals that, in the end, are a building just like houses. We can use them to monitor and control different areas of the hospital of the future.
In the ODIN's catalogue[8] we can find one device that would be suitable to use this standard: the Transparent Robot. It consists of a multi-sensor unit with satellite connection to connect smartphones. We can use the Web of Things standard to describe the sensors it contains:

1. Temperature: It sends a float number with the temperature measured in degrees Celsius. The possible values are between [(-40ºC) – 85ºC].

```
{
  "@context": "https://webthings.io/schemas/",
  "@type": ["Temperature", "Sensor"],
  "id": "tcp://exampleodinserver.com",
  "title":"TR Temperature",
  "description": "The temperature sensor in TR",
  "properties": {
    "temperature": {
      "type": "float",
      "unit": "degree celsius",
      "description": "Temperature measured",
      "minimum" : -40,
      "maximum" : 80,
      "links": [{"href": "tr-temperature"}]
    }
  },
  "links": [
    {
      "rel": "properties",
      "href": "tr"
    }
  ]
}
```

Figure 7: Web of Things example for Transparent Robot temperature sensor

2. Humidity: It sends a float number with the percentage of humidity. The possible values are between [0 – 100%].
3. Pressure: It sends a float number with the pressure in Pa. The possible values are between [300 –1100 hPa].
4. Dust: It sends a float number with the percentage of particulates matter in the environment. The possible values are between: [0 – 2.8 x 10 ⁷ pcs/l] measuring up to 1µm diameter of particles
5. Light: It sends a string describing the light brightness. The possible values are: Very Dark, Dark, Light, Bright, Very Bright.
6. Noise: It sends a string describing the noise. The possible values are: Quiet, Noisy, Very Noisy.

[8] D2.3 ODIN Platform Catalogue

7. Air Quality: It sends string describing the air quality. The possible values are Danger, Low Pollution, High Pollution, Fresh Air.
8. Battery: It sends a float number with the device's battery percentage. The possible values are between [0-100%].
9. Time: Current date and time in date format.

2.2.3.3 OpenAPI

In D4.2 we show that OpenAPI[9] is intended for REST applications. Therefore, the cases when this standard will be used is to describe the interaction with devices that use HTTP/HTTPS requests. It is expected that most telemedicine tools use this protocol along with FHIR to communicate with the hospitals remotely and adding the health-related information to the EMR (Electronic Medical Record).

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9

servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing

paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Figure 8: OpenAPI description

2.2.3.4 FHIR

HL7 FHIR is a fully computable standard that combines the best features of HL7's v2, HL7 v3 and Clinical Document Architecture (CDA) product lines while leveraging the latest web standards

[9] Official OpenAPI site, https://swagger.io/docs/specification/basic-structure/

and applying a tight focus on implementation. FHIR solutions are built from a set of modular components called "Resources". FHIR is mainly designed for REST applications, but it can also support document-based, messaging and services-based interoperability paradigms. FHIR resources are typically accessed through HTTP-based REST APIs and can be represented with XML, JSON, or RDF turtle. The RDF turtle representation will be likely superseded in the next release by JSON-LD 1.1. The last published HL7 FHIR release is R4, including normative content.

FHIR is widely adopted at the global level, and it is supported by a large community of practice.

FHIR profiles and implementation guides play a relevant role in the adoption and usage of the base standard, allowing for validation and increasing interoperability. They define, by means of conformance resources, how FHIR should be used in specific contexts and scopes. They also specify which terminologies (e.g. LOINC, SNOMED CT) to use and how.

FHIR is used to define the structure healthcare information can have to exchange it between different computer systems. A full explanation about this standard was presented in D4.2.

It will be used to describe healthcare information in the ODIN platform. It is expected that the KER that most uses this standard is the hospital's Health Information System (HIS). A FHIR server[10,11] would facilitate the interaction of the ODIN platform with the HIS.

This standard has already clearly defined the resources and the elements it describes, therefore there is no need to write one by one all the descriptors. All the information with respect to the resources can be found in the standard's site[12].

The standing role of HL7 FHIR for health, social and wellbeing-related data is nowadays globally recognized. The adoption of HL7 FHIR in specific contexts of use, including ODIN, can be optimized by profiling the HL7 FHIR standard for these scopes. This profiling activity is a standardized process usually documented into FHIR implementation guides. These implementation guides are purely computable specifications, that support the automatic validation of the exchanged resources; the generation of human readable browsable guides for implementers the pilots and enable the tracing from logical to implementable models if of interest. A HL7 FHIR implementation guide will also provide auto-generated testing environment that are built on top of the FHIR implementation guide.

[10] FHIR server by IBM, https://www.ibm.com/blogs/watson-health/ibm-fhir-server-vs-hapi-jpa/

[11] HAPI server, https://hapifhir.io/hapi-fhir/docs/introduction/table_of_contents.html

[12] HL7 official site, https://hl7.org/fhir/resourcelist.html

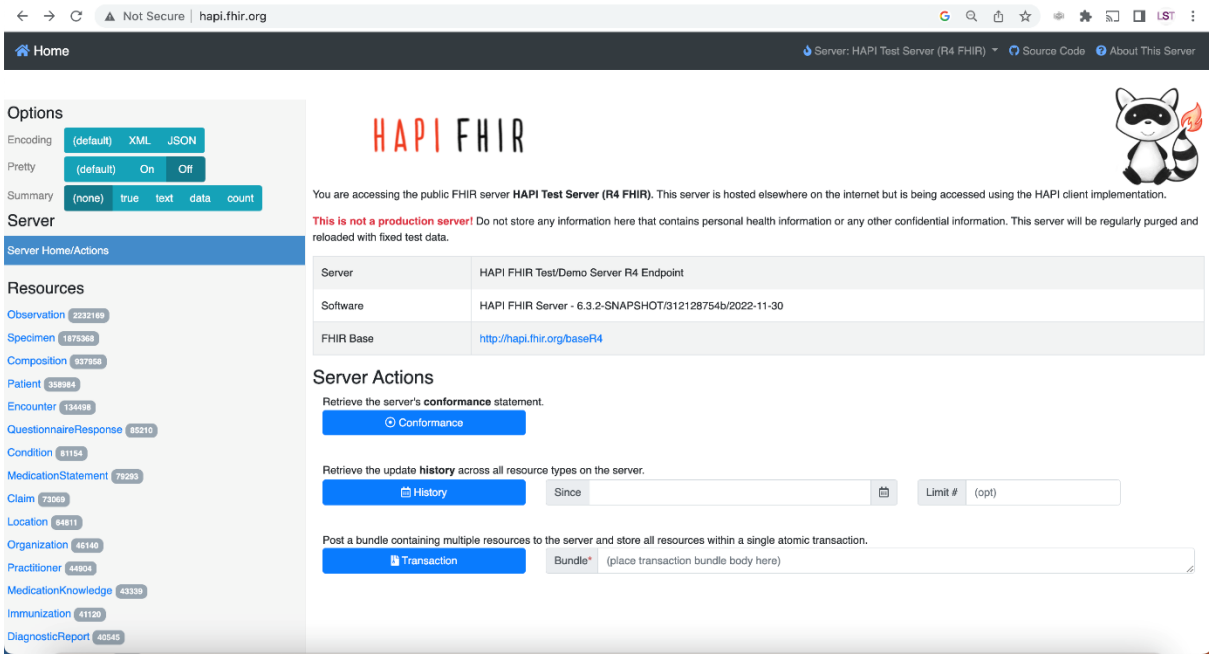

Figure 9: HAPI FHIR main page

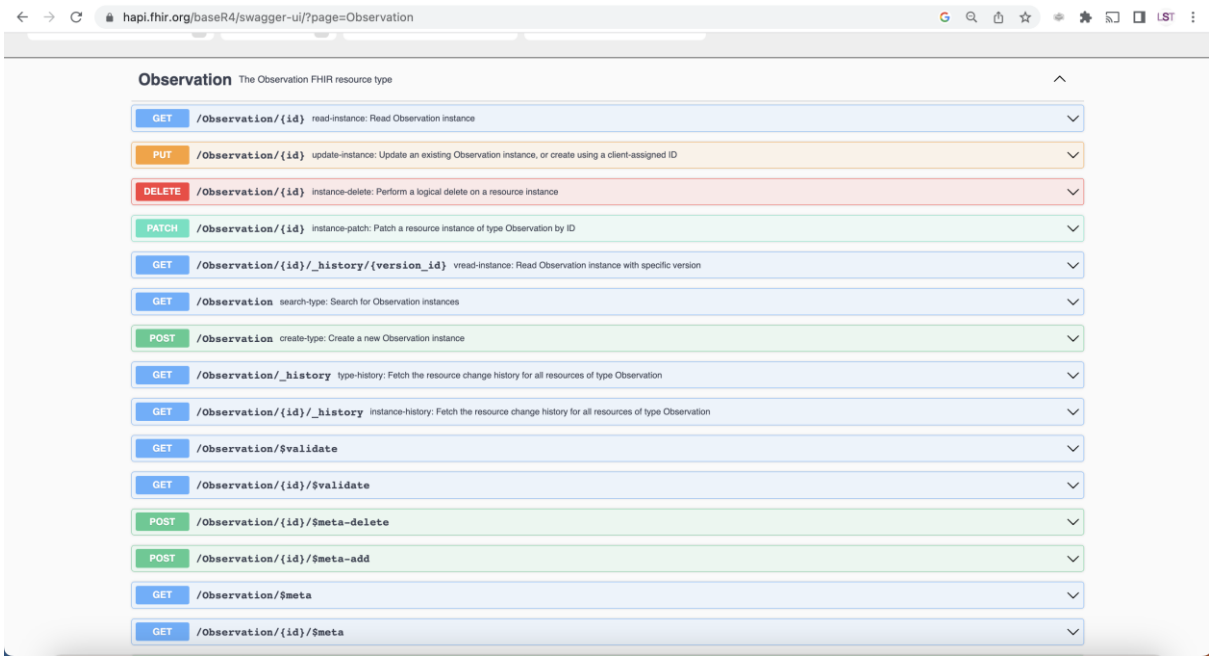

Figure 10: HAPI FHIR observations

2.2.4 Integration of the Selected Technologies

Once we have defined the technologies we will use, it is necessary to implement a tool that integrates all of them and works as the Resource Descriptor manager. The requirements of this tool are:

1. Interaction with a wide range of technologies such as servers, databases, etc.

2. Analyse the incoming data, to check its contents.
3. Adapt the content depending on the functionality described and use the chosen standard for it.
4. Opensource.

Therefore, we need a tool to perform the necessary semantic and syntactic translations. In the end, we the files in the Resource Descriptor registry should be consumed and understand by any component. The selected tool is Apache NiFi.

2.2.4.1 Apache NiFi

Open-source application, initially develop by the US National Security Agency (NSA) and now under the wing of the Apache Software Foundation, design to aid in the integration and automatization of data flows.

The core concepts of Apache Nifi[13] are:

- FlowFile: Each piece of data moving through the system.
- Processor: Component responsible of making certain operation (creating, sending, receiving, transforming, routing, splitting, merging, or processing) on the FlowFile. It is the most important element of NiFi.
- Connection: Link between processors.

It has several characteristics that make it the best tool for the ODIN platform:

- Enables to perform different operations on data flows such as transformation, collection, and uploading.
- Facilitates the interconnection between different systems (databases, mqtt brokers, FHIR servers, etc).
- Can operate within clusters.
- Managed through a graphical interface that helps to visualise the elements interconnected.
- Security mechanisms like TLS encryption.
- More than 300 processors available and the possibility to create new ones. Some of them are: ConsumeKafka, ConsumeMQTT , ConvertJSONToSQL, EncryptContent, ExecuteSQL, PublishKafka, PublishMQTT, PrometheusReportingTask, ExtractHL7Attributes, etc.

The most important characteristic of NiFi is that the processors are like the connectors from Apache Camel, with the added value of the possibility to transform the data. In a following section we will make a performance test between NiFi and Camel to choose the best connecting technology.

[13] NiFi documentation, https://nifi.apache.org/docs.html

2.2.4.2 NiFi in the ODIN Platform

Currently, an instance of NiFi has been deployed in the testing infrastructures in AWS. The application runs as a pod in Kubernetes that exposes its services via a nginx ingress controller. Users connected to the VPN with the appropriate certificate and credentials can access the graphical interface and start creating flows. Additionally, the communication is secured using HTTPS.

Before start creating the flows, we must register to the platform with a user and a password.

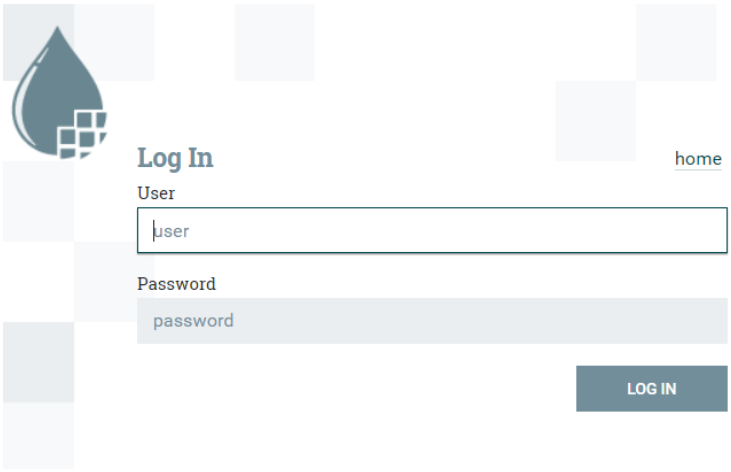

Figure 11: NiFi login page

Once the user is correctly identified he/she can start creating the first flows.

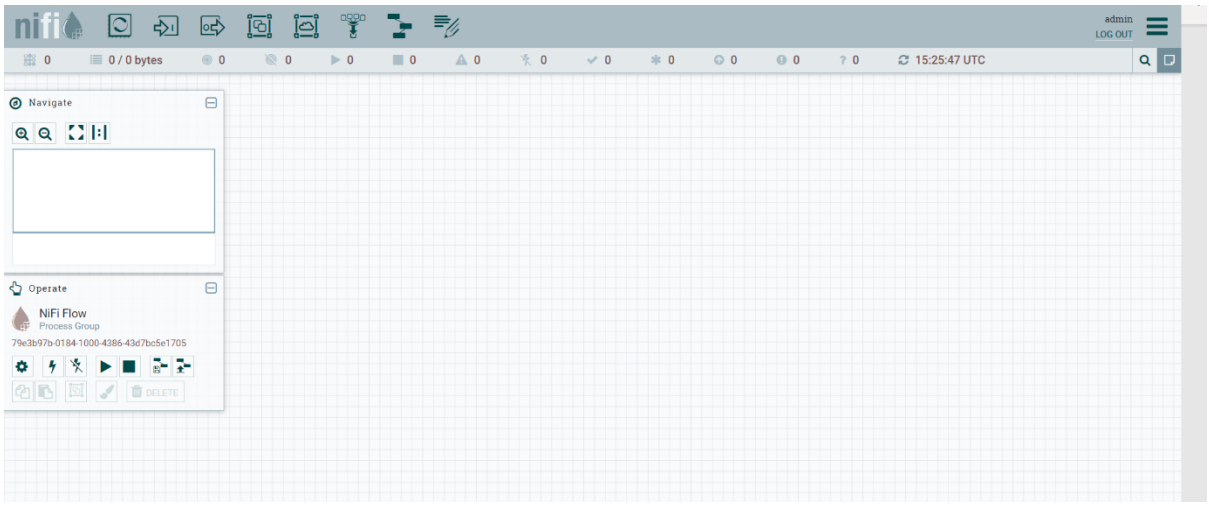

Figure 12: NiFi flows canvas

To test the infrastructure and the tool, an initial data flow between Kafka and a MQTT broker has been successfully created. We have a MQTT consumer that reads data from the MQTT broker and a Kafka Publisher that uploads it to the Enterprise Service Bus implemented with Apache Kafka.

The MQTT broker is also deployed in the testing infrastructure, and it will receive data from the Transparent Robot or other devices that use the MQTT protocol.

2.3 Resource Directory

The Resource Directory will be implemented using Apache Jena Fuseki[14]. Apache Jena Fuseki is a SPARQL webapp, developed by the Apache Software Foundation, it contains a UI for admin and query, and the backend for integration with other systems. It is already published in Docker Hub. It allows users to manage and query RDF (Resource Description Framework) data using the SPARQL query language. RDF is a standard for representing data in the form of statements about resources, where each statement is composed of a subject, predicate, and object. This data model is often used in the semantic web to represent information in a way that can be understood and interpreted by machines. Fuseki provides the SPARQL 1.1 protocols for query and update as well as the SPARQL Graph Store protocol. Fuseki is a part of the Apache Jena open source project to support and boost the adoption and implementation of the semantic web, in fact Fuseki is just the component of the framework which exposes most of the functionality as REST web services, together with a front end as mentioned. Thus, it employs Jena's TDB, the triple store component, to provide a robust, transactional persistent storage layer, and incorporates Jena text query.

SPARQL is a query language for RDF data. It allows users to retrieve, construct, and update RDF data stored in a triple store. SPARQL can be used to query data from multiple sources and retrieve results in a variety of formats. It is a powerful tool for working with RDF data and has been widely adopted in the semantic web community. The Jena framework can further improve the value of SPARQL by applying ontologies and inference to the query. This way relational knowledge embedded in the ontology can be applied to the query and get initially hidden results.

Since ODIN already has a working ontology to describe KERs and other resources (see Deliverable D3.3), it makes sense to employ a triple store and a SPARQL engine to handle the highly dynamic, and heterogenic hospital resources connected to ODIN in a semantic way. Since the backbone of ODIN platform is based on microservice architecture, Fuseki with its REST services fits in the overall concept very well. Thus, the main component of the resource directory is composed of the Fuseki module (bundling other Apache Jena semantic web technologies), enabling raw SPARQL queries to be able to retrieve the appropriate information in the desired format.

In terms of functionality, other components in an ODIN deployment can query for resources of a particular type, with particular capabilities and particular conditions using SPARQL query of the matching elements, this simple query could be enhanced with inference, meaning that the capabilities, or conditions could be expanded. A typical example of this is a location-based query, if the query is, for example, looking for all resources physically located in a room, the inference engine will also include resources <carried by> robots and resources <inside> resources (recursively) which <are located> in the desired the room. The employment of ontologies also means that the semantics of the query can be adapted; for example, a component might be interested in persons, where another component might be interested in employees, when the underlying data is the same semantic technologies are capable of returning the appropriate information to each component without duplicating processes nor data, realising that semantically in certain cases persons and employees are the same datapoint.

[14] https://jena.apache.org/documentation/fuseki2/index.html

On top of the Fuseki engine, the Resource Directory offers a layer of REST services to facilitate common operations, as well as connecting to the ESB to be able to keep the backend model synchronized with the real world; and also ensuring issued queries are secure.

2.4 Possible Limitations

Once we have presented the technologies that will be used for the Resource Descriptor, we must take a step back and analyse them from a more critical point of view to find the possible limitations they might have and in the following deliverable iteration try to tackle them.

Web of Things, AsynAPI, OpenAPI, FHIR

Although these technologies constitute a standard that helps describing a wide variety of protocols, architectures, and implementations. They are not widely as implemented as it was expected yet. Some manufacturers prefer to use their own in-house defined standard for their products. It would be necessary to define a protocol on how to act in these cases were none of the defined technologies suit the manufacturers. In any case, the ODIN platform and its partners will also advocate for opensource and standard technologies.

NiFi

The performance of NiFi in Kubernetes clusters and within a high availability policy are being evaluated.

Additionally, the tool provides with several processors and possibilities, some of them have not been fully analysed yet, since the works have focused on evaluating the alternatives and deploying the chosen one. Nonetheless, those functionalities that would not be available in NiFi by default could be develop and included in the implementation, if necessary.

3 Resource Gateway

The Resource Gateway is in charge of connecting all the resources and services available within a deployment instance but also in other instances from other hospitals to the upper layers of ODIN platform, and finally must allow the interaction of external users or systems with the platform.

As stated in "D3.11 ODIN platform v2" and "D4.2 Implementation of Local CPS-IoT RSM Features v1.0" the connection should be easy, following an Enterprise Service Bus with a canonical messaging system and be scalable and reliable.

3.1 Introduction

3.1.1 Resource Gateway v1

From the ODIN architecture defined in D3.11, here we recall Important services or components for the Resource Gateway:

- The ESB, which is the main communication channel for exchanging messages among services in the platform and other possible channels, such as streaming or peer-to-peer.
- The transport services that connect internal resources and services, such as IoT, robots and other services to the ESB. The transport services read or write information from and to the resources bridging protocols and making them interoperable through the platform.
- The API Gateway, which exposes the platform functionality to external users and external services to the platform in a secure way.
- As part of the work developed during phase 2 of the project, a new component has been attached to the Resource Gateway, the History component. It will be in charge of recording all the messages of the topics shared among the ESB.

As part of the first version of this document, several points were scheduled to be developed and defined for the Resource Gateway.

- The solutions to be used.
- A mechanism to decide the list of the topics that need to be created in the messaging bus related to resources, services, and storage.
- The configuration of the bus to work under the situations to be managed.

3.2 Selected Technology for Messaging Bus

The selected technology for the messaging bus is Apache Kafka. "D4.2 Implementation of Local CPS-IoT RSM Features v1.0" contains the comparison of messaging solutions and Kafka highlighted as one of the best approaches.

Due its capabilities, scalability, and ease of connection, among other features such as rate of adoption in private and public projects, Kafka has been adopted for ODIN.

3.2.1 Kafka

D4.2 already explained the main and general characteristics of Kafka, in this version we will focus on the specific implementation made for the ODIN platform and how the elements have been deployed and work.

The STRIMZI project[15] has been used to deploy Kafka in Kubernetes. In the following lines we will talk about how the architecture looks like using STRIMZI and the final classification for the topics that the KERs will subscribe and publish to.

3.2.1.1 STRIMZI Architecture

STRIMZI eases the deployment of Kafka in Kubernetes be providing with images and Operators, that simplify the process of:

- Deploying and running Kafka clusters and components.
- Configuring and securing access to Kafka
- Upgrading Kafka
- Managing brokers
- Creating and managing topics and users

[15] STRIMZI documentation, https://strimzi.io/docs/operators/0.24.0/full/deploying.html

In the following image we see a scheme of the Architecture.

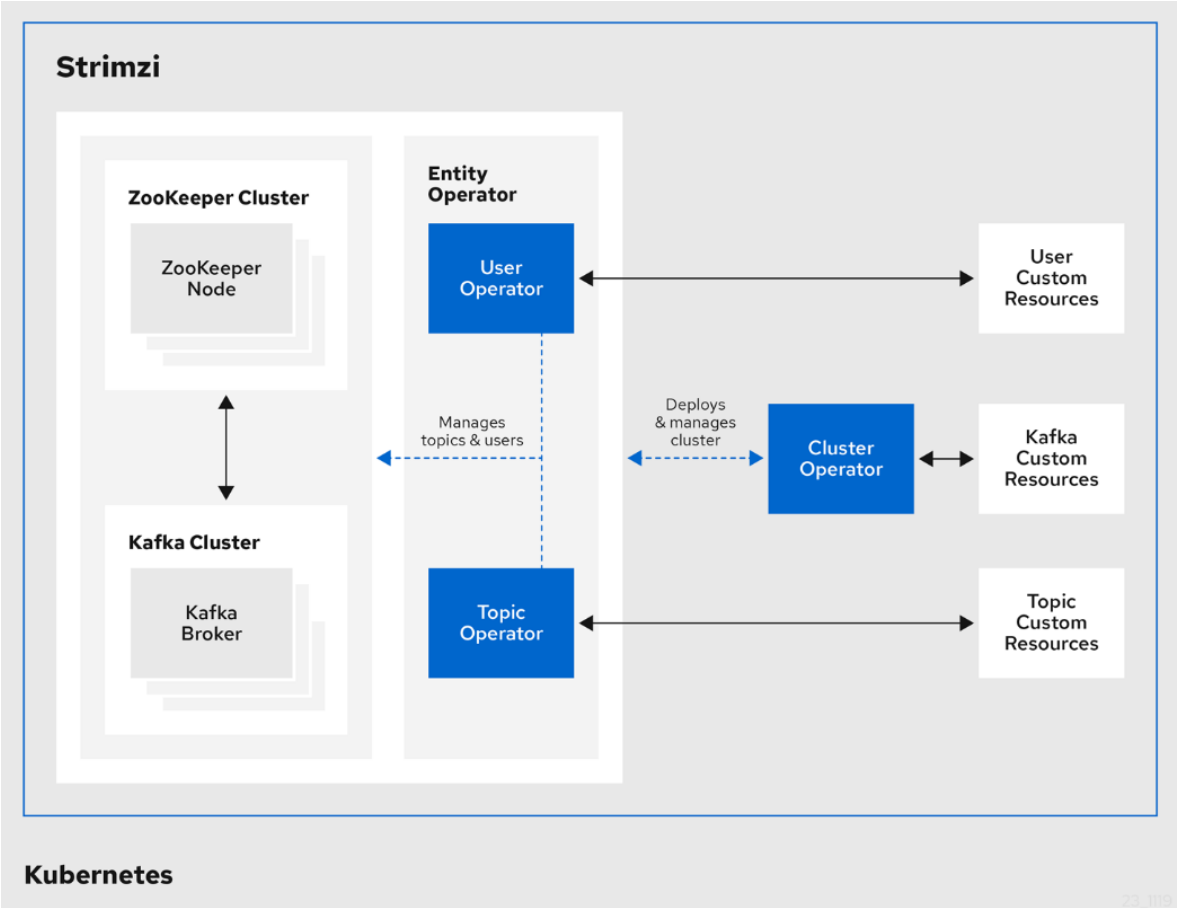

Figure 13: STRIMZI architecture

From right to left we have:

1. Cluster Operator: Deploys and handles Kafka clusters such as Kafka brokers, Zookeeper, Entity Operator and Kafka Connect.
2. Entity Operator: Contains:
    a. User Operator: Handles users using Kubernetes resources, that specifies the authentication and authorization methods, the access to topics and the rights.
    b. Topic Operator: Handles topics using Kubernetes resources. Allowing to perform create, delete, or modify operations on topics and keep them in sync.

We can also see that there are custom resources that are used to define the specific configuration of the clusters.

The figure above presents the standard minimal deployment. Additionally, we can include the Kafka Connect cluster that deploys connectors that enable the communication between Kafka and other messaging systems, databases, and servers for external data streaming.

3.2.1.2 Connectors

The Kafka Connect Operator is the component within the STRIMZI architecture responsible of managing and deploying the specific connectors, but we must indicate and provide with the

connectors we want to use. These come from section 3.4 “Selected Technologies for Transport Services”.

3.2.1.3 Topics

Topics are the message queues that can be created in Kafka to publish and read messages. Part of the work to create a good ESB is to select the types of topics and the number of topics.

After checking the different strategies to create the topics, it was decided to have a low number of general topics, where the resources could publish or read the messages grouped per types. The other option, use one topic per resource, would create a lot of operational work. Thus, the consortium has decided to use an iterative approach and in case a deeper granularity is required in some case, the platform will evolve to manage those situations.

Another important decision was to have 3 levels of topics: Core, Platform, and Custom topics.

- Core topics: those that are intrinsic to the ODIN architecture such as metrics, Resource Manager information…. All defined in Table 3. The structure would be as follows: Base name (i.e. odin.core.) + Component’s name (i.e., metrics.) + Specific topic (i.e., control). That leaves the following topic: “odin.core.metrics.control”.
- Platform topics: Those that are defined by platform components, as well as standard dataflows (e.g. common IoT statements) such as: “odin.platform.iot.rtls.tagid”.
- Custom topics: new Resources may define custom topics to cover dataflows not fitting in the previous topics (e.g. temporal topics) such as: “odin.custom.non-anonymized-data”

The following tables contain the first version of the Topics that would fit into ODIN platform.

Core topics

Table 3: Core topics

Component	Base name odin.core	Subscribers	Publishers	Description
Metrics				
	metrics.control	All	metrics	Messages to control actions on metric publish, etc…
	metrics.status	ODIN control components	metrics	messages about metric component status
Resource Choreographer				
	resourcechoreographer.control	Components and services	Resource Choreographer	messages

Component	Base name odin.core	Subscribers	Publishers	Description
Resource Manager				
	resourcemanager.control	Components and services	R. Manager	Messages about controlling services, general control, not task choreography. For example, the platform kicks a service due to security problems.
	resourcemanager.joined	R.Manager or any service that wants to be aware of KER and services joining the platform	Components and services	Messages sent by components to join the platform. Messages will include information about interfaces published, schemas, etc. (Async API, WoT, OpenAPI, FIHR)
	resourcemanager.left	R.Manager or any service that wants to be aware of KER and services leaving the platform	Components and services	Messages sent by components to leave the platform and stop offering services.
DLT				

	resourcefederation.control	Any resource	resourcefederation	Messages sent from DLT to other resources to perform tasks on behalf of another hospital request
	odin.hospital.hospitalid	resourcefederation	services that want to use toher hospital's resources	
	odin.cloud	resourcefederation	services that want to communicate with cloud services or resources	
Component	**Base name odin.core**	**Subscribers**	**Publishers**	**Description**
DeadLetter				
	deadletter	Metrics	all	Topic to manage several situations. Topic does not exist Message with errors Message reaches a threshold read counter number, because it is not consumed. The message expires due to per-message TTL (time to live)[4] Message is not processed successfully
log				

	log		all	Any information to be logged
	log.error		all	
	log.warning		all	
	log.debug		all	When debug is active

Platform topics

Table 4: Platform topics

Component	base name odin.platform	Subscribers	Publishers	Description
RTLS	iot.rtls			
	iot.rtls.tagID	Any service/KER interested in a concrete tag position	RTLS	Position and events about a tag
Transparent robot	iot.temperature	Any service/KER interested in measurement data	Transparent robot or other sensors	Events about temperature
	iot.humidity			Events about humidity
	iot.pressure			Events about preassure
	iot.airquality			Events about air quality
	iot.light			Events about light
	iot.dust			Events about dust
	iot.noise			Events about noise
Robot	robot.status	Resource Orchestrator, Resource Manager, other KER interested in robot status	robots	Data about robot status. battery, available/busy, etc.
	robot.inventory	Resource Manager, other service	robots, smart boxes	Data about items carried by a robot.

Component	base name odin.platform	Subscribers	Publishers	Description
		controlling inventories		I.e , smart box, catheters.
	robot.control	robots	Resource Manager, Resource Orchestrator, Web applications to control robots	Robots receive instructions and commands to perform actions in this topic.
AI	ai.control	ai	Resource Manager, Resource Orchestrator, other web apps controlling ai jobs	Messages to load/unload AI models, launch jobs, etc.
Data				
HIS	data.his.control	HIS	Any KER interested in sending commands to HIS service	Messages to request a job, send FIHR data, etc.
	data.his.result	Any KER that requested a command	HIS	Job results
SQL BBDD	data.sql.control	sql bbdd	Any KER interested in sending commands to BBDD	Commands for BBDD (update, create, insert, …)
Nosql BBDD	data.nosql.control		any KER interested in sending commands to NoSQL BBDD	commands for NoSQL BBDD (update, create, insert, …)

3.2.1.4 Implementation

Currently, there is a Kafka deployment based on STRIMZI in the testing infrastructures of the project that contains the following features:

- 1 Entity Operator with 3 replicas.
- 3 Kafka brokers.
- 3 Zookeeper pods.
- 1 Kafka Connect Operator.

- 1 MQTT-Kafka source connector. That can read messages from a MQTT broker and write them to Kafka.
- 1 external connection to communicate with Kafka from outside the Kafka cluster.
- A UserOperator and a TopicOperator.

```
NAME                                                  READY   STATUS    RESTARTS          AGE
pod/my-cluster-entity-operator-7788896d67-qdhl4       3/3     Running   91 (3h53m ago)    27d
pod/my-cluster-kafka-0                                1/1     Running   40 (3h54m ago)    27d
pod/my-cluster-kafka-1                                1/1     Running   7 (3h54m ago)     3d3h
pod/my-cluster-kafka-2                                1/1     Running   27 (3h54m ago)    17d
pod/my-cluster-zookeeper-0                            1/1     Running   31 (3h55m ago)    27d
pod/my-cluster-zookeeper-1                            1/1     Running   32 (3h55m ago)    27d
pod/my-cluster-zookeeper-2                            1/1     Running   28 (3h54m ago)    27d
pod/my-connect-cluster-connect-9cf968f59-fflmg        1/1     Running   44 (3h55m ago)    27d
```

Figure 14: Kafka STRIMZI pods

```
NAME                                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
service/my-cluster-kafka-external-bootstrap   NodePort    10.43.68.178    <none>        9094:32100/TCP               27d
```

Figure 15: Kafka STRIMZI service

3.3 Technologies for the API Gateway

The deliverable D4.2 “Implementation of Local CPS-IoT RSM Features v1.0” did not contain a real comparison of solutions for the API Gateway, because the requirements at that moment were perfectly covered by the proposed solution. In time between deliverables, new requirements have arisen that made necessary to evaluate additional technologies. Therefore, a comparison is included here, plus a clearer definition of the expected functionality.

3.3.1 API Gateway functionality

The API Gateway (AG) exposes the platform functionality to external users and external services to the platform in a secure way. Refining the functionality, here is a non-exhaustive list of the functions the API gateway will implement:

- Synchronize with the Resource Manager (RM) to expose, update or hide new resources that are added to the platform. In this way, when a new resource is added, in case it has an API available that has to be exposed outside the platform, the API Gateway will get an update from the RM and read the information available to publish the API in a secure and ordered way.
- Act as a reverse proxy to external clients. So, the AG will forward incoming calls to the right backend component/endpoint and answer with the response. REST is a must, but compatibility with GRAPHQL and GRPC is a plus.
- Unify the component API’s into one homogeneous resource. For example. If two services, RTLS and AI offer an API, those could be exposed as “https://api.odin.eu/ker/ai/ai_1” and “https://api.odin.eu/ker/rtls” for example, with a common root, simplifying the integration.
- Secure the services access with Keycloack integration.
- Transform incoming or outgoing data in case it is needed.
- Force rate and throttling policies.

3.3.2 KrakenD

KrakenD[16] is an opensource API Gateway, focused on linear scalability and low operational costs to offer a single point of access to microservices from external clients. About scalability KrakenD shows better benchmarks for requests/second than their competitors such as Kong or TYK. This is achieved thanks to its stateless design, which does not rely on databases.

But it is not only a simple API gateway, because it aggregates lots of microservices to process requests and responses, as visualized in Figure 16.

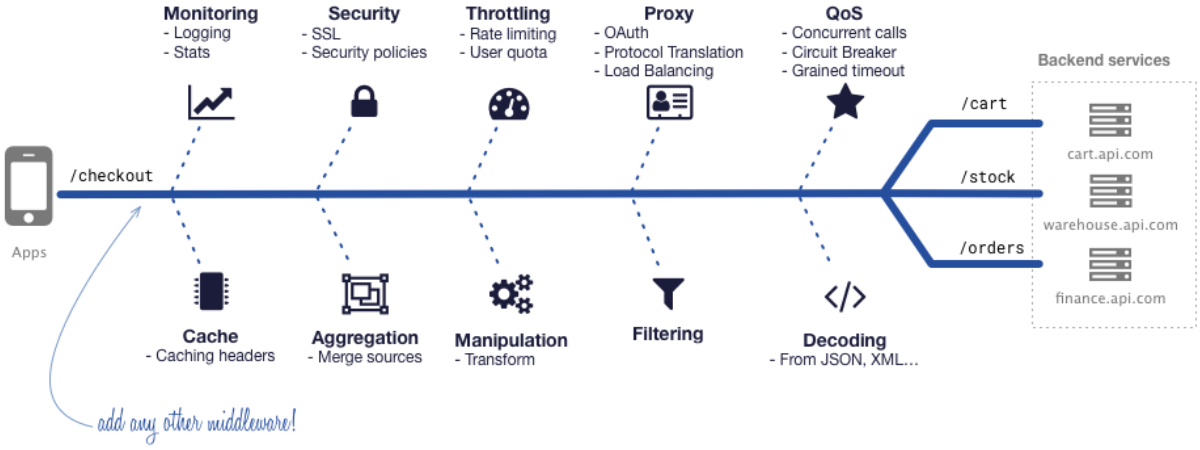

Figure 16: KrakenD conceptual microservice architecture

KrakenD can transform, aggregate, or remove data from your own- or third-party services. It also implements some patterns to hide the complexity to deal with multiple REST services, isolating clients from the micro-service implementation details. This way it can transform the requests and unify them for example under the same root.

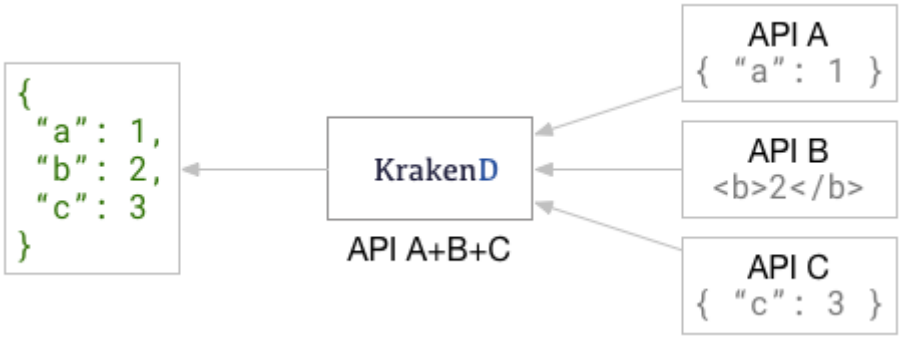

Figure 17: Merging services

[16] https://www.krakend.io/

All the setup for the API Gateway can be done through a web application called KrakenDesigner, but it also can be done writing a json and loading into KrakenD using the console.

Moreover, KrakenD has Graphql support through translating agents, opening the integration to other types of clients, beyond REST style.

KrakenD also provides connection to Async sources and consumers, for example to RabbitMQ or Kafka, easing the tasks to interact KrakenD with the rest of KERs and services in the platform.

To extend its functionality, KrakenD has lots of plugins, but also they that can be created using LUA, Martian DSL, Google Cel or Go. Some of the most important integration it offers for ODIN purposes are Kafka, Prometheus metrics and Keycloack integration, all, solutions already selected for ODIN.

One of the missing features is static content delivery, but due to ODIN platform nature, static content may be reduced to documentation.

Security

KrakenD manages most common security features such as OAUTH, JWT, SSL, mutual certificates, and a list of protections for clickjacking, XSS, HSTS, HPKP, and mime sniffing.

Another security feature beyond securing the API is to remove sensitive data, so it can be used to anonymize responses from the services or inputs from the clients.

Rate limit also is available to prevent flood attacks and manage API rate access.

For ODIN purposes, one of the key features it is the integration with Keycloack. This integration allows that all the Authentication can be moved to Keycloak. So a client connecting to an API through KrakenD, first must get a JWT token from Keycloack, and then use it against KrakenD. KrakenD still manages the Authorization setup, and checking the JWT can determine whether to grant a client to access an API or deny it. Following image shows the process to authenticate and authorize a client.

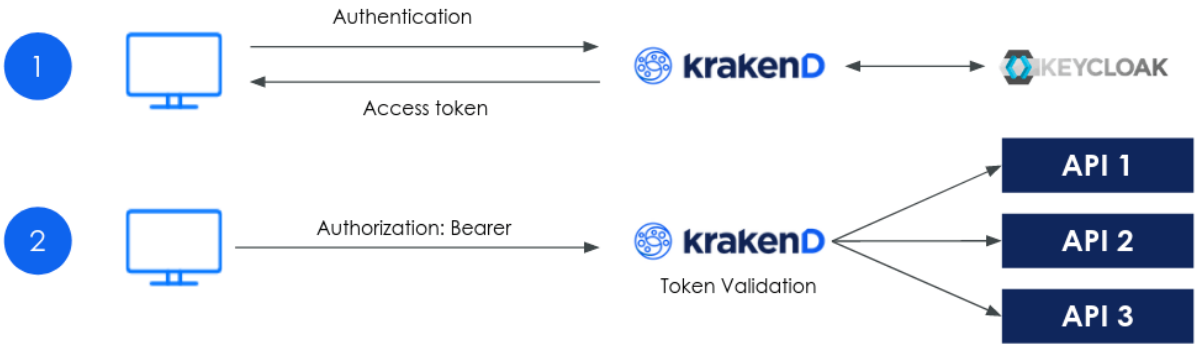

Figure 18: KrakenD security mechanism

Metrics

KrakenD offers access to metrics and has integration to many tools with Open Census telemetry specification. Important integrations are Prometheus, Graphana and Logstash.

For example, the Graphana Dashboard includes:

- Requests from users to KrakenD

- Requests from KrakenD to your backends
- Response times
- Memory usage and details
- Endpoints and status codes
- Heatmaps
- Open connections
- Throughput
- Distributions, timers, garbage collection and a long etcetera

As Prometheus is supported, the ODIN metric system has direct integration.

Setup and Deployment

KrakenD supports most types of deployment as bare metal, clusters, Docker volumes and images and Kubernetes, plus CI/CD facilities.

This eases how KrakenD is adopted and how operational issues are solved.

The setup can be performed from the managing dashboard or setting up the configuration files which follow JSON format. This opens the opportunity to manage KrakenD despite it does not have an API for setup purposes.

3.3.3 TYK

Tyk is an API Management and API Gateway solution, focused on providing a solution which does not rely on previous open-source solutions, with no lockouts among its open source and enterprise paid solution.

Tyk is an open-source Enterprise API Gateway, supporting REST, GraphQL, TCP and gRPC protocols, which is a desired feature for ODIN due to its objective to allow as many integrations as it can with ease of adoption.

It also allows for data and resource transformation, thus it can convert from SOAP to Graphql for example, remove or transform data payload and unify resources.

Importing Swagger and OAS2/3 to scaffold an API is very easy, so you can create the skeleton of a service with very low effort.

Tyk API allows managing all the API Gateway programmatically, which is very appealing to automatize publishing new endpoints.

Another plus is the plug-in architecture. Tyk can add more functionality with plugins developed using Go, javascript, Python or any language which supports gRPC.

To add some features, Tyk supports API versioning, Developer portal to document API's (paid version), hot configuration reloading, webhooks, and has a very nice performance in terms of request resolved per second, but not as good as KrakenD.

Security

Tyk offers OIDC, JWT, bearer Tokens, Basic Auth, Client Certificates and more, plus a very granular access control, which controls to grant access to one or more APIs on a per version and operation basis. Blacklisting and whitelisting are other features part of the security package.

On the other hand Tyk does not have direct integration with Keycloak out of the box for the open source version, but offers OpenID Connect integration through Tyk Identity Broker. Integrating authentication and authorization with Keycloak may involve some sort of setup, harder than using APIMAN or KrakenD.

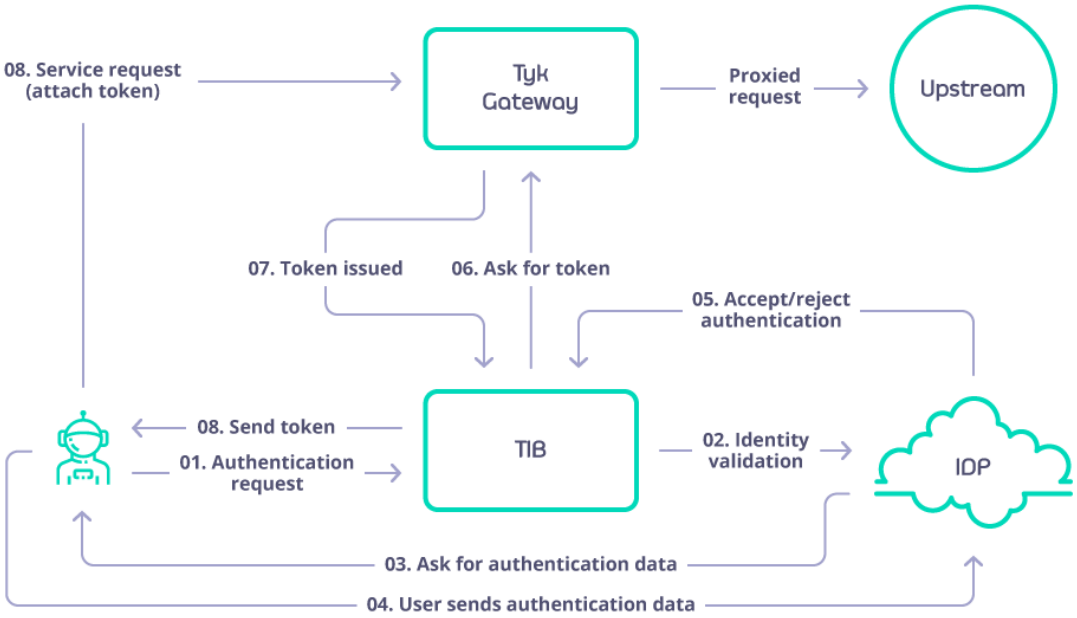

Figure 19: OpenID Connect authentication flow

Quotas and rate limiting can be setup also for the API's, giving a good protection and control over the number of requests allowed. There are global rate limit and per API limit, which controls all the users accessing an API.

Metrics

Tyk uses its Pump server to provide metrics about what is happening in the API Gateway and the endpoints. With pump, the metrics are extracted from Tyk and sent to any compatible store, for example Prometheus which is very interesting to ODIN, but also MongoDB, ElasticSearch, Kafka and others.

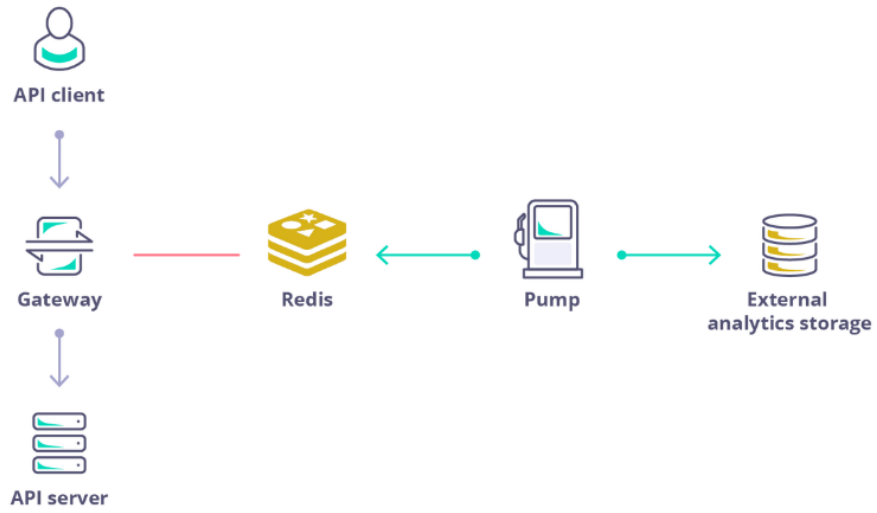

Figure 20: Tyk Pump component extracting data to a store

Setup and deployment

Tyk supports Docker, and Kubernetes. In addition has Kubernetes native declarative API support using Open Source Tyk Operator . Tyk Operator can configure Tyk Gateway in a drop-in fashion, replacing standard Kubernetes Ingress. It allows managing your API definitions and security policies with it.

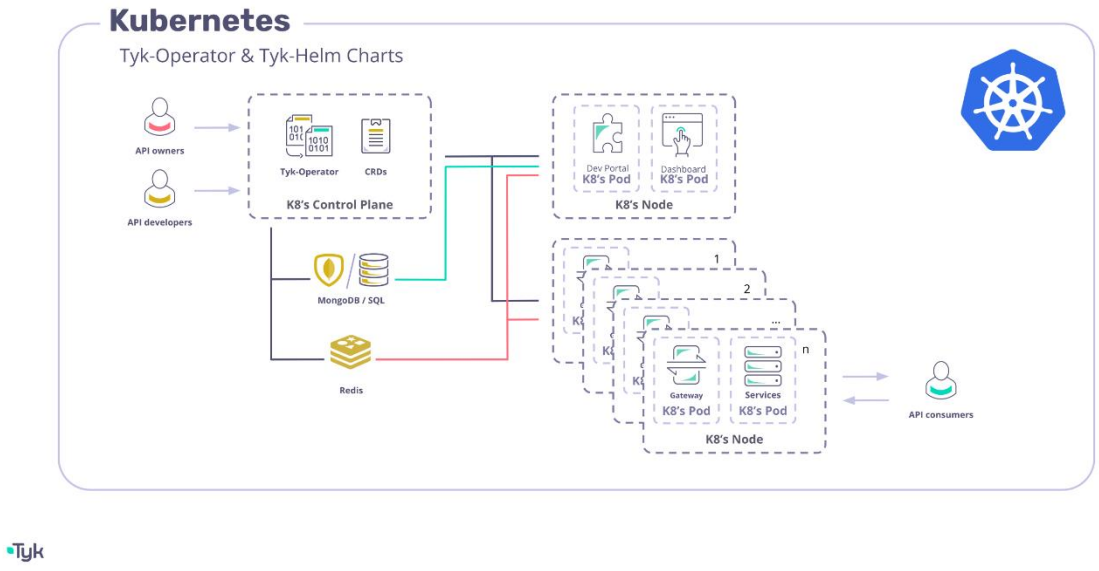

Figure 21: Tyk operator for Kubernetes

Tyk also supports setup through its API and using the setup files, but it lacks a UI to manage it, only available for paid versions.

Regarding dependencies, Tyk only has a dependency on Redis database.

3.3.4 Istio

Istio is a different type of solution, and it is not only an API Gateway. Istio supports Kubernetes to provide a programmable, application-aware network using the Envoy service proxy. Working with Kubernetes Istio brings universal traffic management, telemetry, and security to complex deployments.

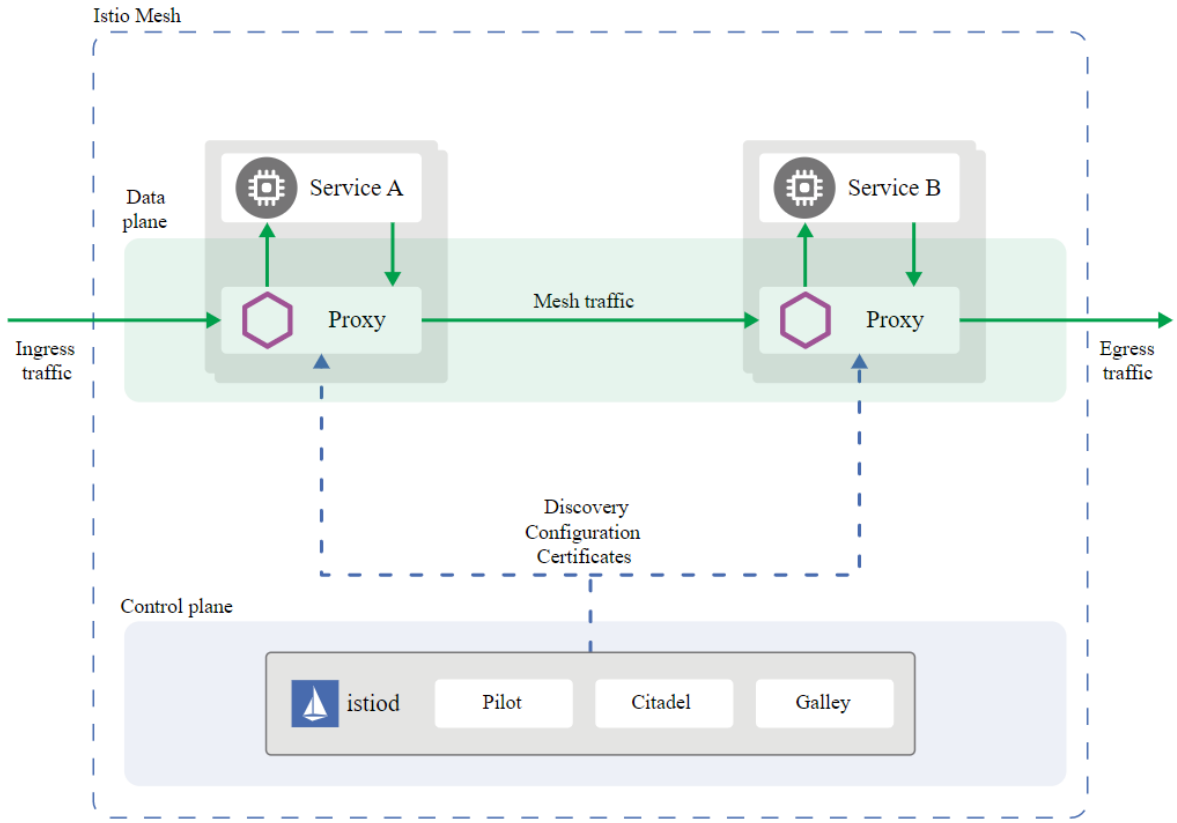

Figure 22: Istio concept

Figure 22 shows the concept of Istio, managing in and out traffic, the side-car proxy to connect the services and some features managed by Istio control plane.

For traffic management, Istio has several features that make it attractive. With the concept of service mesh, Istio handles ingress traffic into the Data Plane services located as a mesh under the control of the Control Plane. Inside the mesh, the traffic can travel between the services, while the Control plane handles service discovery, setup, and metrics.

Istio provides quite interesting services when managing traffic, such as traffic redirection, load balance, dynamic routing requests, circuit break, timeouts, traffic mirroring ot copy data to more than one destination, traffic shifting to test and adopt new versions of an API or service and managing ingress/egress traffic to and from the mesh.

Most of the work for Istio is performed using the concepts of Gateway and Virtual Service components. Gateway is used to manage the incoming and outgoing traffic, while the Virtual Service is a representation for an application or service running in the mesh.

Beyond its traffic management and all the other features, Istio is currently becoming the API Gateway for Kubernetes. Recently has developed an implementation for the Kubernetes Gateway API specification using Istio, which provides a set of Kubernetes configuration resources for

ingress traffic control that overcomes Kubernetes Ingress limitations[17], and can be used together with the service mesh or without it. This new feature is a very interesting one for ODIN's automation and operational processes.

Istio does not have a management console or a developer portal but its integration with Kubernetes help handling operation tasks.

Security

Istio offers security features for the outer perimeter and inner perimeter in the service mesh. For example, it supports traffic encryption to defend against man-in-the-middle attacks, it provides service access control with fine-grained access policies, authentication, and authorization, and it also has auditing tools to determine who did what at what time.

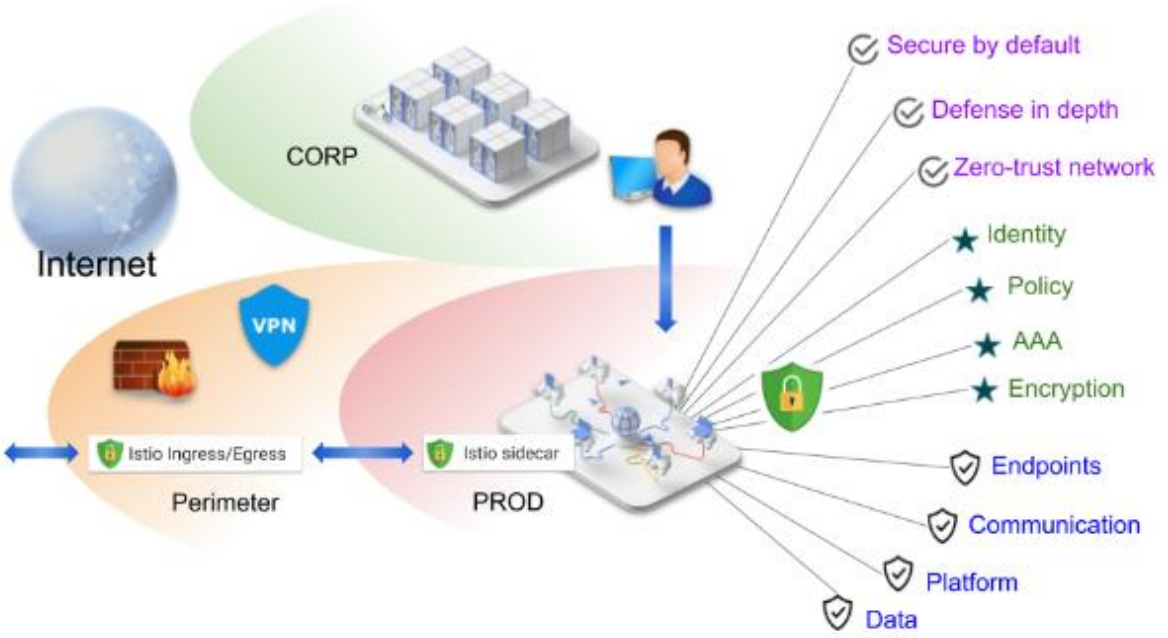

Figure 23: Istio security concepts

Istio has its own Certification Authority to manage the certificates used in the infrastructure, an strong Configuration API Server to manage authentication, authorization and security policies and other services that help keeping the services in the right operation such as side-car extensions to monitor telemetry and manage communication among services in the mesh.

[17] https://istio.io/latest/blog/2022/getting-started-gtwapi/

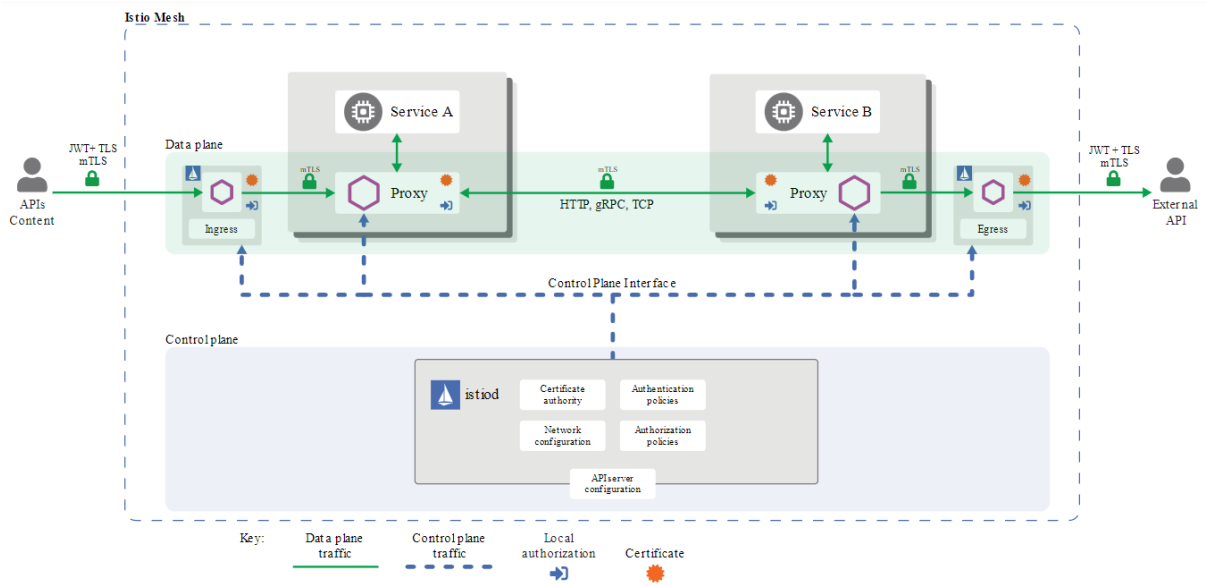

Figure 24: Istio security architecture

Istio also supports Keycloack and other Identity providers to delegate authentication, such as Auth0 or Google Auth.

Managing policies with Istio has some advantages as they are committed on demand to the services, thus, managing changes and operational tasks is easy when adapting the infrastructure and applications to the project needs.

Istio supports HTTP, GRPC and many TPC traffic managing, which opens a wide range of protocols support.

Metrics

Istio has several ways to get metrics from the service-mesh in case you use it but also supports ingress and egress traffic metrics related to the Gateway API. Storing Requests per service or traffic type can be done with the right setup and then inspect the results using Prometheus which is a a desired integration tool.

Examples of supported metrics out of the box are:

- Request Count (istio_requests_total): This is a COUNTER incremented for every request handled by an Istio proxy.
- Request Duration (istio_request_duration_milliseconds): This is a DISTRIBUTION which measures the duration of requests.
- Request Size (istio_request_bytes): This is a DISTRIBUTION which measures HTTP request body sizes.
- Response Size (istio_response_bytes): This is a DISTRIBUTION which measures HTTP response body sizes.
- gRPC Request Message Count (istio_request_messages_total): This is a COUNTER incremented for every gRPC message sent from a client.

- gRPC Response Message Count (istio_response_messages_total): This is a COUNTER incremented for every gRPC message sent from a server.

Setup and deployment

Istio setup and deployment can be performed using several tools, but the most important for ODIN are the support of Helm charts and the Kubernetes Operator, which are aligned with ODIN objectives and tools.

3.3.5 API Gateway solution comparison

The following table contains the comparison of the solutions.

Table 5: API Gateway technologies comparison

Specific requirement	APIMAN	KrakenD	Tyk	Istio	Reason
Is what is needed	9	8	10	9	All are API Gateways and fulfil the needs. APIMAN offers more management that KrakenD, plus an API Rest to manage APIMAN, that Krakend can only be setup using files.
Ease of adoption	9	9	9	8	All the solutions have very good documentation but Istio is closer to Kubernetes
Scalability	7	10	8	10	Tests reports KrakenD is more scalable than the other API Gateway solutions
Reliability	10	10	10	10	With the proper setup all the solutions are reliable.
Security (Kerberos, OAuth, etc.)	10	8	7	10	All can use Keycloak more or less, but APIMAN and Istio can handle authorization better
Data integration	5	8	10	9	Kraken supports Kafka integration out of the box, plus Graphql, while APIMAN not. Tyk offers integration for Graphql, Grpc, Istio is fully integrated with Kubernetes and manages more type of traffic.
Support	8	8	8	8	All have very nice documentation. Direct support is provided under paid versions or through the community forums.

Deployment facilities	5	10	10	10	Deploying Istio, KrakenD and Tyk into Kubernetes have better support in general. Operational task may be easier than using APIMAN
Size of project to be used	9	9	10	10	Due to scalability and operational features Kraken can manage bigger projects in term of throughput and resource and data transformation, but APIMAN has more management features which are enterprise interesting and it also scales pretty well. Tyk also scales well and has more management facilities than Kraken
Cost	10	10	10	10	All are free.
Licence	10	9	9	9	All are open-source but KrakenD community version does not have all the features as the Enterprise version
Number of projects using it	-	-	-	-	No information about the number of projects found.
Number of languages	1	1	3	1	Java for APIMAN. GO for KrakenD plus several scripting notation. Tyk offers Python, Javascript and Go. Istio provides C++
Monitoring	7	10	10	8	KrakenD and Tyk have more integration plugins for metrics and have Prometheus integration
Hard dependencies on other projects or solutions	1	-	1	-	APIMAN uses Elasticsearch to store metrics and Tyk Redis database.
Innovation impact	8	7	9	10	There is no special innovation impact on using an API Gateway, but the way is going to be used in ODIN gives a kind of innovation to all, proportionally to the number of managing facilities and compatible interfaces. Istio approach is more advanced than other solutions as works directly for clusters and with microservice mindset.

Due to ODIN requirements, objectives, and its adoption in the developer community, Istio and Tyk seem to represent the two best options to be used as API Gateway. The facilities, supported protocols, Kubernetes integration, metric reporting, and security, make both a serious choice with long term mindset. Anyway, Tyk offers the API Manager, a lighter solution in terms of setup and a more traditional API Gateway solution.

Therefore, Tyk will be selected as API Gateway for ODIN.

3.3.6 API Gateway final approach

The API Gateway will connect the exposed services directly to the internet. Thus, the Kafka bus will not be used to forward the calls from clients to access the services.

On the other hand, some operational tasks could be handled through the Kafka bus using a connector.

With this approach, the API Gateway would listen to messages on key topics such as when a KER wants to publish a service, and read the Resource Descriptor from the Resource Manager, to publish the API specification, using the right security and transformation policies. Updating an API or security policies could also be performed this way.

Because API Gateways have Kubernetes support can be managed through declarative operators, publishing task could also be driven using Kubernetes facilities.

3.3.7 Limitations of Tyk

A possible limitation for Tyk, the selected technology for the API Gateway, could be that Tyk does not have a Dashboard or Developer portal, as reviewed in the comparison, but has a powerful API to be integrated and documentation could be shared using ODIN documentation component.

Beyond functionalities, Tyk has a wide enough community for support and a company behind its developments. Therefore, there is no reason to worry about future deprecation of the tool.

3.4 Transport Services

The transport services are represented in the architecture by the Connector components.

The initial protocols to be implemented due to the requirements are MQTT and Kafka connectivity for the RTLS.

Those connectors must act as a bridge among the protocols and messages managed outside the platform, and the ESB.

Thus, the functionalities to be implemented are the following.

For input connectors, publishing information into the Kafka ESB:

- Adapt the outer protocols to Kafka
- Transform the information coming from the resources to the messages defined to work in Kafka for each component.
- Publish the information into the right topics
- Be compliant with the messages required by the platform (e.g. Resource Descriptor message to publish to the Resource Manager.

For the output connectors, reading information from Kafka and sending it to the components:

- Adapt the inner messages and protocols of the platform, to the KER's
- Listen to the required topics
- Be compliant with the messages required by the platform (e.g. messages from the Resource Descriptor or the Resource Choreographer)

3.4.1 Implementation

Considering the required features there are three ways to implement the connector components: in-house development (RTLS example), Camel plugins (explained in D4.2 section 3.5.1[18]) or NiFi processors.

3.4.1.1 Apache Kafka Connector (RTLS)

The RTLS has implemented a first version of its connector using Apache Kafka Connect technology, which allows using it as a centralized data broker for simple data integration between databases, search indexes, and files to drop them into Kafka.

Currently, the connector implements getting the messages from the RTLS and processing them to follow a json schema. In this case, a source component from Apache Connect has been used to read the data from the RTLS and publish it to the Kafka bus.

The following image sketches the architecture followed by the RTLS connector, where the connector reads from a source using an adapter, process the data and forwards it to the destination using another adapter.

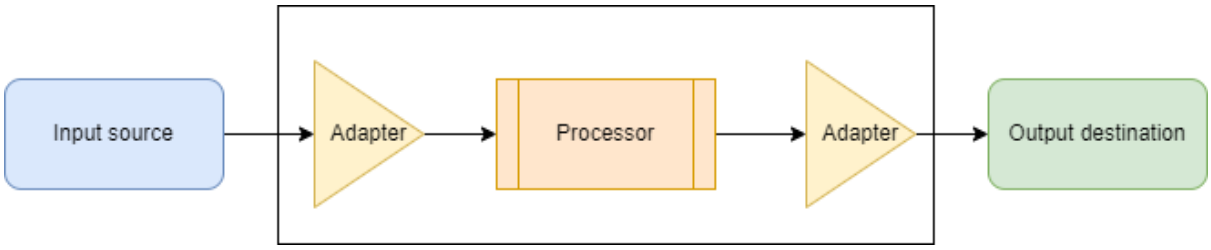

Figure 25: Connector Architecture

The following figure contains the POJO for the RTLS message. Highlighted there are the message's attributes. The most important ones for the RTLS are the x, y and z coordinates, that define the position of the Tag in the map. The attributes more important to the ODIN platform are: the "version" (to differentiate between changes on messages versions), the "message" (that provides additional information with respect to the whole JSON message been sent) and, finally, the "action"(to indicate certain action with respect to the message type).

[18] D4.2 Implementation of Local CPS-IoT RSM v1

```java
package com.mysphera.odin.connect.spheraq;

import org.apache.kafka.connect.data.Schema;

public class SpheraqPositionSchema {

    private SpheraqPositionSchema() {
        throw new IllegalStateException("Utility class");
    }

    // Position fields
    public static final String VERSION_FIELD = "version";
    public static final String ID_TAG_FIELD = "id_tag";
    public static final String TIMESTAMP_FIELD = "timestamp";
    public static final String MESSAGE_FIELD = "message";
    public static final String ACTION_FIELD = "action";
    public static final String X_FIELD = "x";
    public static final String Y_FIELD = "y";
    public static final String Z_FIELD = "z";
    public static final String SEQUENCE_FIELD = "sequence";
    public static final String RADIUS_FIELD = "radius";

    // Schema names
    public static final String SCHEMA_KEY = "position_key";
    public static final String SCHEMA_VALUE_POSITION = "position";

    // Key Schema
    public static final Schema KEY_SCHEMA = SchemaBuilder.struct().name(SCHEMA_KEY).version(1)
            .field(ID_TAG_FIELD, Schema.STRING_SCHEMA).build();

    // Value Schema

    public static final Schema VALUE_SCHEMA = SchemaBuilder.struct().name(SCHEMA_VALUE_POSITION).version(1)
            .field(VERSION_FIELD, Schema.STRING_SCHEMA).field(ID_TAG_FIELD, Schema.STRING_SCHEMA)
            .field(TIMESTAMP_FIELD, Schema.INT64_SCHEMA).field(MESSAGE_FIELD, Schema.STRING_SCHEMA)
            .field(ACTION_FIELD, Schema.STRING_SCHEMA).field(X_FIELD, Schema.FLOAT32_SCHEMA)
            .field(Y_FIELD, Schema.FLOAT32_SCHEMA).field(Z_FIELD, Schema.FLOAT32_SCHEMA)
            .field(SEQUENCE_FIELD, Schema.INT64_SCHEMA).field(RADIUS_FIELD, Schema.FLOAT32_SCHEMA).build();

}
```

Figure 26: POJO for RTLS position message

Each message is published into the Kafka bus in a topic that correspond to each tag MAC address.

3.4.1.2 Apache Camel Kafka Connector

With the practical test of the Transparent Robot (bravely discussed in section 2.2.4.2) in mind, a Camel connector was deployed using the Kafka Connect Operator. The goal was to connect the Kafka broker with a MQTT broker since the Robot uses the MQTT protocol. This was successfully deployed and tested.

3.4.1.3 NiFi connector

Afterwards, the NiFi technology was suggested and deployed, and it was seen that it also allowed to connect different systems, among them, Kafka and MQTT. In the case of NiFi, instead of calling to the elements that established the connection connectors it calls them processors, but in practice they do the same.

One of the strengths of NiFi is that it provides with a graphical interface to interact with the application and generate the flows. Using this interface, a MQTT consumer processor and a Kafka

publisher processor were deployed and connected to each other creating a unidirectional (from MQTT to Kafka) connection between both brokers.

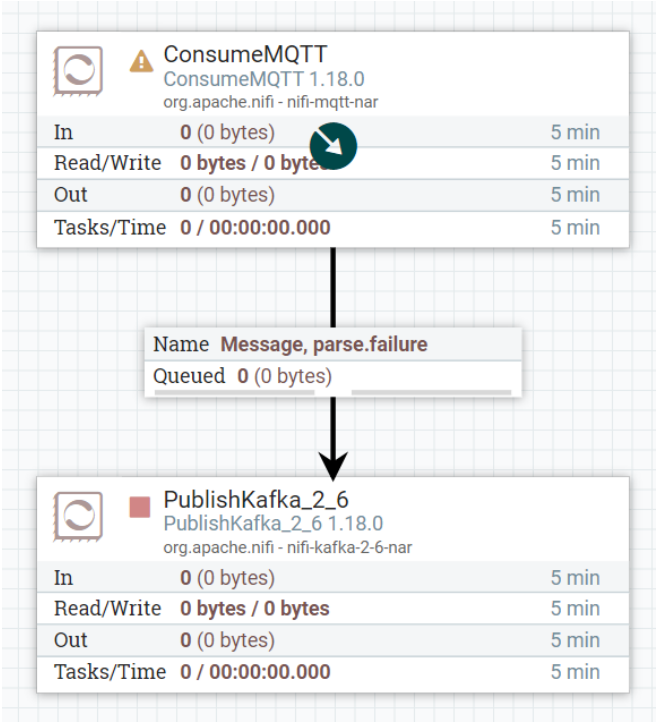

Figure 27: NiFi MQTT-Kafka connector

This way, we were able to extract the messages sent from the Transparent Robot to the MQTT broker from the MQTT broker and publish them to the Enterprise Service Bus, that is Kafka.

3.4.1.4 Performance test between Camel and NiFi connectors

In other to compare the two possible solutions, Camel connectors and NiFi processors, a performance test was design to see how both solutions behave under the same circumstances. The test consisted in sending 100 messages per second during 5 minutes to the MQTT broker and analysing the performance of Kafka depending on how the connection was implemented, either using the Camel connector or NiFi.

The results for the test can be summarize in the following graph.

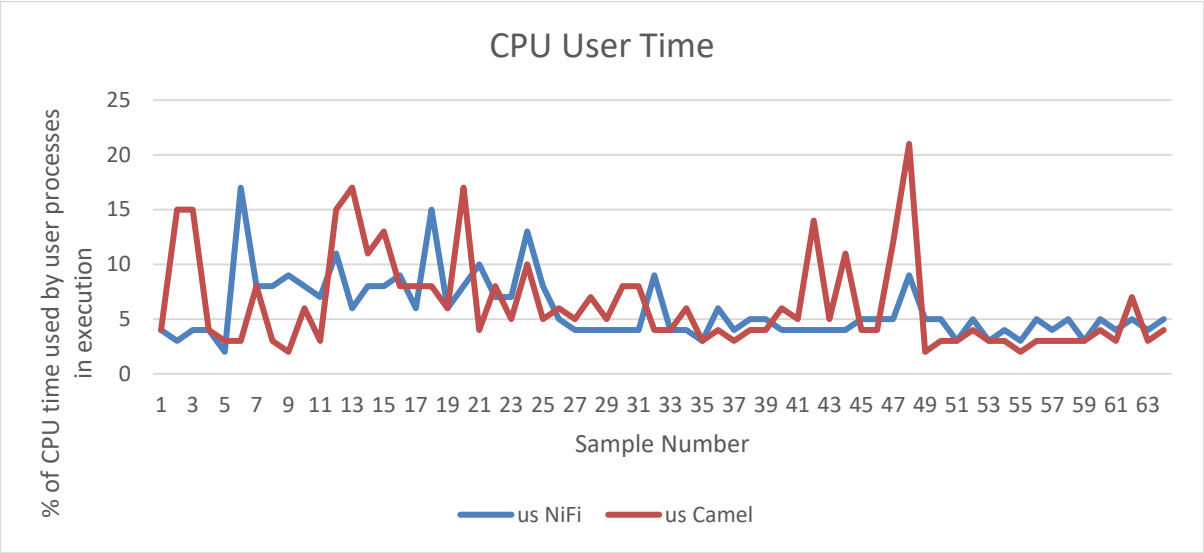

Figure 28: Camel vs NiFi CPU User Time

The graph shows the percentage of CPU time used by user processes. The higher the value the more CPU is being consumed, therefore the application is more demanding in terms of server's resources. In Figure 27, we have the NiFi process in blue and Camel in red. We can see that both lines cross each other several times and, in fact, the mean for both is around 5,3. Nevertheless, there is a difference on how the processes behave over time. Camel is less stable and has very pronounced peaks from time to time. In the case of NiFi we also see few peaks, but they are less pronounced, and it tends to stabilize over time. We can conclude that NiFi is better than Camel in terms of performance so NiFi processors will be used as connectors.

Another reason to choose NiFi is the before mentioned user interface that the application provides to use it. That facilitates a lot the deployment of connections between systems. Whereas with Camel, we have to create a personalized Docker image with the connectors and use it to deploy the Kafka Connect Operator. Finally, we would need to use yaml files to configure the connection in the Linux terminal.

3.5 Possible Limitations

Messaging Bus

The technology selected should full fil all the potential needs the ODIN platform would have. In the case of an overload of the bus due to a massive message arriving, the Kafka deployment could be easily upscaled.

In the case of the topics, we could face some limitations on the topics that have been defined, because they might be insufficient for some KERs. In that cases new topics could be created.

API Gateway

As reviewed in the comparison, Tyk lacks a Dashboard Manager to manage it and does not have a developer portal that could be used to expose API documentation and enrolment.

The Dashboard Manager is an UI interface to the Tyk API functionality, thus, while it is a nice to have, due to ODIN automation objectives and mindset, it is not a problem, as most of the task will be performed programmatically using the API or the Kubernetes Operator.

Regarding the Developer Portal, it is also a nice to have feature, but its main functionality, which is documentation, can be solved by ODIN's documentation components.

The most important drawback it the lack of direct Keycloak integration which should be done through a concrete setup. Another option is to purchase Tyk Dashboard.

Transport services

The connectors will be under the responsibility of the KER that would want to interact with the ODIN platform. The development of those might suppose a limitation. To overcome this limitation a workshop to show how they are deployed and developed will take place and the necessary documentation will be provided.

3.6 Technologies for the History component

The History component oversees being the memory of ODIN, to support several use cases such as check what happened and when it happened, for support purposes and testing, but also be part of the RMS Dashboard to be managed as any dataset in the platform.

Some of the high-level requirements for this component are:

- Be scalable to handle high throughput.
- Support to listen to a set or all the ODIN topics.
- Record all the messages as time series.

Data stored by the History component could be used as a new dataset or 1 dataset per topic stored. This way new use cases could be developed.

Here we present several solutions to create the History component.

3.6.1 Kafka

Kafka can keep all the messages shared in the topics as they are stored in the topic if the retain policy says so. Therefore, by setting up the right retain policy, ODIN could use the ESB itself to store all the data.

Once stored, a consumer could read the data just by setting the reading offset, the position where to start reading, to 0. This way, the consumer could read all the messages from the beginning.

While it is a very lightweight and easy method to store all the information, it lacks query capabilities as it is not a database. This minimizes the cases to use it as datastore, and here are some of them:

- To repeat the processing of the data from the beginning when the logic of processing changes.
- When a new system is included in the processing pipeline, and it needs to process all previous records from the very beginning or from some point in time. This feature helps avoid copying the full dump of one database to another.

- When consumers transform data and save the results somewhere, but for some reason, you need to store the log of data changes over time.

3.6.2 SQL Database

An SQL database can listen through a connector to all the topics or a set of topics and store each message.

This approach would require more development in some cases but most of the databases have Kafka integration.

Once stored, a consumer can read the database using sql queries, which has clear advantages over Kafka message retention approach.

On the other hand, number of tables can grow as many topics exist, plus other drawbacks as data should be stored as blobs of data that cannot be accessed or inspected easily.

3.6.3 No-SQL Database

No-SQL databases can store time series of data, with heterogeneous schemas and scales horizontally very well.

No-SQL databases also provide a query API, not as powerful as SQL, and data can be stored with less constrains than using SQL databases as do not require data normalization. Furthermore, data can be inspected while stored in some solutions.

Usually No-SQL are a better approach for real time and event applications.

3.6.4 Status

By the time of this deliverable is written, several solutions are being analysed, such as InfluxDB for time series sensor data, and other object stores such as S3, ElasticSearch, Cassandra and MongoDB[19]. Apache Kafka Connector project has lots of connectors already created and can interact with those solutions to record messages stored in the topics.

[19] https://db-engines.com/en/system/Cassandra%3BElasticsearch%3BInfluxDB

4 Measurement collection system

4.1 Introduction

Gathering metrics, monitoring components, and configuring alerts is a fundamental piece for setting up and overseeing a service-based system. Having the option to determine what's going on inside a framework, what assets need consideration, and what is causing a stoppage or blackout is necessary. While planning and monitoring can be a challenge, including it from the beginning into the service infrastructure is an important added value that assists the teams with focusing on their work, delegating the obligation of oversight to an automated system.

The ODIN measurement collection system could offer a very dependable service for many stakeholders. System administrators could monitor the whole system to make decisions about:

- Infrastructure resources: by monitoring the utilization of the current infrastructure resources
- Security: by being able to detect intrusions or denial of service attempts
- Configuration: by being able to trace specific cases and share results within the team and with support.

Hospital administrators and clinicians would be able to also take advantage of the measurement collection system, as this system would be able to also handle not so traditional system metrics. A broad classification of these metrics include:

- User perceptions: metrics depicting citizen satisfaction and clinical impact.
- Outcomes: helping better understand clinical treatments and their impact across the patients in the hospital
- Economic aspects: enabling administrators to take smarter decisions.
- Organizational aspects: understanding where the hospital processes could be improved and how.
- Operational aspects: metrics regarding efficiency of the different hospital processes
- Sustainability aspects: helping maintain the service at cost.

4.1.1 Measurement collection system v1

The measurement collection software is architecturally divided in three blocks. For the first block, collection (left side of Figure 28), the architecture contemplates two methods for metric collection. The pushing metrics method is employed when the measured 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, the pulling metrics method is initiated by the measurement subsystem itself (through the polling module), it contacts the module at the module's predefined interface (which has been previously registered in the RMS, see section 2.3) 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.

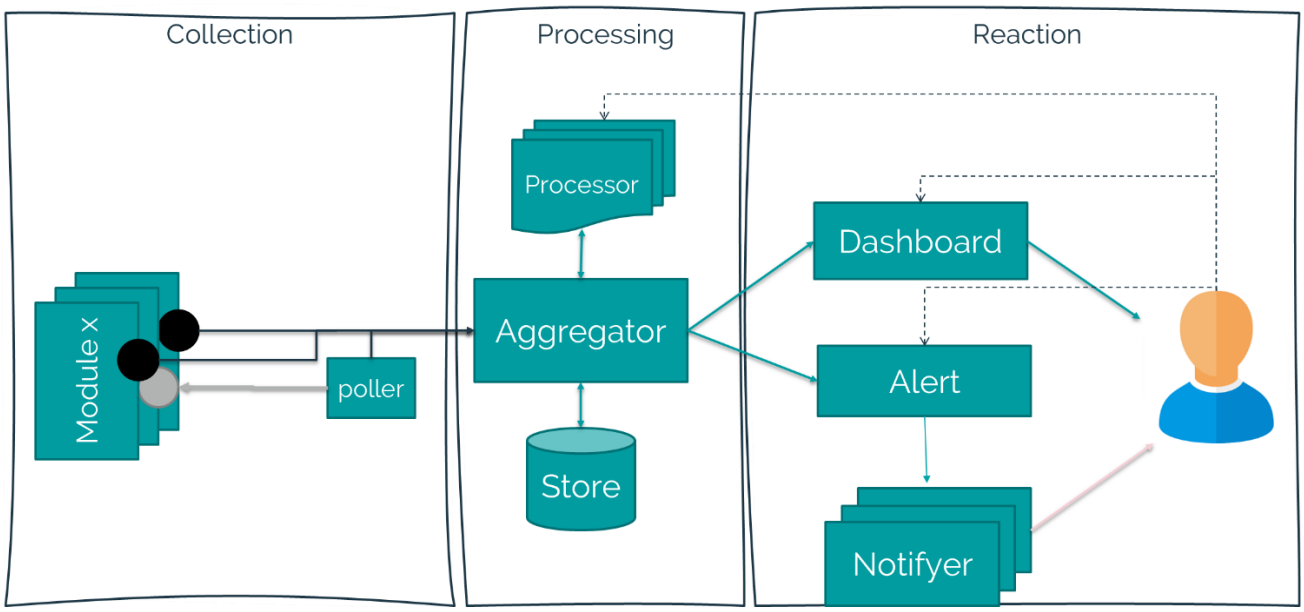

Figure 29: KPI subsystem architecture

The second block is the processing stage (middle of Figure 29) and is related to the collection of the metric 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 metric 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 functionalities.

The final block, the reaction stage (right side of Figure 29), 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.

4.1.2 Final approach

For ODIN v2, the KPI system is limited to monitoring technical metrics, low level operations and metrics of the platform and resources. The system is based on the popular open-source

components Prometheus[20] (for managing metric collection, processing, and part of the reaction stage) and Grafana[21] (for displaying the metrics in customizable dashboards in the reaction stage).

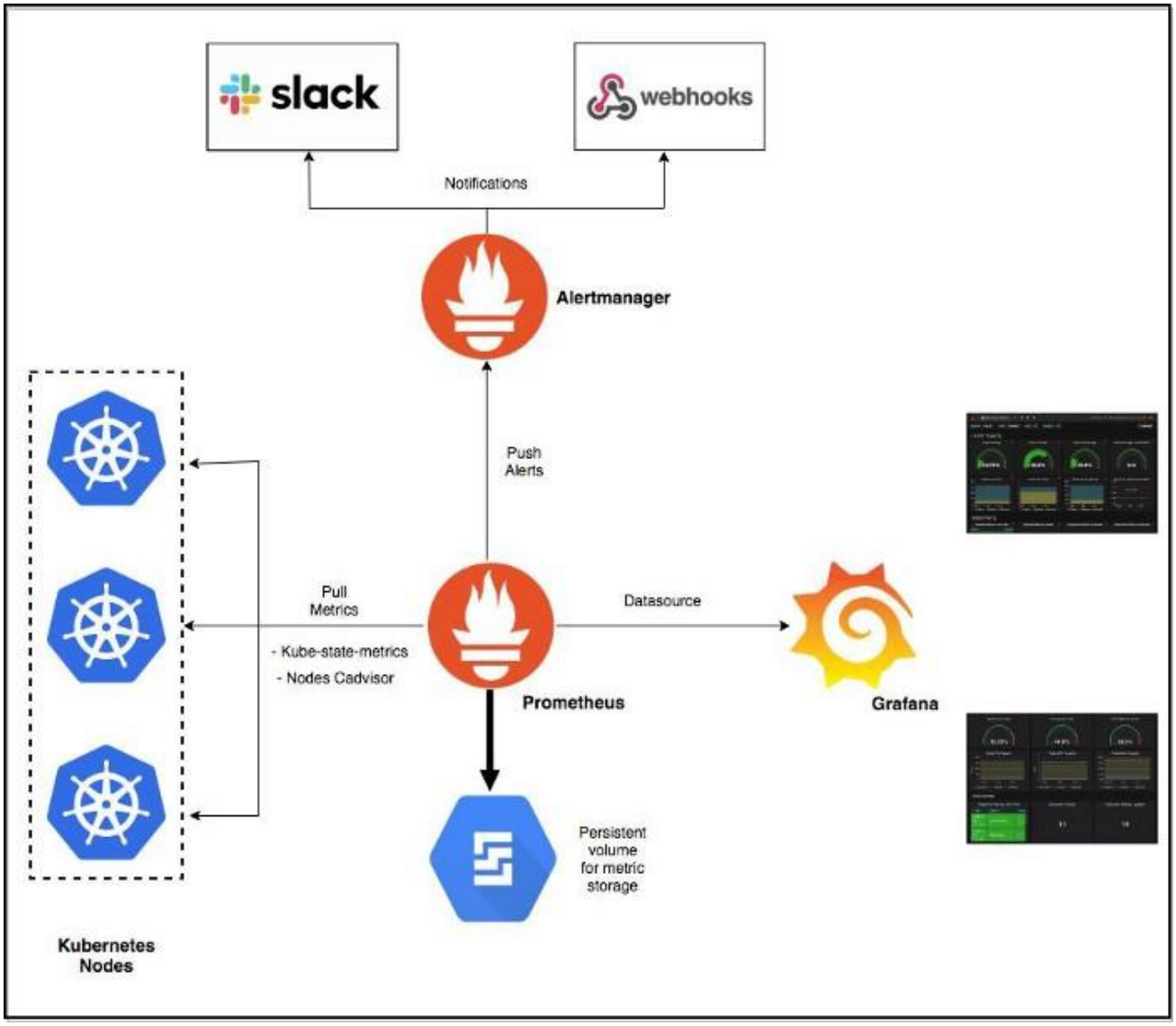

Figure 30: Prometheus, Alert manager (a Prometheus component) and Grafana stack (credit medium.com)[22]

[20] https://prometheus.io/

[21] https://grafana.com/

[22] https://medium.com/avmconsulting-blog/how-to-monitor-kubernetes-cluster-with-prometheus-and-grafana-8ec7e060896f

4.2 Selected technology

Prometheus is an open-source computer monitoring and alerting software. It records metrics in real time in a time series database (with high capture capacity) based on the content of the entry point exposed using the HTTP protocol. These metrics can then be queried using a simple query language (PromQL) and can also be used to generate alerts. The project is written in Go and is available under the Apache 2 license. The source code is available on GitHub and is a project managed by the Cloud Native Computing Foundation along with other projects such as Kubernetes and Envoy.

A typical Prometheus installation includes several building blocks:

- Several agents (exporters) that usually run on the systems to be monitored and will expose the monitoring metrics.
- Prometheus for centralization and archiving of metrics.
- Alertmanager[23] that triggers the issuance of alerts based on rules.
- Grafana[24] for the return of metrics in the form of a dashboard.
- PromQL is the query language used to build dashboards and create alerts.

Prometheus uses the so-called white box surveillance. Applications are encouraged to expose their internal metrics (using an exporter) so that Prometheus can collect them on a regular basis. In case the application (or component) cannot do it directly (database, monitoring server), there are many exporters or agents ready to use to fulfil this role, particularly for popular open-source components. Some exporters also allow the communication management with some monitoring tools (Graphite, StatsD, etc.) to simplify switching to Prometheus during migration.

Default ODIN Prometheus configuration performs service discovery by querying the RMS, eliminating the need for extra configuration to scrape a new endpoint. Services should include specific annotations if they want to be scraped by Prometheus automatically. Prometheus is also configured with the AlertManager component, and some basic rules preconfigured. These rules can be adapted to the pilots' needs by editing the rules file, or through the GUI. The default Grafana dashboards are community developed dashboards for monitoring nodejs[25] applications, since all of the interface layer modules are nodejs applications.

These basic metrics can already provide important KPIs such as number of accesses, response times, and even some derived KPIs like average session time can be calculated. The system already implements all the necessary aspects for automatic KPI collection display, and alert; with potential to add post-processing and analysis capabilities; independently of the metric types, so it is possible to include non-technical KPIs in the system with new metric collection and processing components.

[23] Alertmanager, https://prometheus.io/docs/alerting/latest/alertmanager/ , Last Access Jan. 2022

[24] Grafana, https://prometheus.io/docs/visualization/grafana/ , Last Access Jan. 2022

[25] https://nodejs.org/

4.2.1 Features Implemented

The selected software offers interesting features:

- Network translation of metrics (including logs)
- A basic set of exporters for technical metrics
- Centralization of metric processing and storage
- Querying of metrics
- Alert triggering based on rules
- Custom visualization of metrics (mostly time-based series), and multiple dashboards

On top of it the metric collection system automatically integrates ODIN components and resources and self-configures its components to collect and display standard metrics.

The metrics could be classified in one of three broad categories, depending on the implementation effort:

1. Monitoring system performance. Default exporters and other standard services can already be used to provide these technical metrics.
2. Monitoring activities and tracking of services. Specific exporters and services have to be implemented in order to extract and report these metrics
3. Monitoring of RUC performance and KPIs. High-level collection and post processing is required, at this stage it might be complex to envision the system actually managing these metrics without further implementation.

4.3 Possible Limitations

The metric collection system can manage numeric and discrete metrics. This system is tailored towards technical aspects of the ODIN system operations; however, the hypothesis is that this system could be reused to automatically collect, manage and report other metrics such as those derived from the hospital needs, piloting, and business and sustainability efforts within the project (which would also be applicable for a production hospital environment). As this is still a hypothesis, it needs to be tested whether the effort to be invested in the implementation of the metric agents (Prometheus exporters) for these higher-level metrics is returned in the benefits the system might bring to the hospital decision process.

One clear limitation of the system is the capability of collecting metrics directly from users, for example user satisfaction, PREMs and PROMs, which are very commonly used in the process evaluation. Thus, it is important for future releases that the system incorporates questionnaire management software. Additionally, this software should provide standardized questionnaires such as SUS, TAM, EQ-5, or IEXPAC. In this way system administrators, hospital management, or clinicians could easily launch questionnaire campaigns, particularly if the questionnaires are already preloaded, to collect KPIs directly collected from the users and displayed real-time in the dashboards.

Additionally, if these questionnaires (and answers) are compatible with FHIR standards, it makes it even easier to exchange (semi) standardized questionnaires between instances. In a similar way, compatible applications could better interoperate with the system, which is particularly interesting for processing the results.

5 Integration protocols for Local ODIN instances

5.1 Implementation

The components defined above will run as microservices in the Kubernetes cluster. Each hospital will have their own local implementation and only resources will be shared when necessary. NiFi, Kafka, Fuseki and Tyk can be deployed based on docker images with the necessary connectors and features. In the next steps the possibility of using helm charts will be considered. For each component an installation manual and the necessary images will be provided.

Nevertheless, we must consider that the deployment scenario is heterogeneous, and we could find all kinds of infrastructures in hospitals across Europe, or even the lack of them. Also, the personnel available to maintain the infrastructure or deploy it could be insufficient. For those cases, alternative implementation methods will be considered.

6 Conclusions and next steps

The previous version of this deliverable presented several technologies for the different components and left us with the task to make a choice between the alternatives. In this new version, we have presented the choices made and the improvements in the deployment of the selected technologies. Additionally, new components have been included and the technologies for them.

In the next version of the deliverable, D4.4, the refined components will be presented, their integration with each other and in the ODIN platform.

With respect to each component individually, conclusions and next steps will be covered in the following lines.

Resource Descriptor

In the case of the Resource Descriptor a multi-technological approach has been chosen were all the standards can complement each other. Additionally, we have presented NiFi as integrating tool. Progress has been made in the deployment of a NiFi instance and the application of the chosen standards to some of the technologies in the catalogue. On the other hand, another tool, Fuseki, has been introduced. Fuseki will be the database used to store the Resource Directory data.

The work will continue with:

- The integration of the standard using NiFi.
- Deployment and integration of the Resource Directory using FUSEKI.

Resource Gateway

The chosen technology for the Messaging Bus has been Kafka. It has been deployed using STRIMZI. Next steps will focus on adding security.

In the case of the Resource Gateway a deeper analysis has been made on the possible alternatives and the next version of the deliverable will present the chosen one.

The Transport Services will use NiFi and, for the cases where it would be necessary (like the RTLS), in-house developed connectors. Next steps will focus on further testing the connectors on site.

The API Gateway will use Tyk solution to expose, control and manage the different services published by ODIN platform to external users or systems. A possible procedure to manage how to publish or update the exposed API has been sketched. Following work will go through the concrete messages to implement the sketched process, plus the creation of a specific connector and setup of the solution.

Finally, the History component will be developed in depth with the objective of selecting the right implementation for the requirements stated and the available solutions. Currently the aim is to store all the Kafka messages that are interesting for ODIN.

Measurement collection software

The selected technology for measurement collection system is Prometheus, a widely used open-source system to collect could metrics, together with Grafana, another commonly integrated open-source system to create insightful dashboards from Prometheus metrics.

Although the system is intended to collect technical metrics (such as RAM usage, access rates, resource consumption, etc.) it can be used to collect and manage higher level indicators with little effort, especially if the indicators are derived from existing data, o metadata associated with KERs.

The system will include in the future a subsystem for collecting metrics directly from users and patients, through the selection of an open source, pluggable, questionnaire management system, which will allow the development of FHIR questionnaire definitions, as well as FHIR responses and inclusion of results in the measurement collection system.