DIGITAL MANUFACTURING PLATFORMS FOR CONNECTED SMART FACTORIES

D5.8 QU4LITY Digital Platforms Open APIs (Final Version)

Deliverable Id :	**D5.8**
Deliverable Name :	**QU4LITY Digital Platforms Open APIs (Final Version)**
Status :	**Final**
Dissemination Level :	**PU**
Due date of deliverable :	**30/09/2021**
Actual submission date :	**12/11/2021**
Work Package :	**WP5**
Organization name of lead contractor for this deliverable :	**Synesis - Società Consortile a Responsabilità Limitata (SYN)**
Author(s):	**A. Chiodi (SYN)**
Partner(s) contributing :	**G. Montalbano (SYN)** **F. Larrinaga (MGEP-MON)** **S. Trakic (NXT)** **S. Scholze (ATB)** **D. Pasanisi (IMECH)** **J. H. Simarro (ATOS)** **E. Urkia (DAN-IDEKO)** **I. Metaxa (ATLANTIS)** **N. Weinert (SIEMENS)** **A. Marguglio (ENG)** **J. H. Simarro (ATOS)**

Abstract: The deliverable presents the final details and considerations about the specification and the implementation of Open APIs in the context of the Qu4lity project.

Co-funded by the Horizon 2020 Programme of the European Union

Contents

HISTORY

Version	Date	Modification reason	Modified by
0.0	31/5/2021	Revision of content from version 1	Giuseppe Montalbano (SYN)
0.1	17/9/2021	General revision	Andrea Chiodi (SYN)
0.2	21/9/2021	Analysis of API standard	Andrea Chiodi (SYN)
0.3	10/10/2020	Setup of the proof-of-concept pilot	Andrea Chiodi (SYN)
0.4	29/10/2020	Final version for reviewing	Andrea Chiodi (SYN)
1.0	12/11/2021	Final version for submitting	Andrea Chiodi (SYN)

1. Executive Summary

This is the second of two documents aiming to report about the analysis and the outcomes that have progressed with the activities of *T5.4 - Digital Platforms Open APIs and Process-Level Interoperability*.

This "final version" consolidates most of the contents presented in "version 1" of the same deliverable, adding further details and considerations about the specification and the implementation of Open APIs in the context of the Qu4lity project. Elements and concepts that have been relevant for the progress of the T5.4 activities have been maintained in the document in order to provide the necessary context for the discussion.

Core part of T5.4 activities is the identification of the Open APIs that could enable the composition of the many different digital platforms and tools adopted in QU4LITY, to deploy applications suitable for ZDM processes. The aim of that task is to define the set of APIs that can support the interoperability between tools of different parties and enable the composition of different building blocks based on the specific requirements of the ZDM application to address.

This report starts providing an overview about the general concept of API and then digs deep into some of the most relevant aspects that characterize an API. Interoperability between digital platforms, leveraging an API, can be realized at different levels.

This report then highlights some of the key characteristics that needs to be considered in T5.4 to identify the set of APIs relevant for QU4LITY, providing an introduction to some communication protocol architectures, like for instance the **service-oriented** or **REST** architectural styles as well as the **broker-centric** architecture and mentioning a few of the most common protocols adopted in ZDM processes (*MQTT*, *AMQP*, *OPC UA, HTTP*, etc.).

The portability and flexibility of an API is also characterized by the availability of formalisms that support their use and maintenance. In particular for the web services, the exploitation of specifications like the ***AsyncAPI Specification*** [1] and ***OpenAPI Specification*** (OAS) [2] simplifies and makes more effective the use of APIs via both manual and automatic tools, and results an attractive characteristics to be considered for the identification of QU4LITY's Open APIs. Such an approach has been further analyzed and discussed in this second version of the document.

A possible approach was investigated in the previous version, which would enable the interoperability between tools despite their incompatibility with a specific API. This would be based on the adoption of translation tools aimed to "adapt" the communications based on different APIs. While such a development direction retains its validity, the present task adopted a more radical approach, based on the belief that an interoperability by-design is the necessary foundation to reach the foreseen level of ZDM quality that is the basis of the QU4LITY objectives.

2. Introduction

This document, D5.8 - QU4LITY Digital Platforms Open APIs (Final Version), describes the activities performed within the scope of task T5.4 and the achieved outcomes.

The purpose of Task 5.4 is to provide the basis for data interoperability between technologies with the aim to enable implementation of Autonomous Quality (AQ) processes. Among these technologies, the digital platforms of the project partners are those of main interest for this task although the general aim is to achieve the broader generality for the Open APIs identified in QU4LITY in order to support the implementation of ZDM processes also leveraging the integration and enhancement of technologies developed by third parties.

T5.4's first activity was to identify the APIs used and promoted by QU4LITY partners, as well as to research APIs that enable the integration of technologies and capabilities from various digital manufacturing platforms and digital enablers.

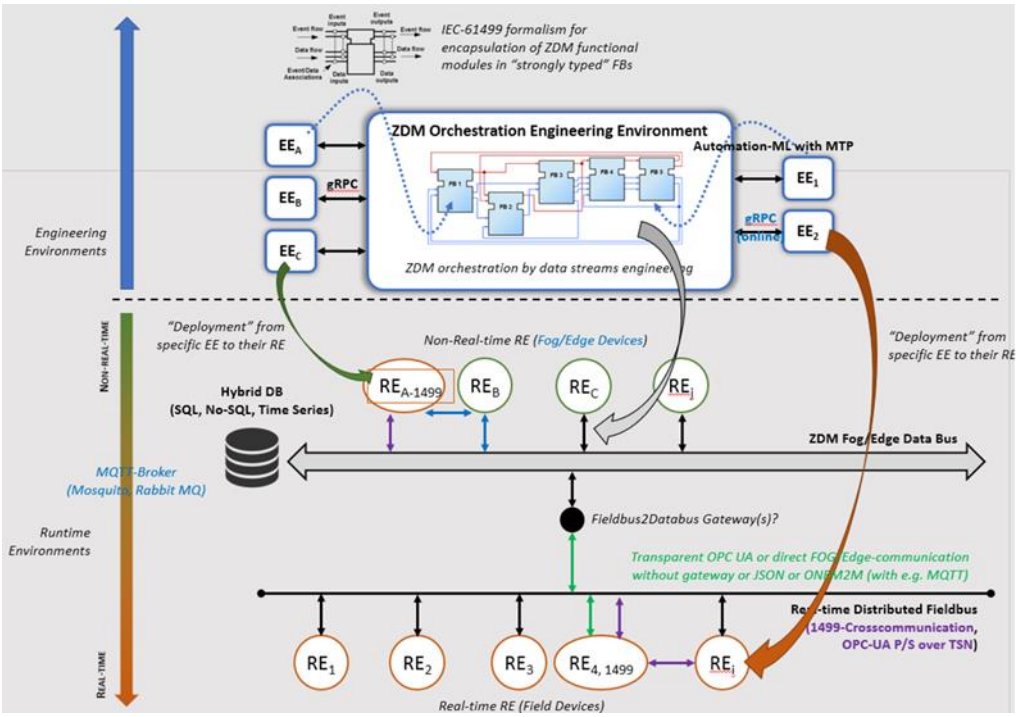

Figure 1 Possible approach for integration of Engineering Environments

A possible approach to enable the integration of different ZDM Orchestration Engineering Environments has been proposed by WP4 and described in *D4.3 - Distributed Communication and Control Infrastructure* (section 3.1). That approach is based on leveraging an orchestration environment to synchronize the several engineering tools that are needed during the design, deployment and operation of ZDM processes. In such a scenario, an orchestration environment based on the *IEC61499* formalism would interact with the other tools via different interfaces and therefore APIs, both at design and at operation phase.

That proposed approach has been a first attempt to identify a composite framework for engineering and orchestration of ZDM processes leveraging the technologies platforms that are available in the QU4LITY consortium, and addressing the distributed communication and control infrastructure in the scope of WP4.

Part of this report focuses on providing an overview of the concepts and aspects of relevance associated to the definition of Open APIs suitable for ZDM processes and on the APIs supported by some of the technologies of project partners.

The overall structure of this document is as follows:

- Section 2 introduces the overall document.
- Section 3 presents the relations between T5.4 and other tasks of WP5, as well as with the overall QU4LITY vision.
- Section 4 clarifies the concept of API and Open API and addresses the key aspects that characterize APIs, which has been considered in T5.4 to evaluate and identify the Open APIs for QU4LITY
- Section 5 provides an overview of APIs currently supported by some of the technologies and tools provided by QU4LITY partners or that will be develop within the project.
- Section 6 introduces the concept of Abstract API specification, to support interoperability by design.
- Section 7 presents alternative approach to the interoperability issue.
- Section 8 presents a Proof of Concept for the proposed Open API approach.
- Section 9 draws some conclusions.

3. How T5.4 fits in the QU4LITY Vision for AQ

Task 5.4 is part of WP5 and as such shares the overall goal of enabling and fostering the implementation of Autonomous Quality, providing the means that support and improve the engineering, deploying and management of ZDM processes.

The framework of Open Autonomous Quality Services Engineering and Processes promoted by QU4LITY and main development objective of WP5 should result as composition of different platforms. Task 5.4 is the task of WP5 that addresses the interoperability aspects between the technologies exploitable to implement the QU4LITY AQ paradigm and that focus on enabling the integration of those building blocks by leveraging Open APIs.

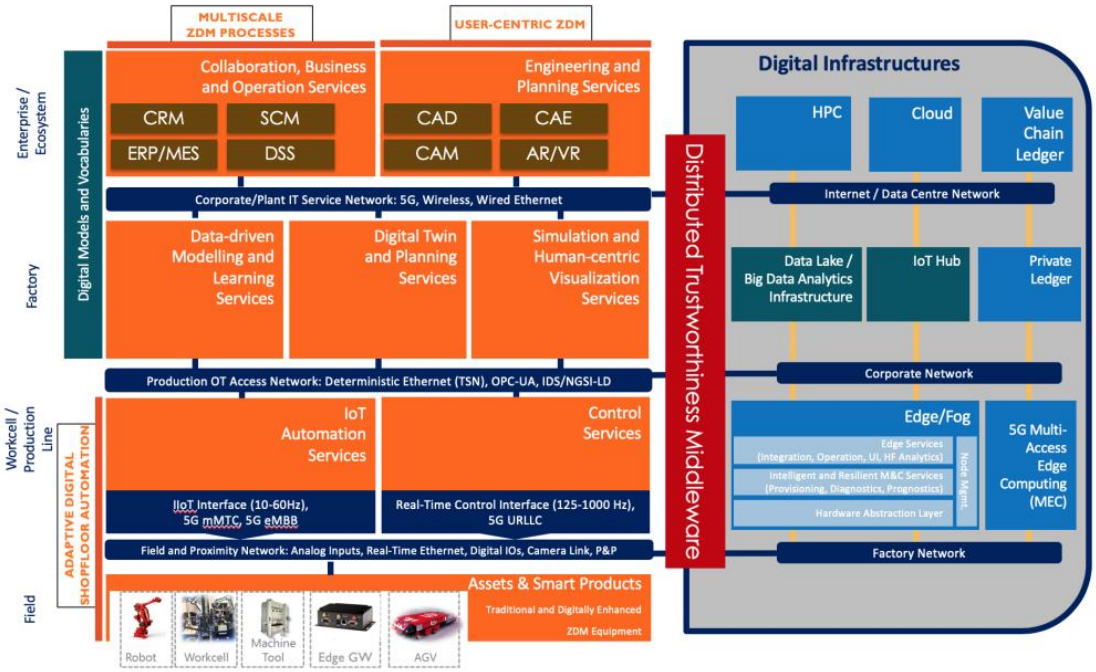

Figure 2 QU4LITY Reference Architecture

By definition, interoperability and interconnection topics regard the composability of more than one technology. In Task 5.4 scope we have to address the possible integration of different technologies, spanning at almost all the three higher levels in which the QU4LITY digital services are structured: Work cell/Production Line layer, Factory layer, and Enterprise/Ecosystem layer.

The synergies with the other tasks of WP5 can be briefly described as follows, based on their specific goals:

- T5.1 addresses the development of a Framework to support the specification and integration of human centric ZDM processes.
- T5.2 mainly focuses on the customization and integration of simulation processes and digital twins to the multi-stage ZDM processes based on AQ.

- T5.3 addresses the capability of digital platforms' operations of being adaptive to changing conditions of the shop floor.
- T5.5 addresses the enabling of autonomous data management operations by integration of the digital platforms to the open secure Industrial Data Space.
- T5.6 focuses on the integration of functionalities from multiple platforms to enable an open and integrated service engineering approach.

In general terms, Task 5.4 will support the enhancement of the project partners' digital platforms to enable the composability of their systems with the other technologies of QU4LITY, towards the implementation of ZDM processes.

The compatibility with Open APIs will enable not only the effective exploitation of the project partners' digital platforms but also their enhancement and extension by third parties.

The Open APIs identified in Task 5.4 represents also a mechanism through which the test and validation activities can be performed on the systems to evaluate their performance and to make validation activities. The results of Task 5.4 are therefore relevant also for the activities of WP6.

4. Characteristics of interfaces and API used in ZDM

In general, when referring to APIs we can actually mean two different concepts: one is based on systems exposing data and operations, and the other one refers to language constructs or in the form of libraries/frameworks [3].

APIs refer to the interfaces that other software tools can access by using, in general, libraries that implement those interfaces, when the focus is on programming elements or the integration possibilities available at the programming code level. APIs are represented in Object Oriented Programming (OOP) languages by the classes that implement those interfaces. APIs are used by third parties to exploit, enhance, and/or expand the core program functionality since they are designed to allow external software modules outside the main section of the application exposing the API to communicate successfully.

To ensure usability to third parties the interfaces and their implementations can be published as packages, where they are possibly bundled with documentation and sample implementations.

API can also refer to a user interface that allows data and operations to flow between separate systems. In this context API refer to the interfacing mechanisms that allow distributed systems to interact by mean of a communication protocol. In this case, the API must provide for a well-defined and well-understood communication method between the systems that want to interact.

Both these interpretation of APIs are relevant to enable the integration of different technologies and therefore T5.4 considers APIs belonging to both the categories as possible candidates for the Open APIs promoted in QU4LITY as enablers for interoperability.

Considering that an API is mainly of interest for parties that aim to leverage that interface to interact and integrate additional functionalities to the tool that support such an API, it is worth differentiating APIs based on the accessibility level. **Private APIs** serve a specific set of stakeholders and are usually not exposed outside these specific parties. Usually they are leveraged within an organization to support the structuring and development of software tools/products. When the APIs are designed to enable third parties to leverage the interface for integration with other tools, they are commonly identified as **Public APIs**. Public APIs are open for consumption by interested parties but this does not mean that they are publicly available for free.

Commonly, to describe an API that is publicly available to all the interested parties that wants to develop tools exploiting that interface, the API is tagged with the "open" adjective. **Open APIs** are therefore leveraged to promote the growing of developers that can have the possibility to extend the functionalities of applications based on those interfaces and to generate a larger interest and market opportunities on the technologies compliant with them.

It is worth to note that "open" does not implicitly means "free". Any possible restriction to use is typically ruled by some license agreement, which is bundled with the library itself.

To promote the interoperability of digital platforms exploitable for the implementation of ZDM processes, QU4LITY aims at assuring that all of them can be composed by means of Open APIs. The following section of this report describes many of the characteristics that can differentiate APIs and, as a consequence, the identification and selection of the set of Open APIs to be adopted in QU4LITY is not an immediate task.

With the aim to support the development and diffusion of more and more Open APIs, in particular in the field of client-server services and broker-based interaction, some communities of developers started working on the implementation of specifications to facilitate the portability and maintainability of existent APIs, regardless their specific implementation details. Two of the most relevant results in this direction are the **AsyncAPI specification** [1], and the **OpenAPI Specification** (OAS) [2]. These standard will be better described in a subsequent section of this document.

When a digital platform exposes a web service API that is based on this specification, it creates an advantageous condition that qualify it as an ideal candidate for supporting and implementing the Open APIs that QU4LITY is looking for.

The use of APIs to enable integration of different tools and technologies is a commonly adopted approach in the field of digital solutions for manufacturing processes. APIs enable to integrate different tools in order to compose frameworks of digital platforms that can address the challenges of the ZDM processes.

When referring to API as a mean to integrate tools at programming code level, one of the two possible interpretations as described in the previous section, their use in the context of ZDM processes is mainly exploited for the enhancement and extension of software features dedicated to the interconnection with other tools.

Remote Procedure Call (RPC) [4] has been one of the first mechanism adopted to "transform" an API defined at programming code level into an API to enable data and operations flow, and their implementations are highly dependent on the programming language and runtime code used. **gRPC** [5] is a modern revision of that same approach, which try to overcome some limitations of the legacy RPC. APIs defined at programming code level can therefore be exploited to enhance the digital tools with communication interfaces and protocol stacks enabling the interaction with other systems via digital networks.

An example of APIs of this type are the *FBDLL* and *#Develop Add-in* APIs described in the section 6 of this report.

To satisfy the need of APIs that are independent from the specific programming language and/or runtime implementation, many tools and digital platforms adopt

APIs of the second nature (communication protocol-based) and follows a *Service-Oriented architectural approach*.

Service-Oriented Architecture (SOA) is a term that has not a unique definition [6] but in general terms refers to a client-server design approach where a server entity exposes a set of services to other systems, which can be called by client entities to trigger data and operations flow.

By means of communication protocol-based APIs different tools and digital platforms can interact with each other, enabling the implementation of highly powerful and flexible solutions providing digital services to address ZDM processes. The use of *service-oriented interfaces* makes the composability of the tools independent from the programming code details of each specific software application involved and limits the interoperability issues to the common understanding of the communication mechanism associated to a certain API.

It is paramount, therefore, that APIs specify all the details needed to enable the correct exchange of data and provide the information needed their correct interpretation.

While the SOA architecture is based on a client-server pattern, so it is designed for synchronous exchange of messages, the **Broker-centric** architecture follows the publisher / subscriber pattern and is more properly suitable for asynchronous dialogue, possibly extended to multicasting. Several use-cases implements such a pattern in the industrial context, where a distributed architecture, based on autonomous subsystems, is increasingly adopted.

Generally APIs supported by tools suitable for ZDM processes leverage some kind of standard protocol (or stack of protocols) to exchange data over the network (like for instance **TCP** and **UDP**). Doing that, tools can exchange data by means of the common and well supported *Internet Protocol* (IP) networks and devices. More frequently the APIs use some kind of application protocol that add functionalities on top of the basic (TCP and UDP). **MQTT** [7] and **AMQP** [8] are for instance two messaging protocols that different technologies for ZDM exploit as part of their APIs to implement in effective way publish-subscribe communication architectures.

To implement APIs that can provide a higher level of flexibility in the composability of technologies, many tools adopt the **Representational State Transfer** (REST) style. REST is an architectural style for services and defines a set of architectural constraints and agreements for implementing service-oriented interfaces. A service that is compliant with the REST constraints is said to be **RESTful**. A REST API can be implemented leveraging for the data delivery many different protocols, however HTTP and HTTPS are the most widely used.

Whatever protocol and architectural style on which an API is based, to provide interoperability and enable the effective interconnection of different digital platforms and achieve the desired objectives of quality in ZDM, the APIs has not only to define the *syntactic aspects* of the messages exchanged via the network connections, but

also it has to provide enough details for the *semantic aspects* needed to exploit the API and make the interaction between tools possible.

In terms of **syntactic aspects**, the data formalism used to structure the information transferred via the communication protocol is another element that an API has to define clearly to enable the interaction between different tools. Commonly, in the context digital platforms and application tools for ZDM, they leverage standard formalisms, like JSON and XML. These standards alone, however, do not fully specify the type of messages exchanged through the API and therefore additional specifications should be part of the API.

The **semantic aspects** are finally the set of information that enable other tools interact effectively via the API and exploit the services offered by means of that API. Specifications for the semantic interpretation of the messages exchanged via an API are therefore even more important than the syntactic ones.

Moreover, the semantics of the exchanged messages is likely at the origin of technological and architectural decisions, which could involve, for instance, the choice between synchronous or asynchronous communication because of the lifecycle of the managed information, or the network protocol based on time-related restrictions.

Among the standard protocols that are relevant for the implementation of ZDM processes, **OPC UA** (IEC 62541) [9] is one of the most interesting. The standard specify many aspects of the protocol to cover a large set of issues associated to the implementation of an effective and standardized interfacing mechanism between heterogeneous systems. The specification provides details about both the syntactic and semantic aspects associated to the use of that protocol, assuring a large extent of interoperability between tools that adopt that standard. This notwithstanding, some aspects are strictly dependent on the particular use that a system/tool makes of the API and therefore even OPC UA compliant APIs need to provide additional specifications if they want to be complete.

Some efforts in this direction are also done by the *OPC Foundation* [10] trying to standardize also the **information model** exploitable in certain contexts (in robotics, in machine vision applications, in CNC system, etc.), however they are not covering the overall spectrum of information that can be encountered in ZDM applications and therefore customized APIs are very frequent.

The availability of an **API definition language** is a possible criteria to evaluate an API. **SOAP** [11], for example, adopts the *Web Services Description Language* (WSDL) language to describe the syntax of the implemented functionalities. Other APIs adopt different dialects of **IDL** (*Interface Definition Language*) [12] which offer similar functionalities. These forms of structured documentation also enable several design-time functionalities, like syntax checker, testing tools, generation of reference manuals, etc. The API description language could even extends beyond mere syntax, including information about the semantics of operations, based on some known and shared ontology.

While most of the existing definition languages are intended to extend a specific protocol, their last generation approach is moving toward a complete independence from any specific implementation. The mentioned AsyncAPI and OpenAPI specifications are example of such a trend.

5. APIs supported by the technologies of QU4LITY partners

The identification of the Open APIs to be supported and promoted by QU4LITY for the implementation of ZDM processes began with an analysis of the APIs currently supported by the technologies of the QU4LITY partners. Such analysis supported the progress of T5.4 on defining the most relevant aspects that needs to be addressed by the Open APIs suitable for ZDM.

A first step toward the collection of details on the APIs possibly supported in QU4LITY has been the analysis of the technologies described in the **IoT-Catalogue**. The *IoT-Catalogue* is a web-based catalogue for Internet-of-Things (IoT) solutions, available at *www.iot-catalogue.com* and described in D2.5. The general goal for the IoT-*Catalogue* is to bring IoT users and technology providers together, from the domain needs to IoT products (and back) via validated solutions with components, assembly guides, and more.

This list of interfaces supported by QU4LITY partners' technologies is representative of the wide spectrum of interfaces that are available in the technologies suitable for implementation of ZDM processes. Some of them refer to the communication protocol on which they are based (for instance MQTT, OPC UA, AMQP, WebSocket, etc.) others refer to the software tools exploited for their implementation (Flink, Kafka, Spark, RabbitMQ, etc.).

To achieve better understanding of those APIs a questionnaire has been circulated among the QU4LITY partners to collect more details. The questions to be answered aimed not only to collect more insight into the partners' APIs but also to understand at which level of QU4LITY Reference Architecture (RA) functional view (D2.11 section 5.2.4.1) a specific API can be exploited.

The document containing the questionnaire circulated is hosted in the files repository of the project. As a result, the partner's APIs are described in the following of this section.

The integral description provided by the partner is available in the D5.7 deliverable, while it has been reduced in this document.

UMI

Name of the technology/ tool that uses this interface/ protocol/ API	M3MH	
Name of the partner	UNIMETRIK	
Name of the interface/API	UMI	
This interface provides (ref. D2.7 page 23)	**Technical interoperability**	Yes
	Syntactic interoperability	Yes

	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	
		Control Services	x
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	x
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	UNIMETRIK aims to leverage this interface in the pilot where participates: GF It will be exploited to make for the connectivity between the Machine Tool and the M3MH software.		

The UMI (Universal Module Interface) interface is integrated in the smart factory of the Automotive Intelligence Centre. This API controls the connectivity of the M3MH and the Machine Tool.

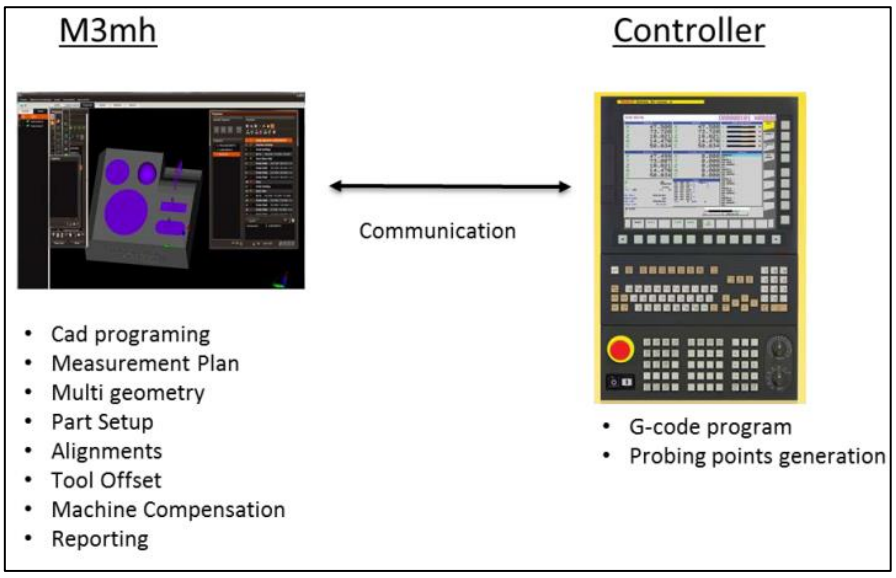

Figure 3 Unimetrik M3mh Communication

IMECH MQTT-based API

Name of the technology/ tool that uses this interface/ protocol/ API	Decision Support System for ZDM		
Name of the partner	IMECH		
Name of the interface/API	MQTT		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	X
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	X
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	IMECH aims to leverage this interface in WHR pilot case.		

The MQTT protocol enables the publication of the data predicted by data-driven model and allows visualization of quality information on innovative HMI.

The data messages are packed in JSON format and it is possible to customize the structure and semantics of data based on the application needs, both on request and in response.

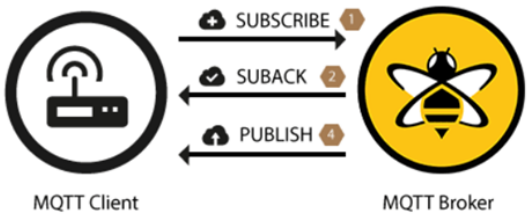

Figure 4 Communication between MQTT Broker and Client

The interface that IMECH is willing to adopt for the WHR pilot is mainly based on this standard protocol. The technology aims at providing an alarm based on the most important variables that can be measured in the WHR plant. The alarm should provide an indication about the possibility of producing a defective product.

Savvy M2CAPI

Name of the technology/ tool that uses this interface/ protocol/ API	DANOBAT DATA SYSTEM		
Name of the partner	DAN		
Name of the interface/API	Savvy M2CAPI		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		No
	Semantic interoperability		No
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	X
Pilot(s) where this interface will be exploited	DANOBAT DIGITAL MACHINE		

Savvy M2CAPI is a *REST API*, which enables the automatic connection of third-parties to the Big Data environment managed by the Industrial Cloud platform of DANOBAT. Thanks to this API, different types of systems can consume data collected by the platform in an automated, secure, and efficient way. It operates over *HTTPS* protocol.

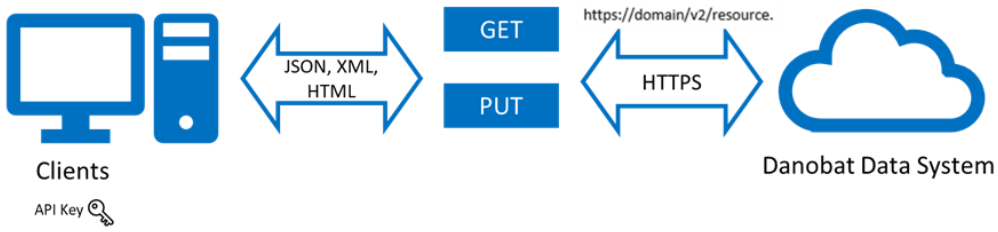

Figure 5Communication schema based on M2CAPI

Representational state transfer (*REST*) is a software architectural style that defines a set of constraints to be used for creating Web services.

By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

In this **RESTful Web service**, requests made to a resource's *URL* will elicit a response with a payload formatted in *HTML*, *XML* or *JSON*. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When *HTTP* is used, as is most common, the operations (*HTTP* methods) available are *GET*, *HEAD*, *POST*, *PUT*, *PATCH*, *DELETE*, *CONNECT*, *OPTIONS* and *TRACE*.

The API design is based on REST, and it operates over HTTPS protocol. Each request contains all the necessary information to execute, so the server will never maintain a session for an API client; that is, each request will be treated independently and a user's identification will be born and die with every request.

```
v2
|__ segment
|__ productionLines
|             |__ lineId
|__ locations
|             |__ locationId
|__ machines
              |__ machineId
              |__ groups
              |             |__ groupId
              |__ indicators
              |             |__ indicatorId
              |__ alarms
              |             |__ alarmId
              |__ severities
              |             |__ severityId
              |__ categories
                            |__ categoryId
              |__ files
                            |__ fileName
```

Figure 6 M2CAPI REST Topology

The API has two types of requests:

- **Resource requests**. Information about different Savvy Industrial Cloud resources: machines, indicators, locations, alarms, files, etc..
- **Data retrieval / download requests**
  - Data requests: historical machine indicators during a timeframe.
  - Data Alarm requests: historical machine alarm during a timeframe.
  - Machine streaming requests: subscribes the client to a machine data flow through a persistent connection. The API client will receive all indicator data sent from the machine to the cloud in real time.

Edge OPC-UA

Name of the technology/ tool that uses this interface/ protocol/ API	DANOBAT DATA SYSTEM		
Name of the partner	DAN		
Name of the interface/API	Edge OPC-UA		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	DANOBAT DIGITAL MACHINE.		

DAN has developed an API based on **OPC-UA** (*IEC-62541*) on Danobat gateway module, to abstract from the details of the low-level communication mechanisms implemented by the exploitation of the *DANOBAT DATA SYSTE*M. The set of

functionalities offered by that API enables to implement flexible and effective CPS data collection mechanisms.

The OPC UA servers are implemented on the IoT gateways each machine following the IEC 62541 standard, allowing to get the maximum data gathering speed, as the OPC UA servers are implemented using the binary protocol.

The binary protocol offers the best performance/least overhead, takes minimum resources (no XML Parser, SOAP and HTTP required, which is important for embedded devices), offers best interoperability (binary is explicitly specified and allows fewer degrees of freedom during implementation) and uses a single arbitrarily choosable TCP port for communication easing tunneling or easy enablement through a firewall.

The OPC UA server uses the server/client message exchange mechanism.

The implementation of each OPC UA server is automatically configured, set and deployed for every corresponding machine, with a unique Information Model, Moreover, the deployment and managing of the OPC UA server can be directly done in *Danobat Data System Cloud*. The information Model implemented in the OPC UA server consist of a set of folders and variables that represent the machine(s) connected to the IoT gateway.

ATB Kafka-based API

Name of the technology/ tool that uses this interface/ protocol/ API	Context Extractor & Device Centric Context Model		
Name of the partner	ATB		
Name of the interface/API	Kafka		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	X
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
		Engineering and Planning Services	X

	Enterprise/ Ecosystem layer	**Collaboration, Business and Operation Services**	
Pilot(s) where this interface will be exploited	Continental		

The technology developed by ATB is composed of different modules.

The ***Context Extractor*** is a generic solution for extracting current context from monitored data. The *Context Extractor* is communicating to its outside via to components, the *Context (System) Monitor* and the "Context Provision".

The objective of the ***context monitoring*** component is to receive raw data and provide aggregated context data. It is a generic solution for the monitoring data sources and customisable for different communication protocols and data structures. It enables also data pre-processing or data aggregation.

The following Figures shows the conceptual architecture of the *Context Awareness* and *Context Monitoring modules*.

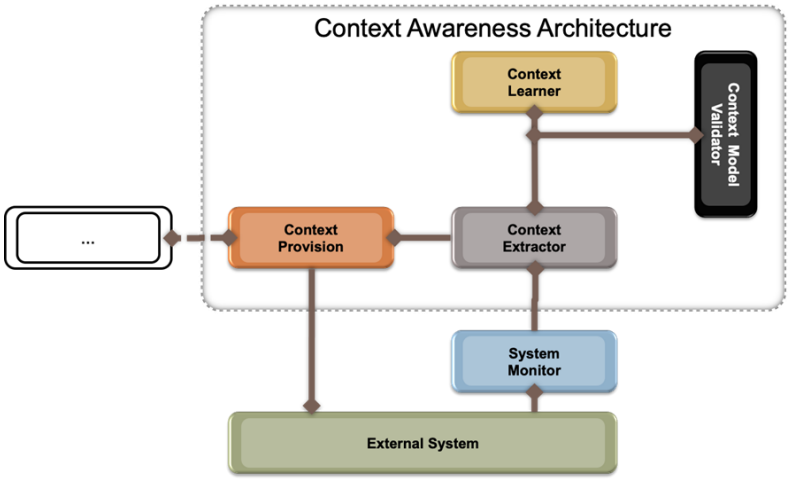

Figure 7 Conceptual Context Awareness Architecture

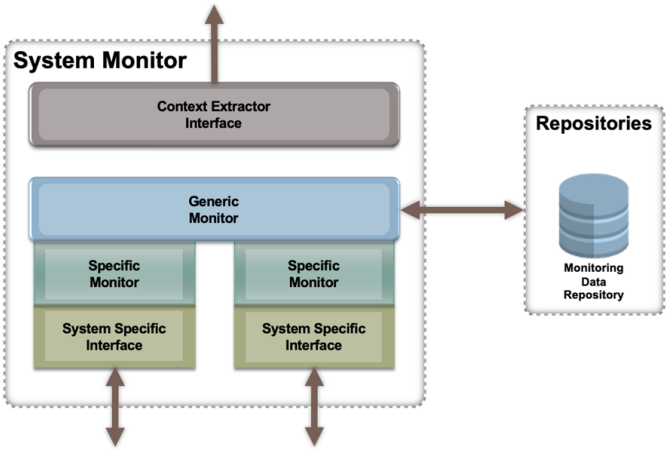

Figure 8 Conceptual Context Monitoring Architecture

The objective of the ***context extraction and provision*** *component* is to identify the context of products/machines, production processes or specified systems (e.g. legacy systems) and to provide it for further use to other modules or external systems. The following Figure shows the conceptual architecture for the *Context Extraction module*.

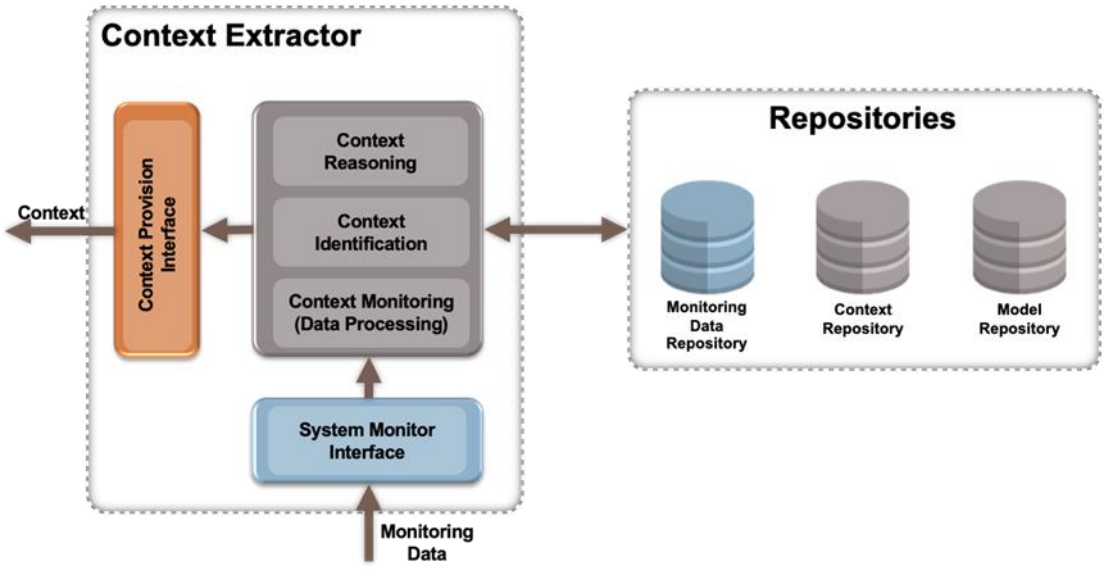

Figure 9 Conceptual Situation Determination Architecture

For both module several possibilities to communicate with external systems are available.

Kafka is used twofold by the *Context Extractor* module: as in input for the *Context Monitoring* and as an output for the *Context Provisioning* module.

The *Context Monitoring* is able to use an interface to the distributed messaging system *Kafka* to retrieve information from the external systems. Since the

communication channel is a *Kafka* node/cluster every service which wants to use this channel has to subscribe to a specific topic in order to get the relevant information.

The *Context Provision* is able to use an interface to the distributed messaging system *Kafka* to publish the identified context for usage by other services/systems.

The data formats for the Kafka topics to be exchange between *Context Extractor* modules and external systems are based on JSON standard. In order to monitor an external system via the *Context Monitoring*, the data which will be monitored via Kafka has to be defined case by case. Also the data format of the Kafka topic for provisioning of identified context has to be defined case by case depending on the service to which the service will be send.

ATB MQTT-based API

Name of the technology/ tool that uses this interface/ protocol/ API	Context Extractor & Device Centric Context Model		
Name of the partner	ATB		
Name of the interface/API	MQTT		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	X
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	X
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	Continental		

This API is part of the *Context Extractor & Device Centric Context Model* that is introduced in section "*ATB Kafka-based API*". As previously mentioned, different APIs are supported and this section provides some details about the MQTT API.

MQTT can be used twofold by the *Context Extractor* module: as in input for the *Context Monitoring* and as an output for the *Context Provisioning* module.

The *Context Monitoring* is able to use an interface to the distributed messaging system MQT to retrieve information from the external systems, specifically IoT devices. The *Context Provision* is able to use an interface to the MQTT messaging system to publish the identified context for usage by other services/systems.

The data formats for the MQTT topics to be exchange between *Context Extractor* modules and external systems are based on **JSON** standard.

In order to monitor an external system via the *Context Monitoring*, the data which will be monitored via MQTT has to be defined case by case. Also the data format of the MQTT topic for provisioning of identified context has to be defined case by case depending on the service to which the service will be send.

For an example of the data format, please take a look at the *Context Extractors* Kafka API described in section "ATB Kafka-based API".

ATB REST API

Name of the technology/ tool that uses this interface/ protocol/ API	Context Extractor & Device Centric Context Model		
Name of the partner	ATB		
Name of the interface/API	REST		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	X
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	

	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	X
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	Continental		

This API is part of the *Context Extractor & Device Centric Context Model* that is introduced in section "*ATB Kafka-based API*". As previously mentioned, different APIs are supported and this section provides some details about the REST API.

REST is used twofold by the *Context Extractor* module: as in input for the *Context Monitoring* and as an output for the *Context Provisioning* module.

The *Context Monitoring* is able to use REST APIs from the external systems to monitor them. The *Context Provision* is able to provide identified context via a REST API and using this API other external systems can get current identified context.

The data formats for the REST API topics to be exchange between Context Extractor modules and external systems are based on **JSON** standard.

In order to monitor an external system via the Context Monitoring, the data which will be monitored via a REST API has to be defined case by case. Also the data format of the REST API topic for provisioning of identified context has to be defined case by case depending on the service to which the service will be send.

For an example of the data format, please take a look at the Context Extractors Kafka API described in section "ATB Kafka-based API".

NXT FBDLL

Name of the technology/ tool that uses this interface/ protocol/ API	nxtSTUDIO Engineering Tool		
Name of the partner	NXT		
Name of the interface/API	FBDLL		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	

		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	NXT & ASTI use case		

The ***FB_DLL*** function block implements an interface between a FB and arbitrary functions residing in a dynamically loaded library. It is a generic function block which means that the count of event ports and data ports can be defined.

It provides the possibility to implement basic or service *IEC 61499* function blocks in a custom programming language that are compiled in a dynamical loadable library (*DLL*). The goal of this FB is to include external code written in a custom programming language into an *IEC 61499* application.

More than one instance of the *Generic DLL function block* (FB_DLL) can be instantiated in an IEC 61499 application and the parameters provided as input to those FBs are exploited to select the appropriate DLL. All the FB_DLL instances are characterized by an INIT input event that is used to load the DLL: in particular, when the INIT event of any FB_DLL is received for the first time, the associated DLL is loaded and the IEC 61499 runtime registers the function block with that DLL. Furthermore, if the constructor is implemented in the custom code, than it is run afterward.

To leverage this **flexible customization mechanism** for implementing distributed automation applications, the custom code has to expose a data structure whose specification is detailed in the nxtControl's documentation material. That interfacing structure defines different elements that characterize the generic DLL function block, like:

- the number of input and output events;
- the number of data values that are associated to the input and output events;
- the data type associated to data values.

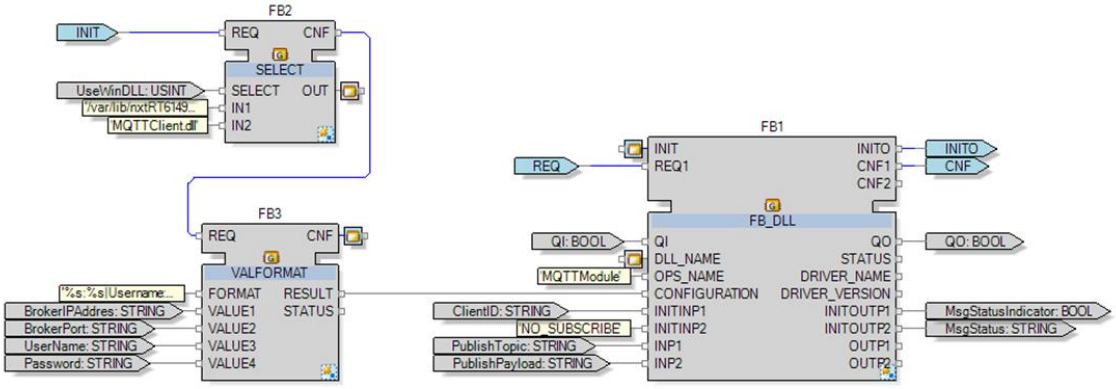

Figure 10 Usage of generic DLL FB in an application

The previous Figure shows the usage of the generic DLL function block to implement the **MQTT** protocol. To be more precise this example shows the implementation of the MQTT client publisher.

All the relevant data (*Broker IP Address, Broker Port, Client ID, Publish Topic and Payload*) needed to establish a connection with the MQTT broker and to publish a message on a topic are provided via the input variables of the DLL FB. The output variables provide the status of the message.

This is just one example on how a generic DLL function block can be used to implement a protocol and to provide interoperability between different technologies.

nxtSTUDIO #Develop Add-in

Name of the technology/ tool that uses this interface/ protocol/ API	nxtSTUDIO Engineering Tool		
Name of the partner	NXT		
Name of the interface/API	#Develop Add-in		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		No
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	X
		Digital Twin and Planning Services	

		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	X
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	Currently not used in any Pilot or Use case		

SharpDevelop (also styled as *#Develop* or *#D*) is a free and open-source integrated development environment (IDE) for the *.NET Framework*, *Mono*, *Gtk#* and *Glade#* platforms. It supports development in *C#*, *Visual Basic .NET*, *Boo*, *F#*, *IronPython* and *IronRuby* programming languages.

Within a previous EU research project (*Daedalus*), one of the activities was to integrate Virtual commissioning in the *nxtStudio* leveraging the existing *IEC61499* technology to establish real-time communication with a shop floor digital twin counterpart. This was realized by using the *#Develop* Add-In concept to create a plugin in the *nxtStudio*.

From the architecture point of view, the plugin for the Virtual commissioning hosts the implementation of the **GRPC** client, which serves as a communication protocol between two engineering environments.

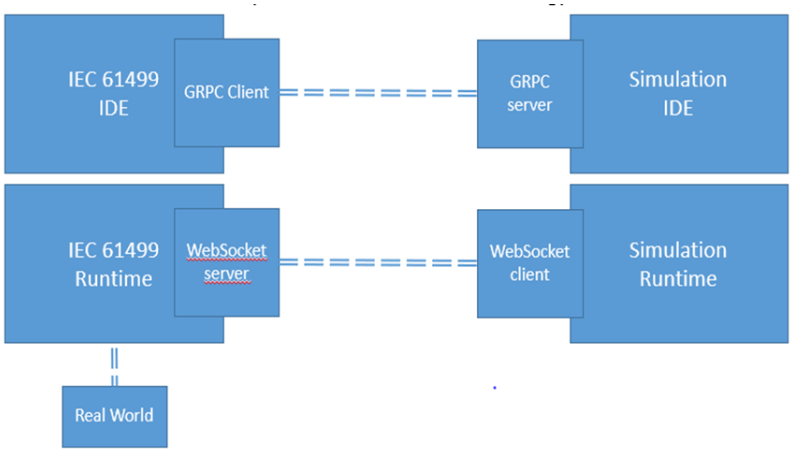

Figure 11 Communication between IEC61499 and simulation environment

ATLANTIS REST API#1 (Prediction RUL Component)

Name of the technology/ tool that uses this interface/ protocol/ API	Prediction of defects based on assets deterioration rate		
Name of the partner	ATLANTIS		
Name of the interface/API	REST APIs – GET, POST requests		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	X
Pilot(s) where this interface will be exploited	MONDRA-1, DANOBAT		

This API enables the integration of Prediction component with DSS component and other AI components. It allows the results of the Prediction component to be exposed to partners who need them. Especially, it exposes them to the DSS, which can than apply rules and trigger responses based on the prediction of failures and remaining useful life estimations. In particular, the goal of this interface is to enable the:

- Identification of deterioration for asset's condition parameters.
- Correlation of identified deteriorations.
- Rising of alarms for detected / predicted conditions.

The data messages are packed in **JSON** format and it is possible to customize the structure and semantics of data based on the application needs. The goal is to make

results of RUL Prediction available to other components that need them, using an agreed upon schema that consists of four variables.

The tool expected to be integrated by means of this interface is the Input storage subcomponent of the DSS component.

Main Inputs

- Real-time asset's conditions
- Historical data from external systems
- Maintenance logs, production information related to produced product

Main Outputs

- Calculation of the deterioration rate of the monitored assets

Functionalities

- Analysis of asset's conditions in real-time
- Detection of emerging failures
- Prediction of future failures
- Incorporation of complex event processing and trend analysis
- Identification of deterioration for asset's condition parameters
- Correlation of identified deteriorations (to be examined, if possible)

The diagram below is applicable for both APIs of ATLAS, as it incorporates functionalities from both digital technologies. It should be mentioned that this is the generalized diagram of ATLAS solutions. The exact implementation for the QU4LITY project may vary a bit, depending on the needs of the pilots and the project's specifications.

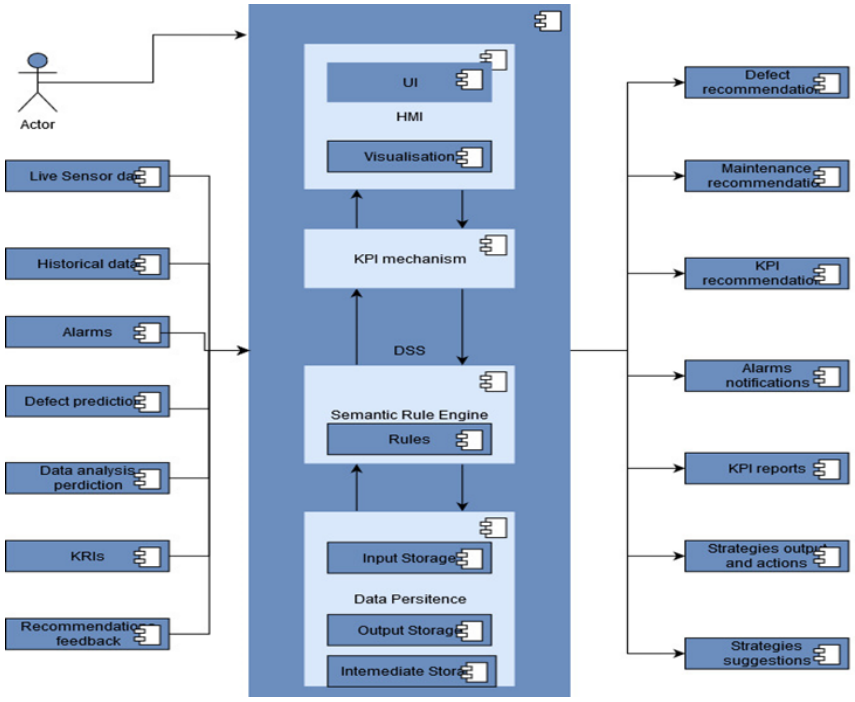

Figure 12 ATLAS Component Diagram and Architecture

All aspects of the API are based on a stateless client/server REST protocol and HTTP protocol.

Both http and https are supported. For **https** protocol bearer token will be used.

The **syntactic definition** of the API is based on the *JSON* format. When we version the Media Type and extend the language, we go through Content Negotiation based on the header. The REST API would make use of custom vendor MIME media types instead of generic media types, such as application/json. We are going to version these media types instead of the URIs.

The client makes no assumptions about the structure of the response beyond what is defined in the media type. This is why generic media types are not ideal. These do not provide enough semantic information and force the consumer REST API client to require additional hints to process the actual representation of the resource.

This is a component for which ATLAS holds the IPR, so restrictions may apply to the level of information and the extent that it can be shared.

ATLANTIS REST API#2 (DSS Component)

Name of the technology/ tool that uses this interface/ protocol/ API	Decision Support System (DSS) and Strategies for ZDM		
Name of the partner	ATLANTIS		
Name of the interface/API	REST APIs – GET, POST requests		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	
		Control Services	
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	X
Pilot(s) where this interface will be exploited	MONDRA-1, DANOBAT, PRIMA		

This API enables the integration of defect prediction/detection sources of the pilots with the DSS and other AI subsystems. It allows the outcomes of the DSS to be communicated to systems that may need them, such as the feedback engine or to trigger Strategies for ZDM.

In particular, the goal of this interface is to enable the:

- Interfacing with the specific defect prediction/detection sources of the pilots.
- Filtering out false alarms.

DSS can be considered an Artificial Intelligence (AI) subsystem which manages zero-defect processes, by:

- **Filtering** out false alarms originated from predictive analytics (i.e. from Condition Monitoring and sensorial/inspection data assessed vs tolerance bands, being processed through the defect prediction and detection layers).
- **Incorporating semantic rules** and a rule-based engine to cope with detected/predicted defects identified from corroborating sources (e.g. different sources of defect detection).
- **Deciding the mitigation actions** to cope with defects and triggering the activation of the appropriate ZDM Strategy(ies).
- **Providing recommendations** for performance improvements, based on KPI assessment and dashboards (e.g. scrap level, rework level, throughput, FAR, Precision in warning signals, defects per stage, etc, depending on availability of pilots' data).

Main Inputs

- Detected/predicted failures
- Tolerance bands
- Feedback from the shopfloor, via the Notification subsystem.

Main Outputs

- Filtering out false alarms originated from predictive analytics
- Recommendation for actions predicted failures, detected defects, maintenance and performance improvements
- KPIs
- Triggering/Activation of the appropriate ZDM Strategy(ies)
- Notifications to personnel.

Functionalities

- Management of zero-defect processes
- Filtering out false alarms originated from predictive analytics
- Incorporating semantic rules and rule-based engine
- Identification of detected/predicted failures
- Suggestions for mitigation actions to cope with failures
- Triggering/Activation of the appropriate ZDM Strategy(ies)
- KPIs mechanism

DSS will provide recommendations based on the strategies will be provided. Continuously incoming data might be used to train the system's algorithms and improve its performance. A database will be used for storing the **recommendations feedback**, which allows data storage and retrieval for training purposes. The DSS database will also include the implemented strategies outputs and will exploit them in the machine learning processes of the system.

The data messages are packed in **JSON** format and it is possible to customize the structure and semantics of data based on the application needs.

All aspects of the API are based on a stateless client/server REST protocol and HTTP protocol. Both http and https are supported. For **https** protocol bearer token will be used.

The **syntactic definition** of the API is based on the *JSON* format. When we version the Media Type and extend the language, we go through Content Negotiation based on the header. The REST API would make use of custom vendor MIME media types instead of generic media types, such as application/json. We are going to version these media types instead of the URIs.

The client makes no assumptions about the structure of the response beyond what is defined in the media type. This is why generic media types are not ideal. These do not provide enough semantic information and force the consumer REST API client to require additional hints to process the actual representation of the resource.

This is a component for which ATLAS holds the IPR, so restrictions may apply to the level of information and the extent that it can be shared.

SYN MQTT-based API

Name of the technology/ tool that uses this interface/ protocol/ API	MQTT-based data monitoring tool		
Name of the partner	SYN		
Name of the interface/API	MQTT		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	X
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	

Pilot(s) where this interface will be exploited	SYN aims to leverage this interface in the pilots where participates: PRIMA and RIASTONE.

MQTT (MQ Telemetry Transport) is an open OASIS and ISO standard (ISO/IEC PRF 20922). lightweight, publish-subscribe network protocol that transports messages between devices. It is a publish / subscribe protocol, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium [7].

The protocol usually runs over TCP/IP; however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.

The SYN MQTT protocol stack is integrated in a data monitoring tool composed of different modules, including databases (e.g. InfluxDB) and visualization tools (e.g. Grafana) represented in the following figure.

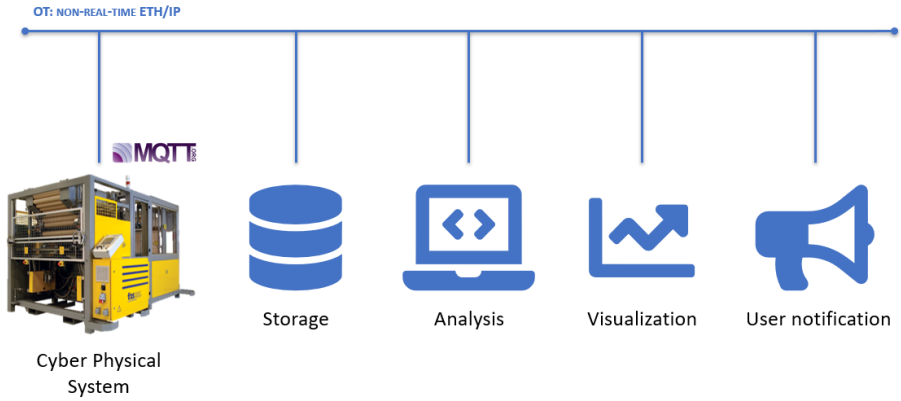

Figure 13 Monitoring Architecture exploiting the MQTT API

The data messages are packed in JSON format and it is possible to customize the structure and semantics of data based on the application needs.

Part of the communication within the modules is implemented by means of other proprietary protocols based on TCP/IP.

ATOS MQTT-based API

Name of the technology/ tool that uses this interface/ protocol/ API	MQTT- MQ Telemetry Transport		
Name of the partner	ATOS		
Name of the interface/API	MQTT		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	ATOS aims to leverage this interface in the pilot where it participates: CONTI.		

The **MQTT** protocol is used by *MASAI* to receive data from IoT based data sources. The figure below represents how this protocol is used. As it can be derived from the figure, the MQTT broker is the central component to facilitate the communication between devices and *MASAI*.

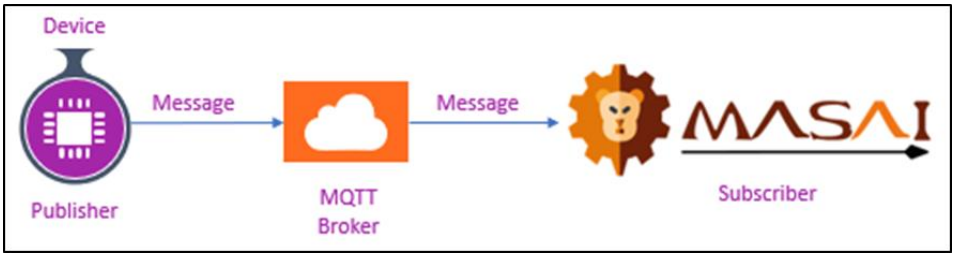

Figure 14 Communication schema based on MQTT API

The architecture of MQTT follows the publish/subscribe pattern, using topics to send and receive data. In this regard, when a new device *MASAI* is registered in *MASAI*, the necessary topics for data publishing and subscribing are created. Each device, in *MASAI*, has a unique identifier, and so the topics are specific for the device. For example, if a device with ID **device1** is created in *MASAI*, the following topics will be created in the MQTT broker:

"/ATOS/device1/attributes/+" and *"/ATOS/device1/attributes"*.

The device will publish the data into the topic while the consumer will be subscribed to any change occurred in the topic. *MASAI* expects to receive data in **JSON** format, for example:

'{"temperature":"24.4","humidity":"55.0", "type":"office"}'.

Considering the data model used internally in *MASAI*, it is important to clarify that *MASAI* is a **FIWARE** based component, and so it internally uses the *NGSI* language to describe devices. As the scope of this document is not to describe deeply the data formats used, more information about how devices are described using the *NGSI* is available at on *FIWARE* website [13].

ATOS REST API

Name of the technology/ tool that uses this interface/ protocol/ API	REST - Representational state transfer		
Name of the partner	ATOS		
Name of the interface/API	REST		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
		Engineering and Planning Services	

	Enterprise/ Ecosystem layer	**Collaboration, Business and Operation Services**	
Pilot(s) where this interface will be exploited	ATOS aims to leverage this interface in the pilot where it participates: CONTI.		

Among its functionalities, *MASAI* offers devices virtualization and data handling functionalities. But before those capabilities are ready to be used, some configuration needs to be performed in *MASAI*. To receive the necessary configuration, in form of configuration files, *MASAI* offers REST interfaces. Figure below presents a high-level view of the usage of REST interfaces.

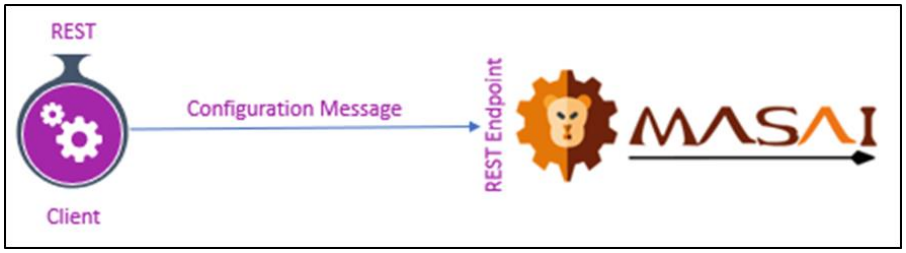

Figure 15 Communication schema based on REST API

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

By using a stateless protocol and standard operations, RESTful systems aim for fast performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole, even while it is running.

In a *RESTful Web service*, requests made to a resource's URI will elicit a response with a payload formatted in **HTML**, **XML**, **JSON**, or some other format. The response can confirm that some alteration has been made to the stored resource, and the response can provide hypertext links to other related resources or collections of resources. When HTTP is used, as is most common, the operations (HTTP methods) available are *GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS* and *TRACE*.

The payload to register data handling functionalities is mainly composed by three main groups:

- Incoming events: represents the incoming data in form of events.
- Outgoing events: outgoing events are created when incoming events match the business rules defined at the statements.

- Statements: represents the business rules to handle the data.

ATOS OPC UA API

Name of the technology/ tool that uses this interface/ protocol/ API	OPC UA - OPC Unified Architecture		
Name of the partner	ATOS		
Name of the interface/API	OPC UA		
This interface provides (ref. D2.7 page 23)	**Technical interoperability**		Yes
	Syntactic interoperability		Yes
	Semantic interoperability		Yes
At which level of QU4LITY RA functional view (D2.11 section 5.2.4.1) your tool is located	**Workcell/ Production Line layer**	**IoT Automation Services**	X
		Control Services	X
	Factory layer	**Data-driven Modelling and Learning Services**	
		Digital Twin and Planning Services	
		Simulation and Human-centric Visualization Services	
	Enterprise/ Ecosystem layer	**Engineering and Planning Services**	
		Collaboration, Business and Operation Services	
Pilot(s) where this interface will be exploited	ATOS aims to leverage this interface in the pilot where it participates: CONTI.		

MASAI offers an **OPC UA client** capable to interoperate with OPC UA servers. The figure below shows an approach to that type of communication.

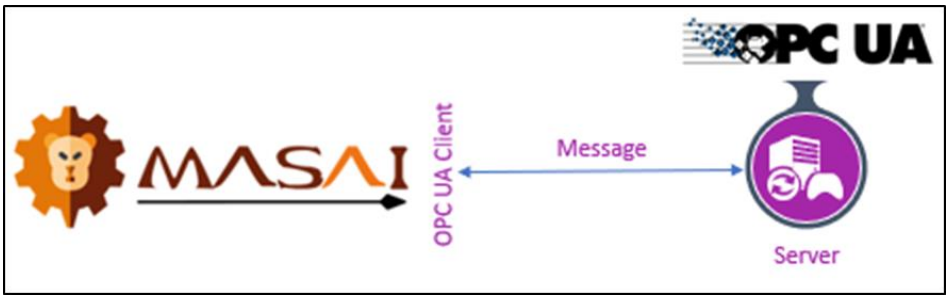

Figure 16 Communication schema based on OPC UA API

OPC Unified Architecture (OPC UA) is a machine to machine communication protocol for industrial automation developed by the OPC Foundation. Distinguishing characteristics are:

- Focus on communicating with industrial equipment and systems for data collection and control
- Open - freely available and implementable under GPL 2.0 license
- Cross-platform - not tied to one operating system or programming language
- Service-oriented architecture (SOA)
- Inherent complexity - specification of 1250 pages in 14 documents
- Robust security
- Integral information model, which is the foundation of the infrastructure necessary for information integration where vendors and organizations can model their complex data into an OPC UA namespace to take advantage of the rich service-oriented architecture of OPC UA. There are over 35 collaborations with the OPC Foundation currently. Key industries include pharmaceutical, oil and gas, building automation, industrial robotics, security, manufacturing and process control.

MASAI supports the binary protocol, i.e. opc.tcp://Server, which offers the best performance/least overhead, takes minimum resources (no XML Parser, SOAP and HTTP required, which is important for embedded devices), offers best interoperability (binary is explicitly specified and allows fewer degrees of freedom during implementation) and uses a single arbitrarily choosable TCP port for communication easing tunneling or easy enablement through a firewall.

MASAIs' OPC UA client is based on the *FIWARE OPC UA Agent*. This client interface is capable to get data made available from the OPC UA server, responsible for fetching sensor data from factory-level machinery. Before the client can receive sensor data values, sensors are mapped to the OPC UA Server Address Space as variables (or attributes). Additionally, it is also possible to control the machinery invoking methods exposed by the server.

Sensor values access is provided through a subscription mechanism. For each sensor value the OPC UA client wants to have access to, it creates a subscription specifying some parameters. Using these parameters, the client asks the server to send data according to some particular modalities. At that point the server determines if the requests can be fulfilled, otherwise it will continue sending data in a best effort mode [14].

6. Abstract API specification

The diversity of the several communication solutions adopted by the Qu4lity partners, who contributed to the questionnaire-based overview, suggests some fundamental criteria, to be considered as a rationale for the adoption of an Open API strategy.

The subject of the present section is about the adoption of a set of well-known standards, which could support the interoperability of the endpoints in a complex communication network, still respecting the design choices that brought to their specific implementation solutions.

Three aspects of interoperability are discussed hereafter.

First, and more important, the semantics of the exchanged information must be defined to the maximum possible extent. Some of the several semantic aspects could be easily agreed, because of the availability of some underlying common knowledge. A possible example could be: the unit-of-measure of an exchanged value. If this could be agreed in advance, it should be properly documented and included in the "contract" between the parties. If it cannot be defined, it should be included in the foreseen messages as an auxiliary information, which in turn needs to be semantically well defined, for instance against a list of possible values either derived from standard or agreed locally between partner. And, again, documented and reciprocally agreed. Such a level of cooperation necessarily involve the developer teams, as is hard to be automated in any way. In other and more challenging cases, the actual meaning of the exchanged data could not even be represented in the message itself, as it require a deep and reciprocal understanding of the collaborating subsystems. Even more so, the effort to document it at the best detail is a design duty.

Second, the language to be adopted in order to make the exchanged message reciprocally understandable must be well defined in advance. This brings to the selection of some specific syntax, which should be common to all or most of the endpoints in order to reduce the effort of interpreting them. However, several techniques exists to support the automated translation between different syntaxes, based on the assumption that the underlying logic is similar. As an example, conversions between XML and JSON syntax could be easily automated, even if some variety of resulting dialects needs to be constrained and controlled during the process. Another common method, on the other side, is to dynamically instruct an endpoint to produce a message selecting the requested syntax. This is a common practice, for instance, in REST based services, where a simple additional parameter could instruct the server application to select a specific syntax for the response *(e.g. &fmt=xml or &fmt=json, &fmt=html, etc.)*

Last, but obviously relevant, the variety of adopted communication protocol should be taken in account. However, the convergence toward IP based protocols, which could support interoperability at various level of the automation stack, from field to cloud, is nowadays a reality so no longer a matter of opinion. Nevertheless, several higher level protocols have been mentioned in the QU4LITY overview, which should be considered.

It is worth to notice that one of the main qualifying aspects that lead to the adoption of a certain protocol, depends on the choice between synchronous or asynchronous communication. Demanding interoperability between these distinct dialogue approaches is not always possible, as the underlying design decisions still pertain to the semantic layer of the information exchange or, event worst, they depend on the behavior of the collaborating subsystems along the flowing of time.

The availability of abstract languages oriented to the formal description of the various aspects of an API is therefore a critical building block in the process of support its interoperability. The present section concentrates on describing two candidates for such a role, which have been previously mentioned in the general discussion and which will be discussed here in better details.

The OpenAPI [2] specification (OAS) defines a standard, language agnostic, interface specification for service-oriented APIs, which allows both humans and computers to discover and to understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. By defining an API by means of the OAS, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

AsyncAPI [1] is an open source initiative that seeks to improve the current state of Event-Driven Architectures (EDA), whose long-term goal is to reach a compatible representation of EDAs and REST APIs, aimed to support the automated management of protocol specifications.

AsyncAPI derives from the OpenAPI specification, aiming at extending its synchronous REST based orientation to the world of asynchronous protocols. Compatibility and interoperability between the two specifications is by design.

The OpenAPI Initiative (OAI) is an open governance structure under the Linux Foundation. It is focused on creating, evolving and promoting a vendor neutral description format.

Since 2021, the AsyncAPI initiative is also hosted by the Linux Foundation, a condition that paves the way to a continuous convergence, also ensuring the interest for sponsors and business implementations.

Both the initiatives are supported by important actors in the business and technological world, a condition that ensure continuous development and general acceptance. Moreover, the actual adoption in business application is more and more growing, so bringing experience from the real word and consolidating their inclusion in mission critical DevOps and workflows.

While OpenAPI and AsyncAPI are growing in parallel, easing their interoperability keeps being a primary design goal.

Goals for both the environments span from automatic documentation to code generation, from discovery to event management.

OpenAPI or AsyncAPI definitions can be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

They helps in the automation of high quality documentation and ready to reuse code generation. Other applications are API management, testing, and monitoring. They support the entire development cycle of service-oriented or event-driven architecture by providing a language for describing the interfaces of the systems, regardless of the underlying technology.

Possible side-product of such a unification effort could be:

- the automated production of eloquent documentation of the protocols
- the automated generation of scaffolding code
- automated control of syntactical and protocol related constraints
- automated generation of validation code
- etc.

While OpenAPI is expressly designed to support service-oriented architecture, the AsyncAPI Specification aims to describe and document message-driven APIs. It is protocol-agnostic, so you can use it for APIs that work over any protocol (e.g., AMQP, MQTT, WebSockets, Kafka, STOMP, HTTP, Mercure, etc.). However, it does not assume any kind of software topology or architecture: messages could be exchanged through a message broker, a web server or any other kind of computer program capable of sending and/or receiving data.

In order to manage protocol specific requirements, a language construct called "bindings" helps in detailing information about each of them.

Besides helping the process of defining and automating the technological and message exchange aspects of the communication between endpoints, the adoption of a meta-language could prove to be a powerful tool to help in the definition of a common ontology of concepts and related attributes among the project's participants.

The AsyncAPI and OpenAPI languages define some prime-class concepts, which are shortly described hereafter, for further discussion.

An AsyncAPI message **broker** is a piece of infrastructure that receives messages and distributes them to software components who have expressed an interest in receiving them. The broker frequently retains messages until they are delivered, making them highly resistant to failure. RabbitMQ, Apache Kafka, and Mosquitto are some examples of brokers. An OpenAPI **server** is the piece of infrastructure where the API services are provided. It is also protocol agnostic, as it supports HTTP, CoAP, etc

A **publisher** (also known as producer) is a program that delivers messages to the broker. A **subscriber** (also known as consumer) is an application that connects to the broker, expresses an interest in a specific sort of communication, then leaves the connection open, so that the broker can send them the proper messages.

Channels represent the means through which the endpoints communicates. Because the industry lacks a standard term, they are referred with several names by different protocols: topics, routing keys, event types and so on. The **paths** are the OpenAPI endpoints which identify all the operations provided by a server. They have a hierarchical structure and could be defined in terms of their REST specific request and response.

A **message** is a piece of information that the publishers send to the broker and that all interested subscribers receive. The message's content can be anything, and they're typically classified as events, orders, commands or the like.

The message **payload** is where the actual information is packaged, to be exchanged between the communicating endpoints. Here is where most of the syntax and the semantic aspects of the messages are defined and require to be documented.

Neither OpenAPI nor AsyncAPI impose constraint for the syntax to be adopted for the actual messages. Their structure and all the information that could be useful to document their semantics are collected in so called **schemas**.

A detailed specification of the AsyncAPI syntax is beyond the scope of this deliverable: the details about the language are freely available on the AsyncAPI official website. However, some examples of their usage is included in the present document, in a subsequent section.

An AsyncAPI document is human-friendly and also machine-readable. Even in its native, raw form, it represent a useful and effective design and documentation tool. The file format must be JSON or YAML. Being composed by a simple text file, an AsyncAPI specification does not require any special IDE to support its development.

As it is easily parse-able, several useful applications could be developed over them with a minimal effort. Indeed, a very active community is constantly generating new tools, which offer services like

- generation of documentation and code;
- validation of the messages received by an application;
- application of management policies to messages;
- development of bridges between endpoints, to support different protocols

Beside the elements that structurally define the asynchronous or synchronous applications, additional details could be included in the specification file. Examples are:

- security mechanisms
- parameters and parameter substitution
- protocol dependent details

7. Tools to enable interoperability

Many companies have their own proprietary platforms and IoT gateways. They communicate using standard data formats such as JSON or XML, but there is not always common agreement on the structure or the semantics of data shared. As presented in section 6, the APIs adopted in QU4LITY rely on different technologies, protocols and architectures. Thus, we can find an API that uses the RESTful architecture to offer data and another that publishes messages using an MQTT broker.

Version 1 of the present document described a possible approach to support interoperability between digital platforms, based on the exploitation of APIs that provide data and operations flow between systems, generally distributed and interacting via a communication network. These kind of APIs aim to specify the mechanisms that tools need to implement for exchange of data and coordination/synchronization of operations.

A possible approach to face the difficulties just highlighted, enabling the interoperability between different APIs, could be to build translator according to the requirements and specifications set by each of the QU4LITY cooperating systems.

Such a development path was followed by task T3.5 and described in the corresponding deliverables, so the discussion about such a methodology will not be repeated in this document.

Nevertheless, the adoption of translators between endpoints imposes to the parties the responsibility to clearly and unambiguously describe the constituent aspects of the translation, which is based on the reciprocal expectations by the involved subsystems. In other words, the semantic, syntactical and technological facets of the communication need to be documented and shared, in order to appropriately configure the middleware tool with the correct information.

This requirement, again, do force the teams to collaborate on the basis of a common language, which could simplify and therefore enable the translation specification. The adoption of such a common set of languages is the core of the proposal resulting from the analysis carried out in this task.

8. Proof of Concept implementation

In order to demonstrate the effectiveness of the suggested approach, the QU4LITY "PRIMA" pilot was selected as a workbench for experimentation, with the aim of supplying a proof-of-concept of the applied methodology.

The PRIMA pilot builds around one of their additive machine tool products, a Power Bed Fusion (PBF) system. The main goal of the pilot is to enhance the machine with advanced monitoring capabilities and other features that enable to increase the quality of the production. The main goal of the tool is to process a set of KPIs that are representative of the process quality and that can support the operator in the identification of roots for possible problems in the quality of the manufactured parts.

The overall system will be developed by different partner, each developing its own specialized subsystem, all of them interconnected by a dense exchange of messages:

- The **PRIMA** additive machine, integrated in the overall system;
- A vision system and a real time image processing system, developed by **FHG ILT**, monitoring the quality of the layers while printing is in progress.
- A 3D visualization tool, developed by **FHG IGD**, offering a detailed vision of the manufactured part to an operator, so enabling the investigation of possible defects recognized by the vision system.
- A simulation tool, developed by **TTS**, simulating the manufacturing process before the actual run on the machine.
- A Decision Support System (DSS), developed by partner **Atlantis**, providing relevant indications aimed at guaranteeing the expected production quality.
- A Data analytics tool for Additive Manufacturing, developed by **SYN**, to harmonize the information provided by the companion tools, adding metrics to help the Decision support process.
- An augmented reality (AR) system, developed by **VTT**, integrated to the machine tool to support the operator during maintenance tasks.

The richness of the development team, both in terms of number of participants than diversity of the reciprocal roles, along with the inherent need to create a network of exchanged information, make this pilot an ideal workbench to share a common Open API.

Moreover, the development team lacked a common Open API at the beginning of the project, as all of them are independent parties, each adopting its internal DevOps methodologies. This condition also offers the opportunity to adopt the proposed QU4LITY ZDM data standard, in those cases where an Open APIs was not already provided or it could be adapted with a minimal effort.

While the development of this pilot is still ongoing during the writing of this documents, it is possible nevertheless to document the work-in-progress regarding the definition of the message exchange mechanism, which is the argument of the present discussion.

Semantics and syntax

The following diagram depict the expected communication between modules, at the initial stage of the analysis.

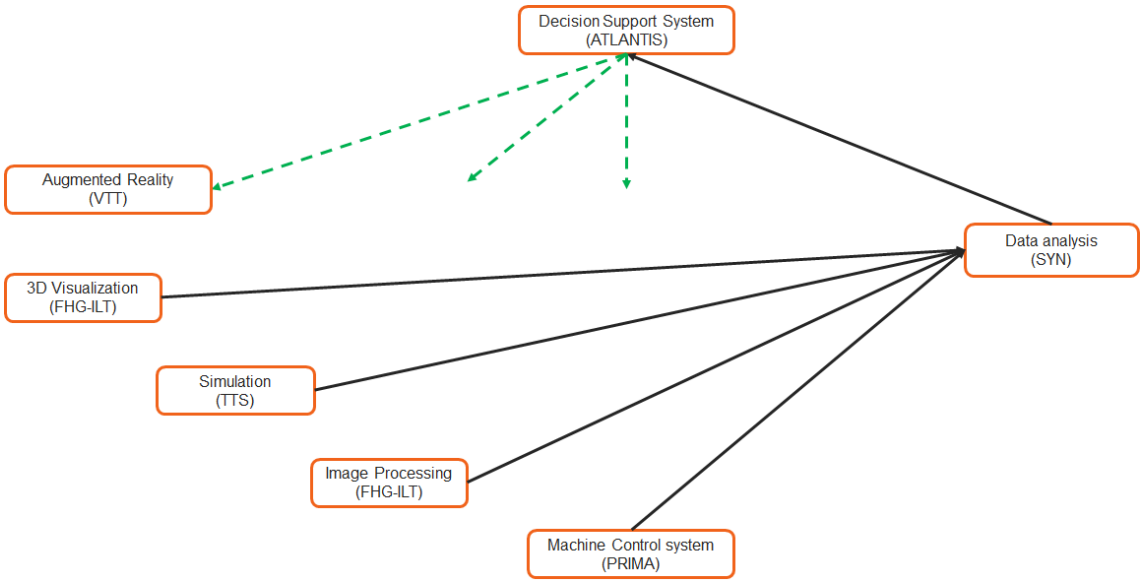

Figure 17: PRIMA pilot actors and connections

The communication needs, as depicted by the previous diagram, would suggest a client-server architecture, where several tools could communicate their actions to a master node, responsible in turn for exposing the collected information to external entities. A REST oriented architecture could therefore be selected as a first solution.

The adoption of the **Open API** specification, therefore, could appear as a good choice for the description of the expected communication scenario.

Even at such a preliminary stage of development, the need to define the basic aspects of the exchanged messages arises, as they are essential to define the reciprocal roles of each subsystem. These aspects could be, and should be, independent from the selected communication protocol. They refer indeed to the semantics of the exchanged information, first of all.

Then, they could refer to a common syntax, to be agreed with the aim to simplify the reciprocal effort for interpreting the messages.

It is worth to note that syntactical decisions could tolerate some degree of freedom. Several formal languages, indeed, allow automatic translations, which could be supplied by intermediate proxies, like we discussed in the previous sections. On the contrary, the semantic aspects must be defined and agreed to the minimal possible ambiguity.

To this aim, the schema of each of the exchanged messages has been defined adopting the "**JSON schema**" formalism. Such a standard is included as an integral

element of **both the OpenAPI and AsyncAPI** specification, despite their different orientation toward synchronous or asynchronous nature of the communication.

Some excerpt of the common schema defined by the PRIMA pilot team is reported hereafter. Despite the name 'JSON schema', the language adopted here is YAML, which is an extremely terse formalism, mostly interoperable with JSON, offering several advantages in terms of readability and write ability.

```
components:
  schemas:
    primaDatasetParameters:
      type: object
      properties:
        DatasetParameters:
          type: object
          description: "just a wrapper"
          properties:
            BuildJob:
              type: string
              description: "unique 3D printing job id"
            TotalNumberOfLayers:
              type: integer
              minimum: 1
              description: "Total Number Of Layers"
            EvaluationTags:
              type: array
              description: |
                The list of evaluation tags is set by the dataset initialization message
                with the requirement to be sorted with 'good' to bad' interpretation.
              items:
                type: string
    layerEvaluation:
      type: object
      properties:
        InspectionedData:
          type: object
          properties:
            Id:
              type: integer
              minimum: 1
              description: |
                index (starting from 1) of the current feedback message,
                i.e. a enumeration of all feedback messages until now"
            BuildJobId:
              type: string
              description: "unique identifier string of the current building job"
            Timestamp:
              type: string
              format: date-time
              description: "time stamp when the current data layer was created"

[… omissis …]

    logEntryLayerwiseInfo:
      type: object
      properties:
        "LogEntry_layer-wise_info":
          type: object
          description: "just a wrapper"
          properties:
            Id:
              type: integer
              minimum: 1
              description: "id of this layer, relative to its job"
            BuildJob:
              type: string
              description: "unique 3D printing job id"
```

```
            Timestamp:
              type: string
              format: date-time
            Event:
              type: object
              properties:
        ProcessParameter:
          type: object
          description: "just a wrapper"
          properties:
            LaserPower:
              type: integer
[… omissis …]
```

Again, it is important to note that the languages YAML/JSON has been selected as a lingua franca for the editing of the schemas, but they do not constraint the target language to be adopted for the actual messages (their 'payload'). The target language could be, just to mention typical examples: XML, HCL, CSV, JSON, YAML, TOML, CSON, plain text, base64 binary, etc.

Each of the message schemas could be stored as individual files, in order to be reused by different endpoint of the communication. The presented example combines some common schema in a single file.

Synchronous vs Asynchronous communication

During the subsequent stages of the development it became apparent that the exchange of message would be more articulated, involving not only a central node ("SYN" in this diagram) receiving most of the information, but also a direct communication between nodes, in order to coordinate their actions.

The following UML communication diagram depicts such a scenario.

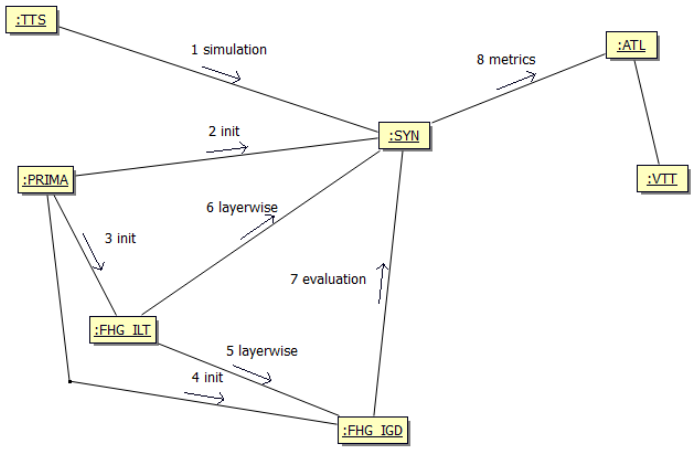

Figure 18: PRIMA pilot - raw communication diagram

Such a scenario configures the need of multicast communication, which undermines the appropriateness of the client-server architecture assumed so far.

Moreover, most of the exchanged message reveal themselves as asynchronous messages, also presenting a diversity of cause-effect. Some message from an endpoint, indeed, prelude to a sequence of messages from another endpoint (e.g. after an "init", several "layerwise" messages will follow). Other messages are just 'send and forget', with no need for a feedback (e.g. 'simulation').

Such a renewed scenario configure a better approach, based on a **broker based** architecture. A central broker will appropriately support both the mentioned cases. Each of the cooperating subsystems will "publish" messages of interest for other parties and/or it will "subscribe" for the reception of specific messages of its interested in.

The following UML communication diagram describes the updated network.

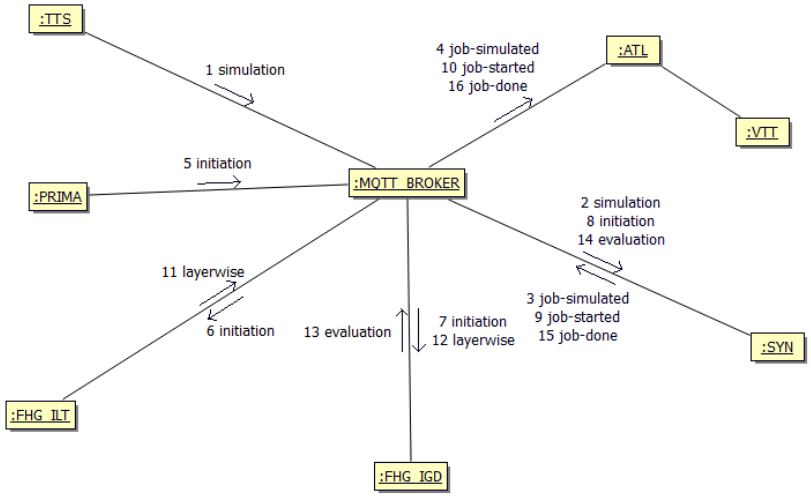

Figure 19: PRIMA pilot - broker-based collaboration

After such a modification to the architecture, the AsyncAPI specification appears to be more appropriate for the description of the interaction between subsystems. It allows, indeed, to define several details about the constituent components of the network.

However, **the schema specification** already detailed and shared by the development team **remains unchanged and still valid**.

On the contrary, the choice of a broker-based architecture outlines a possible choice between several alternative protocols. The AsyncAPI specification does not constraint such a choice, as its formalism has been abstracted precisely to allow such a freedom, still providing all the necessary details to specify the details of the selected implementation.

Based on the specific requirements of this pilot, MQTT was select as the more appropriate.

Application description

The following short excerpt of AsyncAPI specification defines the characteristics of one of the cooperating application. In this case it is the PRIMA machine controller.

This specific case is very simple, as it is composed by a single channel ('topic' in the MQTT jargon) that uniquely identify the flow of messages sent by this subsystem, which are destined to all the subscribers.

```
asyncapi: 2.2.0
info:
  title: PRIMA 3D printing machine
  version: '1.0.0'

servers:
  demonstration:
    url: q4prima.qu4lity-project.eu
    protocol: mqtt
    description: Test & demonstration MQTT broker
    bindings:
      mqtt:
        clientId: "YEP"
        qos: 0
        retain: true

channels:
  FraunhoferIGD_Visualizer/initialization:
    publish:
      operationId: primaDatasetParameters
      message:
        $ref : '#/components/messages/PrimaDatasetParameters'
        description: |
          Currently, the list of evaluation tags is set by the dataset initialization message with
the requirement to be sorted from some kind of â€˜goodâ€™ to â€˜badâ€™ interpretation.

components:
  messages:
    PrimaDatasetParameters:
      name: primaDatasetParameters
      title: Dataset Parameters from PRIMA printing machine
      summary: Inform about a new job started on the 3D printing machine
      contentType: application/json
      payload:
        $ref: './#/components/schemas/primaDatasetParameters'
```

Figure 20: PRIMA pilot - AsyncAPI application definition

Notices as the JSON schema describing the message payload is a reference is to an external file, shared with the other application which will receive such a message.

The following UML Sequence diagram better illustrates the interaction between those applications. While it is equivalent to the previous communication diagram, it highlights the need to share the same payload schema between several of them.

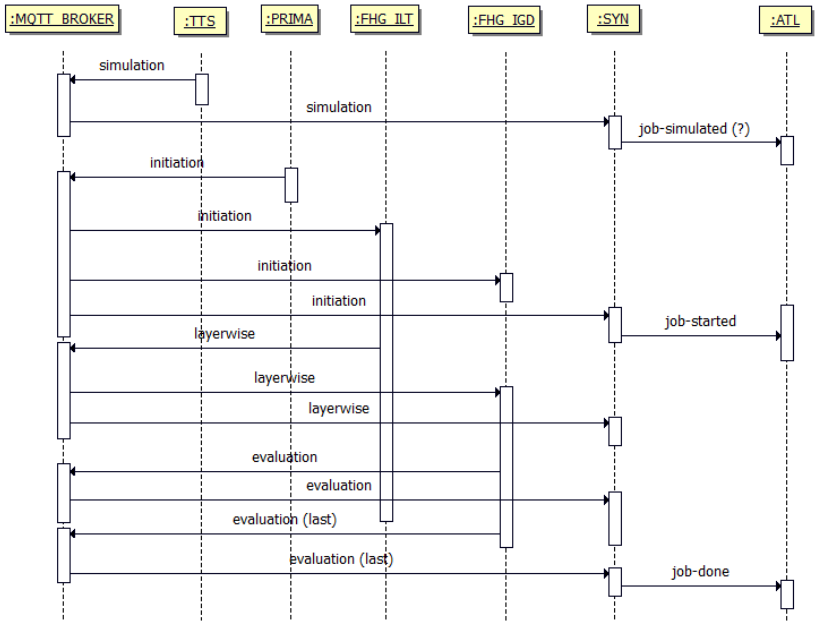

Figure 21: PRIMA pilot sequence diagram

Each AsyncAPI file describes a single application, so the present pilot will count six different descriptors. *As all of them are very similar, they will not be included here.*

Asynch and Synch coexistence

The strict closeness which binds AsyncAPI and OpenAPI comes in handy in case we need to compose asynchronous and synchronous services in the same network. In such a case, the description of both services will be mostly reusable and homogeneous, so reducing the development effort.

This could be the case, indeed, in the PRIMA pilot, where the role of the SYN node is twofold: on the one hand, it is a subscriber of most of the messages exchanged in the system, to be enriched and published to the ATL endpoint; on the other hand, it could implement local repository with a few management and visualization feature. On this respect, the "SYN srv" depicted in next diagram could be an HTTP server, willing to expose REST services, which could supply information on demand.

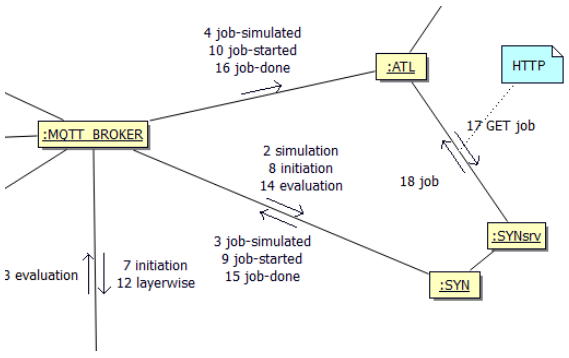

Figure 22: additional REST service

In such a configuration the JSON schema of the message payload would be fully reused and the application description would only describe its REST specific peculiarity.

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: PRIMA SYN server
  license:
    name: public
servers:
  - url: http://q4prima.synesis-consortium.eu
paths:
  /jobs:
    get:
      summary: List all PRiMA 3dprint jobs
      operationId: listJobs
      tags:
        - jobs
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of jobs
[… omissis …]
  /jobs/{jobId}:
    get:
      summary: Info for a specific job
      operationId: showJobById
      tags:
        - jobs
      parameters:
        - name: jobId
          in: path
          required: true
          description: The id of the job to retrieve
          schema:
            type: string
[… omissis …]
components:
  $ref: "./#/components/schemas/Job"
```

Notice as the final `./#/components/schemas/Job` would refer to the same JSON schema compiled for the AsyncAPI definition, so ensuring its full re-use.

As mentioned in the general presentation of the AsyncAPI and OpenAPI standards, several code generators are available, especially oriented to the creation of scaffolding applications, ready to be enriched with specific functionalities. These could support both the initial phase of development and the refactoring of existing code, in order to make them compatible with additional network connections.

In the context of the PRIMA pilot content, this opportunity has been adopted by some of the involved partners, while others found more convenient to enhance their existing solution with the agreed functionalities, which are described by the common specifications.

Finally, several tools are available to generate automatic and high quality documentation. Uniform and automated documentation is a powerful enabler to the development of interoperable code, as it supports a common and reciprocal understanding of the problem space and the interaction contracts established between cooperating endpoints of a complex system.

The following is an example of an interactive HTML page, offering an easy to navigate description of a complete (and potentially complex) application, automatically generated. Several alternative tools are available, while the developers' community relentlessly develop new solutions and enhancements to the standards.

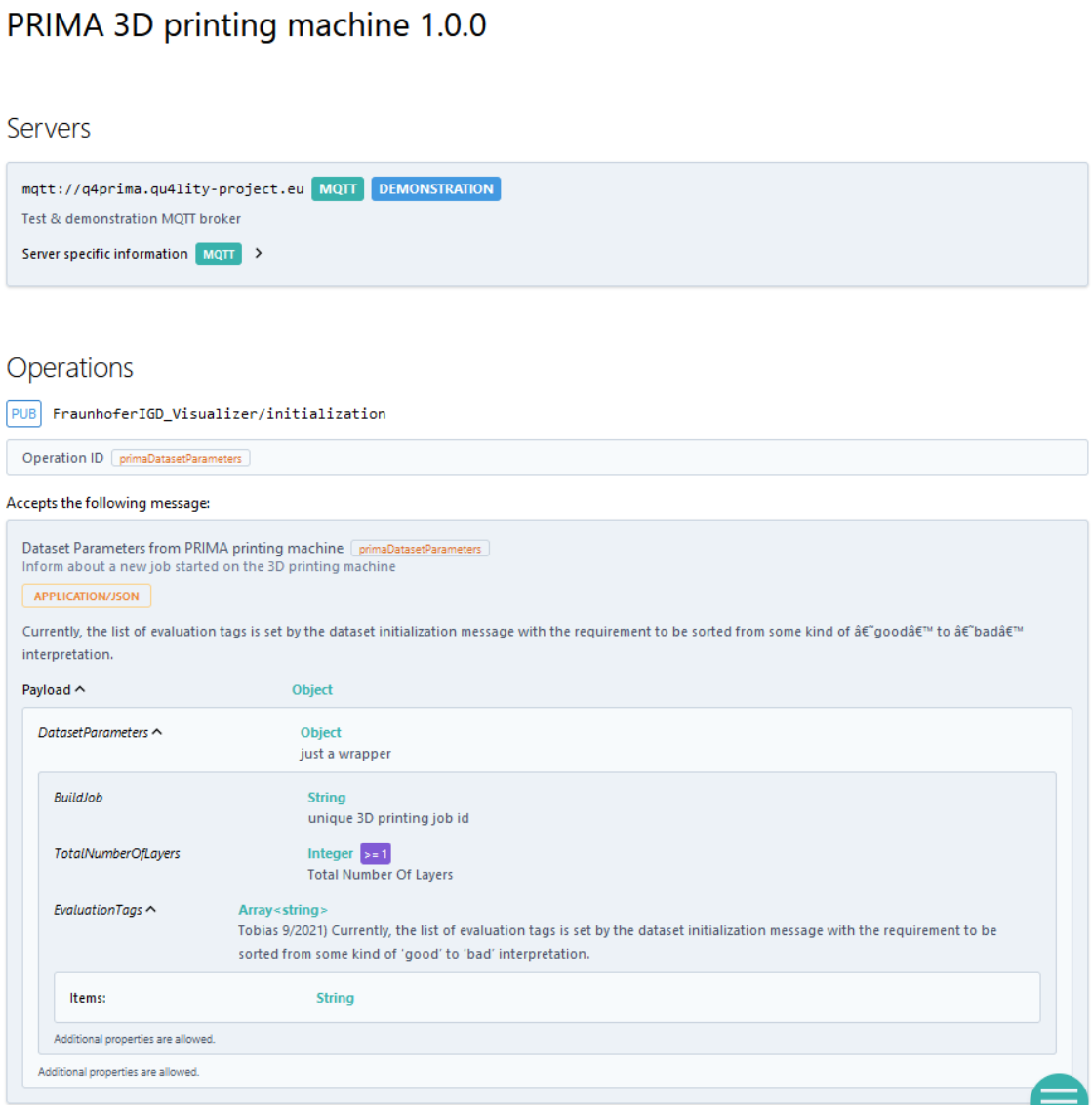

Figure 23: AsyncAPI automated documentation (1)

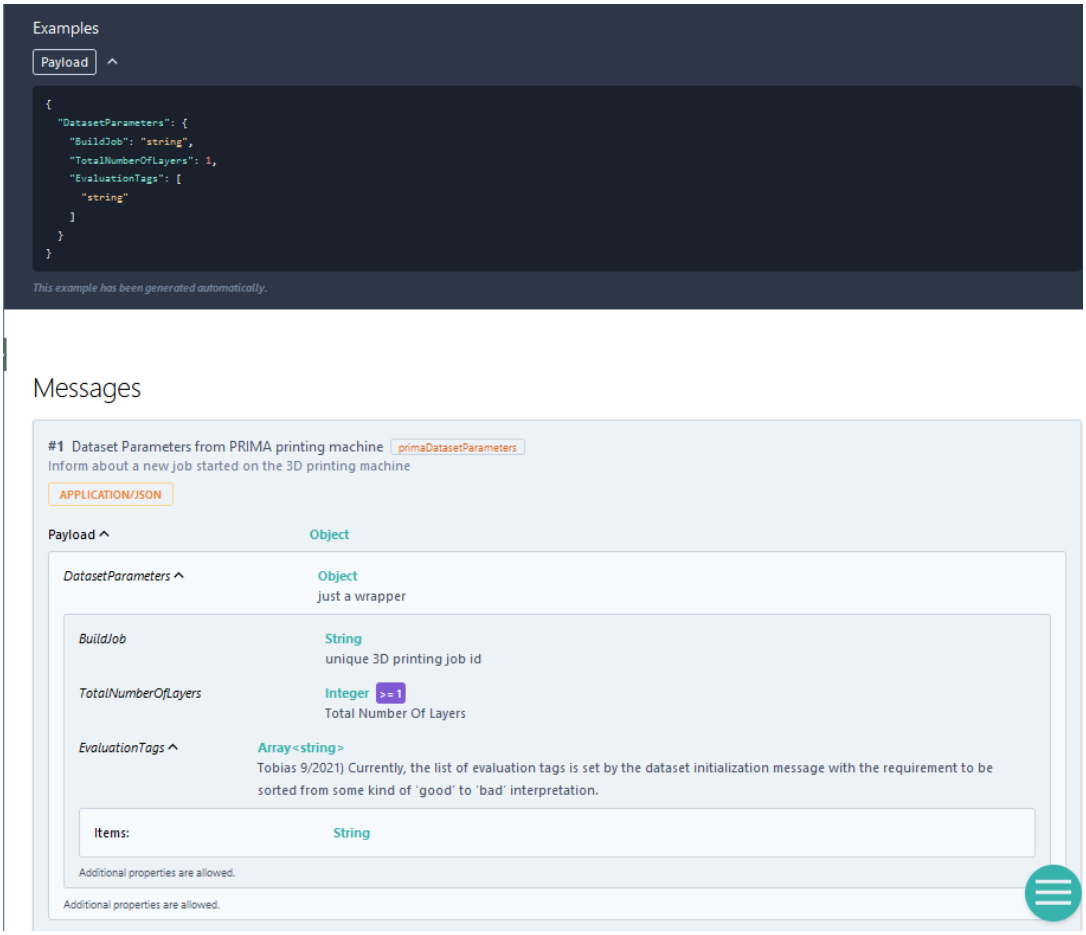

Figure 24: AsyncAPI automated documentation (2)

9. Conclusions

Task 5.4 is expected to provide the basis for data interoperability between digital platforms, enabling the implementation of the AQ paradigm in ZDM processes. To achieve that objective, Task 5.4 proposed a set of Open APIs specifications to be adopted in QU4LITY.

The interoperability between the QU4LITY technologies could will be supported by the enhancement of those digital platforms that suffer some incompatibility of APIs, so that they can interface effectively with other tools by means of Open APIs.

In general terms, the goal is to enable the use of the digital platforms from the project partners in various ZDM processes. At the same time, the Open APIs identified in Task 5.4 should also make more accessible and effective the enhancement of those technologies by third parties. That possibility will make the AQ paradigm proposed by QU4LITY largely applicable and adaptive to a wider set of concrete ZDM processes. Overall Task 5.4 will work on providing the means for composing the different digital platforms addressed in the project and their capabilities in holistic ZDM solutions for AQ.

This document introduced the key concepts that are instrumental for the identification of the Open APIs that QU4LITY should leverage and promote for implementing the AQ paradigm in ZDM processes.

Many aspects can characterize an API and therefore it is important to understand and evaluate the elements that should support on the identification and then selection of those interfaces the QU4LITY will promote as Open APIs for the implementation of ZDM processes.

An overview of the APIs supported by some of the QU4LITY partners' technologies has been presented, detailing their characteristics and highlighting protocols and data formalism adopted.

A set of Abstract API specification has been presented, namely AsyncAPI and OpenAPI, which have the potential to cover most of the interoperability requirements described by the QU4LITY partners.

A specific use-case has been selected among the QU4LITY pilots, as a proof-of-concept of the enforceability and the effectiveness of the proposed Open API solution for ZDM processes.

10. References

[1] "AsyncAPI Specification," [Online]. Available: https://www.asyncapi.com/docs/specifications/v2.2.0..

[2] OPENAPI initiative, "The OpenAPI Specification," [Online]. Available: https://www.openapis.org/.

[3] T. Vijayakumar, Practical API Architecture and Development with Azire and AWS, Apress, 2018.

[4] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems,* vol. 2, no. 1, 1984.

[5] T. L. Foundation, "gRPC," [Online]. Available: https://grpc.io/docs/.

[6] "Chapter 1: Service Oriented Architecture (SOA)," msdn.microsoft.com, [Online]. Available: https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx.

[7] "MQTT.ORG," [Online]. Available: http://mqtt.org/.

[8] "AMQP.ORG," [Online]. Available: https://www.amqp.org/.

[9] OPC Foundation, "OPC Unified Architecture Specification," [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

[10] OPC Foundation, "OPC Unified Architecture Information Models," [Online]. Available: https://opcfoundation.org/developer-tools/specifications-opc-ua-information-models.

[11] W3C, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," [Online]. Available: https://www.w3.org/TR/soap12/.

[12] Object Management Group, "Interface Definition Language," [Online]. Available: https://www.omg.org/spec/IDL.

[13] "FIWARE Device," [Online]. Available: https://fiware-datamodels.readthedocs.io/en/latest/Device/Device/doc/spec/index.html.

[14] "OPC UA Agent," [Online]. Available: https://iotagent-opcua.readthedocs.io/en/latest/opc_ua_agent_tutorial/index.html.

[15] OpenJS Foundation, "Node-RED," [Online]. Available: Web Services Oxygenated 2.

List of figures

List of Abbreviations

Abbreviation	Explanation
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AQ	Autonomous Quality
AR	Augmented Reality
CAT	Composite automation topology
CPPS	Cyber Physical Production System
CPS	Cyber Physical System
CRUD	Create, Read, Update and Delete
D2.5	D2.5 Catalogue of ZDM Assets (Version 1)
D2.11	D2.11 Reference Architecture and Blueprints (Version 1)
D4.3	D4.3 Distributed Communication and Control Infrastructure
DSS	Decision Support System
ESB	Enterprise Service Bus
IDE	integrated development environment
IDL	Interface Definition Language
IT	Internet Technology
KPI	Key Performance Indicator
MQTT	Message Queuing Telemetry Transport
MR	Mixed Reality
OAS	OpenAPI Specification
OOP	Object Oriented Programming
OPC UA	Open Platform Communication Unified Architecture
OT	Operation Technology
RA	Reference Architecture
REST	Representational state transfer
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
ZDM	Zero Defect Manufacturing

Partners: