Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

D2.4 The COMPOSITION architecture specification II

Date: 2018-09-18

Version 1.1

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file: D2.4 The COMPOSITION architecture specification II_v1.1.docx
Document version: 1.1
Document owner: CNET

Work package: WP2 Use Case Driven Requirements Engineering and Architecture
Task: Task 2.3 COMPOSITION Architecture
Deliverable type: R

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Mathias Axling	2018-06-17	Initial version
0.2	Vasiliki Charisi	2018-07-17	ATL contributions
0.21	Matteo Pardi	2018-07-18	NXW contributions
0.22	Giuseppe Pacelli	2018-07-23	ISMB contributions
0.23	Paolo Vergori	2018-07-27	ISMB contributions
0.24	Alexandros Nizamis, Thanasis Vafeiadis, Vagia Rousopoulou, Dimosthenis Ioannidis	2018-07-27	CERTH contributions. Input related to Matchmaker, SFT, Ontology, DFM and the overall scalability perspective
0.25	Farshid Tavakolizadeh, Jose Angel Carvahal Soto	2018-08-08	FIT contributions
0.3	Mathias Axling	2018-08-20	Integrated contributions
0.4	Mathias Axling	2018-08-27	CNET contributions, editing
0.5	Vivian Esquivias	2018-08-28	HMI framework
0.6	Mathias Axling	2018-08-29	Editing
0.61	Mathias Axling, Peeter Kool	2018-08-30	CNET contributions
0.7	Mathias Axling	2018-09-05	Restructuring, editing, additional content
0.71	Nacho González	2018-09-12	ATOS contributions
0.8	Mathias Axling, Matts Ahlsen	2018-09-12	Ready for peer review
0.9	Helene Udsen, Mathias Axling	2018-09-14	Peer Review comments incorporated
1.0	Mathias Axling	2018-09-18	Final version
1.1	Mathias Axling	2018-10-08	CERTH authors missing in document history

Internal review history:

Reviewed by	Date	Summary of comments
ELDIA	2018-09-14	Approved without comments
FIT-WI	2018-09-14	Approved with minor comments

Index:

1 Executive Summary

In this deliverable, the second version of the software architecture for the COMPOSITION project is described.

COMPOSITION has two main goals: The first goal is to integrate data along the value chain inside a factory into one integrated information management system (IIMS) combining physical world, simulation, planning and forecasting data to enhance re-configurability, scalability and optimisation of resources and processes inside the factory to optimise manufacturing and logistics processes.

The second goal is to create a (semi-)automatic ecosystem, which extends the local IIMS concept to a holistic and collaborative system incorporating and interlinking both the supply and the value Chains. This should be able to dynamically adapt to changing market requirements.

The objectives are achieved by the use of number of IoT enabling technologies and services together with sophisticated big data analytics and deep learning as well as a trusted framework based on blockchain technology. The main services realised by COMPOSITION are:

- Material and Component Tracking
- Product Quality Monitoring
- Manufacturing Forecasting
- Automated Procurement
- Ecosystem Collaboration Framework

The COMPOSITION architecture has been designed with consideration to compliance with RAMI 4.0 (Reference Architecture Model Industrie 4.0).

1.1 Content and structure of this deliverable

The deliverable closely follows the structure outlined by the selected architecture documentation approach described in Section 3.2. The remainder of the document is structured as follows:

Section 2 - Terminology: defines the used in the deliverable and terminology specific to the COMPOSITION domain.

Section 3 - Introduction: identifies the purpose, scope and context of the deliverable, and the architecture design and description methodology used. Provides a summary of architectural design decisions.

Section 4 – Stakeholders, provides an overview of the stakeholders, concerns, and requirements that drive the architecture design.

Section 5 - Architectural views: documents the architecture in five views: Context, Functional, Information, Development and Operational.

Section 6 - System Quality Perspectives: documents quality attributes cross-cutting several views in two architecture perspectives: Security and Scalability.

Section 7 - Summary and future work: presents a summary of the current state of architecture development and how future architecture design will proceed.

2 Terminology

Commonly used acronyms and the currently adopted domain-specific terminology used in the remainder of the document is presented in Table 1: Acronyms and COMPOSITION-specific terminology below.

Table 1: Acronyms and COMPOSITION-specific terminology.

Term	Definition
Agent Container	An agent container is a set of intelligent agents interacting through the same, shared transport protocol and referring to shared platform services such as the Directory Facilitator, DF and the Agent Management Service, AMS.
AMQP	Advanced Message Queuing Protocol, an open standard application layer protocol for message-oriented middleware (ISO/IEC 19464).
Closed Marketplace	COMPOSITION Marketplace owned by one stakeholder and typically offered to a trusted subset of other COMPOSITION stakeholders. The Closed Marketplace can be public or private. A public, closed market will accept join requests by agents living in the Open Marketplace A private, closed marketplace will accept agents only by invitation. A Closed Marketplace is structurally equivalent to the open marketplace A Closed Marketplace is physically separated to the Open Marketplace and has typically a separate infrastructure of shared platform services including the broker, AMS, DF, etc.
COMPOSITION Ecosystem	The supply chain part of a COMPOSITION system, implemented by a COMPOSITION Marketplace and involving suppliers, producers and logistics services.
COMPOSITION Marketplace	A COMPOSITION Marketplace is an agent container.
Computerised Maintenance Management System (CMMS)	A software system to schedule, manage, plan and track maintenance operations, equipment, inventory and workflows.
Decision Support System (DSS)	The component the helps the decision-making process based on a rule engine and retrieving data from other components. It also visualises COMPOSITION components data in various ways, sends notifications to users and extracts knowledge with an imbedded KPIs tool.
GUI	Graphical User Interface
HMI	Human Machine Interface

Integrated Information Management System (IIMS)	The Integrated Information Management System is a digital automation framework that optimizes the manufacturing processes by exploiting existing data, knowledge and tools to increase productivity and dynamically adapt to changing market requirements.
IoT	Internet Of Things
JSON	JavaScript Object Notation is an open-standard human-readable data format.
JSON-LD	JavaScript Object Notation for Linked Data I a standard for embedding metadata in JSON documents, linking them to an RDF model.
Key Performance Indicators (KPI)	Key Performance Indicators are extracted from factory data data in the Decision Support System and KPIs tool to create graphs.
Message Broker	A message broker is an architectural pattern for message validation, transformation and routing. A message broker can receive messages from multiple destinations, determine the correct destination and route the message to the correct channel. Used interchangeably with “Real-time event broker” in this report.
MQTT	MQ Telemetry Transport or Message Queue Telemetry Transport. A binary, lightweight messaging protocol for small sensors and mobile devices (ISO/IEC PRF 20922).
OPC-UA	OPC Unified Architecture, IEC 62541, is an open, SOA-based, platform-independent machine to machine communication protocol for industrial automation.
RDF-A	Resource Description Framework in Attributes is a W3C Recommendation for embedding metadata in HTML and XML documents types, linking them to an RDF model.
Rule Engine	The heart of the Decision Support System, where rules about operational and maintenance processes are created based on finite state machines or non – deterministic state machines.
SSL	Secure Sockets Layer is a standard technology for securing internet connections.
Supply chain	The sequence of processes involved in the production and distribution of a commodity
TLS	Transport Layer Security is the successor to version 3 of the SSL protocol,
Virtual Marketplace	A Virtual Marketplace, or group is a "multicast" group of agents interacting with each other in the context of a negotiation. The group can be: – persistent over negotiations or

	– just be defined for a single negotiation exchange. A Virtual Marketplace lives in, and exploits the infrastructure of the Open Marketplace.
Value chain	The process or activities by which a company adds value to an article, including production, marketing, and the provision of after-sales service.
XML	Extensible Markup Language is an open-standard human-readable data format.

3 Introduction

This deliverable, D2.4 “The COMPOSITION architecture specification II”, describes the current state of the architecture and results of the software architecture design activities for the COMPOSITION system up to and after the M2 milestone in month 10 of the project. The results up to milestone M2 were reported in D2.3: “The COMPOSITION architecture specification I”. The system architecture design activities are carried out in Work Package 2 (WP2), “Use Case Driven Requirements Engineering and Architecture”, in the COMPOSITION work package structure defined by the project specification (COMPOSITION, 2016).

3.1 Purpose, context and scope of this deliverable

The purpose of this report is to provide a high-level overview of the design of the COMPOSITION system. It documents the main elements of the system and the relations between these elements. It also documents the stakeholder concerns - expressed in the project specification and user requirements – that drive the architecture design and the resulting design decisions that affect the system on an architectural level. This deliverable will focus on the fundamental concepts and properties of the COMPOSITION system. Properties and design decisions for architectural elements are described when these affect the overall design or are needed for the understanding of the components’ impact on architecture design.

Detailed descriptions of the elements of the architecture, e.g. the Security Framework, Decision Support System or Digital Factory Model, are available as separate deliverables. The reader should refer to these for implementation details and specifications; references have been included in the appropriate sections. This report will include some diagrams and descriptions from detailed deliverables. Sections in D2.3 that have since been provided as separate deliverables have been abbreviated or left out of this document.

Several key functional requirements and architectural constraints are defined in the project specification, available at the start of the project. Gathering and validation of requirements and definition of pilot scenarios and use cases have been performed in parallel to the architecture definition process. The results of these activities have been reported in D2.1 “Industrial use cases for an Integrated Information Management System” and D2.2 “Initial requirements specification”. The D2.5 report “Lessons Learned and updated requirements report I” provided an update of the requirements which has served as input to the architecture design activities in WP2.

3.2 Architectural Design and Documentation Approach

As in D2.3, the documentation will adhere to the IEEE 42010 standard, using several viewpoints to frame the concerns of the system stakeholders and illustrate the design decisions taken. Specifically, the IEEE 42010 compliant framework presented in (Rozanski & Woods, 2012) will be used. This has been extended with the concept of perspectives, which are used to evaluate quality attributes cross-cutting several viewpoints, e.g. security, evolvability or scalability.

The architecture reference model RAMI 4.0, developed in the Industrie 4.0, is used for integration of research and technical development efforts in the area of industrial IoT. This collaboration and integration with other initiatives is a strategic objective of the project (COMPOSITION, 2016).

3.2.1 Methodology

The inception phase (Kruchten, 2004) of the architecture design is documented in the project specification, which introduces several canonical architectural elements connected to technical objectives, tasks and deliverables, providing a basic functional decomposition of the system. An initial list of system components was derived from this source in architecture workshops early in the project. Developing and integrating these components is necessary to ensure that the strategic and technical objectives of the project can be met.

In subsequent workshops, this bottom-up design approach has been complemented by additional components and design decisions on standards and architectural mechanisms (Kruchten, 2004) to integrate the components. The design of individual components has been carried out in parallel to the architecture design. Evaluation and revision of this design is conducted continuously in workshops and design meetings (no formal architecture evaluation has been performed). As the components have matured and feedback from pilot development and revised requirements have produced, the architecture design has become predominantly top-down. The development work has been driven by high-priority use cases, and cross-cutting concerns with architectural scope has been handled in separate design tracks involving key developer partners. With the

component design and key architectural decisions in place, strategies and mechanisms for scalability, evolvability and other quality attributes can be elaborated.

The COMPOSITION architecture design process and the architecture description in this document follows the ISO/IEC/IEEE 42010 “System and software engineering – Architecture description” (ISO/IEC/IEEE42010, 2011), which superseded the IEEE 1471 “Recommended Practice for Architectural Description for Software Intensive Systems” (IEEE, 2000). See the conceptual model of architecture descriptions from (ISO/IEC/IEEE42010, 2011) below.

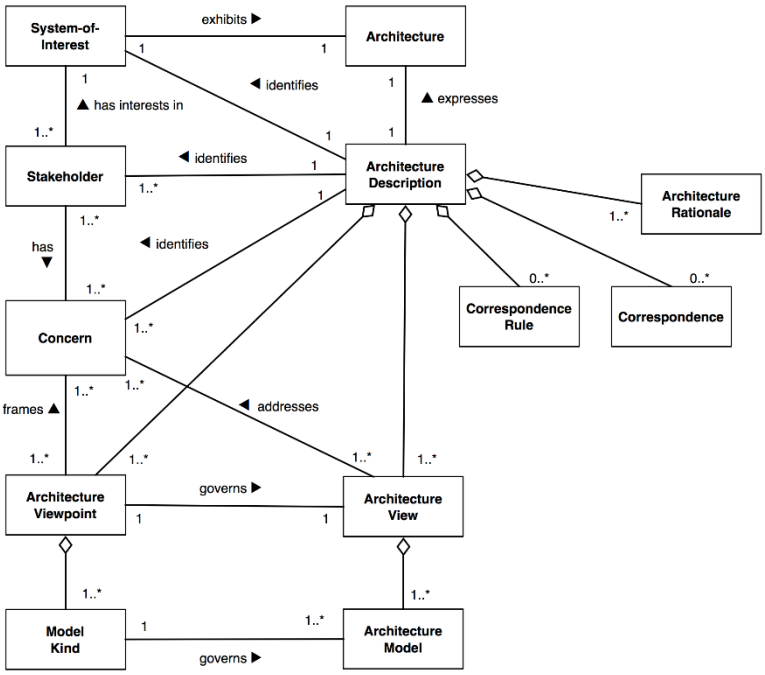

Figure 1: ISO/IEC/IEEE 42010 Architecture Description Conceptual Model[1]

As can be seen from the ISO/IEX/IEEE 42010 conceptual model of architecture descriptions, a viewpoint uses a set of model kinds to frame a specific set of concerns that stakeholders have about a system. However, quality properties such as security, performance or availability need to be considered across several viewpoints. In (Rozanski & Woods, 2012), the complementary concept of architectural perspectives is introduced to address these cross-cutting concerns.

We have addressed the system design from five viewpoints – context, functional, information, deployment and operational - and two perspectives, the security perspective and the scalability perspective.

3.2.2 Reference Architecture Model Industrie 4.0

In COMPOSITION, the Reference Architectural Model Industrie 4.0 (RAMI 4.0)[2] will be adopted to communicate the scope and design of the system, to further collaboration and integration with other relevant initiatives by framing the developed concepts and technologies in a common model.[3] COMPOSITION alignment with RAMI will be described in section 5.

RAMI 4.0 is a reference architecture model for Industrial Internet of Things (IIoT). It has been developed by the Industrie 4.0 platform, submitted as DIN SPEC 91345 and is available as IEC Publicly Available Specification 63088:2017. RAMI 4.0 is modeled on Smart Grid Architecture Model (SGAM), IEC 62262, Enterprise-control system integration (IEC62264, 2013) and the IEC 62890 ”Life-cycle management for systems and products used in industrial-process measurement, control and automation” (IEC, 2013). The focus of RAMI 4.0 is on manufacturing, primarily modelling systems for the production process and product life cycle.

[1] http://www.iso-architecture.org/42010/cm/
[2] https://www.zvei.org/fileadmin/user_upload/Themen/Industrie_4.0/Das_Referenzarchitekturmodell_RAMI_4.0_und_die_Industrie_4.0-Komponente/pdf/5305_Publikation_GMA_Status_Report_ZVEI_Reference_Architecture_Model.pdf
[3] Pictures in this section copyright “Umsezungsstrategie Industrie 4.0 – Ergebnisbericht, Berlin, April 2015”

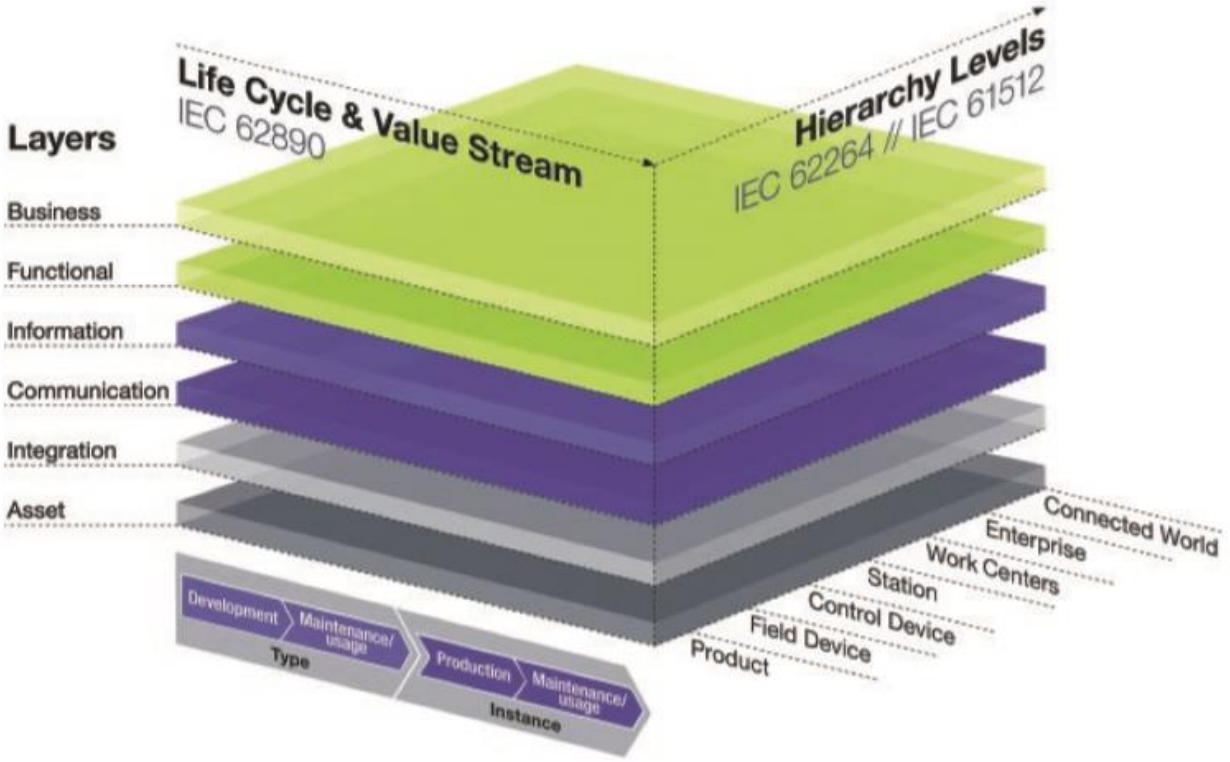

Figure 2: The three dimensions of the RAMI 4.0. (Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0), 2015)

In the three dimensional model, existing standards and architectures and candidate solutions can be plotted, overlaps and gaps can be identified and resolved. It provides a map of Industry 4.0 components, solutions and requirements by the three axes IT Layers, Hierarchy Levels and Life Cycle and Value Stream.

The purpose of the reference architecture model is to promote common understanding of different architectures for industry 4.0. It can be used to derive specific architecture models and align existing solutions. Examples of applications are:

- Provide a shared understanding of the function provided by every layer and the defined interfaces between the layers.
- To see where existing and emerging architectures fit in and allow discussing associations and details of components.
- Identification of overlaps and the scope of preferred solutions
- Identification of existing standards, closure of gaps and loopholes in standards, minimization of the number of standards involved
- Identify new business models and applications
- Identification of use cases for Industry 4.0

A more comprehensive description of the RAMI4.0 model can be found in Appendix 1: The RAMI4.0 Model.

4 Stakeholders, Concerns and Architecture Decisions

This section describes the stakeholders of the COMPOSITION system and their concerns. These concerns are expressed in different form and in different artefacts. Scenarios and requirements express some of these concerns. Although the system envisioned in the project specification is scoped to address the needs of all these, priorities must be made. Finally, architecture decisions pertaining to fundamental concerns are documented.

4.1 Stakeholders

The COMPOSITION system has several stakeholders, whose interests and concerns are expressed in the project governing documents. These may be categorized in groups, here we use the canonical ones from (Rozanski & Woods, 2012). The three key stakeholder groups for COMPOSTION have been identified as developers and maintainers (grouped together since these are basically the same in this case), acquirers, and users.

Acquirers are the European commission in H2020 framework, whose goals and concerns are stated in the project specification (COMPOSITION, 2016) and the technical and strategic objectives therein. These describe the main goals of the system, some software artefacts that will be delivered, and the need for collaboration with other projects, and re-use of results, in the industrial IoT and factory of the future programmes.

The developer stakeholder group consist of the technical partners in the project, commercial- and research-oriented. The concern of commercial partners is to produce exploitable results that can be sold as products or services and produce innovations that can provide a competitive advantage in their respective market. Research organizations need to produce significant contributions to their respective field and build platforms and knowledge for further research. The concerns of these stakeholders are captured in the innovation and exploitation documents, the DOA and to some extent in the requirements.

The user stakeholder group are the pilot partners and future users of the system, whose concerns are mainly expressed in the scenarios, use cases (D2.1 “Industrial Use Cases for an Integrated Information Management System”) and requirements (D2.2 “Initial requirements specification”, D2.5 “Lessons Learned and updated requirements report I”.). These capture the needs of the manufacturing industry and the priorities of the pilot partners.

4.2 Requirements

In a process parallel to the scenario development, described in report D2.2 “Initial requirements specification”, several user requirements have been elicited. These have been entered into the project management system (Atlassian JIRA) and complemented by additional non-functional and operational requirements added by the developer stakeholders. In the initial requirements phase, 105 requirements have been gathered, quality checked and improved.

The development efforts have been be guided by the tasks in the project management system directly connected to the requirements. The initial list of requirements has been revised and the results reported in D2.5 “Lessons Learned and updated requirements report I”. Further updated requirements will be reported in D2.6 “Lessons Learned and updated requirements report II”.

4.3 Scenarios and Use Cases

Scenario workshops with mainly the user stakeholder group and some participants from the developer stakeholder group have been conducted to evaluate how COMPOSITION could optimise processes for manufacturing, logistics and supply chain collaboration within the scope of the pilots defined by Technical Objective 3.1. This resulted in nine functional intra- and inter-factory scenarios for describing application areas of the COMPOSITION system. These were detailed in 16 use cases for the pilots that capture user stakeholder concerns, presented in D2.1 “Industrial Use Cases for an Integrated Information Management System”. These use cases have undergone some revision in the second year of the project.

The design and development work have been organized around on a set of prioritized use cases from these scenarios, to ensure that the architecture provides coverage of the base functional requirements. The use cases have been organized in three tiers by priority based on importance to user stakeholders, developer stakeholders, acquirers (by impact on COMPOSITION objectives). User stakeholders and developer

stakeholders rated the use cases by the coverage they provided of the systems intended functionality, innovation potential and exploitation potential. The use cases in Tier 1 have been first in priority to be implemented, tier 2 started being implemented as Tier 1 were nearing completion and Tier 3 are to wait until Tier 1 and 2 are ready. All use cases have been considered in design decisions, however. A set of business modelling use cases have recently been introduced which are also to be implemented, if possible.

Table 2: Prioritized Use Cases

Tier	Use Case	Scenario
Tier 1	UC-BSL-2 Predictive Maintenance	
	UC-KLE-1 Maintenance Decision Support	INTRA-2
	UC-KLE-4 Scrap metal collection and bidding process	INTER-1
	UC-ELDIA-1 Fill-level Notification – Contractual wood and recyclable materials management	INTER-2
Tier 2	UC-BSL-5 Equipment Monitoring and Line Visualization	INTRA-1
	UC-KLE-2 Delayed Process Step	
	UC-BSL-3 Component Tracking	INTRA-3
	UC-KLE-7 Ordering raw materials	INTER-3
	UC-ATL-3 Searching for recommended solutions	INTER-4
Tier 3	UC-KLE-3 Scrap Metal and Recyclable Waste Transportation	
	UC-BSL-7 Automatic long-term tracking of high value materials for physical security	INTRA-3
	UC-BSL-4 Automatic Solder Paste Touch Up	INTRA-4
	UC-ATL-1 Selling software/consultancy	
	UC-ATL-2 Searching for solutions	INTER-4
	UC-ATL/NXW-1 Integrate external product into own solution	
	UC-NXW-1 Decision support over marketplace	INTER-5
Business Modelling Use Cases	UC-BM-1 Waste notification, certificates and collection	BM Subcase of UC-KLE-4
	UC-BM-6 Contract fulfilment and supply chain management	BM Subcase of UC-KLE-7

The use cases have driven the development work and will be the primary instrument for analysing the functional suitability of the design. Architectural design work, e.g. for communication or persistence mechanisms, has been performed in parallel with the involved development stakeholders.

4.4 Concerns and Architectural Decisions

4.4.1 Concerns

The goals of the COMPOSITION system are stated in the strategic and technical objectives in the project specification (COMPOSITION, 2016) and can be found summarized in the table below. These are necessary objectives stated by the acquirers of the system. There is an emphasis on interoperability, integration and analysis of information from heterogenous sources, dynamic adaptation to market requirements and innovativeness.

- Strategic Objective 1: Create a digital automation framework (the COMPOSITION IIMS) that optimizes the manufacturing processes by exploiting existing data, knowledge and tools to increase productivity and dynamically adapt to changing market requirements.
  - Technical Objective 1.1: Innovate and extend the FI-WARE and FITMAN catalogues of Generic Enablers with an innovative CPS-aware library of open, standard connectors specialised for real-time architectures for interoperability in manufacturing to ease the integration and coupling of data, information and knowledge from existing, heterogeneous, sources in the factory.
  - Technical Objective 1.2: Research and develop innovative, multi-level, cross-domain analytics detecting complex patterns in manufacturing big data sets, and implementing a continuous deep learning toolkit for re-adaptation and adjustments of operational metrics, in real time.
  - Technical Objective 1.3: Develop a set of modelling and simulation tools including a Decision Support System (DSS) to help users build the digital models of processes and products and to forecast impacts of reconfigurations of the production process.
- Strategic Objective 2: Enable the *COMPOSITION ecosystem* by designing and implementing a technical operating system supporting connected and interoperable factories, with their stakeholders and, by optimizing manufacturing and logistics processes through new innovative services and practices.
  - Technical Objective 2.1: Design and implement *a Log Oriented Architecture*, based on blockchain technology, ensuring the trusted, secure and automated exchange of supply chain data among all authorized stakeholders, to connect factories and support interoperability and product traceability along the supply chain.
  - Technical Objective 2.2: Provide *end-to-end security* from factory floor to cloud services encompassing major mechanisms in a seamless and fully integrated manner including authentication and access control, transport security, as well as system security, while maintaining suitable levels of IPR and knowledge protection.
  - Technical Objective 2.3: Develop an *interoperable agent-based marketplace*, where each party is represented by one or more agents, endowed with sufficient autonomy to set up exchanges and to enable new economic collaboration models.
- Strategic Objective 3: *Demonstrate and validate reference implementations* of the full COMPOSITION ecosystem in real value and supply chains to foster take-up and re-use at European level.
  - Technical Objective 3.1: Implement, demonstrate and validate the COMPOSITION operating system in *two multi-sided pilots*.
  - Technical Objective 3.2: *Collaborate and integrate* successful concepts and technologies with other relevant initiatives such as Industrial Data Space and FITMAN.

Figure 3: The strategical and technical objectives of COMPOSITION

The concerns of the user stakeholders are covered by the use cases listed in section 4.2.

The developer stakeholders, i.e. the technical partners, are interested in the exploitability of COMPOSITION results. The system should be compatible with the existing products and stakeholders should be able to supply components and services complementing and extending the system on the COMPOSITION aftermarket. Developer stakeholders use different programming languages and make use of existing software frameworks in the project.

Developer and maintainer stakeholders also have an interest of offering their (software) services using the COMPOSITION system (D2.1).

This creates requirements for the extensibility and evolvability qualities of the system, and the need of a set of standards and interfaces that companies developing a component extending the COMPOSITION system can adhere to. Components should not use programming language or platform specific inter component communication. Opens standards should be used and special consideration should be taken to the ones already supported by the development stakeholder products.

The formats, protocols and interfaces (“open, standard connectors”) should be designed to enable both use of and extension of FI-WARE and FITMAN Generic Enablers as well as the integration of concepts and technologies from other initiatives in Industrial IoT.

The context in which the COMPOSITION system will be deployed is expected to be heterogenous, with different factories using different infrastructure. The architecture design will have to take this into account and allow for flexibility in deployment of components and adaptation to existing infrastructure.

COMPOSITION services and applications should be possible to deploy independently of each other under different licences to accommodate for the interests of the commercial consortium partners. Licensing must allow for commercial usage of individual components or the entire system. Incorporating or applying open source licensing affecting the possibility of commercial exploitation, such as GPL, is explicitly forbidden (COMPOSITION, 2016).

Security should be seamlessly integrated in the entire system and allow for integration of components from external sources into the COMPOSITION platform. The use of open standards is thus a requirement from the security perspective as well.

4.4.2 Architectural decisions

The COMPOSITION project specification (COMPOSITION, 2016) provides a basic functional decomposition of the system. This considers the objectives of the acquirers, the frameworks and components brought to the project by developer stakeholders and provides a division of development work in alignment with the project plan. The decision was made to build the system bottom up starting from the components given by the breakdown in the project specification and revise this as needed. The aim has been to support an extensible modular design where other components could be added as needed.

Existing components form developer partners are integrated in the system. COMPOSITION re-uses earlier results, frameworks and standards familiar to the partners, e.g. the LinkSmart Middleware and the Symphony BMS. This provides a code base to build on and provides compatibility with existing product lines, enhancing exploitability of the results for the partners. The design of external interfaces and extension points have been made to allow for the use of other frameworks providing similar functionality.

With these preconditions, a number of architectural design decisions, or choices for architectural mechanisms (Kruchten, 2004), have been made to address the stakeholder concerns. These have been made in architecture workshops, dedicated discussions, using input from the design process for individual components. Below is a summary of architectural decisions.

The following sections will use some terms and concepts that will be explained in other sections.

4.4.2.1 Development

The COMPOSITION system is comprised of existing and specifically developed components from several development stakeholders with specific expertise. The use of heterogenous platforms and frameworks as well as existing products from several development stakeholders within the COMPOSITION system will result in different build chains and platforms being used. External actors in the inter-factory ecosystem are also likely to use different technologies. This has made the project suited to a development approach where teams dedicated to a specific system service apply the technology stack most appropriate to the task. Integration is performed through shared use of standards, well defined interfaces and componentization.

This approach is also applied in the HMI development where several “micro frontends”[4][5] are developed independently and integrated in the portal.

Extensions and additions to the COMPOSITION system will have to support the standards used but may use the frameworks and technology best suited for that component. No alignment of technical platforms or software build chains is necessary.

4.4.2.2 Deployment and System Management mechanism

The development stakeholders will be free use the most appropriate technology stacks and target different runtime platforms. To provide both a consistent deployment and system management mechanism, all components will be made available as pre-configured, container-based instances. As described in section 5.5.1, COMPOSITION have chosen Docker as the container implementation and uses Portainer as a management tool. Orchestration of clusters of nodes in larger installations for load-balancing and failover may

[4] https://www.thoughtworks.com/radar/techniques/micro-frontends
[5] https://micro-frontends.org/

be performed using e.g. Kubernetes or Docker Swarm. Portainer is compatible with Docker Swarm mode, which consequently is the first choice for managing clusters of deployment nodes.

4.4.2.3 Communication Mechanism

Given the emphasis of extensibility, interoperability, analysis of heterogenous data and loose coupling in the COMPOSITION system, the general communication mechanism for the system will be data-centric and messaging-based, where factory data is published and interested components (performing e.g. analytical or supervisory functions) subscribe to this data without direct addressing between components. This will be built using standard message broker components with extensions for security, multi-protocol and multi-format support.

The focus of COMPOSITION is on functionality that requires “human scale” response time, e.g. visualization, simulation, forecasting rather than real-time device control in the sub-millisecond range. It is therefore not required to build on very fast device-device integration protocols (e.g. Data Distribution Service (DDS)[6]) as a communication layer, but rather include such protocols as a possible asset layer should it be needed. Interoperability and integration of heterogenous data sources for analysis, optimization and decision support are the primary concerns for the communication mechanism design.

The intra-factory IoT interoperability functionality builds on LinkSmart and Symphony BMS, which use MQTT as its message-based communication mechanism. The developer stakeholders have extensive experience with the LinkSmart platform, which has been used in several large IoT projects previously and using this will be effective in developing the core interoperability functionality of the project. However, alternatives were considered. Standards such as the Foundation Open Platform Communications-Unified Architecture (OPC-UA)[7] and DDS are already used in industrial applications. In terms of architectures for industrial applications, the proposed solution has more similarities with the message-centric design of DDS than the more device-centric model of OPC-UA. However, the platform allows both for directly addressing devices, requesting data and subscribing to data by type without knowledge about the hardware involved. An OPC-UA adapter for integration with compatible installations has been developed to address exploitation concerns. (OPC-UA is the recommended standard for implementing the RAMI4.0 communication layer, and MQTT and AMQP are defined transports in the OPC-UA Pub/Sub Architecture.)

4.4.2.3.1 Inter-factory communication

Data sharing between marketplace actors and agent communication is message-based and use the COMPOSITION eXchange Language (CXL) extension to the Foundation for Intelligent Physical Agents (FIPA) ACL language specification. Some external management interfaces and security (e.g. log-on, token validation) will expose REST-based services over HTTP.

The AMQP protocol is used for intra-factory messaging in the COMPOSITION Marketplace. It is a very flexible protocol that may be configured for different message routing schemes and emulation of other protocols such as MQTT, STOMP, XMPP or the Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)[8]. It also supports fine-grained access control for message exchanges and queues. The project has selected RabbitMQ as the implementation of this mechanism. RabbitMQ is open source software, extensible and has support for multiple platforms.

4.4.2.3.2 Intra-factory communication

The external interfaces of components in the COMPOSITION system use RESTful HTTP interfaces for request-response communication with other components. For message-based communication, e.g. sensor and forecasting data, the MQTT protocol are used. Widely used in IoT applications, and with a low message overhead, this protocol is already supported by several components in COMPOSITION. MQTT may be transparently used by clients on top of an AMQP broker architecture.

The Intra-factory Integration Layer uses custom adapters for integration with sensor platforms and existing systems.

The OGC SensorThings API Data Model is used for system-generated factory information passed between COMPOSITION components.

[6] http://www.omg.org/spec/DDS/
[7] https://opcfoundation.org/about/opc-technologies/opc-ua/
[8] https://www.rabbitmq.com/community-plugins.html

4.4.2.4 Security mechanism

The Security Framework will manage authentication and authorization of actors in COMPOSITION, and access to the system data, service endpoints and HMI. The security framework has been integrated with the message broker, thus allowing all components to use the security system in a uniform manner. Standards used will be TLS, Open ID Connect, and Oath 2.0. Keycloak and EPICA are used to implement these standards, together with components developed in COMPOSITION.

A Security Information and Event Management (SIEM) solution is being implemented to analyse large volumes of messaging data and raise security alerts.

Blockchain functionality will also be integrated in the broker functionality, providing distributed trust for any message sent through this mechanism. Multichain is used as the blockchain implementation in COMPOSITION. This implementation and extension to the bitcoin protocol supports logging of immutable data streams – not only asset transactions – which was a good match for COMPOSITION requirements.

A Reputation Model for the marketplace agent system is currently in the design phase.

4.4.2.5 Data persistence mechanism

Component-specific configuration data and caching is handled inside the components, whereas regarding the shop-floor data, the approach of reusing existing components from technical partners has been followed. Thus, instead of implementing everything from scratch, COMPOSITION is relying on Symphony BMS not only for collecting real-time data from external sources, but also for their persistence. Symphony BMS storage service leverages on its internal mechanisms to save the information collected during its operation and provides interfaces for retrieving it. In order to allow for the use of other frameworks that may provide similar functionality, the design choice has been to create RESTful APIs that might be compliant also with most common standards (or to-be), such as FIWARE and OGC Sensor Things API. This is discussed in more detail in section 5.4.2.

Data generated internally in COMPOSITION, e.g. output from a trained artificial neural network, uses the OGC SensorThings format. Consequently, an OGC SensorThings compliant data store is used. DFM implements a subset of the standard and has the necessary storage capacity for the pilot installations. However, there are several implementations available for OGC SensorThings API, e.g. GOST[9] and FROST[10].

4.4.2.6 Metadata mechanism

The Digital Factory Model (DFM) (COMPOSITION, 2016), described in section 5.4.1.2, is the system source of information on classes and instances in the factory. It contains information on production lines, sensors, the id of a sensor, what phenomenon it reports data for, format and unit of measurement.

Other parts of the system, such as the middleware, the message broker and the human computer interfaces, will need this information when searching for or subscribing to messages containing data on specific objects or types of objects. E.g., the intra-factory interoperability layer will publish information coming from a temperature sensor.

This may be published containing metadata in-band, e.g. containing information on the unit of measurement or associated production line, or the metadata may be located out-of band. In the latter case, components subscribing to data for a production line will have to first locate the relevant data sources using the DFM and then subscribe to data based in the identifiers of these data sources.

Some components are the source of all metadata regarding the data streams, e.g. the BDA IoT Learning Agent, which publishes metadata in-band by default. The default is to communicate metadata out-of-band. When new data sources are added, the metadata is communicated to the DFM via the message broker. There is a well-defined mapping between the OGC SensorThings data model and the DFM schema.

4.4.2.7 External interfaces, standards and protocols

JSON is selected as the internal and external communication data format. It is a text format that is completely language independent but uses conventions that are familiar to programmers. Also, it is easy for machines to parse and generate this format. These properties make JSON an ideal format for data-exchange.

Agents in the inter-factory marketplace communicate through messages encoded in a dedicated language named COMPOSITION eXchange Language (CXL). CXL has been designed as a dialect of the well-known

[9] https://www.gostserver.xyz/
[10] https://github.com/FraunhoferIOSB/FROST-Server

Foundation for Intelligent Physical Agents (FIPA) ACL language specification, with a dedicated syntax and with reference to a well-defined set of ontologies for representing the message payload data. This is the external interface through which actors in the marketplace interact and exchange data. The messaging protocol used is AMQP.

Adaptation to external data sources in the intra-factory system is handled by the Intrafactory Adaptation Layer, providing implementations for sensor communication protocols as well as custom adapters for e.g. existing ERP systems. The internal interfaces and standards used are OGC SensorThings, with communication over MQTT and REST-based services.

5 Architectural views

5.1 Overview

The strategic objectives of COMPOSITION state two main deliverables of the project: a digital automation framework to integrate data along the value chain inside the factory, and a largely automatic ecosystem to interconnect different stakeholders in the supply chain.

COMPOSITION has two main goals: The first goal is to integrate data along the value chain inside a factory into one integrated information management system (IIMS) combining physical world, simulation, planning and forecasting data to enhance re-configurability, scalability and optimisation of resources and processes inside the factory to optimise manufacturing and logistics processes.

The second goal is to create a (semi-)automatic ecosystem, which extends the local IIMS concept to a holistic and collaborative system incorporating and interlinking both the supply and value chains. This should be able to dynamically adapt to changing market requirements.

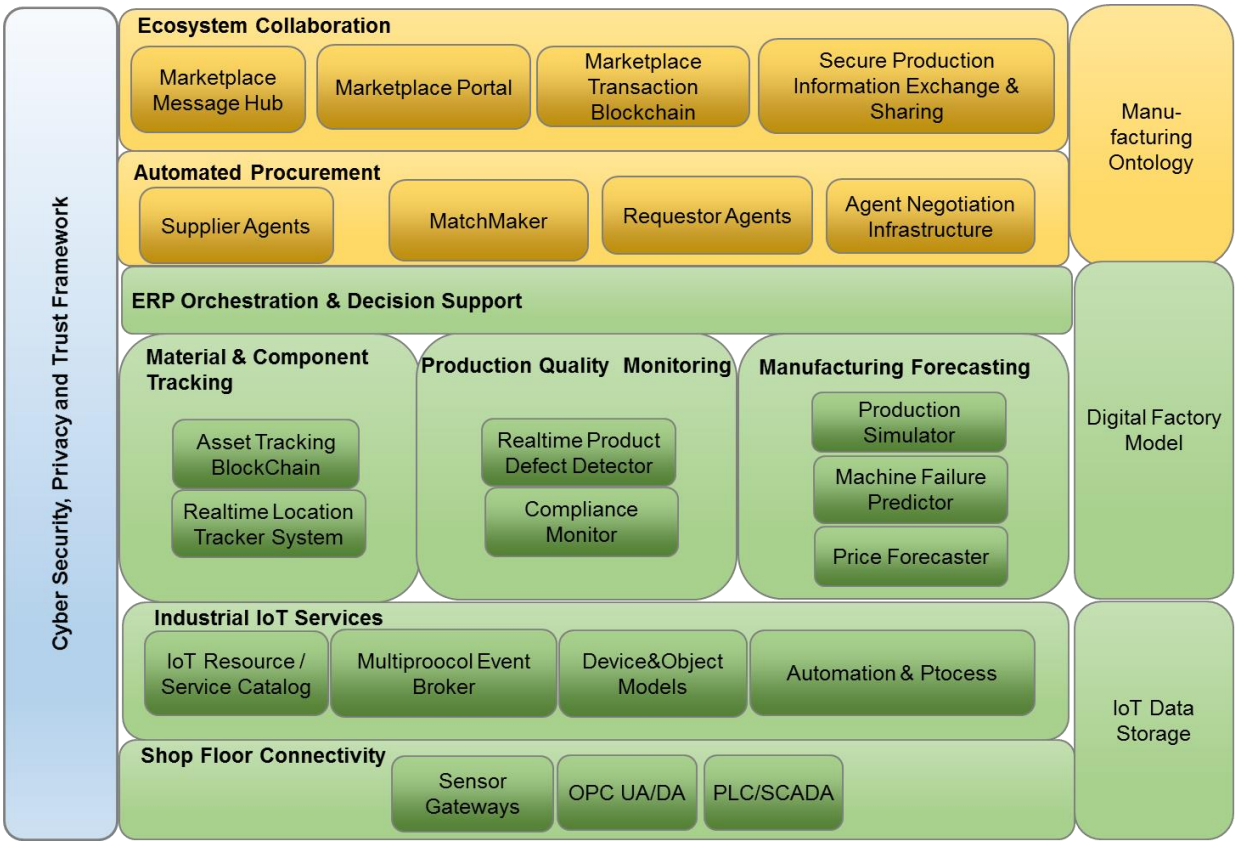

Figure 4: COMPOSITION conceptual architecture

The digital automations framework combines the data sources in the factory value chain, data from the production lines, ERP systems, forecasting, simulation and analytics data to form an integrated information management system (the COMPOSITION IIMS). At the lowest level the *Shop Floor Connectivity* provides access to devices, machines, equipment and sensors installed in the factory. The *Industrial IoT Services* layer creates an Internet of Things environment and enables standardised communication, discovery, data exchange and service innovation mechanisms.

The *Industrial IoT Services* feeds a number of business services with collected IoT and other production data:

Material & Component Tracking

A *Realtime Location Tracker System* keeps track of where products and other valuable components are on the shop floor while an *Asset Tracking Blockchain* is used to log transfer and movements of components in the manufacturing chain.

Production Quality Monitoring

The *Compliance Monitor* is responsible for checking that a product is manufactured and handled according to relevant regulations. The *Realtime Product Defect Detector* uses advanced data fusion and big data analytics to detect any deficiencies in a product.

Manufacturing Forecasting

The *Machine Failure Predictor* uses deep learning and advanced big data analytics to predict failures of machine and needs of maintenance. The *Price Forecaster* uses trained artificial neural networks to forecast the price of products and components. A Production Simulation and Forecasting Engine allows shop managers to simulate effects of re-configuration of processes inside the factory to optimise manufacturing and logistics processes.

Automated Procurement

One of the main innovations of COMPOSITION is the use of agent technologies to automate the procurement and negotiation process. Autonomous *Supplier* or *Requestor Agents* that negotiate and reach agreements with other stakeholders. A *Matchmaker* helps in find and matching best available offers with request.

Ecosystem Collaboration Framework

A virtual marketplace is envisioned where each party is represented by one or more semi-autonomous agents. To enable the COMPOSITION ecosystem an infrastructure for an *Agent Marketplace* is developed to support dynamic and automated connections between stakeholders in the supply chain, making manufacturers, suppliers and logistics interoperable and optimizable. The *Market Event Broker* propagates message between different actors in the marketplace. Trust is achieved by the use of an *Audit Log Blockchain* to maintain an immutable ledger of agreements and transactions.

Meta Data and Storage

Finally, *IoT Storage* allows for logging and storing of historical data from the shop floor. The *Digital Factory Mode*l is a high-level representation of the shop floor, stations, cells, productions lines and all the IoT sensors. The *Manufacturing Ontology* contains semantics about the market place.

Cyber Security, Privacy and Trust Framework

The Security Framework managing Cyber Security, Privacy and Trust, is a cross-cutting concern spanning the entire platform, providing end-to-end security by means of standard and widely used protocols for identification and distributed trust (e.g. OpenID and the Bitcoin blockchain protocol).

5.2 Context View

The Context View describes the system boundaries and interactions with its environment: how the system is connected to actors in the marketplace and other systems, e.g. existing factory infrastructure.

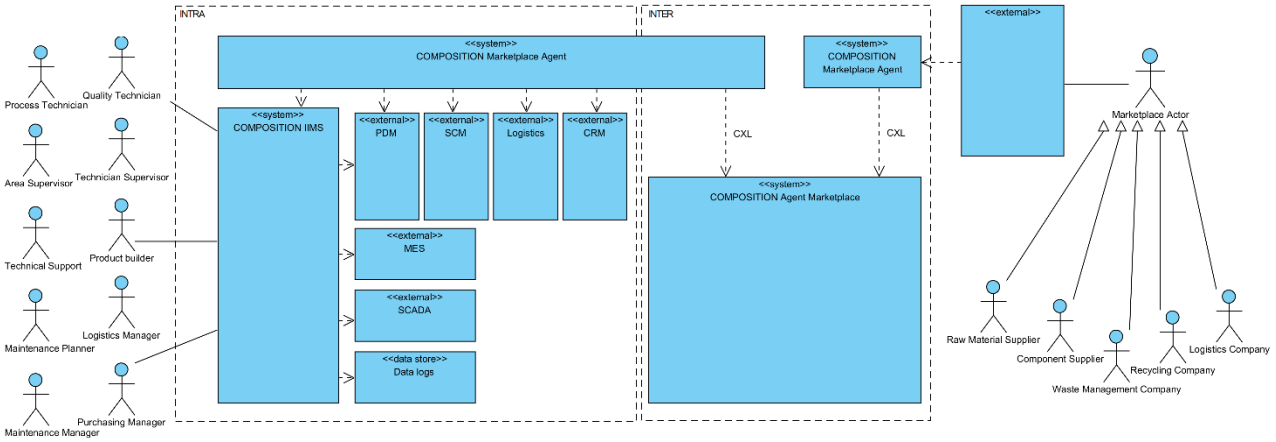

Figure 5: The COMPOSITION system context view

The value chain IIMS interacts with the actors in the value chain and external systems in the factory, e.g. Product Data Management (PDM), Manufacturing Execution System (MES) and Supervisory Control And Data Acquisition (SCADA). Some analytics components in COMPOSITION use external data logs as input. The COMPOSITION Marketplace Agent is the intermediary between the factory IIMS and the COMPOSITION

Marketplace. The agent uses information from external systems for Product Data Management (PDM), Supply Chain Management (SCM), Logistics or Customer Relationship Management (CRM) and/or data from COMPOSITION to initiate and guide the actions it takes in the Marketplace. The Agents use the discovery, communication and data sharing facilities of the COMPOSITION Agent Marketplace to create supply chains and share factory information with business partners.

The intra-factory system manages assets in the manufacturing value chain (the RAMI4.0 Life Cycle and Value Stream). For each such asset, whether it is a product or manufacturing equipment, data is collected and stored by the industrial IoT services. From this data, key performance indicators and analysis models are extracted, to support business services such as material and component tracking, product quality monitoring, manufacturing forecasting and automated procurement in the ecosystem collaboration framework of the COMPOSITION marketplace.

The COMPOSITION marketplace can be seen as a particular variation of a Multi-Agent System (MAS). MAS have been widely investigated in research, and their application domains range from Distributed Constraints Optimization (DCO) problems to coordination and delegation of computational tasks. While the adoption of agent systems in automatic negotiation, i.e., for DCO problems, is not new, as witnessed by the huge amount of literature available, application of such techniques in real-industrial environments, in a fully decentralized set-up still presents some research challenge and offers possibilities for advancing the state of the art. As part of the architecture specification process documented in this deliverable, activities on the agent marketplace mainly lead to a fully de-centralized definition of MAS, including the de-materialization of traditional agent containers into a much lighter set of collaborating software (agents) sharing a common communication infrastructure and common agency services (i.e., white and yellow pages).

According to the COMPOSITION approach, agent containers are defined as follows.

An agent container is a set of intelligent agents interacting through the same, shared broker (can be a cluster) and referring to shared platform services such as the Directory Facilitator[11] and the Agent Management Service."

Differently from approaches, in which the agent container is seen as a central runtime environment where all the agents belonging to a certain system live, in COMPOSITION agents are designed to live at the stakeholder premises (or in its IT infrastructure). This permits on one hand, to improve trustworthiness of agents, and acceptance, as no real code access is possible for entities other than the agent owner itself. On the other hand, it permits to remove typical constraints of traditional MAS systems, e.g., (a) the single point of failure represented by the Agent Container, (b) the scalability issues, (c) the techniques for enabling container-to-container communication, (d) the performance issues related to central deployment of computationally intensive agents. Moreover, the fully distributed approach proposed in COMPOSITION, reduces as much as possible the typical overhead of intercommunicating agent containers. Agency services are in fact shared naturally among distributed agents, thus removing the typical issues of duplication among containers and the related synchronization and/or delegation problems of activities needed for effectively supporting agent search and/or directory services.

While being decentralized by design, the COMPOSITION marketplace definition is centred on a so-called communication broker, which might be identified as single-point-of-failure for the architecture. However, several studies, and results in literature, show that design solutions can be adopted, based on clustered deployment, which can ensure high resilience to failures for these kind of broker-centric messaging infrastructures (see Section 6.1).

In COMPOSITION, an agent-based marketplace is simply defined as an "agent container".

Despite this simple, technical, definition, several variations of the marketplace concept are introduced including the distinction between open and closed marketplaces as well as the introduction of temporary association of agents, or "virtual marketplaces".

COMPOSITION foresees the possibility to have more than one market place running at the same time, serving different communities. However, according to the project specification (COMPOSITION, 2016), the marketplace must support the discovery of stakeholders not part of established supply chains. Assuming that in first instance a single market place corresponds to an extended supply chain, the concept of a so-called "open marketplace" can be introduced.

[11] In COMPOSITION a more advanced version of such an agent, namely the MatchMaker, operating based on ontology models is adopted.

A COMPOSITION Open Marketplace is "a COMPOSITION Marketplace open to any stakeholder having valid COMPOSITION credentials".

All players of the COMPOSITION ecosystem shall have a representative in the Open Marketplace. However, some stakeholder might decide to invite other stakeholders to participate in a Closed Marketplace, e.g., to protect/isolate certain supply chains. Such an invitation is managed through suitable agent interaction (i.e., messages) and/or through a dedicated marketplace portal. Closed Marketplaces are structurally equivalent to open marketplaces. The main difference with respect to an open marketplace is that a closed marketplace is a separated marketplace with its own infrastructure, e.g., AMS, DF and communication broker. Closed Marketplaces typically run on the premises of the marketplace owner and are subject to additional join and/or participation policies defined by the marketplace owner. The closed market place operations and exchanges are "isolated" from the open marketplace. A closed marketplace is defined as follows.

"*A COMPOSITION Closed Marketplace is a COMPOSITION Marketplace owned by one stakeholder and typically offered to a trusted subset of other COMPOSITION stakeholders. The Closed Marketplace can be public or private. The former will accept join requests by agents living in the Open Marketplace while the latter will accept agents only by invitation. A Closed Marketplace is physically separated by the Open Marketplace and has typically a separate infrastructure including the broker, AMS, DF, etc."*

In case collaboration within agents shall occur on a temporary basis, *Virtual Marketplaces*, are supported, e.g., through grouping mechanisms similar to multicast communication. In particular,

"*A Virtual Marketplace, or group, is a "multicast" group of agents interacting with each other in the context of a negotiation. The group can be persistent over negotiations or can just be defined for a single negotiation exchange. A Virtual Marketplace lives in, and exploits the infrastructure of an Open Marketplace*."

While these technical innovations are still subject of active research, and will certainly be refined during the project lifespan, they already open new exploitation possibilities for the COMPOSITION marketplace concept and contribute to lower the technology acceptance level for industrial stakeholders.

More specifically:

- The *Distributed Marketplace* derives from strict interactions with the COMPOSITION industrial partners and provides means to ensure trust on the system, as the involved stakeholders retain full control on their software representatives on the marketplace. Moreover, it opens possibilities for new businesses in the supply chain, e.g., the marketplace infrastructure provider, which can be independent from involved stakeholders and might require a fee for using provided services. Such services include basic connectivity, agency services and the possibility for stakeholders to define and run their own *Closed Marketplaces*.
- The *Closed Marketplace* allows the marketplace owner, typically the "central actor" of a supply chain, to keep control on involved partners and to ensure a certain degree of reliability of actors involved in the chain(s). This concept provides a tuneable tool to trade-off the need of marketplaces open to possibly new stakeholders (*Open Marketplace*) and the contrasting need of having trusted, certified suppliers able to guarantee proven quality in provided materials / services. This ability to tune the "openness" of a certain marketplace is a relevant factor for effective adoption of COMPOSITION, possibly opening access to very controlled supply chains, e.g., those subject to strict certification processes.
- While being central to the marketplace, supply chain formation and related activities (e.g., post-sell services) are not the only focus of the marketplace. Active advertisement and support to service / stakeholder search is a valuable asset, as witnessed by explicit requirements set by the project SME providing added value services, e.g., consultancy, integration and customization. The inclusion of such needs in the initial design of the COMPOSITION marketplace shall increase the overall exploitability of the project outcome, by widening the possible stakeholder base.
- To form and integrate supply chain as discussed above, sharing of information along the supply chain is crucial. Products are nowadays composed of parts from different suppliers and are possibly being assembled in more than one manufacturing plant. This has often been achieved through custom point-to-point integrations with specific partners. The agent marketplace will enable *data sharing* for products and production processes with other actors in the supply chain in a secure, flexible and standardized manner. Access-controlled closed or virtual marketplaces and a reputation model for agents will make it easier to select trustworthy partners with which enterprise data can be shared.

5.3 Functional View

The purpose of the functional view is to describe the main functional elements of the system; their roles and responsibilities, interfaces and dependencies.

5.3.1 High-level functional view

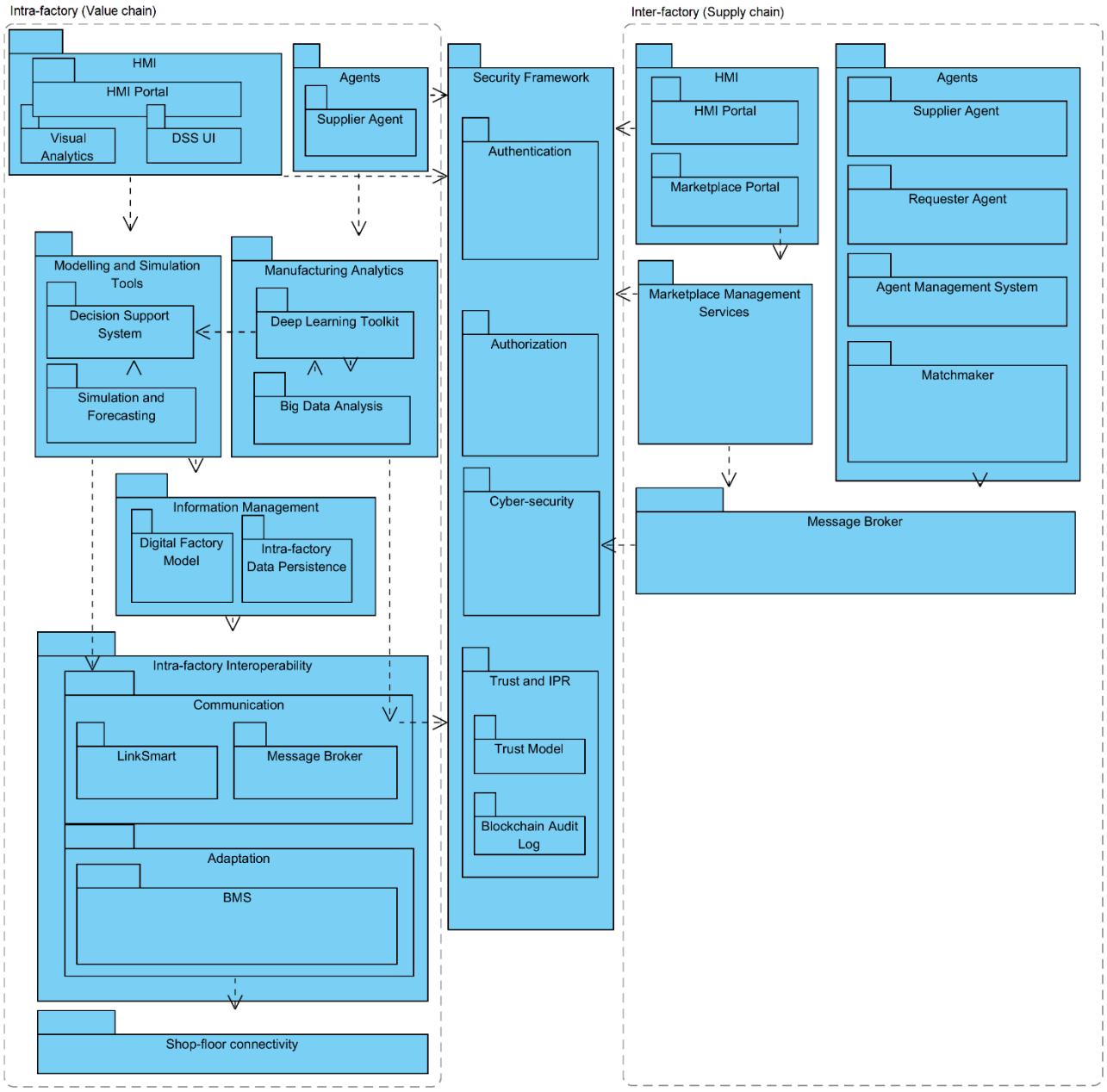

Figure 6: High-level functional view of COMPOSITION architecture

The above diagram describes the COMPOSITION system from business architecture functional view. Generic functional components like Complex Event Processing (CEP) and Deep Learning ANNs (Artificial Neural Networks) are used to implement business specific functionality, e.g. machine failure prediction. The Intra-factory Interoperability Layer connects external factory systems and heterogeneous sensors and provides a uniform model and set of protocols for handling this information to the other packages. The Security Framework provides authentication, authorization, SIEM (Security Information and Event Management), blockchain services and a trust model to the inter – and intra-factory system. Some functional packages are part of both the inter- and intra- factory system, e.g. the common HMI framework and the Message Broker.

It is worth mentioning that the main differences between the Deep Learning Toolkit and Dynamic Reasoning Engine have been highlighted during the architecture definition workshops. The former acts as a continuous learning toolkit for providing predictions on both historical and live data streams from the shop floor level based on Artificial Neural Networks models and supervised learning techniques. The latter provides simulations to needed components, such as the Decision Support System, based on both live and virtual data in a bidirectional manner, simulating possible criticalities adding hypothetical data perturbation to live streams.

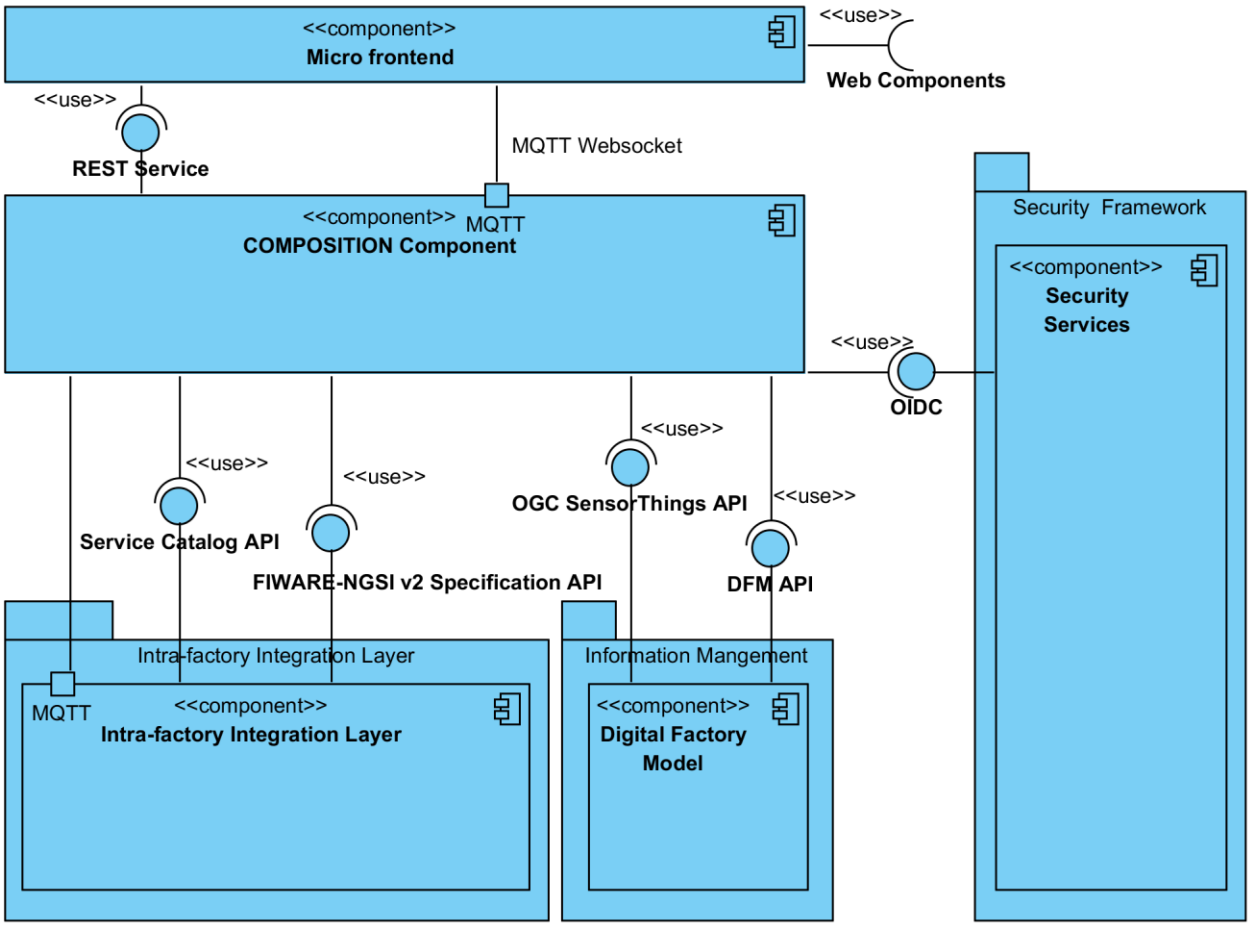

Figure 7: COMPOSITION Component dependencies

The external interfaces of a generic COMPOSITION component providing business function (in the RAMI4,0 Functional Layer) is illustrated in the above diagram. New analysis tools need only to conform to the relevant interface specifications (and the deployment design) to be integrated into the system.

5.3.1.1 RAMI 4.0

A mapping of the COMPOSITION system components to RAMI 4.0 Layers can be seen in Figure 8.

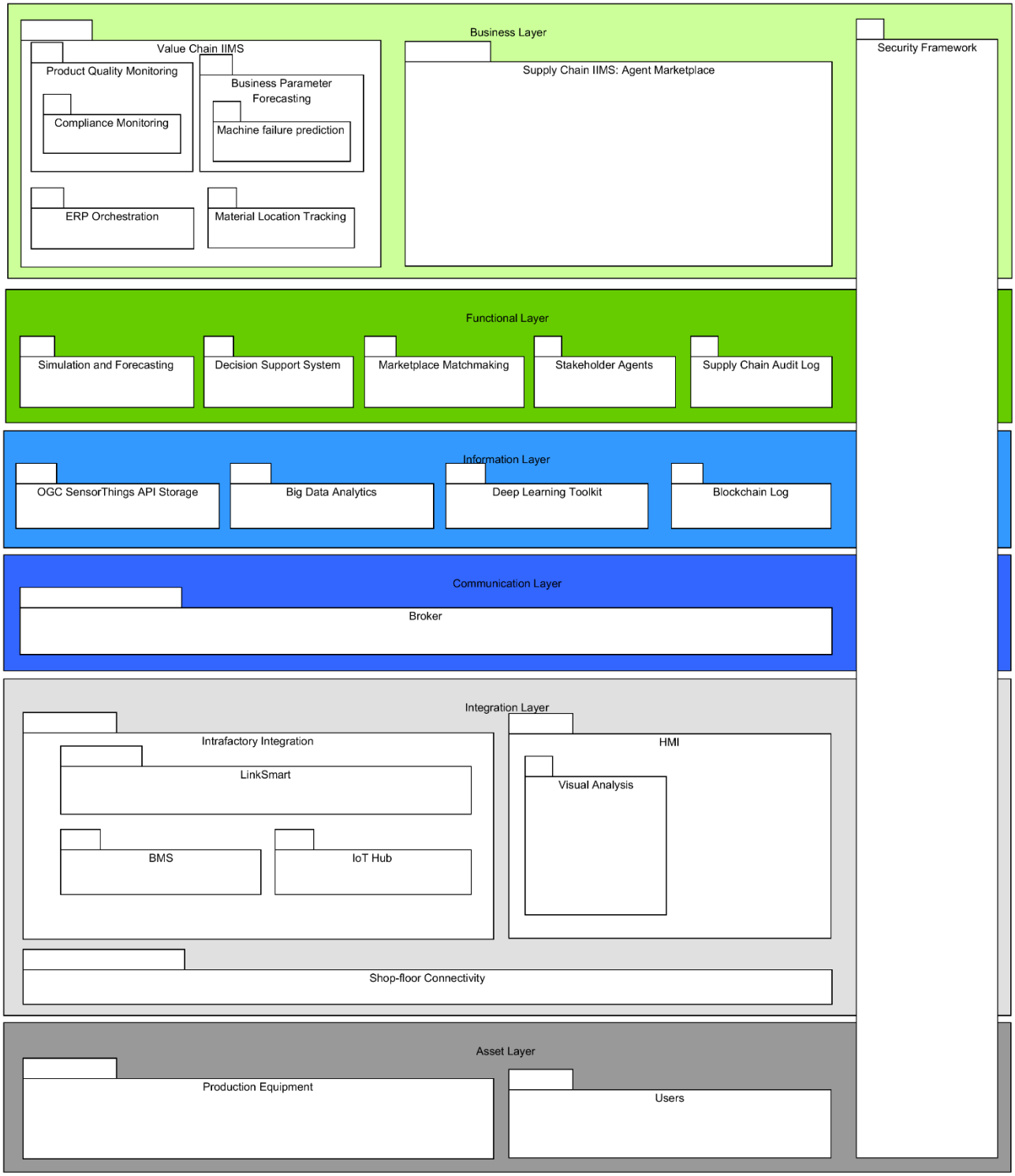

Figure 8: A mapping of COMPOSITION functional packages to the RAMI 4.0 Layers

The COMPOSITION system scope and pilots cover the intra-factory functionality from "Field Device" to "Work Center" via the IIMS and has a special emphasis on the inter-factory ecosystem of the "Connected World", provided by the interoperable agent-based marketplace and the blockchain-based log-oriented architecture, providing secure and trusted exchange of supply chain data between independent parties.

Life cycles of both types and instances of products and machines is covered by COMPOSITION, where complex pattern detection, deep learning networks and simulation capabilities may be used both for operational management and continuous improvement of factory equipment and products.

The administrative shell can be implemented at various levels in the COMPOSITION system. The BMS (or other possible implementation mechanisms of the Adaptation Layer of the Intrafactory Interoperability Layer) create administrative shells for the connected assets (see section 5.3.3.2). More complex administrative shells for production lines are implemented inside the IIMS using other components such as the Big Data Analytics, Decision Support System or Simulation and Forecasting Tool. The I4.0 components will be layered on top of each other and more than one administrative shell may exist for the same asset or combination of assets.

The project has implemented an adapter for OPC-UA, the recommended communication standard in RAMI4.0. MQTT and AMQP, which are defined transports in the OPC-UA Pub/Sub Architecture. The Administration Shells for these assets are realized in the Intra-factory Integration Layer.

5.3.2 Market Event Broker and Real-time Multi-Protocol Event Broker

The role of the Market Event Broker is to manage message-based communication in the agent-based marketplace. The Real-time Multi-Protocol Event Broker manages the streams of factory data in the intra-factory integration layer and loosely coupled communication between components in the intra-factory system.

Primary concerns when designing both components were security, scalability and extensibility. Multiple protocols and formats should be supported. The use of open standards, ease of integration of the chosen implementation and compatibility with software brought into the project was desired.

The Market Event Broker and Real-time Multi-Protocol Event broker have been merged in one component referred to as the Message Broker. This can fulfil both roles, using different configurations.

The Message Broker is described in further detail in D6.1 “Real-time Event Broker I”. Scalability design for the inter-factory system is described in D6.3 “COMPOSITION Marketplace I”.

The AMQP protocol will be used for component communication and message routing in the inter-factory system. It is a very flexible protocol with high-level configurability for different message routing schemes and emulation of other protocols. The more lightweight MQTT protocol will be used for the components in the intra-factory IIMS. Most COMPOSITION components already implement support for MQTT. MQTT may be transparently used by clients on top of an AMQP broker architecture.

The COMPOSITION project selected RabbitMQ[12] as the implementation mechanism for the message broker. This was suggested in the inception phase documented in the project description (COMPOSITION, 2016). RabbitMQ is a widely used open source message broker[13] with an extensible architecture. It implements the AMQP 0-9-1 protocol[14] and can through extension mechanisms, plugins, support the most common messaging protocols, e.g. MQTT, STOMP and XMPP. Extensions and adapters can be written to support other messaging patterns, protocols and security management solutions.

RabbitMQ implements AMQP 0-9-1 and the AMQP concepts of brokers, messages, producers, exchanges, queues and consumers. A publisher – an application that produces messages - sends a message to an exchange, where it is routed to one or more queues. The message is then pushed to (or pulled by) a consumer – an application that processes messages - for processing. Exchanges and brokers may reside on different brokers. The topology of the message routing is controlled by the publisher and consumer, which allows for very flexible communication design. Exchanges and brokers are access-controlled, which allows for fine-grain security control over the communication.

To provide an integrated security solution for COMPOSITION, an adapter has been developed to allow the authentication and authorization mechanisms of RabbitMQ to be managed by Keycloak, the RabbitMQ Authentication/Authorization Service (RAAS). The same security system can thus be used for intra-factory business user identity, marketplace partners and system components. An adapter for the blockchain distributed trust mechanism is being built to allow the integrity and non-repudiation of broker messages.

When the broker is used for inter-component communication, logical addressing of components can be used – a component identifier instead of a network address and port – decoupling components and providing a consistent way to address and find them for other components. As mentioned above, authentication and authorization can also be managed in a uniform manner via the broker. As extensibility is a concern for the developer stakeholders, it is desirable to use the broker for all component communication. De-coupled, message-based communication suits the data-centric nature of the COMPOSITION system well, where

[12] https://www.rabbitmq.com/
[13] At the time of writing 35.000 production deployments , https://www.rabbitmq.com/
[14] http://www.amqp.org/sites/amqp.org/files/amqp0-9-1.zip

several components independently subscribe to the same information. However, some exchanges are more suited for request-response interaction, e.g. REST APIs used for querying or administration. An adapter for RabbitMQ has been developed provide transparent request-response messaging (tentatively named “RabbitHole”). A bit simplified, this routes HTTP requests through an HTTP Proxy, resolves the base URL to a queue where the HTTP request is put. Clients (the REST services) may subscribe to the requests directed at them and return the response without exposing any public HTTP ports. The RPC Executer handles the request-response transparently to the service. This implements request and response buffering, work queues, load balancing, logical addressing and the RAAS provides integrated security for all calls. Further work (outside the scope of COMPOSITION) will extend this to a general purpose microservice execution framework.

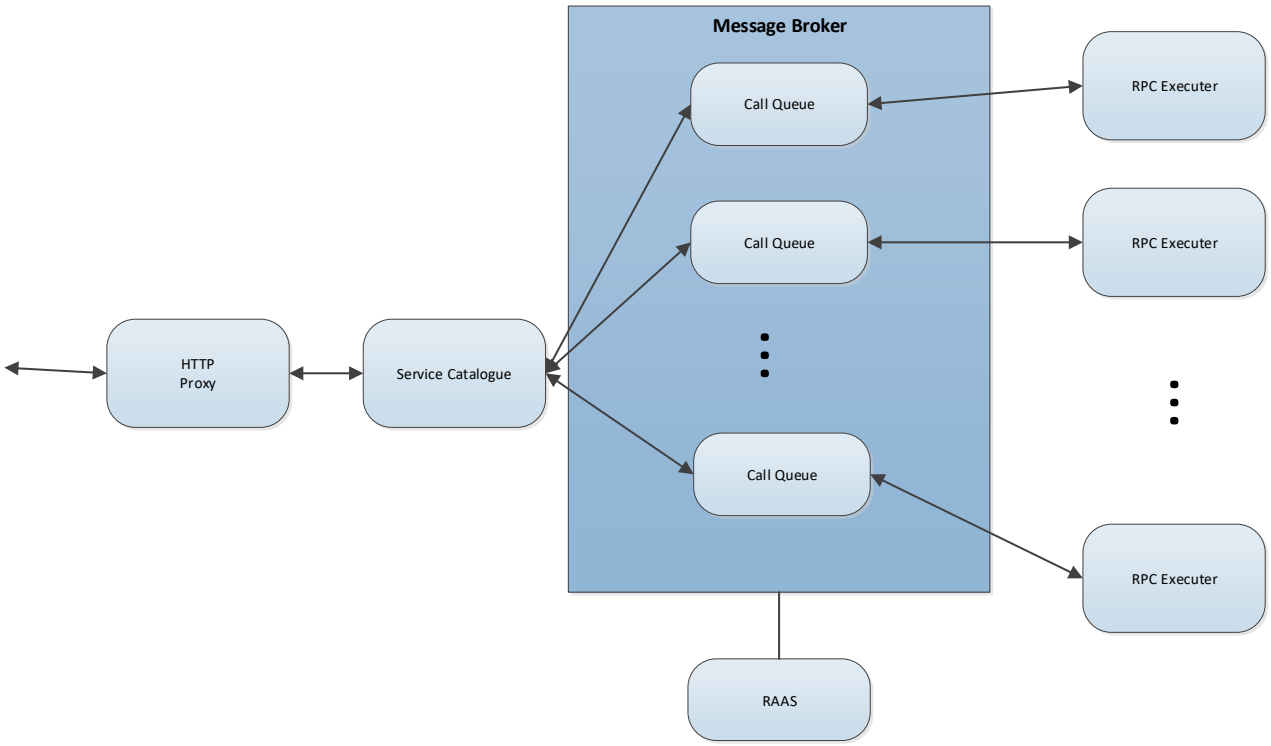

Figure 9: RPC over AMQP

5.3.3 Intra-factory Interoperability Layer

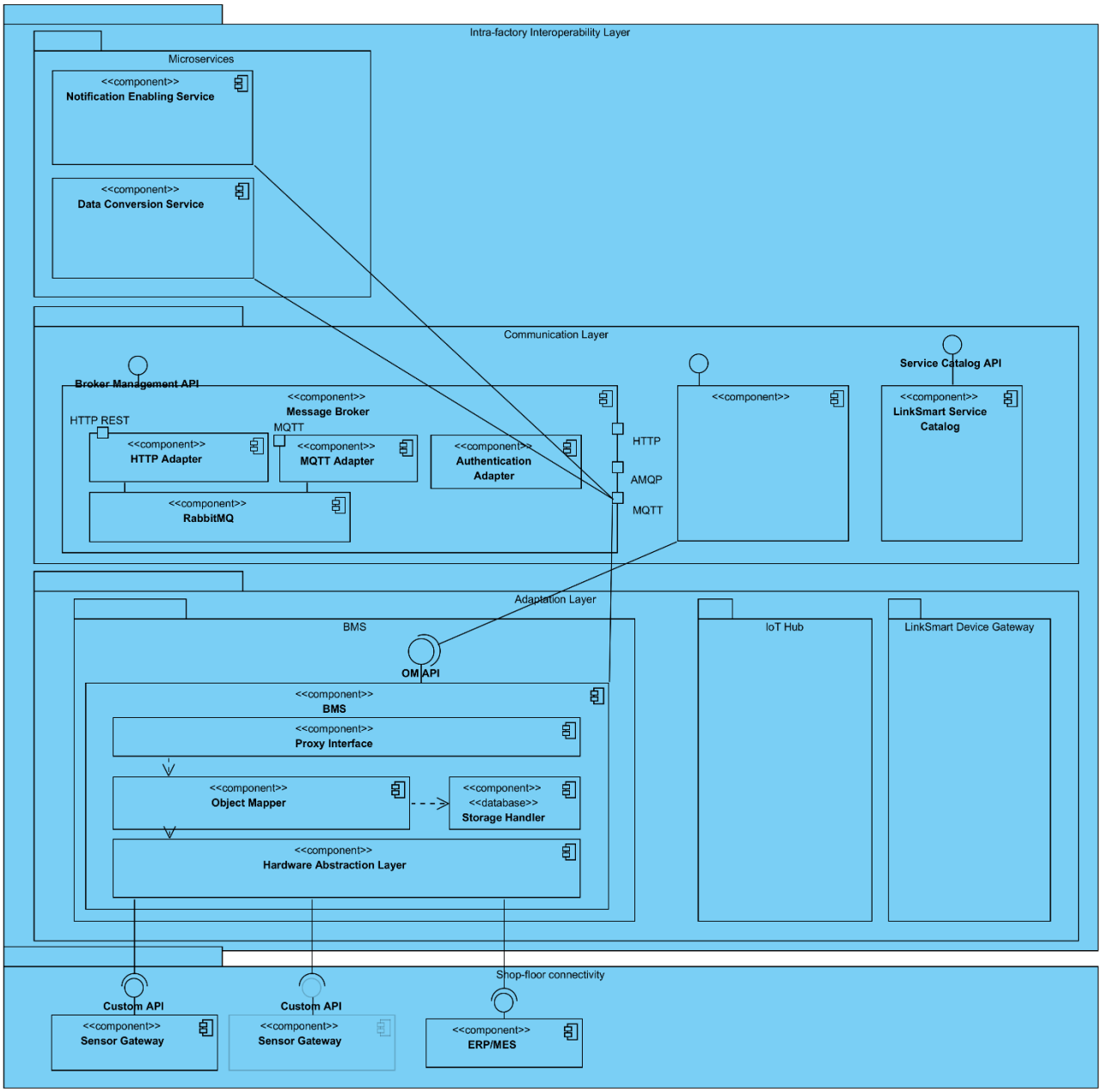

Figure 10: Intrafactory Interoperability Layer and Shop-floor

The Intra-factory Interoperability Layer (IIL) has two main goals, defined in the Description of Action (COMPOSITION, 2016). The first one is to provide a model for interconnecting the COMPOSITION ecosystem in the intra-factory scenario, providing integration and adaptation in the COMPOSITION IIMS of shop-floor data sources, i.e. sensors, control units (e.g. PLCs) and existing software systems (e.g. Manufacturing Execution System (MES). The second one is to ensure the conformity between communications among interconnected components. The involved technology has been provided by development partners of COMPOSITION, with extensions and additions of the connectors that will be defined, developed and deployed to integrate the assets used in the pilot installations.

A detailed description of the IIL can be found in deliverable D5.8 “Intrafactory Interoperability Layer I”.

The IIL spans two RAMI4.0 Layers: the Interoperability Layer and the Communication Layer. The RAMI4.0 Integration Layer performs digitization of assets; the mapping from the physical world to the digital and provides virtualization of shop-floor resources. The main component here is the Building Management System (BMS) which fulfils the requirements for COMPOSITION – other possible implementations, e.g. the

IoT Hub described in D2.3, are optional for the exploitation phase of the project. The RAMI4.0 Communication Layer provides standardized data formats, protocols and interfaces from the Integration Layer to the Information Layer, which processes and stores data and events. The Message Broker and connected micro services are responsible for this task. Interface endpoints for components are managed by the Service Catalog. The intra-factory communication system manages all internal communication in COMPOSITION.

The interfaces exposed by the IIL are the Service Catalog API, the BMS OM API, and the Message Broker MQTT publish-subscribe mechanism. It interacts with the heterogeneous factory CPS systems, other COMPOSITION components and the security framework. All data from the factory and data generated by COMPOSITION is published using JSON format according to the OGC SensorThings Data Model.

Sensors, Sensors Buffering and Sensors Gateways will be developed and adopted from existing technology. The BMS is provided by a project development stakeholder and is the translation layer providing shop floor connectivity from sensors to the COMPOSITION system. Raw data storage will be added for offline debug purposes. Consideration will be taken to Technical Objective 1.1.

- Individual partners' responsibilities and work package outputs are highlighted in the followings:
- The middleware is the main recipient in which the interoperability of single components acts.
- LinkSmart is a well-known middleware solution per se and will be customized to satisfy COMPOSITION requirements.
- Keycloak is a virtual layer that ensures authorization and authentication. Like all security related measures, it will be deployed by the Security Framework.
- The Big Data Analytics provides Complex Event Processing (CEP) capabilities for the data provided by the intra-factory integration layer.
- The Hidden Storage is a storage not accessible from the outside in which aggregated data are stored for debug purposes, i.e. re-bootstrapping already trained artificial neural networks belonging to the Deep Learning Toolkit and to the Dynamic Reasoning Engine.
- The Deep Learning toolkit component for this intra-factory scenario and as described in next section, it foresees a private connection with the Big Data Analytics that mediates all interconnection with the IIMS and all other components connected through the Intra-factory interoperability layer.
- The Visual Analytics component is the reporting interface of the Decision Support System and Simulation and Forecasting Toolkit.
- The Dynamic Reasoning Engine is part of the Simulation and Forecasting Toolkit.
- The Decision Support System uses process models to guide the production process.

Human Machine Interfaces are to be considered connected at the very end of these data streams exchanging components and serve the interaction with human beings whereas the automated processing happens underneath the surface. The aggregated data is also forwarded to the COMPOSITION Agents where it is used to support the agent decision making.

BMS, LinkSmart and the RabbitMQ message broker are mature and components that have been deployed in many other systems. Final deployment won't foresee any meaningful change in the architecture of the Intra-factory interoperability layer, in fact a well-established and reliable communication layer dwells its foundations in the homogenized components' pool that has been specified, developed and ready to be deployed.

5.3.3.1 **LinkSmart**

LinkSmart was originally developed within the Hydra co-founded EU project (The Hydra Project, 2018) for Networked Embedded Systems. It is an enabler allowing heterogeneous physical devices to be incorporated into their applications through easy-to-use web services for controlling any device. In spite of its inclusion in the Intra-factory scenario, a reduced set of LinkSmart functionalities have been required by components, so a stripped down version has been supposedly envisaged, leaving in the Inter-factory Interoperability Layer an agile tool for improving the versatility of the broker-based communication system infrastructure. The design iterations in the project will produce a LinkSmart configuration that suits the COMPOSITION ecosystem. Furthermore, the LinkSmart middleware has been mentioned above and while its inclusion in the intra-factory

layer is certain, there are a number of modifications made to it and its deep connection with the Big Data Analytics has created the IoT Learning Agent which is a key component of the IIMS.

- Components used include
    - Resource catalogue, works as resources index
    - Service catalogue, works as service index
    - Event Aggregator, parses messages to ensure well-formed and conformity in data streams

5.3.3.2 Building Management System

The Adaptation Layer is part of the Integrated Information Management System (IIMS) of COMPOSITION. The main purpose of this layer is allowing a seamless, homogeneous representation and interconnection among all the cyber-physical systems in the factory and the software modules in the upper layer (data processing, decision support, etc.). It has been designed considering the general principles set in the RAMI 4.0 specification, and is split into two logical sub-layers, highlighted in yellow and green in the picture below.

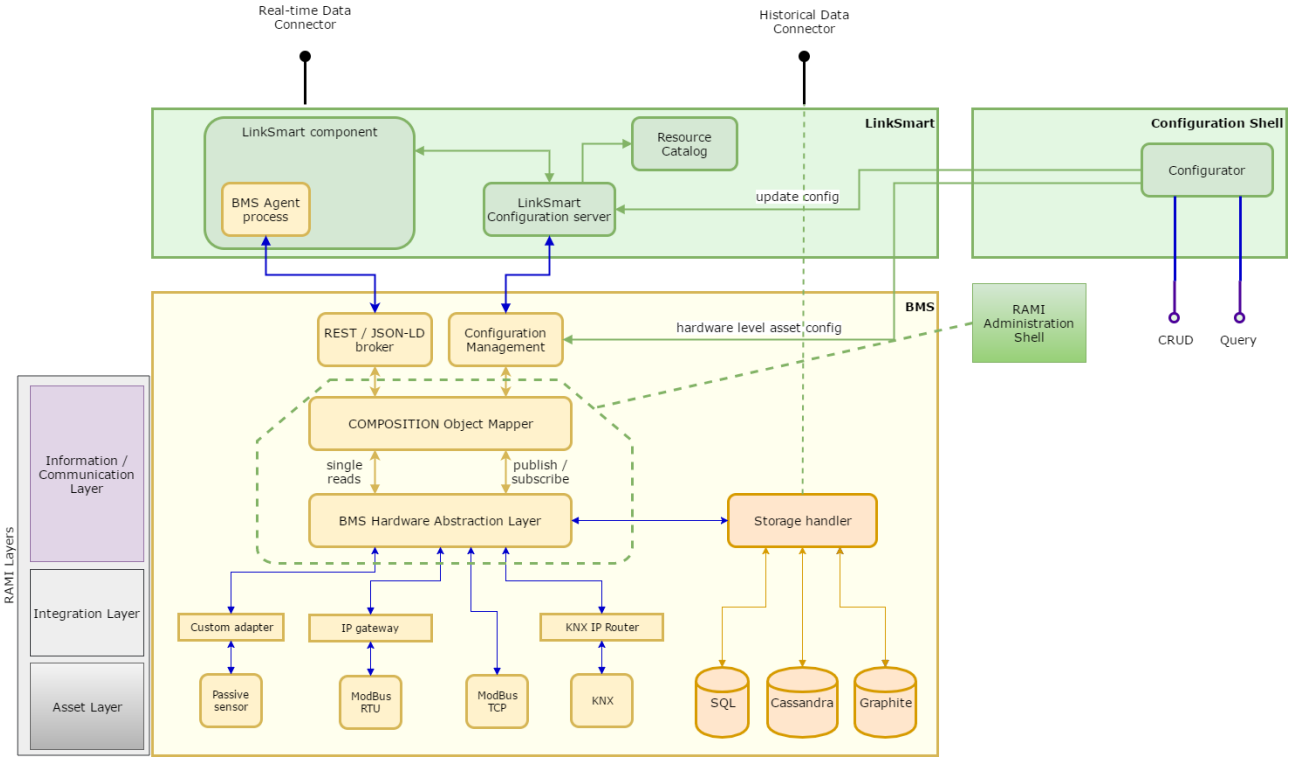

Figure 11: Components and interactions of the BMS: LinkSmart middleware, Configuration Shell, BMS (Building Management System), RAMI Administration Shell

The lower part (yellow in picture above) is built on top of the existing BMS software modules provided by NXW, which guarantee low level interoperability with a number of different field buses (this is positioned at the Asset / Integration RAMI layers). Such modules gather data read from the sensors installed in the local environment, interconnected through different field buses (e.g. KNX, Modbus, BACnet), and organize it into a uniform Data Model. This model provides a representation of sensor and actuator data which is independent of the physical type of underlying devices (Information/Communication RAMI layers).

It supports KNX, BACnet, Modbus/TCP and, Modbus/RTU as well as, several other proprietary control protocols. It can be interconnected with specific field buses either directly - such as via RS232/485 serial ports or GPIOs - or through the use of IP based gateways - such as KNX IP router and/or interface, Modbus/TCP gateways. It, can be extended by developing modules that can be dynamically plugged into its core. Regarding to COMPOSITION, the HAL component has been enhanced in order to support communications via MQTT, which is the protocol used by sensors that are going to be deployed in the project use cases (e.g. vibrometer sensor, fill level sensor, etc.).

In general, the HAL exposes a virtualized version of the underlying physical objects to the upper layers, from which information can be read and actuations can be performed. Moreover, in order to be flexible towards the configuration of the integrated devices, the component provides a user interface as well, that is the equivalent of an Administration Shell in the RAMI architecture.

The BMS HAL and COMPOSITION Object Mapper expose a virtualized version of the underlying physical objects from which information can be read and actuations can be performed, thus providing the equivalent of an Administration Shell in the RAMI architecture.

The upper part (green the picture above) is made of components belonging to the LinkSmart architecture and provides both real-time and historical data connectors for the other IIMS components. Communication between LinkSmart and the BMS components will be done through standard LinkSmart interfaces, implemented into the BMS Agent Process.

5.3.3.3 OPC Connector

OPC (OPC Foundation, 2018) is the most common standard used when interfacing factory equipment, such as PLCs and HMIs. In order to be able to integrate these data sources and address exploitability concerns, a connector has been developed within COMPOSITION. The responsible development stakeholder is a member of the OPC-UA Foundation, which allows use of OPC Foundation source code in commercial products and Distribution of OPC Foundation source code.

The OPC standards are governed and maintained by the OPC-Foundation[15]. The connector developed within COMPOSITION supports both the older OPC-DA standard, which is still very common, and the newer OPC-UA (OPC Universal Access) standard.

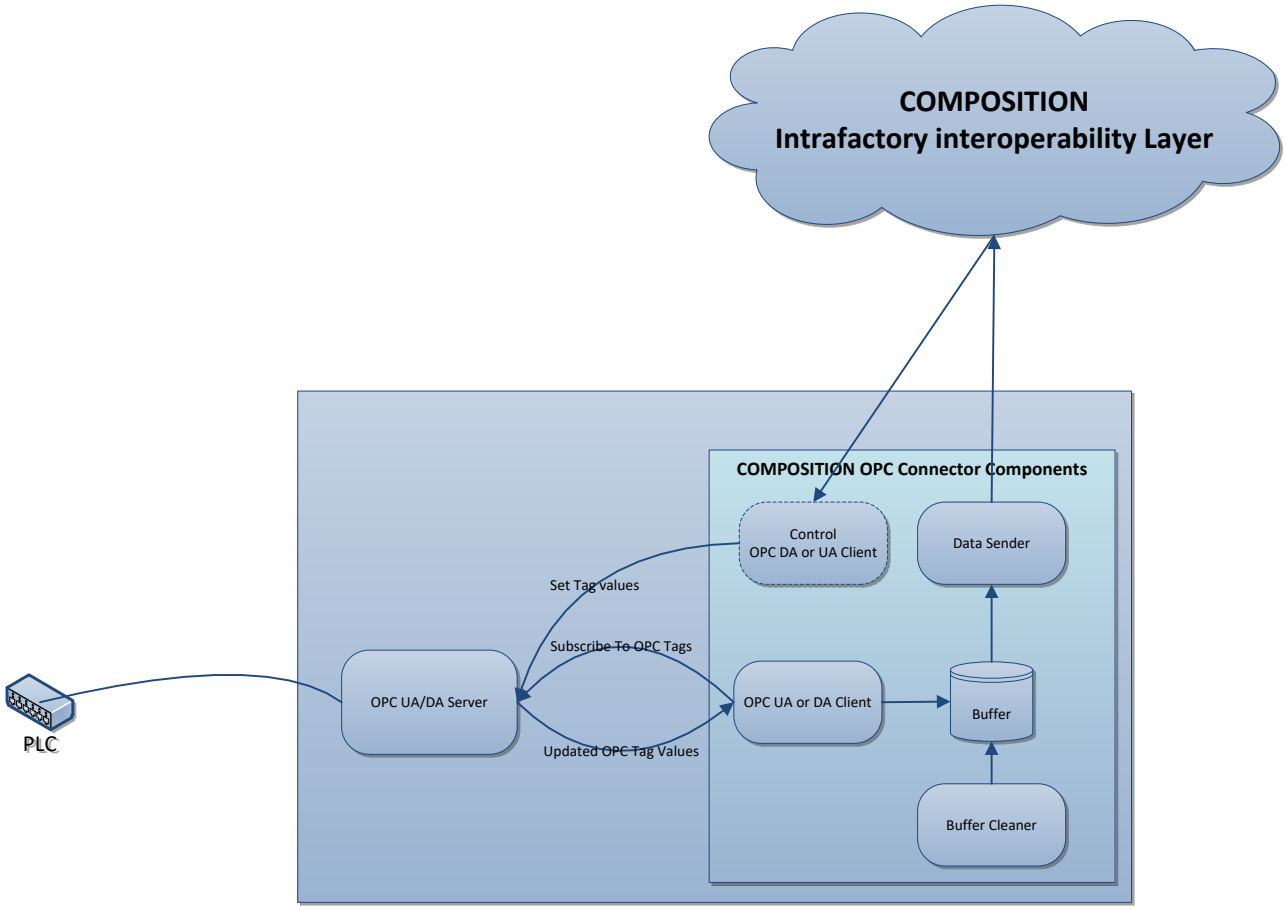

Figure 12: COMPOSITION OPC Connector

In Figure 12, the components of the COMPOSITION are shown:

- OPC UA or DA Client: Communicates with OPC server.

[15] OPC-Foundation https://opcfoundation.org/

- Buffer: Buffers the values before sending them to the Intra-factory interoperability layer.
- Data Sender: Reads the Buffer and send the acquired data to the Intra-factory interoperability layer.
- Buffer Cleaner: Cleans the data buffer, removing all transmitted data.
- Control OPC-DA or UA Client: Used to set tags in the OPC server, i.e. doing control. This component is not yet developed.

Because the very different nature of the OPC-DA and OPC-UA Client these will be described in more detail in the following sub chapters. The other components are developed in .net core and can be deployed in Windows, Docker Containers and Linux (Including Raspbian).

5.3.3.3.1 OPC-DA Client

OPC-DA is an older standard that is built on top OLE in Windows and it requires Windows to be able to run. The OPC-DA standard includes a simple "tag" model where it is possible to list the available tags. The metadata information available for the tags is quite simplistic. Basically, it contains the name, datatype, update frequency and finally if it is a read only or writable tag. OPC-DA includes a messaging model where the clients can subscribe to tag value changes.

The OPC-DA Client in COMPOSITION is built on top of the reference OPC-DA Client provided by the OPC-Foundation which should allow for good interoperability. The basic steps of the functionality in the OPC-DA Client is:

- List all the available tags in the OPC-DA server.
- Create subscriptions for the tags
- Listen to tag value changes and update the buffer database.

As mentioned before the OPC-DA client needs to run on a machine with Windows operating system. Typically, it is installed in the HMI or SCADA PC.

5.3.3.3.2 OPC-UA Client

OPC-UA is a more modern standard that is open source and can run on multiple platforms. The standard includes a certificate-based security model. The OPC-UA standard has a more advanced data model where items can belong to different namespaces and can describe methods, variables and objects. For instance, it is possible to describe that variables belong to a specific machine that has a location. OPC-UA includes a messaging model where the clients can subscribe to item value changes.

The OPC-UA Client in COMPOSITION is built on top of the reference OPC-UA Client provided by the OPC-Foundation which should allow for good interoperability. The basic steps of the functionality in the OPC-UA Client is:

- List all the available items that are of type variable in the OPC-UA server.
    - The OPC-UA Client can filter items using namespaces or objects to only include information regarding certain equipment.
- Create subscriptions for the tags
- Listen to tag value changes and update the buffer database.

The OPC-UA client can run in all environments supported by .net core which means that most Linux based environments such as Raspbian on Raspberry PI can be used for deployment of the OPC-UA client.

5.3.4 HMI Framework

The human machine interfaces of COMPOSITION are comprised by front ends to different monitoring, analytics, and management backends. These are developed by the specialist teams in the respective area as self-sustained, vertically integrated components. However, these are integrated into a coherent user interface with a common look-and-feel as "micro frontends"[16][17], a design analogous to the well-known concept of micro services.

[16] https://www.thoughtworks.com/radar/techniques/micro-frontends
[17] https://micro-frontends.org/

The implementation mechanism used in COMPOSITION to realize this is Web Components, a set of features that allow for extending HTML with reusable custom elements with encapsulated styling and custom behaviour. These features are under review by W3C[18] to be added to the HTML and DOM specifications. The Web components de-facto standards are based on existing web standards and consist of four specifications:

- Custom Elements – The Custom Elements specification defines the APIs for designing and using new types of custom DOM elements.
- Shadow DOM – The Shadow DOM specification defines encapsulated style and HTML markup that can be rendered by the browser without being included in the main document DOM tree
- HTML Imports – The HTML Imports specification defines how to include and reuse HTML documents in other HTML documents.
- HTML Templates - The HTML template element specification defines an HTML fragment which is not rendered when the page is loaded but stored until it is instantiated via JavaScript.

These work across the major browsers (Chrome, Opera, Safari, Firefox), with backward compatibility implemented using JavaScript libraries for browsers that do not support a specification (“polyfill”). Custom web components can be used with any JavaScript library or framework that works with HTML.

5.3.4.1 HMI Integrations

Web components are used for the common parts of the HMIs such as login and menu in order to give the user the impression of one single application when in fact it is multiple applications developed by different partners using different frameworks. These components are created separately from the marketplace applications with their own style and functionality. Once created they need to be implemented by each application.

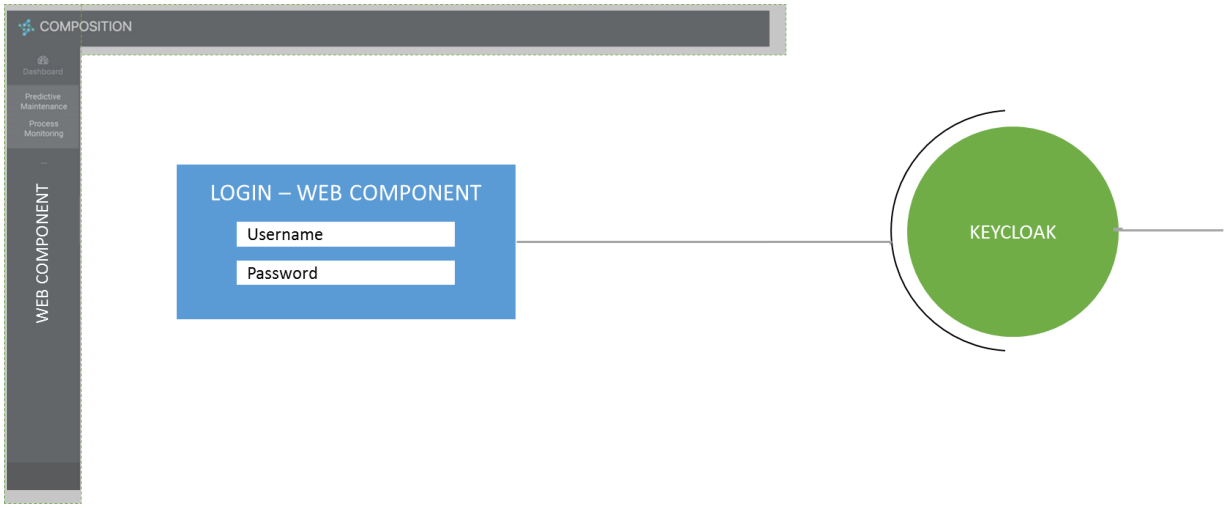

Figure 13: Common HMI Components

The navigation and login will be shared, configurable components. The menu can be configured through a RESTful API with a persistent backend, where each application adds the menus and submenus related to that application. The menu will receive information about the signed in user, since the menus look different depending on the user role.

Custom components can communicate in a loosely coupled fashion via DOM events and attribute updates on custom components through Javascript. Security mechanisms and style information is shared, like the common menu.

5.3.5 Big Data Analytics

Manufacturing in assembly lines consist in a set of hundreds, thousands or millions of small discrete steps aligned in a production process. Automatized production processes or production lines, they produce for each of those steps small bits of data in form of events. The events possess valuable information, but this information

[18] https://www.w3.org/standards/techs/components#w3c_all

loses the value through time. Additionally, the data in the events usually are meaningless if they are not contextualized, either by other events, sensor data or process context. To extract most value of the data, it must be process as it's produced. In other words, in real-time and on demand. Therefore, we prose for the Big Data Analysis; the usage of Complex-Event Processing for the data management coming from the production facilities. In this manner, the data is processed at the moment when it is produced extracting the maximum value, reducing latency, providing reactivity, giving it context, and avoiding the need of archiving unnecessary data.

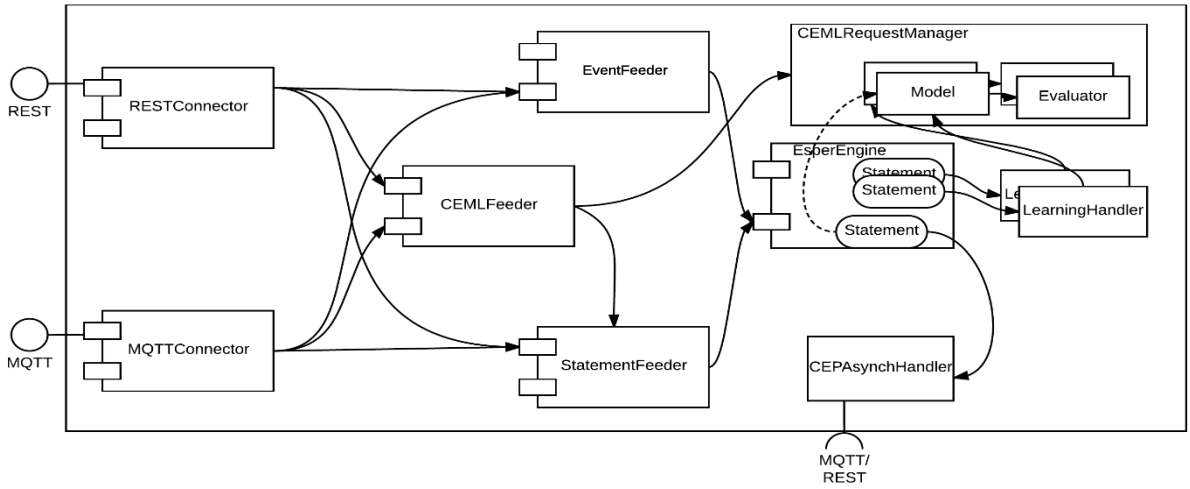

Figure 14: LinkSmart® Learning Service Architecture Sketch

The Complex-Event Processing service is provided by the LinkSmart® Learning Service (LS). The LS is a Stream Mining service that provide means to manage real-time data for several propose. In the first place, the LS provide a set of tools for collect, annotate, filter, aggregate, or cache the real-time data incoming from the production facilities. This set of tools facilitate the possibility to build applications on top of real-time data. Secondly, the LS provide a set of APIs to manga the real-time data lifecycle for continuous learning. Thirdly, the LS can process the live data to provide complex analysis creating real-time results for alerting or informing about important conditions in the factory, that may be not be seeing at first glance. Finally, the LS allows the possibility to adapt to the productions needs during the production process.

Below, we discuss the most relevant developments related to the Big Data Analytics. For more detail, please see deliverable D5.1 "Big data mining and analytics tools I".

The Learning Agent (component in implementing the Big Data Analytics) started development in 2014 in the ALMANAC project as a simple CEP for Smart Cities and was presented in (Bonino, et al., 2015). Since then, the LA has been developed and transformed in a self-managed learning orchestrator service that combined Complex-Event Processing and Machine Learning and other techniques. Specifically, in COMPOSITION there has been the following improvements:

- Python interoperability layer for programmers or Python SDK
- Micro-batch learning handling for non-iterative learning models
- Implementation and testing of a default detection model for SMTs using the Python SDK and Random Forest model.
- Implementation of the JWS standard for the I/O API.
- Full Dockarized distribution
- Introduction of CI for quality assurance using automatic testing. This includes
    - Development of Docker based Integration Test for Statement API
    - Development of Docker based Integration Test for CEML API

- Other smaller improvements and fixes had been done. For more detailed information please check the LinkSmart® project documentation[19] and source[20] code release notes.

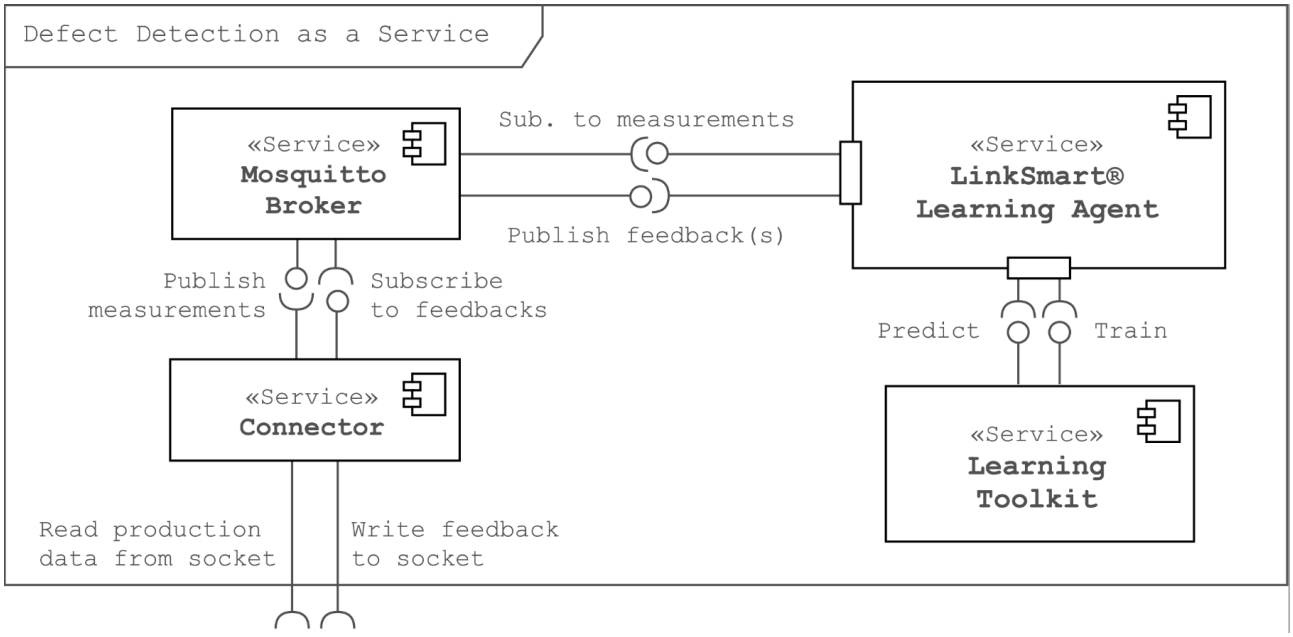

Figure 15: DLT and LA integration

Additionally, an important development is the integration between DLT and the LA. The DLT chapter, the integration of the DLT using the Python Pyro adapter will be explain by introduction the different parts of the adapter. The adapter will implement the different phases of the CEML (explain below).

5.3.5.1 The Complex-Event Machine Learning methodology

The Complex-Event Machine Learning (CEML) (Carvajal Soto, Jentsch, Preuveneers, & Ilie-Zudor, 2016) is a framework that combines Complex-Event Processing (CEP) (Cugola & Margara, 2012) and Machine Learning (ML) (Andrieu, De Freitas, Doucet, & Jordan, 2003) applied to the IoT. This means that the framework was developed to be deployed everywhere, from the edge of the network to the cloud. Furthermore, the framework can manage itself and works autonomously. The following section briefly describes the different aspects that CEML covers. The framework must automate the learning process and the deployment management. This process can be broken down in different phases: (1) the data must be collected from different sensors, either from the same device or in a local network. (2) The data must be pre-processed for attribute extraction. (3) The learning process takes place. (4) The learning must be evaluated. (5) When the evaluation shows that the model is ready, the deployment must take place. Finally, all these phases happen continuously and repetitively, while the environment constantly changes. Therefore, the model and the deployment must adapt as well.

5.3.5.2 Data Propagation Phase

Data in the IoT is produced in several places, protocols, formats, and devices. Although this deliverable does not address the problem of data heterogeneity in detail, the learning agents require a mechanism to acquire and manage the heterogeneity of the data. The mechanism must be scalable and, at the same time, the protocol should handle the asynchronous nature of IoT. Finally, the protocol must provide tools to handle the pub/sub characteristics of the CEP engines. Therefore, we have chosen MQTT[21], a well-established Client Server publish/subscribe messaging transport protocol. The topic based message protocol provides a mechanism to manage the data heterogeneity by making a relation between topics and payloads. It allows deployments in several architectures, OS, and hardware platforms; basic constraints at the edge of the

[19] https://docs.linksmart.eu/display/LA
[20] https://code.linksmart.eu/projects/LA/
[21] *MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. Source http://mqtt.org/*

network. The protocol is payload agnostic, and as such allows for maximum flexibility to support several types of payloads.

5.3.5.3 Data Pre-Processing (Munging) Phase

Usually ML is tied to stored datasets, which incurs several drawbacks. Firstly, the learning can take place only with persistent data. Secondly, usually the models generated are based on historical data, not current data. Both constrains, in the IoT, have dire consequences. It is neither feasible nor profitable to store all data. Also, embedded devices do not have much storage capacity which makes it impossible to use ML algorithms on them. Furthermore, IoT deployments are commonly exposed to ever-changing environments.

Using historical data for off-line learning could cause outdated models learning old patterns rather than current ones, producing drifted models. Although some IoT platforms like COMPOSITION support storage of historical data, it may be too time and space consuming to create large enough times series. Therefore, there is also a need for non-persistence manipulation tools. This is precisely what the CEP engine provides in the CEML framework. This means, the CEP engine decides which and how the data is manipulated using predefined CEP statements deployed in the engine. Each statement can be seen as a topic, to which each learning model is subscribed. Any update of the subscribers provides a sample to be learnt in the learning phase.

5.3.5.4 Learning Phase

There is no pre-selection of algorithms in the framework. They are selected by the restrictions imposed by the problem domain. For example, in extreme constrained devices, algorithms such as Algorithm Output Granularity (AOG) (Gaber, Advanced Methods for Knowledge Discovery from Complex Data) may be the right choice. In other cases where the model changes quickly, one-shot algorithms may be the best fit. Artificial Neural Networks are good for complex problems but only with stable phenomena. This means that the algorithm selection should be made case-by-case. Our framework provides mechanisms for the management and deployment of the learning models, and the process of how the model is fed with samples. In general, the process is based on incremental learning (Syed, Huan, Kah, & Sung, 1999) albeit with online and non-persistent data. The process can be summarized as follows: the samples, without the target provided in the last phase, are used to generate a prediction. The prediction will then be sent to the next phase. Thereafter, the sample is applied to update the model. Thus, all updates are used for the learning process.

5.3.5.5 Continuous Validation Phase

This section describes how the validation of the learning models is done inside the CEML. This phase does not influence the learning process nor validate the CEML framework itself.

ML model validation is a challenging topic in real-time environments and the evaluation for distributed environments or embedded devices is not addressed extensively in the literature, which is why we think it needs further research. There are two addressed strategies. Either we holdout an evaluation dataset by taking a control subset for given time-frame (time window), or we use *Predictive Sequential*, also known as *Prequential* (Dawid, 1984), in which we asses each sequential prediction against the observation. The following section describes the continuous validation we applied for a **classification** problem, even though it can be applied for other cases as well.

Instead of accumulating a sample for validation, we analyse the predictions made before the learning takes place. All predictions are assessed each time an update arrives. The assessment is an entry for the confusion matrix (Stehman, 1997) which is accumulated in an *accumulated confusion matrix*. The matrix contains the accumulation of all assessed predictions done before. In other words, the matrix does not describe the current validation state of the model, but instead the trajectory of it. Using this matrix, the accumulated validation metrics (e.g. Accuracy, Precision, Sensitivity, etc.) are being calculated. This methodology does have some drawbacks and advantages, explained more extensively in (Carvajal Soto, Jentsch, Preuveneers, & Ilie-Zudor, 2016).

5.3.5.6 Deployment Phase

The continuous validation opens the possibility for making an assessment of the status of the model each time a new update arrives, e.g. if it is accrued or not. Using this information, the CEML framework has the capability to decide if the model should or should not be deployed into the system at any time. If the model is behaving well, then it should be deployed, otherwise it should be removed from the deployment. The decision is made by user-provided thresholds with regards to evaluation metrics. If a threshold is reached, the CEML inserts the

model into the CEP engine and starts processing the streams using the model. Otherwise, if the model does not reach the threshold then its remove form the CEP engine.

5.3.6 Deep Learning Toolkit

5.3.6.1 Role and responsibilities in the COMPOSITION Acrhitecture

The Deep Learning Toolkit delivers predictions and forecasting of relevant indicators based on machine learning models. It is a component of the COMPOSITION ecosystem and belongs to both the intra and inter-factory scenarios.

In the former it is in charge of analysing the shop floor parameters, feed to the component by the IIMS and more specifically by the BMS (section 5.3.3.2) through the middleware. So, despite not being directly connected to the broker-based messaging system, it heavily depends on data transported by all COMPOSITION components attached to it, since it belongs and rely on the same Intra-factory Interoperability Layer. The main difference with the other components dwells in the theoretical fact that it is mediated by the Big Data Analytics tool for all the activities. In fact, the Deep Learning Toolkit component has a private 1:1 connection with the Learning Agent framework implemented by the Big Data Analytics through a specific architecture that foresees the usage of Remote Procedures Calls between the two components within the Intra-factory scenario. The technology used for this communication is called Pyro (Pyro - Python Remote Objects, 2018), and allows message translation and interoperability among different languages as well as seamless communication overlay physical dislocated processes among different Docker containers.

In the latter, it has a 1:1 mapped connection to the Agent-based marketplace, in specific one deployed agent corresponds to one deployed instance of the Deep Learning Toolkit component. In this scenario, the Agent is provided with the intelligence required for making future assumption on specific market behaviours.

The twofold nature of the Deep Learning toolkit serves both the Intra and Inter scenarios. In the Intra-factory scenario, the use case UC-BSL-2 is addressed and the Deep Learning Toolkit is deployed to operate as predictive maintenance intelligent tool. In the Inter-factory scenario, the Deep Learning Toolkit works as a REST service, providing intelligence to the correspondent Agent, and provides market estimations.

Both implementations foresee a common pattern regarding the internal architecture. In specific, a continuous learning process can be broken down to a predetermined number of phases which constitute the core of the component itself, and its information lifecycle can be envisaged as follows:

- offline training phase
- validation phase
- testing phase
- continuous learning phase

The offline training phase, as it's named after, starts with an offline analysis of the historical data and takes place outside the shop-floor. It is the longest by far of the four phases and it embeds sub-phases such as the data gathering, validation, preparation, filtering and formatting. Moreover, a humongous number of tests is required for optimally or sub-optimally shape the Artificial Neural Network and its hyper-parameters.

The validation phase takes also place offline and it's the phase in which the network parameters and hyper-parameters the of the Artificial Neural Network are adjusted in order to reach the threshold set for an acceptable accuracy level. This phase is also iterative and has the empowerment of rolling back to the previous phase. In fact, a not adequate result in this phase leads right back to square one.

The training phase is also consequent to the validation and it's the phase, where the component and the Artificial Neural Network has consumed all historical data. The result is a network that has finally embedded the most appropriate stochastic gradient descent and therefore a robust algorithm ready to be deployed and tested on the field.

The continuous learning phase is the longest of the four phases and takes place at the shop-floor level, at the end-user's premises, that has provided the shop floor data the Artificial Neural Network is base on. In this phase, the component is online and connected to the COMPOSITION ecosystem, where it learns from near real time data, accurately formatted and batched to resemble the training set, coming from target sensors at the shop-floor level.

5.3.6.2 Deep Learning Toolkit interactions

As it is an active part of the IIMS, it would be easy to suppose that the Continuous Deep Learning Toolkit module would receive both its raw and pre-processed input data from the component that acts as a middleware as the Adaptation Layer for Intra-factory Interoperability does. In spite of being this the first implementation choice, the final deployment has foreseen the complete integration with the Learning Agent framework implemented by the Big Data Analysis module. In fact, thanks to the Pyro integration performed in the second project year, the two components now rely on a 1:1 private connection among their Docker containers.

The Deep Learning Toolkit when operated in its continuous mode, foresees communications that are all mediated by the Learning Agent framework that is in charge of feeding the data in a pre-formatted manner, mimicking the training data and also is responsible for publishing and propagating the prediction results coming from the component.

The stack diagram in Figure 1Figure 16 are depicted the most relevant interactions between the two aforementioned modules, highlighting the difference between the first and the second iteration of the implementation phase.

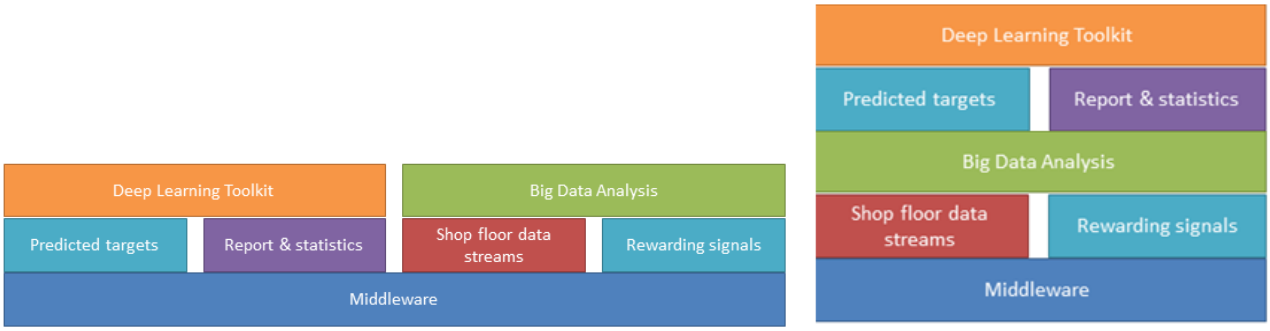

Figure 16: Deep Learning Toolkit in COMPOSITION architecture, before and after first implementation

5.3.6.3 Deep Learning Toolkit interfaces for Intra-factory

As for the Pyro interfaces used for communicating within the 1:1 mapped private connection among the Deep Learning Toolkit and the Learning Agent components, it has been clearly identified a superset of Remote Procedure Calls that will be beneficial for the correct functioning of both. In the following the superset is provided.

5.3.6.3.1 Method build: def build(self, classifier)

Builds the model. The classifier parameter is not required. Can throw exceptions. This method must be called before the learn, predict and exportModel methods. The model status is preserved across agent restarts. The destroy method is required to build a new model from scratch.

5.3.6.3.2 Method destroy: def destroy(self)

Destroys the current model and reset the internal status of the model. Can throw exceptions. The next call to build method will create a new model from scratch.

5.3.6.3.3 Method predict: def predict(self, datapoint)

Generates output predictions for the input datapoint. Must be called after the build method. Can throw exceptions. The datapoint argument contains the time series required for the prediction. It is a numpy array with shape (n_samples, n_features). The n_samples is dependent by the DLT model; currently this value is 32 but can updated with the model refinements.

5.3.6.3.4 Method batchPredict: def batchPredict(self, datapoints)

Generates output predictions for the input batch. Must be called after the build method. Can throw exceptions. The datapoints argument contains the data required for multiple predictions. It is a numpy array with shape (n_slices, n_samples, n_features). The n_slices can be any value greater than 0. The n_samples value is dependent by the DLT model; currently this value is 32 but can updated with the model refinements. Each

slice can be independent from the other. The prediction result is the list of fault probabilities as a float in the range 0 (no faults) to 1 (faults) for each slice of the batch.

5.3.6.3.5 Method learn: def learn(self, datapoint)

Trains the model with the provided datapoint. Must be called after the build method. Can throw exceptions. The datapoint argument contains the inputs data required to update the model. It is a numpy array with the shape (n_samples, n_features). The n_samples is dependent by the DLT type; at the moment, it must be greater than 64 to allow the creation of the batch from the time series. This value can change with the model refinements. This method doesn't return anything.

5.3.6.3.6 Method batchLearn: def batchLearn(self, datapoints)

Trains the model with the provided datapoints. Must be called after the build method. The datapoints argument contains the data required to update the model. It is a numpy array with shape (n_slices, n_samples, n_features). The n_slices can be any value greater than 0. The n_samples value is dependent by the DLT model; currently this value is 32 but can updated with the model refinements. This method doesn't return anything.

5.3.6.3.7 Method exportModel: def exportModel(self):

Serializes and returns the model as a json object. Must be called after the build method. Can throw exceptions.

5.3.6.3.8 Method importModel: def importModel(self, model)

Loads a serialized JSON model provided as parameter. Must be called before the build method. Can throw exceptions. Doesn't return anything.

At the current stage of this deliverable, it has not been decided if all interfaces will be implemented and if some parameters will vary, but any modification will be considered as minor from the architecture perspective. Furthermore, the sequence diagrams provided in the first iteration of this deliverable (D2.3) are to be considered still valid, but the only difference is that the main interlocutor is now the Learning Agent framework instead of the Intra-factory Interoperability Layer as implemented previously. For this reason, the sequence diagrams are not reported since they have been overcome by the interfaces defined above.

5.3.6.4 Deep Learning Toolkit interfaces for Inter-factory

The inter-factory scenario foresees the Deep Learning Toolkit as an instrument for providing intelligence to the Agent-based Marketplace. In specific, each agent will have a tailored version of the Deep Learning Toolkit trained with specific data and custom hyper parameters. Similarly, to what has been designed for the intra-factory scenario, also in this one the components will have a dedicated point to point connection. In fact, this implements the security-by-design paradigm that the COMPOSITION ecosystem evangelize.

In the followings the main REST interfaces that are exposed on the private network between the two Docker containers, are detailed:

Figure 17: REST service interfaces details

For a more detailed insight on the interfaces details, the full description of the interfaces compatible with JSON RFC 4627, is provided in section 9.

5.3.7 Decision Support System

5.3.7.1 Role and responsibilities in COMPOSITION architecture

The main purpose of the COMPOSITION DSS is to aid managers to the decision – making process on a manufacturing shop floor. It is mainly oriented to maintenance processes, but it can also be implemented on all manufacturing processes on shop floors.

DSS exploits historical data from CMMS (Computerised Maintenance Management System) and live data coming from sensor networks and uses it, for its rule engine. The rule engine is based on finite state machines algorithms, which include states, parameters and transitions for the rule. The rules provide suitable suggestion to many different situations on the shop floor. Rule engine also uses data and predictions that come from other COMPOSITION components and are fed to it.

All incoming data is also valuable for the KPIs tool of the DSS. The tool creates KPIs based on the data and visualises them in graphs such as time series, bars or pie charts. DSS also visualises live data from sensors and the predictions that come from other tools.

Finally, the system provides a personnel and task database, where the users are logged. Along with the embedded notifications mechanism, it provides a complete environment where visualisation, notification and decision – making processes are intertwined.

A detailed description of the DSS is provided in D3.8 “Manufacturing Decision Support System”.

5.3.7.2 Architectually significant design decisions

The COMPOSITION Decision Support System communicates with the rest of the COMPOSITION components using the establish MQTT topics to retrieve data.

It also integrates the security aspects developed for the COMPOSITION project. The received data is processed by the DSS Rule Engine and it should follow the data streaming process established for the project. Communication between DSS and components such as DFM and DLT are based on the streaming process. Knowledge extraction based on KPIs and decision – making process are the basic rationales of the DSS in the COMPOSITION project.

The knowledge is propagated to the DSS users with a build – in notifications mechanism that can send several kinds of notifications using WiFi or Internet connectivity.

Application dockerisation is one of the final steps in the design analysis of the component. Dockerisation is essential due the fact that all components should operate as a whole bundle and the operation should be seamless to the user.

5.3.7.3 Functional View

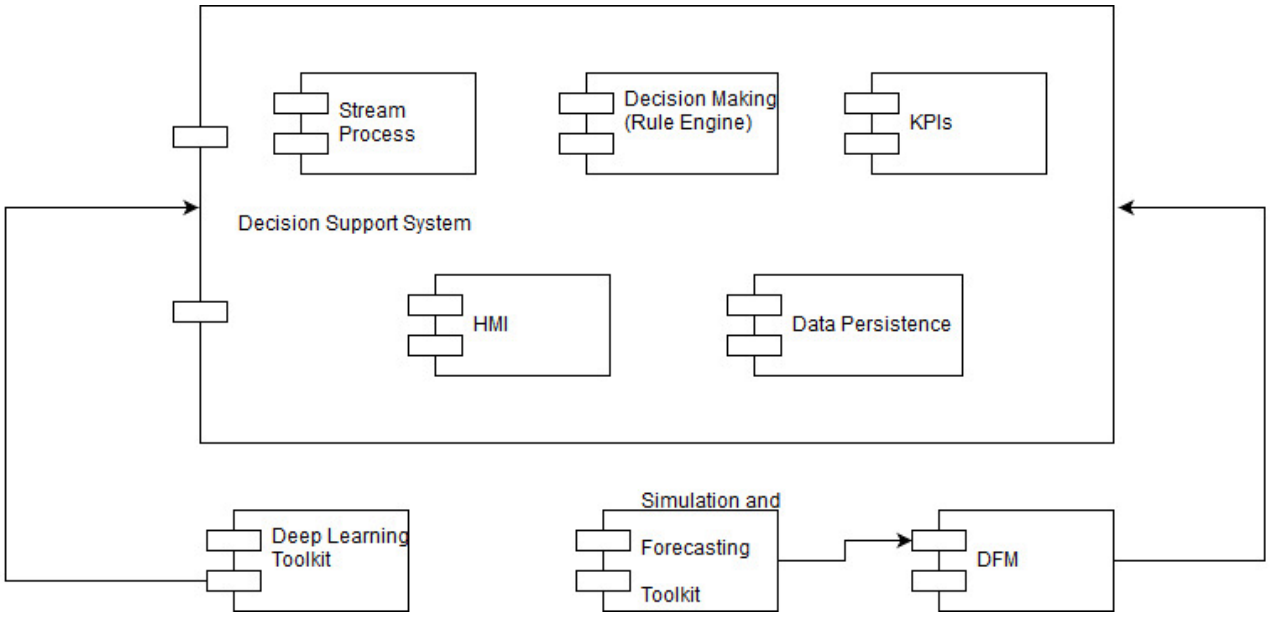

Figure 18: Component Diagram - Decision Support System

Figure 18 shows the component diagram of the DSS. As it is shown, DSS consists of five sub – components and communicates with other three COMPOSITION components, either directly or non – directly. The five sub – components are: Stream Processing, Decision Making (Rule Engine), KPIs, HMI and Data Persistence. These sub – components communicate with GET/POST requests. On the other hand, there are three components: Deep Learning Toolkit, Simulation and Forecasting Toolkit and DFM which communicate with the DSS with MQTT topic on the Message Broker tool.

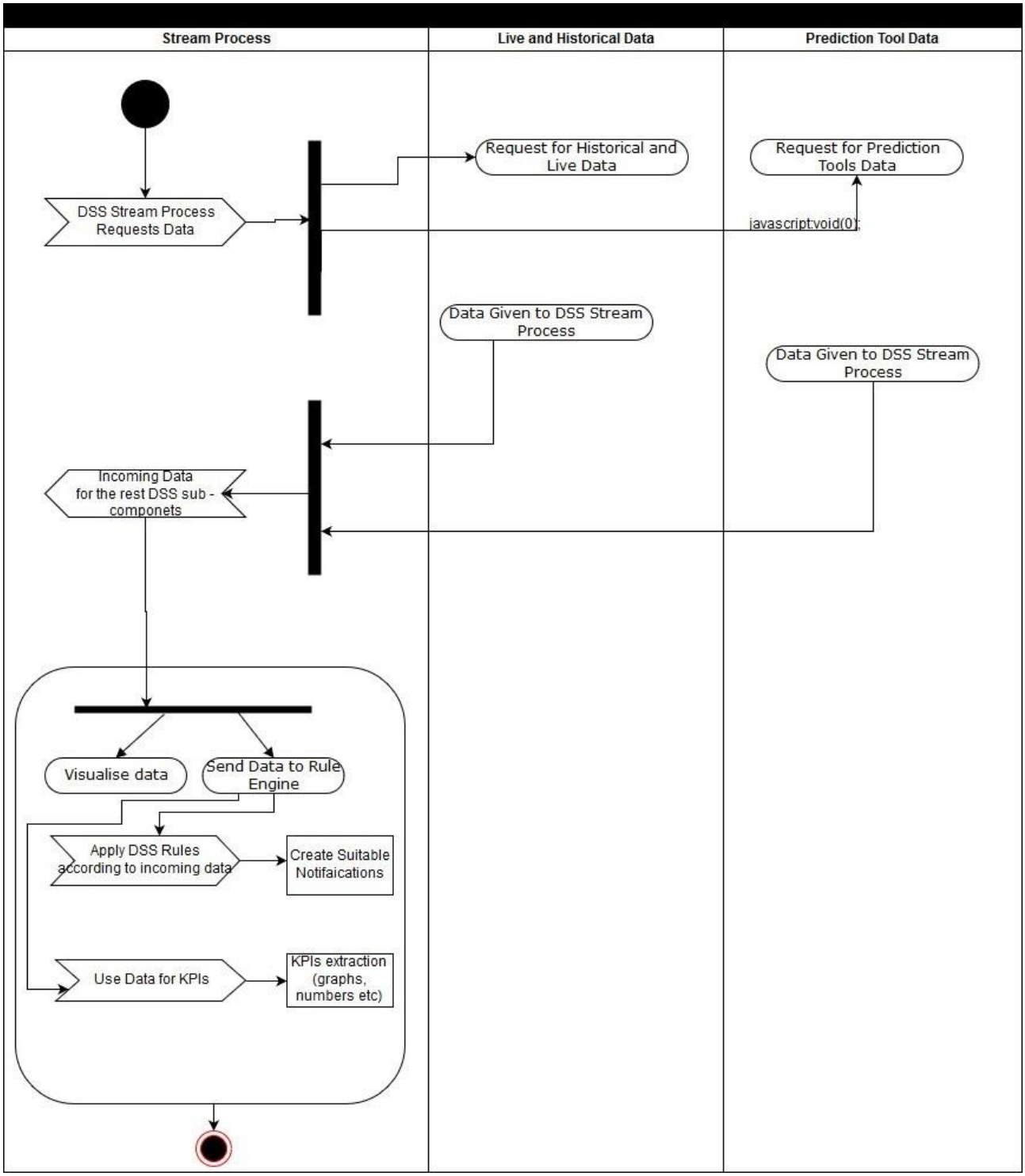

Figure 19: DSS Sequence Diagram

5.3.7.4 **DSS HMI**

The DSS HMI consists of the Log In screen, the main dashboard where visualisation elements exist, the KPIs tool and their visualisation, the Rule Engine HMI. The main design of the HMIs is based on the principled described below.

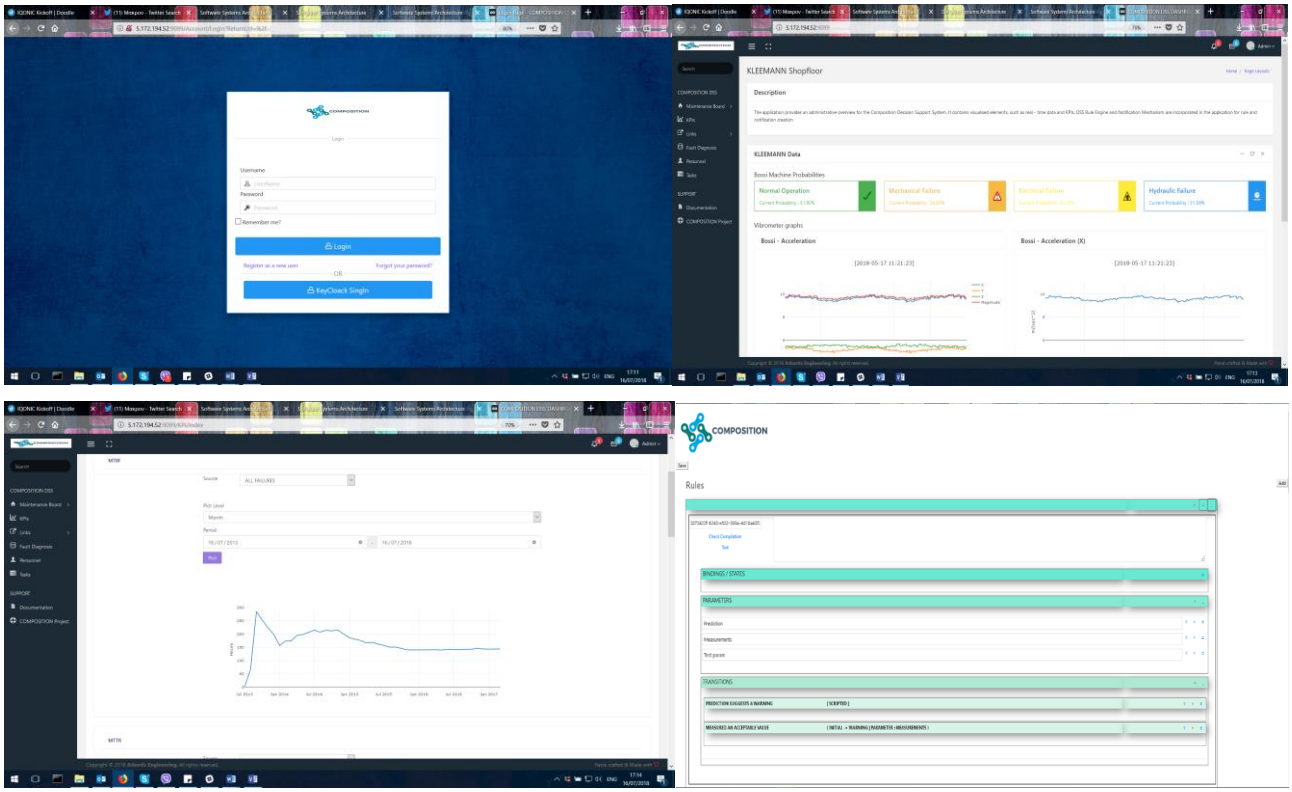

Figure 20: DSS HMI Screens: a) Log In Screen, b) Main Dashboard, c) KPIs tool and d) Rule Engine

Figure 20 shows the HMI screens of the DSS, where there is still working to be on the Rule Engine screen.

The DSS HMI has been updated and redesigned from the previous version. The whole functionality of the previous version remains in the new one, while new features are added. The new design of the DSS is based on the basic HMI designing principles:

- **Clarity:** the interface provides all the necessary information to use the component
- **Flexibility:** the component HMI is quite flexible and can be shown in different screens without problems, implementing Bootstrap standard
- **Familiarity:** HMI follows most of the known administrative HMI and its interface is already familiar to most of the users. No further user training is needed while using the system
- **Efficiency:** The HMI design allows users to follow their work in the rule engine or the KPIs tool and also have an overview of the whole component without feeling that the task requires them to dedicate much and be very detail – oriented.

The DSS HMI will be integrated as a micro-frontend in the HMI framework.

5.3.8 Simulation and Forecasting

5.3.8.1 Role and responsibilities in COMPOSITION architecture

The Simulation and Forecasting Tool (SFT) component is part of the high-level platform of COMPOSITION, the Integrated Information Management System (IIMS), and its main purpose is to simulate processes models and to provide forecast of events whose actuals outcomes have not yet been observed. This component will provide a constantly updated sensing layer regarding the integration of different sensors so as to support a Dynamic Reasoning Engine (DRE) and alarming services in production and logistics.

5.3.8.2 Functional view

The main inputs of Simulation and Forecasting engine component are real time data coming from installed sensors on industrial machines, historical machine data coming from COMPOSITION Database, historical

sensor data coming from Building Management System (BMS) and models of processes from Digital Factory Model (DFM). The schema is presented below:

Main input(s):
- Sensors
- Databases
- Building Management System (BMS)
- Digital Factory Model (DFM)

Main output(s):
- Digital Factory Model (DFM)
- Visual Analytics (VA)
- Decision Support System (DSS)

Main functionalities:
- Simulation of process or logistic models
- Forecast future outcomes based on models

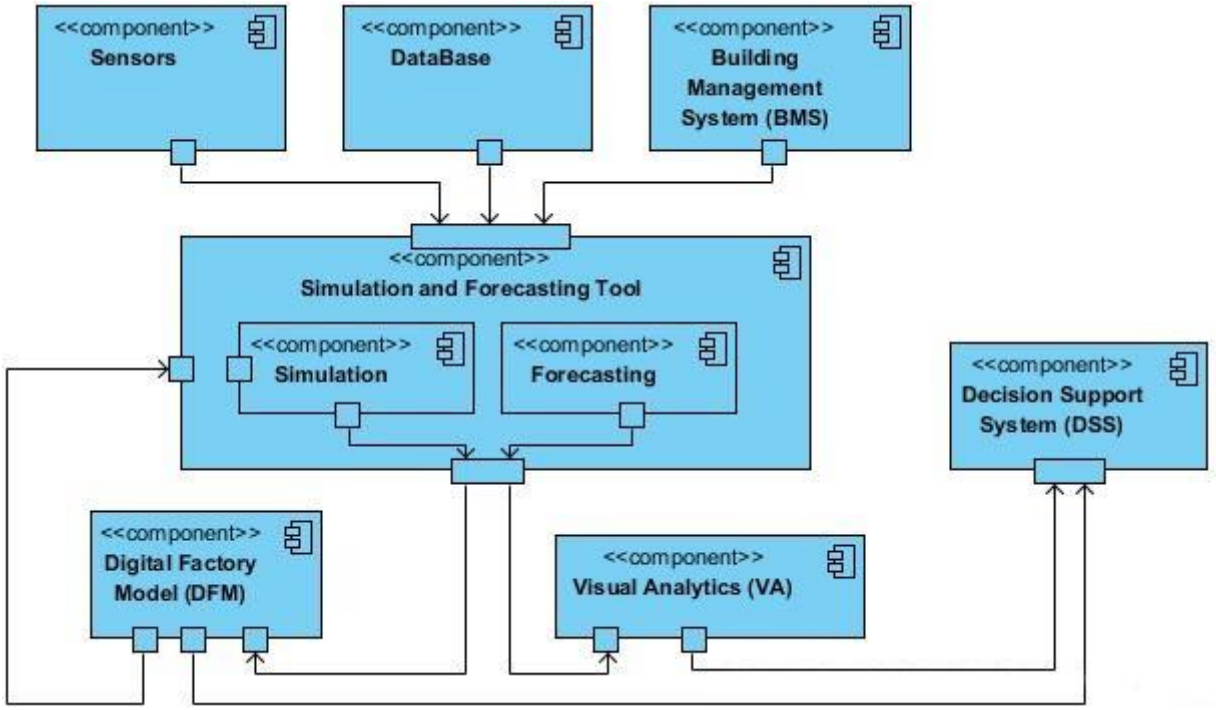

Figure 21: The updated Simulation and Forecasting Tool and dependencies

Simulation and Forecasting Tool component is divided in to sub-components: Simulation and Forecasting. Simulation sub-component will simulate models (provided by DFM component) on historical data (provided by COMPOSITION Database) or real-time data (provided by COMPOSITION sensors) so as to provide results on several process or logistic scenarios, according to projects' use case. The initial internal parameters of a simulation scenario will be defined be the user, accordingly. The simulation results will be fed into the COMPOSITION Digital Factory Model (DFM) so as to update models. Possible models that at first fit to a process and subsequently simulated could be several approaches of regression, such as linear, ridge, lasso, and elastic net regression. Forecasting sub-component will provide predictions of future events for the selected process model, based on the model parameters decided by the COMPOSITION Decision Support System (DSS) for the specific process when the iterations of simulation process end. The simulation and forecasting scenarios will be fed into COMPOSITION Visual Analytics (VA) component so as to present with the most compelling way with advanced and innovative data visualization techniques utilizing an interactive human-machine interface. The components of DFM and VA will be a main input to Decision Support System (DSS),

where by an internal procedure there will be a decision(s) for more scenarios to be simulated or not, regarding the tested process.

5.3.8.3 Visual Analytics Tool

The COMPOSITION Visual Analytics (VA) tool imports data from Simulation and forecasting tool and Big Data Analytics tools. As it is based on the data of the aforementioned tools, and especially from the SFT we choose to present it here as a sub-component of SFT. The VA offers an interactive user interface for the SFT algorithms and apply visual analytics techniques present the output to the users as graphical representations. The Visual Analytics tool will provide the ability to manufacturers/end-users to evaluate the simulation results and identify possible problems.

Based on the COMPOSITION architecture, the VA was designed as a completely web-based component. It is developed in AngularJS[22] and a template similar to FUSE[23] template that follows Google's material design specifications. Many different widgets and directives are offered from the VA tool. A wide variety of charts, pies, line charts, tables and time series representation is available in the Visual Analytics tool as the open source Chart.js [24]library is adopted.

The Visual Analytics Tool communicates with SFT using MQTT and REST protocol as both of them are supported by the aforementioned tools. In particular, the SFT output that contains analysis results transferred to the VA tool using these two protocols. After that, visualizations of these results are available to the end-users. Moreover, the user is able to demand further visualization and analysis results using the interactive interface. The Visual Analytics Tool will be integrated as a micro-frontend in the HMI framework.

5.3.9 Marketplace

The COMPOSITION marketplace is a fully distributed multi-agent system designed to support industry 4.0 exchanges between involved stakeholders. It is aimed at supporting automatic supply chain formation and negotiation of goods/data exchanges. The COMPOSITION marketplace exploits a microservice architecture and relies upon a scalable messaging infrastructure provided by the Message Broker. Trust and security are granted in every negotiation step undertaken by automated agents on behalf of involved stakeholders.

The COMPOSITION marketplace is described in further detail in D6.3 "COMPOSITION Marketplace I".

The COMPOSITION marketplace is composed of four main building blocks: The Agents, the Management Portal and Services, the Communication Infrastructure (Message Broker configured for intra-factory) and the Security Services (see Figure 22).

[22] https://angularjs.org/
[23] http://fusetheme.com/admin-templates/angular
[24] https://www.chartjs.org/

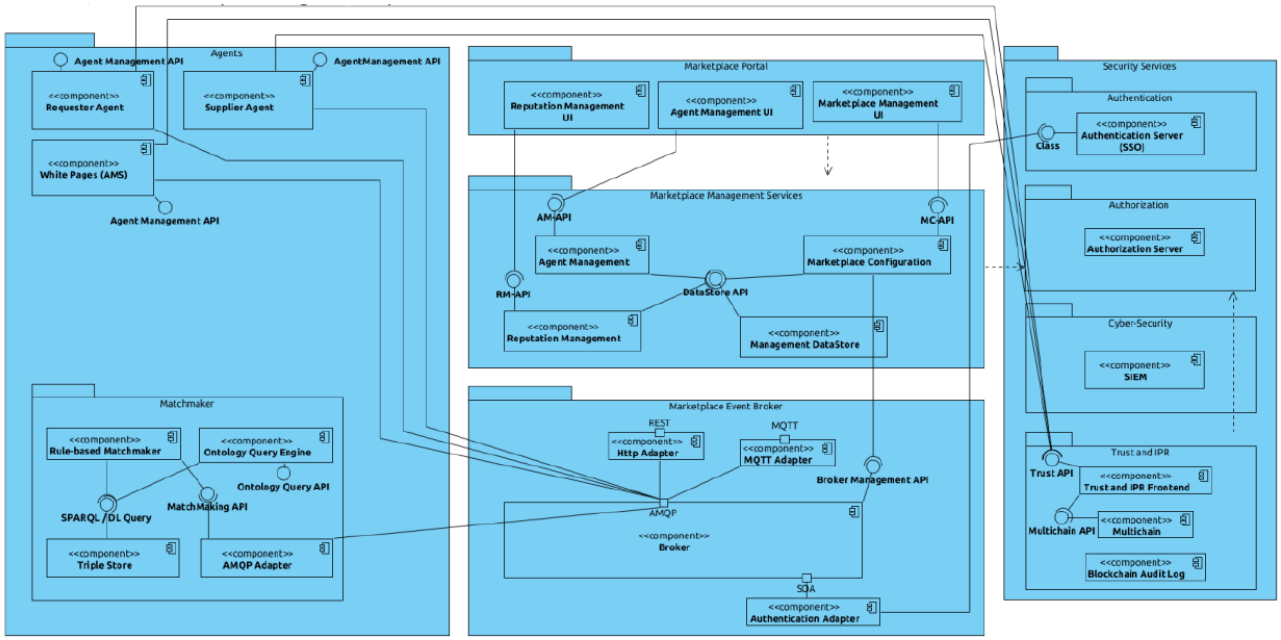

Figure 22: Marketplace components

Agents may implement market-specific services, such as the white pages agent or the matchmaker, or they can act on behalf of industry stakeholders participating in the marketplace. Required communication infrastructure is provided by the Message Broker which provides message delivery services to all other components through a well-known, publish-subscribe, interaction paradigm.

The set of components formed by the Marketplace Portal and the Marketplace Management Services has been designed to offer suitable means to administer marketplaces, register new market stakeholders, provide access credentials and connection parameters for agents to be deployed on the COMPOSITION market, and the like. This design choice allows stakeholders to easily manage the entire marketplace infrastructure, e.g., for defining new Closed Marketplaces.

Transactions and interactions between components in the platform are subject to a certain number of security checks and procedures aimed at ensuring a high degree of trust and reliability of exchanged information. These involve, among the others, restricted access to the marketplace communication infrastructure, channel encryption, provenance assessment techniques for messages, audit logs on message trails, etc. To support marketplace components in achieving such a trusted and secure operation, a dedicated set of components is purposely part of the marketplace design: the so-called Marketplace Security Services (described in Section 5.3.12).

5.3.10 Agent Management System

According to FIPA specifications (FIPA, 2004), an Agent Management System (AMS) is a mandatory component of every agent platform, and only one AMS should exist in every platform. It offers the white pages service to other agents on the platform by maintaining a directory of the agent identifiers currently active on the platform. A White Pages service is required to locate and name agents on the system, making it possible for one agent to connect with one another. In the current implementation of the Agent Management Service, the agent identifiers are stored in a MySQL Database. MySQL has been chosen because it offers relevant features for the project such as on-demand scalability, high availability and reliability. Other agent platforms, like SPADE[25], use MySQL as well for offering the White Pages service.

The current development of the component includes the White Pages Service implemented with a MySql database. (More details in the next sections)

The main component's interfaces are described in D6.5 "Connectors for inter-factory interoperability and logistics I".

[25] https://pypi.python.org/pypi/SPADE

5.3.10.1 Functional View

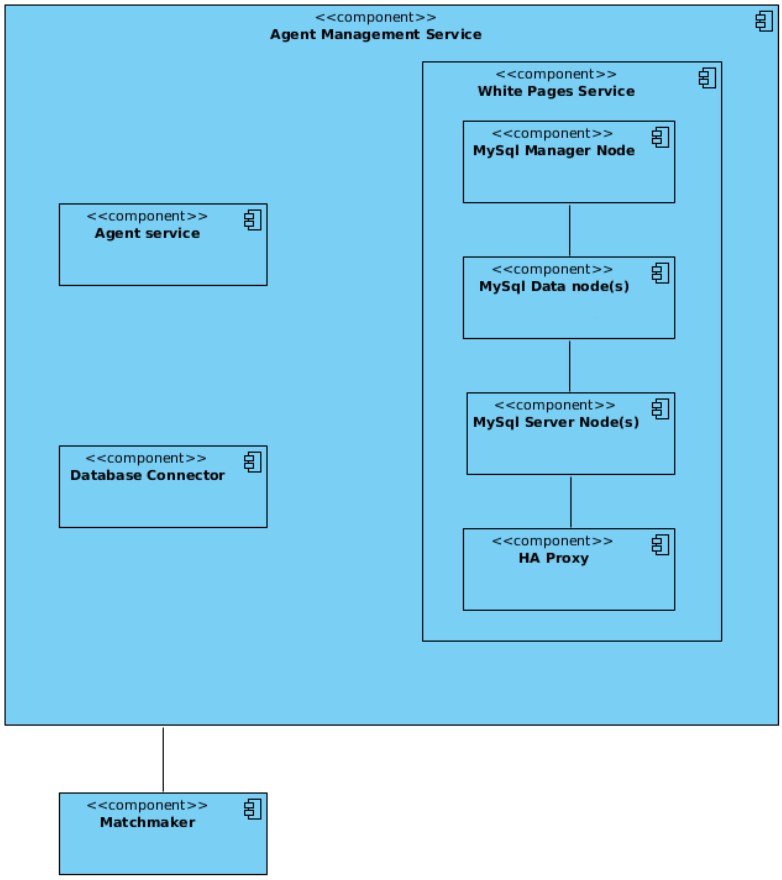

Figure 23: Design and dependencies of the Agent Management System: Matchmaker, Database for storing agents' data

AMS provides the agents with the necessary means to:

- Register on the marketplace
- Deregister from the marketplace
- Update agent information on the marketplace
- Interact with the Matchmaker

The main internal components are:

- Agent Service: provides REST interfaces for interacting with the agent
- Database connector: provides an abstract layer of interaction with the database underlying the White Pages Service
- White Pages Service: Provides access to the database where agent identifiers for the marketplace are stored.

Agents on the marketplace depend on Agent Management System in order to be able to participate in the marketplace. This component cannot be missing from the marketplace deployment.

In order to guarantee a correct registration of the agents on the White Pages database, the expected input for registration is the following:

```
{
 "description" : "A message used by user to register an agent",
 "type" : "object",
```

```
  "properties" : {
    "agent_id" : {
      "description" : "The unique identifier of the agent",
      "type" : "string"
    },
    "agent_owner" : {
      "description" : "Identifier for agent's owner",
      "type" : "string"
    },
    "agent_role" : {
      "description" : "An agent can be either requester or supplier",
      "type" : "string",
      "enum" : ["requester", "supplier"]
    }
  }
}
```

This schema is only temporary and will be modified when the whole set of ontologies for agent management will be ready (to be reported in D6.6 “Connectors for Inter-factory Interoperability and Logistics II”, M34).

5.3.10.2 Architecturally significant design decisions

As mentioned before, the AMS is a key component of any Multi Agent System, therefore it must provide high-availability, scalability and fault-tolerance. The AMS offers the White Pages Service to all the agents which want to participate in the marketplace, therefore it is important that the storage of agents’ identifiers (and other important info) are always available with guarantee of not being lost.

To address scalability while providing high availability, a deployment combining MySql cluster[26] and HAProxy[27] has been studied and setup. MySQL Cluster has been chosen since it is a distributed database combining linear scalability and high availability. Moreover, it provides in-memory real-time access with transactional consistency across partitioned and distributed datasets, and it is designed for mission critical applications. HAProxy has been chosen since it is done for the purpose (load balancing), it is very fast and reliable.

5.3.11 Marketplace Agents

Agents are primary actors of the COMPOSITION marketplace. They typically instantiate the supply-chain formation strategy of industry stakeholders and are therefore crucial for the success of the project inter-factory solutions. Although in the long term many different agent types are expected to coexist in the same marketplace, 2 main categories of agents can be defined a priori, depending on the kind of provided services: Marketplace agents and Stakeholder agents.

The former category groups all the agents providing services that are crucial for the marketplace to operate. The latter category, instead, groups agents developed and deployed by the marketplace stakeholders to take part in chain formation rounds.

5.3.11.1 Marketplace agents

Following FIPA specifications, an Agent Management System (AMS) is a mandatory component of every agent platform, and only one AMS should exist in every platform. It offers the White Pages service to other agents on the platform by maintaining a directory of the agent identifiers currently active on the platform.

5.3.11.2 Stakeholder Agents

Stakeholder agents are deployed at the stakeholder’s premises and their purpose is to fulfil the stakeholder’s interests. In the following sections the reference implementations for the two different kinds of stakeholder

[26] https://www.mysql.com/it/products/cluster/
[27] http://www.haproxy.org/

agents will be described. The set of APIs for the interaction with the agents will not be described here, since they have been thoroughly analysed in deliverable D6.5: Connectors for Inter-factory Interoperability and Logistics I.

Two types of stakeholders' agents have been identified: the Requester agent and the Supplier agent.

5.3.11.3 Supplier Agents

5.3.11.3.1 Role and responsibilities in COMPOSITION architecture

The Supplier agent is the counterpart of the Requester agent on the COMPOSITION marketplace. It is usually
adopted by actual suppliers to respond to supply requests coming from other stakeholders in the marketplace. Factories transforming goods typically employ at least one Requester agent, to get prime goods and one supplier agent to sell intermediate products to other factories.

The current status of implementation includes the capability of acting according to the base contract-net negotiation protocol using COMPOSITION eXchange Language. Also, the connection with IIMS and GUI has been performed.

5.3.11.3.2 Functional view

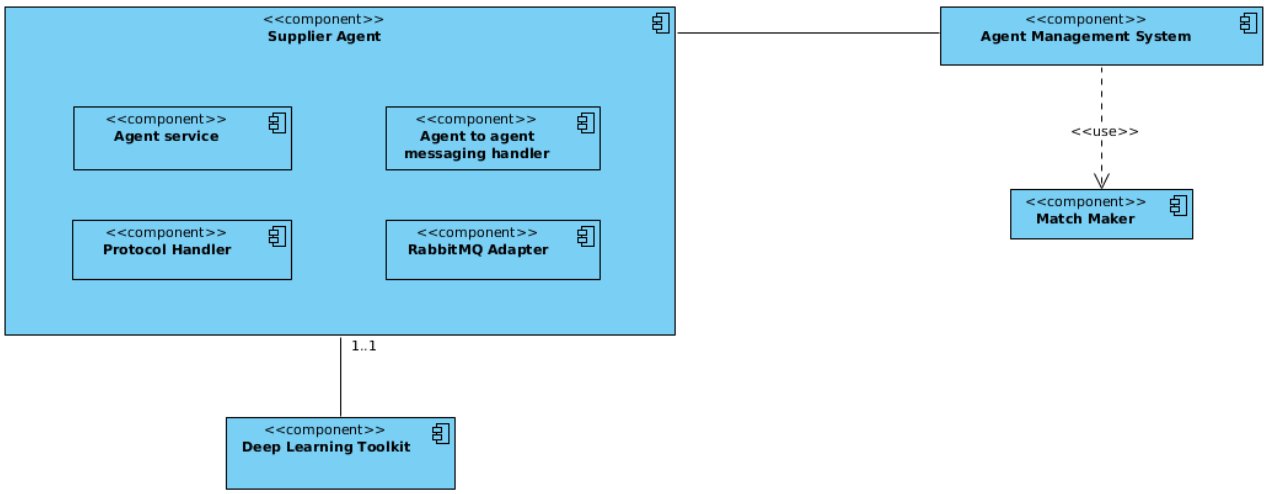

Figure 24: Design and dependencies of the Supplier Agent: Agent Management System, Matchmaker, Deep Learning Toolkit

As shown in Figure 26, agent's main internal components are:

- Agent Service: This component oversees the exposure of the services offered by the agent through REST endpoints, interfacing with GUI and IIMS according to the specific endpoint.
- Protocol Handler: This component handles all the protocol-related activities of the agent, such as state transitions, providing appropriate behaviour according to the current agent state.
- Agent to agent messaging handler: This component handles the incoming and outgoing messages from/to other agents on the marketplace, by providing language and ontology syntax check.
- RabbitMQ adapter: This component handles the communication with the RabbitMQ broker. In future it might be replaced by a more generic transport adapter, according to the broker in use on the marketplace.

To communicate with the Matchmaker agent, the Requester agent needs to have a connection with the Agent Management System.

The communication with the Deep Learning Toolkit is guaranteed by a point-to-point connection. The Deep Learning Toolkit is used by the Supplier agent to obtain features such as the price predictions for a certain good.

All the messages exchanged between the agents over the marketplace happen over AMQP through the RabbitMQ broker.

The messages flowing from the IIMS are received to a dedicated REST endpoint.

The messages from and toward the GUI flow through HTTP and dedicated REST endpoints.

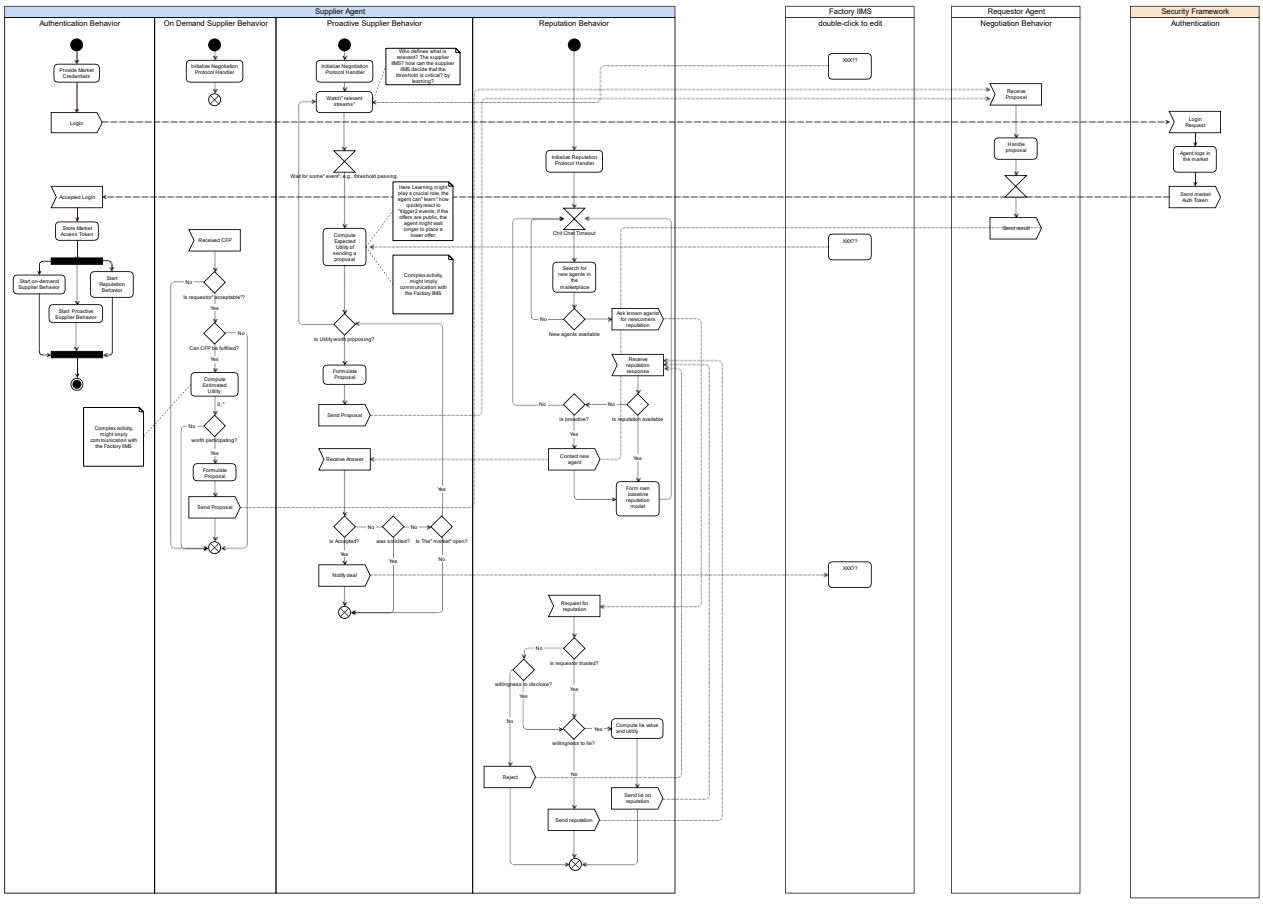

Figure 25: Supplier Agent sequence diagram

5.3.11.4 **Requester Agents**

5.3.11.4.1 Role and responsibilities in COMPOSITION architecture

The Requester Agent is the agent exploited by a factory to request the execution of an existing supply chain or to initiate a new supply chain. Due to the dynamics of exchanges pursued in COMPOSITION, there is no actual distinction between the two processes, i.e., for any supply need a new chain is formed and a new execution of the chain is triggered. The Requester agent may act according to several negotiation protocols, which can possibly be supported by only a subset of the agents active on a specific marketplace instance.

The current status of implementation includes the capability of acting according to the base contract-net negotiation protocol using COMPOSITION eXchange Language. Also, the connection with IIMS and GUI has been performed.

5.3.11.4.2 Functional view

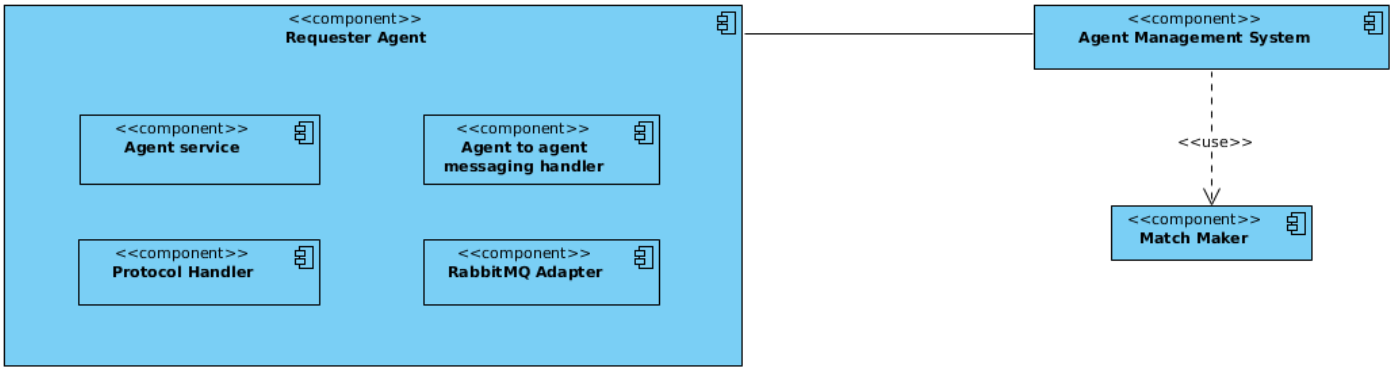

Figure 26: Design and dependencies of the Requester Agent: Agent Management System, Matchmaker

As shown in Figure 26, agent's main internal components are:

- Agent Service: This component oversees the exposure of the services offered by the agent through REST endpoints, interfacing with GUI and IIMS according to the specific endpoint.
- Protocol Handler: This component handles all the protocol-related activities of the agent, such as state transitions, providing appropriate behaviour according to the current agent state.
- Agent to agent messaging handler: This component handles the incoming and outgoing messages from/to other agents on the marketplace, by providing language and ontology syntax check.
- RabbitMQ adapter: This component handles the communication with the RabbitMQ broker. In future it might be replaced by a more generic transport adapter, according to the broker in use on the marketplace.

To communicate with the Matchmaker agent, the Requester agent needs to have a connection with the Agent Management System.

All the messages exchanged between the agents over the marketplace happen over AMQP through the RabbitMQ broker.

The messages flowing from the IIMS are received to a dedicated REST endpoint.

The messages from and toward the GUI flow through HTTP and dedicated REST endpoints.

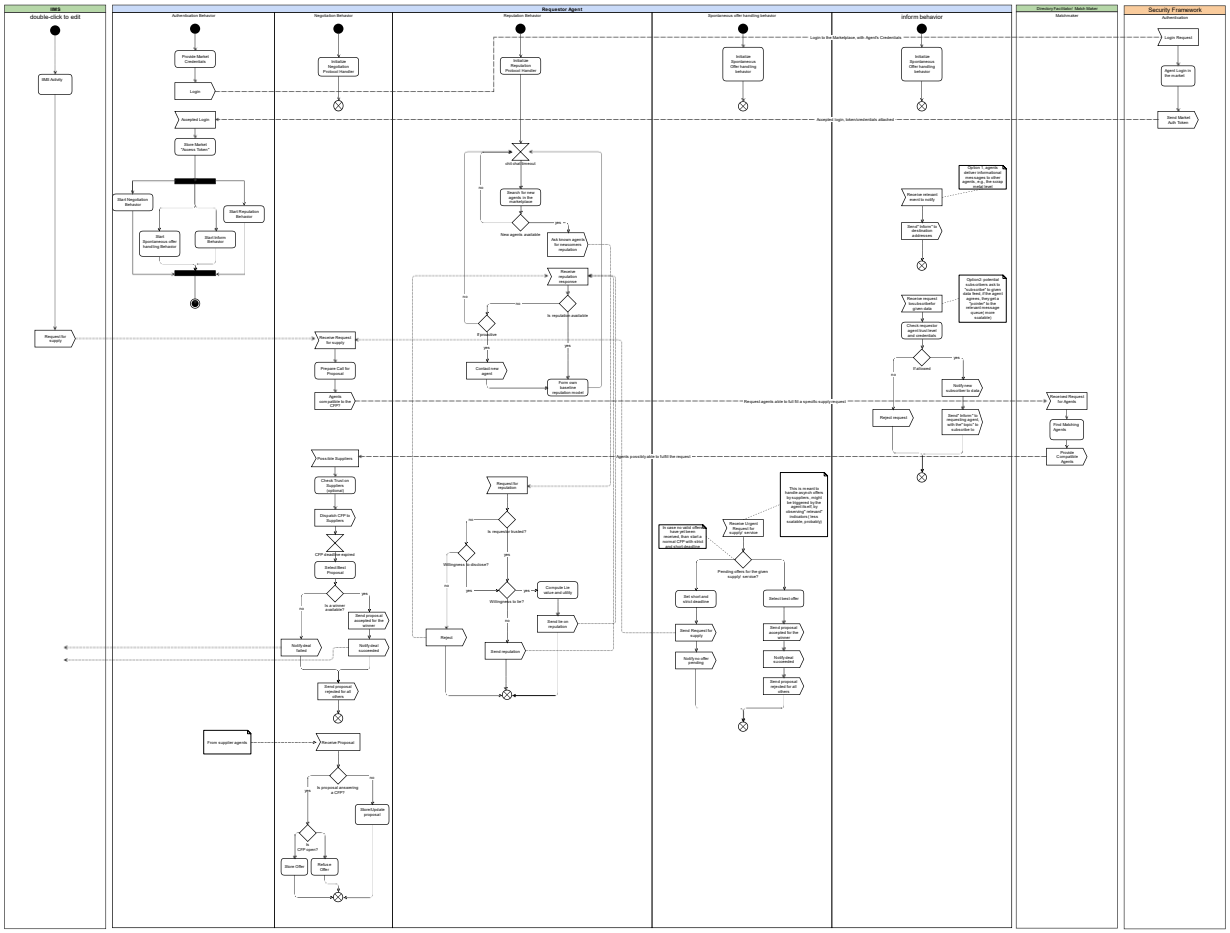

Figure 27: Requester Agent sequence diagram.

5.3.12 Marketplace Portal UI

5.3.12.1 Bidding Process Management UI

Messages are sent from the Supplier and Request agents to the respective GUI, these messages are sent through an API created in NodeJS and stored in a Firebase Realtime database which keeps the GUI updated at all times. The GUI is created using the Angular framework.

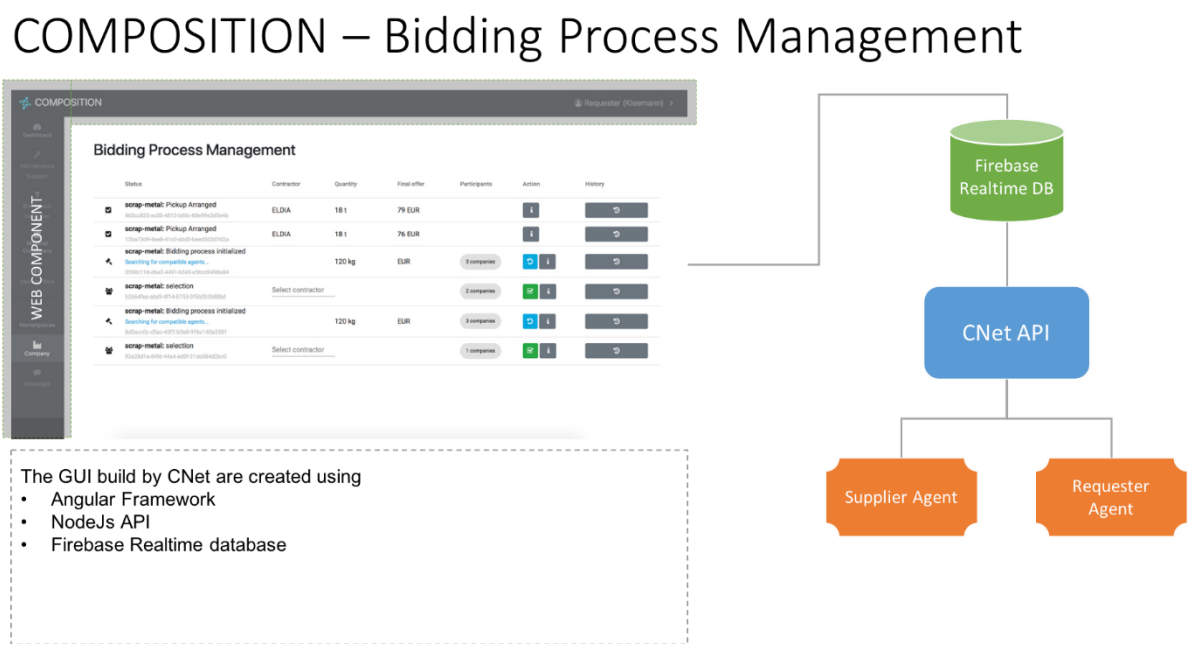

Figure 28: Bidding Process management

5.3.12.2 Material Management

The API, database used here are the same as for the Bidding process manage, only the source of the messages is different. Messages containing current information about containers etc. are sent by the BMS through an API and stored in the Realtime database that is connected to the GUI.

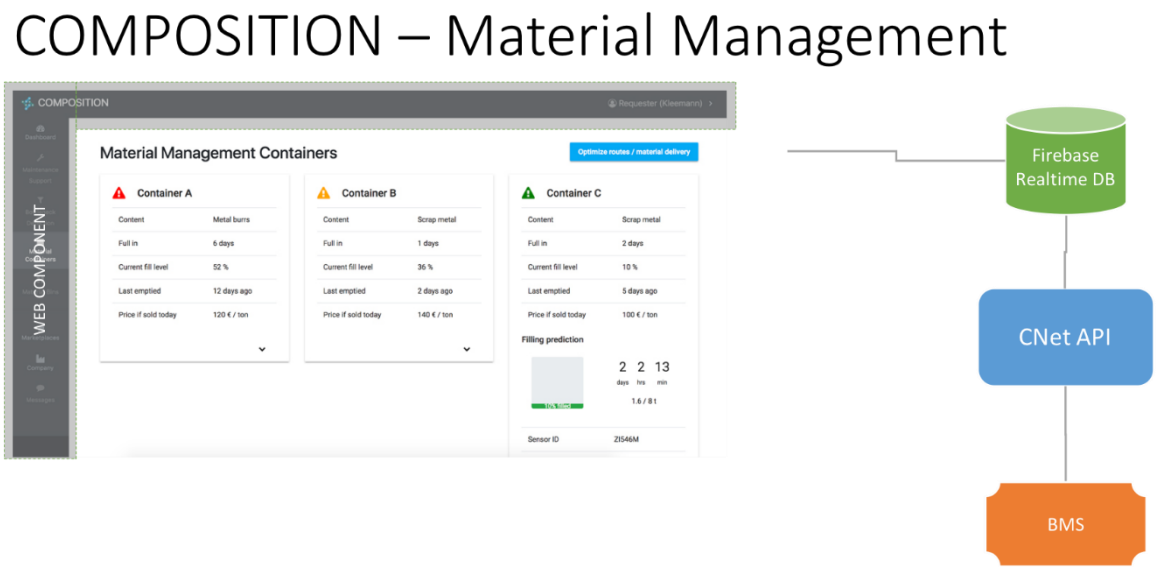

Figure 29: Material Management GUI

5.3.13 Security Framework

5.3.13.1 Introduction

The Security Framework implements the security core mechanisms aiming to ensure the security, confidentiality, integrity and availability of the managed information for all authorized COMPOSITION stakeholders. Below there is an overview of the current identified components that will conform the Security Framework.

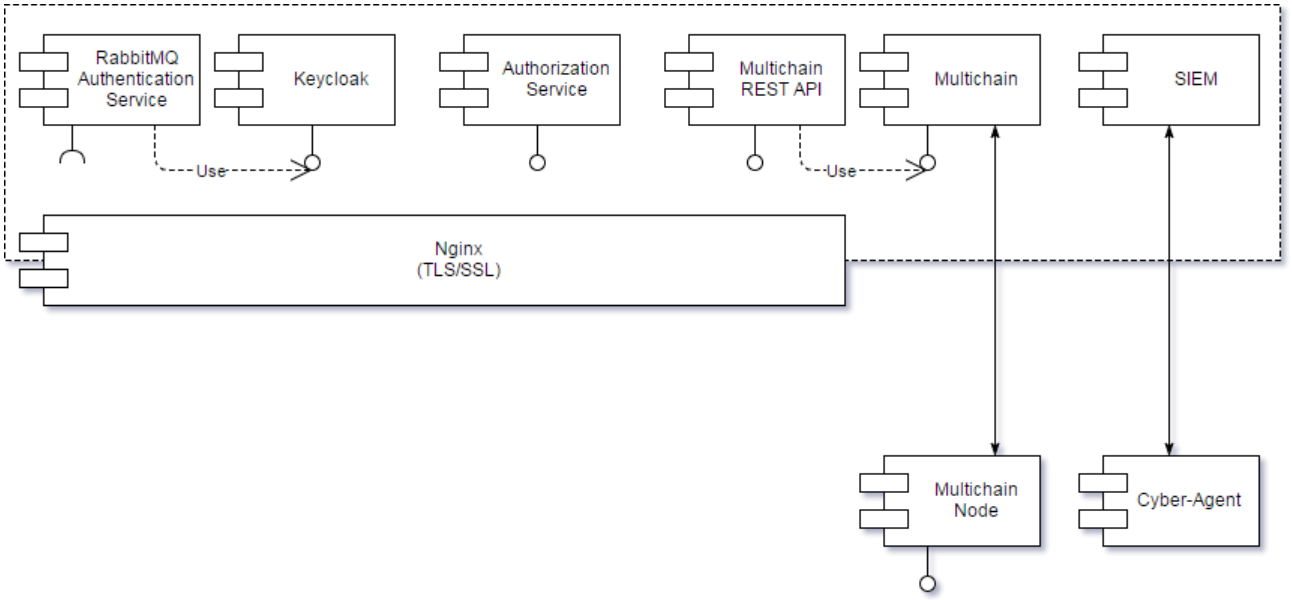

Figure 30: Components of the Security Framework

The current components in the Security have been grouped in for main categories, each of them focusing on different security tasks:

1. Authentication:
    a. Keycloak: Open source Identity and Access Management solution.
    b. RabbitMQ Authentication Service: Service that relays in Keycloak and Authorization Service to override built-in RabbitMQ authentication mechanisms.
2. Authorization:
    a. Authorization Service: Atos tool EPICA based on XACML3.0 that provides authorization and privacy access control to resources
3. Log-Trust-IPR
    a. Multichain: Blockchain based on Bitcoin with added functionalities.
    b. Multichain REST API: Will provide functionalities based on blockchain.
4. Cybersecurity:
    a. SIEM: Atos tool that provides the capabilities of a Security Information and Event Management (SIEM) solution with the advantage of being able of handling large volumes of data and raise security alerts from a business perspective.
    b. Cyber-Agents: These components are responsible to catch the events that later will be analysed by the SIEM.

In front of all web applications and services, in this case Keycloak, RabbitMQ Authentication Service, Authorization Service and Multichain REST API; Nginx will be used as reverse proxy configured for using TLS/SSL.

The following sections will provide details on each of the components and their categories.

5.3.13.2 **Authentication**

The components in this category are the responsible of providing the authentication mechanisms for users, applications, services and devices.

5.3.13.2.1 Keycloak

Keycloak[28] is an open source Identity and Access Management solution. Some of the features are:

- Single-Sign On: Authenticate on Keycloak rather on different applications. One single login will allow access to multiple applications and/o services.
- Identity Brokering and Social Login: Enable login with social networks such as Google, Facebook, Twitter and GitHub.
- User Federation: Connect directly to LDAP and Active Directory servers.
- Standard Protocols: OpenID Connect OAuth 2.0 and SAML.

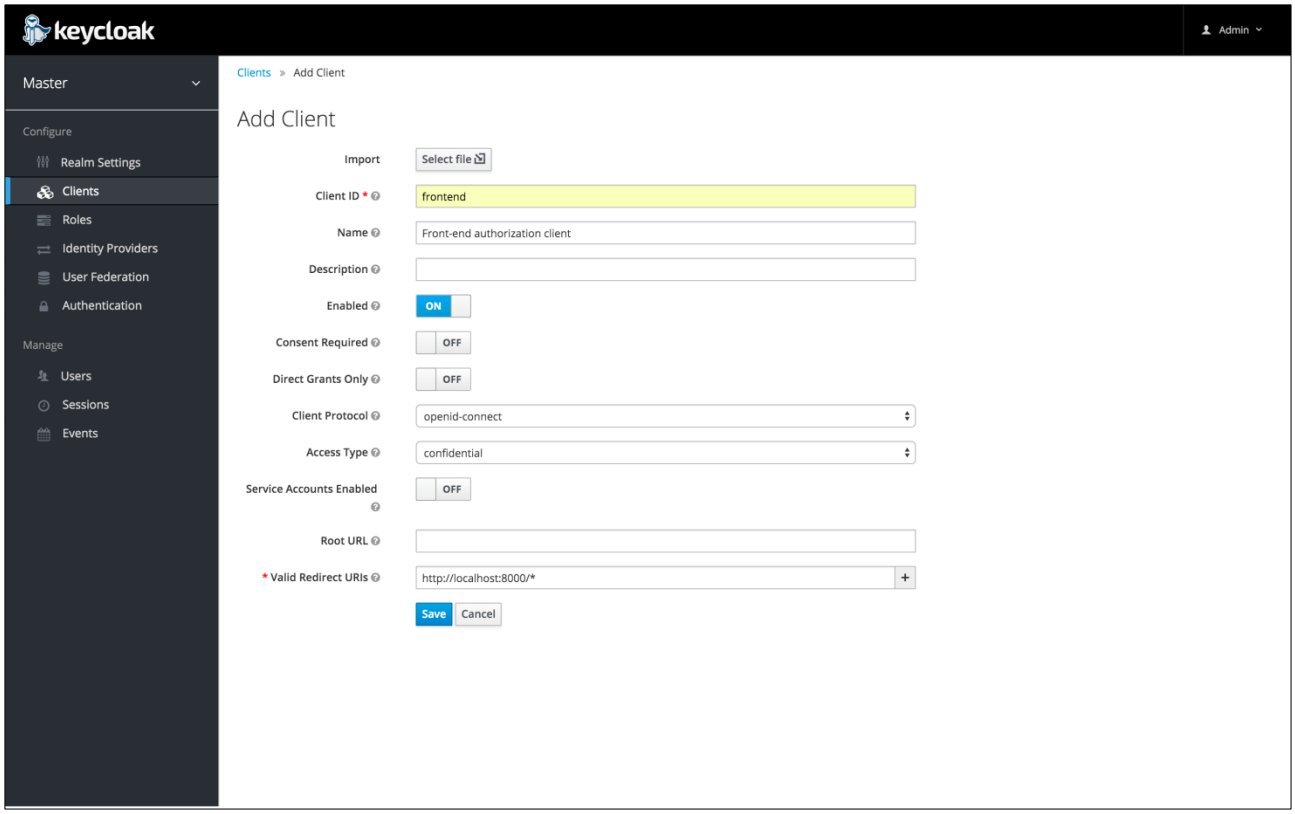

Figure 31: Keycloak administration interface

COMPOSITION components requesting access to system endpoints or Message Broker resources and HMI Framework components managing end-user access will use the standard protocols provided by Keycloak to e.g. request access tokens.

5.3.13.2.2 RabbitMQ Authentication Service (RAAS)

This component implements the needed interfaces to override RabbitMQ built-in authentication and authorization engine and it makes use of Keycloak and the EPICA Authorization Service for authentication and authorization instead.

5.3.13.3 Authorization

This category is responsible of all aspects about authorization mechanisms. Only one component has been identified under this category, the EPICA Authorization Service, which is a tool based on XACML 3.0

[28] http://www.keycloak.org

5.3.13.3.1 Authorization Service (EPICA)

The EPICA Authorization Service is a tool based on XACML 3.0 that provides authorization and privacy access control to resources. It provides two different functionalities:

- Policy management: Ability to manage policies. This means generating, storing, removing and modifying policies.
- Policy enforcement: Ability to enforce that a given access request for a specific resource fulfils the requirements of the policies applicable to the resource trying to be accessed.

5.3.13.4 Log, Trust and IPR

This category is responsible of the component that will contribute to the protection of IPR, the creation of trust and the audit trail for manufacturing and supply chain data.

5.3.13.4.1 Multichain

Multichain[29] is a private blockchain platform based on Bitcoin enhanced with added functionalities like managed permissions and data streams. Data streams are separately permissioned entities in the blockchain optimized for logging data in key-value pairs, as opposed to transactions involving assets (e.g. bitcoins). Several blockchains may be run in parallel, with managed permissions and several data streams per chain. The ability to run multichain in a consortium with a controlled set of block validators ("miners") negates the need for proof-of-work mining, making the generation of blocks, and consequently transaction validation, much faster.

Multichain is designed for compatibility with Bitcoin Core[30], with extensions to e.g. more conveniently manage data streams.

5.3.13.5 Cyber-Security

The components on this category focuses on the analysis of the cyber security in collaborative manufacturing and logistics ecosystems, identifying the variety of attacks (such as abuse of privileges, denial of access…) that could affect and be more relevant for the availability and reliability of the platform and infrastructure and potential remediation measures to mitigate their effects.

5.3.13.5.1 SIEM

SIEM provides the capabilities of a Security Information and Event Management (SIEM) solution with the advantage of being able of handling large volumes of data and raise security alerts from a business perspective thanks to the analysis and event processing in a Storm cluster. The main SIEM functionalities can be summarized in the following points:

- Real-time collection and analysis of security events.
- Prioritization, filtering and normalization of the data gathered from different sources.
- Consolidation and correlation of the security events to carry out a risk assessment and generation of alarms and reports.

5.3.13.5.2 Cyber-Agents

These components are responsible to catch the security events and transmit them to SIEM to be analysed. They are installed on the systems that need to be secured and their configuration may differ from one installation to another depending on the events to be monitored.

5.3.13.6 Nginx

Nginx[31] is a free and open-source web server software, which can also be used as a reverse proxy, load balancer and HTTP cache. Currently it´s only envisioned to be used as a reverse proxy in front of the web applications and services providing an additional security layer. It will also provide Transport Layer Security (TLS) encryption capabilities to all the applications and services behind it.

[29] http://www.multichain.com
[30] https://bitcoin.org/en/bitcoin-core/
[31] https://nginx.org/

5.3.14 Matchmaker

The Matchmaker component is a complete semantic framework for the COMPOSITION Collaborative Ecosystem. It contains the Rule-based Matchmaker, the Ontology Querying Component, the Ontology Store and corresponding APIs.

5.3.14.1 Role and responsibilities in COMPOSITION architecture

COMPOSITION Matchmaker package's role is to offer a complete semantic framework to the Agents Marketplace. The Ontology store is initialized with Collaborative Manufacturing Services Ontology and consists the main knowledge base of the Marketplace. The Ontology Querying Component offers CRUD operation to agents. The operations are applied to the Ontology Store. The COMPOSITION Rule-based Matchmaker is designed to be the core component of the COMPOSITION Broker. It supports semantic matching in terms of manufacturing capabilities, in order to find the best possible supplier to fulfil a request for a service or products involved in the supply chain. Different decision criteria for supplier selection, according to several qualitative and quantitative factors, are considered by the Matchmaker. Furthermore, the Matchmaker acts as a broker for the Marketplace's bidding processes and enables the automation of these processes as well. The Matchmaker evaluates the available offers from the providers in order to suggest the best one to the supplier.

The current status of the implementation of the Matchmaker component is a stable version able to offer a complete semantic framework in order to support ontology storage, ontology manipulation services, possible customers and suppliers' matchmaking and available offers' evaluation as well. The Matchmaker component is deployed in a Docker container to the project's inter-factory server.

5.3.14.2 Functional view

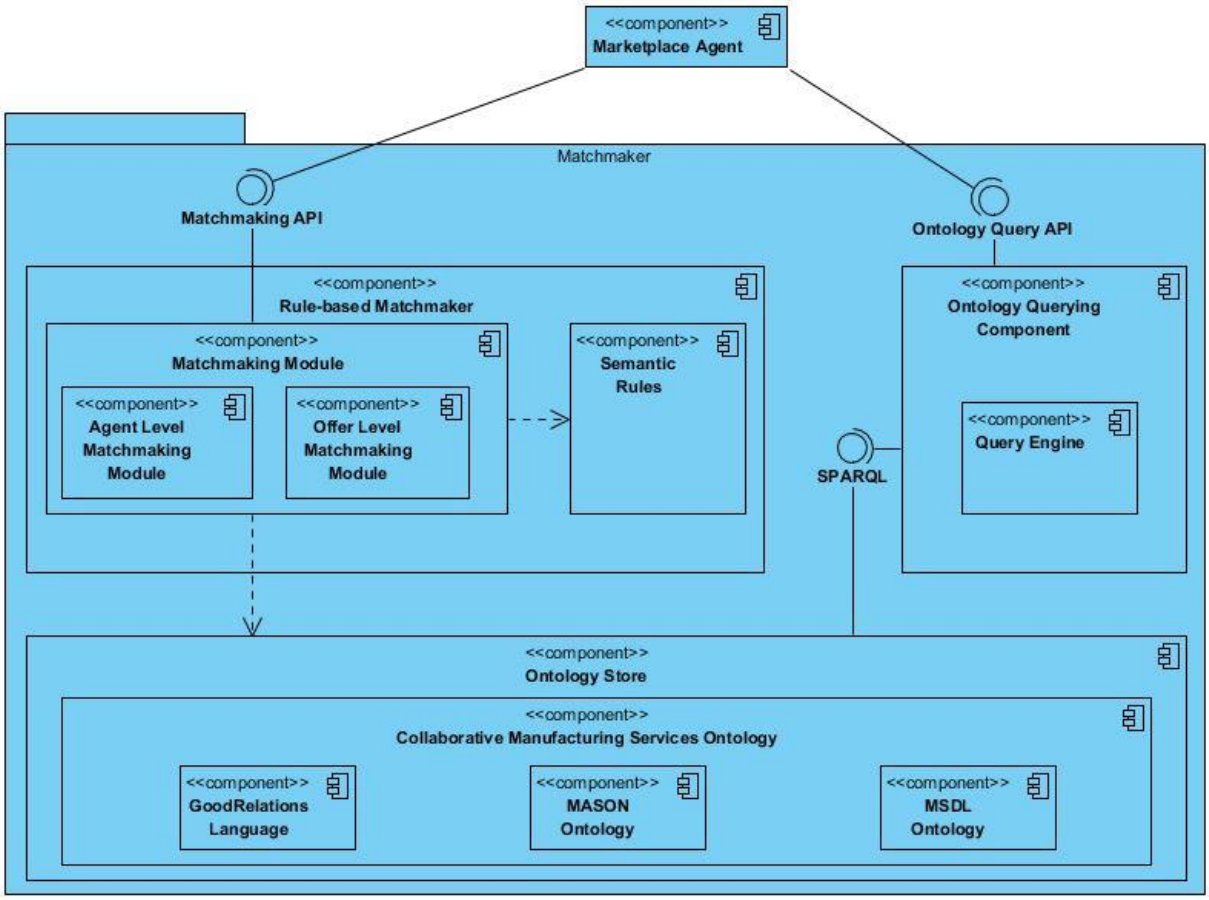

Figure 32: Functional view of COMPOSITION Matchmaker package

Three are the main components from the Matchmaker package:

1. **Ontology Store**: The Ontology Store is the main knowledge base for the COMPOSITION Marketplace. All the created ontology instances (business entities, manufacturing services, offers etc.) will be stored in the Ontology Store. More precisely, the Collaborative Manufacturing Services Ontology initialises a Jena TDB store. By using TDB, the Ontology is saved as a Model in the file system. Jena's component TDB used as a high-performance RDF store instead of a classic SQL database (it is faster and supports the storage of millions of individuals). Every change at the ontology takes place at the Model stored in the file system leaving the original ontology immutable.

2. **Ontology Querying Component**: This component enables agents' access to the Ontology Store. Agents can use Ontology Query API services by sending requests in a compatible to CXL JSON format. Then the Ontology Query engine will create SPARQL queries based on the agents's request and these queries will be applied to the Ontology Store. In this way, the agents will be able to create, read, update and delete instances from the Ontology Store.

3. **Rule-based Matchmaker**: The Rule-Based Matchmaker will be used by Marketplace's agents in order to match customers with suppliers and requests with offers in the Marketplace. Its core component is the Matchmaking Module which consists of the following sub-components:

- Agent Matchmaking Module: This module interacts with agents. An agent sends a request for a service to the Agent Matchmaking Module which applies a set of rules in ontology stored to the Ontology Store. Then the matchmaking module sends a response to the agent with a list contains the agents who support a matching offer for this request.

- Offer Matchmaking Module: This module interacts with agents too. An agent sends a request for available offers' evaluation to the Offer Matchmaking Module which applies a set of rules in the Ontology Store. The set of rules considers several qualitative and quantitative factors to match the agent's request with the best available offer and it is not limited to match the agent with all the other agents that can support his request as the Agent Matchmaking Module does. So, the response of the Offer Matchmaking Module is the best available offer.

The Semantic Rules component is a sub-component of Rule-base Matchmaker which contains all the files with rules. These rules will be applied by Matchmaking Module to the Ontology Store which is initialized by the Collaborative Manufacturing Service Ontology in order to extract the requested matching. The rules are in Jena format.

Dependencies:

The Matchmaker framework's provided functionalities are depended from the contained in the framework Collaborative Manufacturing Services Ontology. The Rule-based Matchmaker services are exclusively designed for the aforementioned ontology's concepts. Moreover, the SPARQL queries from the Ontology Querying component are designed based on the Collaborative Manufacturing Services Ontology terms.

Besides the dependency with the ontology, the Marketplace Agents can be considered as a type of dependency as well. Actually, the agents asks for the Matchmaker functionalities however all the Matchmaker's services triggered by agents' requests. This interaction defines the input and the output of the Matchmaker. The input is the agents' requests described in JSON format (CXL compliant format). A request contains information about a new instance that is going to be added in the marketplace, or information about a requested service or a set of provided offers that the Matchmaker should evaluate etc. The produced output is in the case of CRUD operations from Ontology Querying component a message of successful operation in JSON format (CXL compliant format). In the cases of matchmaking services will be a list of matching agents or offers in the same format.

Communication and Interactions:

The Matchmaker component communicates with the agents using HTTP protocol. The components functionalities are offered to the agents though RESTful web services. Two APIs is provided by the Matchmaker and they presented in the next table. More details about the services of these APIs are available to the corresponding deliverable.

Table 3: Matchmaker APIs

Matchmaker APIs	Description

Ontology Query API	This API receives as input agents' requests (related to CRUD operations to the ontology) and response back to the agent with a message of a successful operation or an error message. Inputs and outputs are in a predefined common format (JSON and CXL). The API is connected with the Querying component that applies a SPARQL query (e.g. Insert/Select commands) into the Ontology Store based on the agents' request. This interface is defined by the REST protocol.
Matchmaking API	This API receives as input agent's requests for customer/suppliers matching or offers/demands matching and response back to the agent with the matchmaking result. Inputs and outputs are in a predefined common format (JSON and CXL). The API is connected to Rule-based Matchmaker that performs the appropriate level of matchmaking based the request. This interface is defined by the REST protocol.

5.3.14.1 Architecturally significant design decisions

The Matchmaking Module is developed in Java and it is built upon the Apache Jena API[32]. Java was selected as it is one of the most popular programming languages in use, especially for client server web applications. The Apache Jena API is a free and open source tool which supports OWL and RDF languages and offers querying, reasoning and storing capabilities. All these criteria suggest the Jena framework as the perfect tool for COMPOSITION Matchmaker implementation. The Matchmaker is offered to other components through RESTful web services. Its core functionality is to receive Marketplace Agents' requests via Matchmaker API and to apply sets of semantic rules to the Ontology Store based on these requests. New knowledge will be inferred by the rules' appliance, and then the Matchmaking Module responses to the Agents by using the Matchmaker API. Furthermore, agents can access and manipulate the Ontology Store using Ontology Querying Component and the corresponding Ontology API.

JSON is selected as the communication data format. It is a text format that is completely language independent but uses conventions that are familiar to programmers. Also it is easy for machines to parse and generate this format. These properties make JSON an ideal format for data-exchange. More precisely, a format compatible with COMPOSITION eXchange Language (CXL) is used in order to offer easy communication with the Marketplace Agents.

Apache Tomcat[33] was the selected web server environment. It is an open-source Java Servlet Container developed by the Apache Software Foundation. It provides an HTTP web server environment in which Java code can run. The complete Matchmaker package is deployed to Tomcat server. After that, the Tomcat server is deployed as a Docker image.

5.4 Information View

The purpose of the information view is to describe how information is represented, persisted, and distributed in the architecture.

[32] https://jena.apache.org/
[33] http://tomcat.apache.org/

5.4.1 Data Models

5.4.1.1 Overview

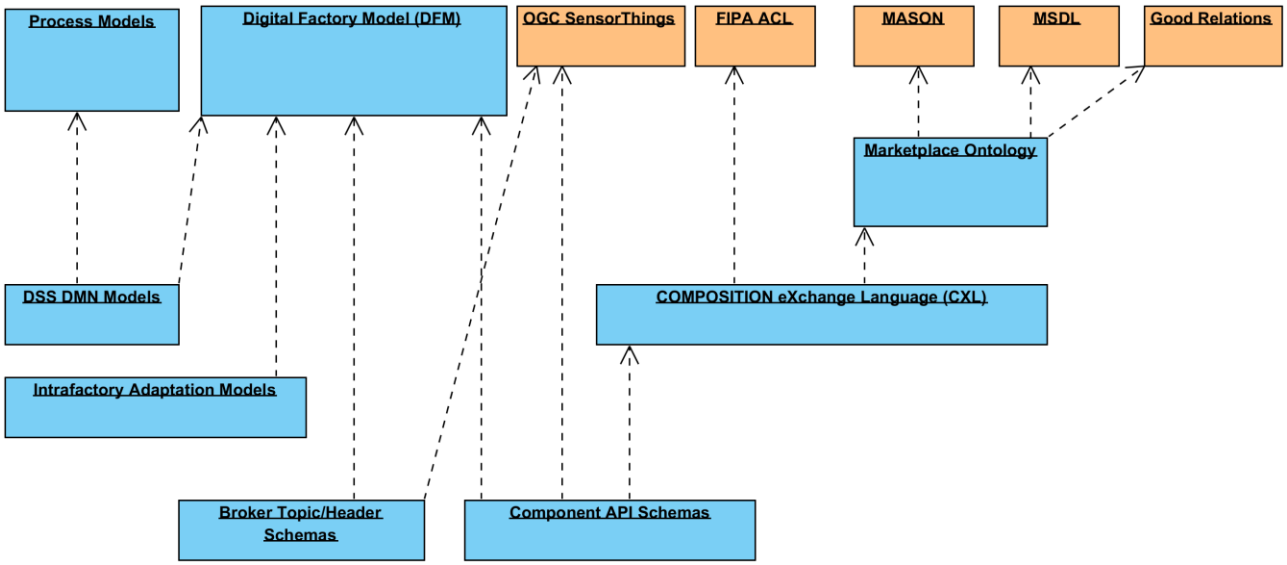

Figure 33: Dependencies of data models used in the system

The Digital Factory Model (DFM) contains both types and instances of the intra-factory components, e.g. production lines, products and sensors. This information will be used by e.g. the BMS to connect the physical sensors to the DFM instances and propagate this information to the LinkSmart middleware to identity the sensor data. The information is also used to build the topics in the message broker by which other components can subscribe to live data. The broker topic schemas have not yet been defined.

The process models describe the production process, linking information in the system to the process context used in the Decision Support System.

The OGC SensorThings Data Model is the used for system-generated data, e.g. data in the IIMS that has passed through the LinkSmart middleware and is exposed in inter-component communication will use the OGC SensorThings Data Model, with links to the DFM types and instances.

The inter-factory domain is modelled in the Marketplace Ontology and expressed in the COMPOSITION Exchange Language (CXL) used for agent communication.

5.4.1.2 Process Models

The goal of process models in COMPOSITION is to use common formats or standards to describe the production process. With such process models, process-oriented monitoring is made possible. By definition, process-oriented monitoring is a monitoring strategy that builds correlation measured values from sensors to a specific process procedure and a specific product instance in a production line, so that those sensor values could be further analysed within context. For example, with process-oriented monitoring it is possible to investigate how much energy is consumed while producing a specific PCB panel in solder printing. Process-oriented monitoring opens up possibilities for different big data analysing strategy, such as real-time abnormalities detection, product quality prediction etc.

5.4.1.2.1 BPMN

The process models of the industrial processes follow the Business Process Model and Notation (BPMN) standard. BPMN is a standard for business process modelling that provides a graphical notation for specifying business processes in a Business Process Diagram (BPD), based on a flowcharting technique very similar to activity diagrams from Unified Modelling Language (UML). Besides the graphical representation, the standard also specifies the XML schema for describing BPMN, which makes it easy to communicate between different systems.

Different elements from BPMN are adopted to model manufacture process in process models. First of all, production procedures will be modelled as *activities* in BPMN. The procedure will be modelled according to

its property, such as if it is a manual task, or if it is an automatic task finished by machines. Between activities there are intermediate message events, which matches to the corresponding sensor signals. These message events act as a transition between activities, which is triggered only when the matching sensor signals is received. With this structure, we can ensure that the BPMN virtual process is always synchronized with the real product process. Gateways are also utilized to model conditional forks during manufacture process.

During runtime, the process models will be instantiated and managed by a BPMN engine, such as the Activiti BPMN Engine. One can imagine the relation between the process model and an instantiated process as the relation between class and object in object oriented programming. Typically, each product on the line is represented by one instantiation of the model, tracking its current activity. This strategy enables a real-time matching between sensor values and the correspondent workpiece in the production line.
Figure 34 shows an example of a process model describing the production line of BSL. Notice that the process consists of many activities (rectangles in the diagram), each of which represents one step in production, such as laser marking PCB, screen printing solder, inspecting solder, etc. Between activities are intermediate message events, which will only be triggered by the matching sensor signals. Exclusive gateways are also used to model choices in process, such as if panel fails to pass *Inspect solder* test, it will be rejected to conveyor belt for either manual touch-up or touch-up in machine.

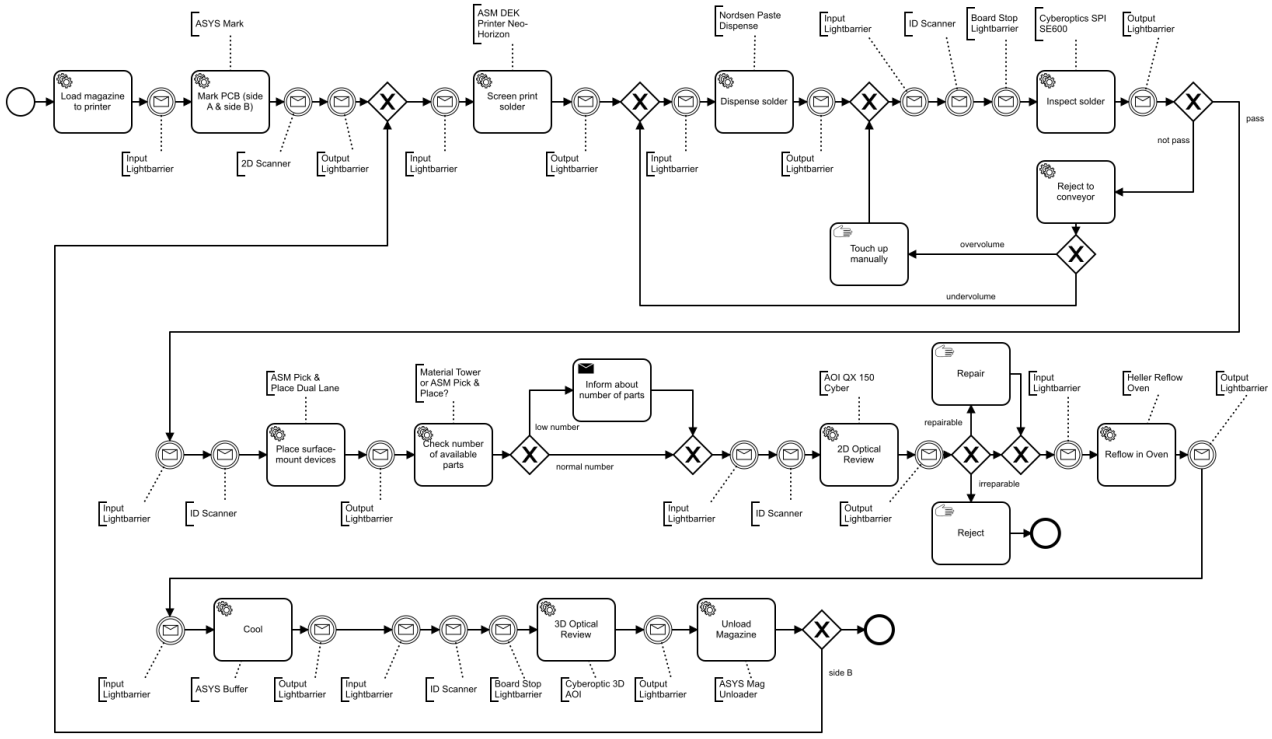

Figure 34: Initial BPMN diagram of BSL production line

5.4.1.2.2 DSS Process Model

DSS process model is based on Finite State Machines models and algorithms. Furthermore, the language, states and transition function are modified to accommodate the creation of the rules. States are defined based on the already existing states of the system. Alphabet is the conditions for each state. Each condition can be mathematical expressions, which when they change the state should change also, regular alphanumeric expressions and strings or a combination between all of them. The transitions are defined from the alphabet and they are a subset of it. The transition function for each transition is evaluated as true or false and when the transition is evaluated true, the system moves from the transition's initial state to the transitions final state.

A state diagram is created for each rule. State diagrams graphically represent the FSM, and contain initial and final states, transitions for different conditions and each transition's condition. The more complicated the rule, the more complicated the state diagram also is. The initial rules contain a few states and transitions, even

though the transitions are more than the states, because there are different ways from transitioning from one state to the other, back and forth. Figure 35 shows an initial state diagram for a rule in the rule engine[34]

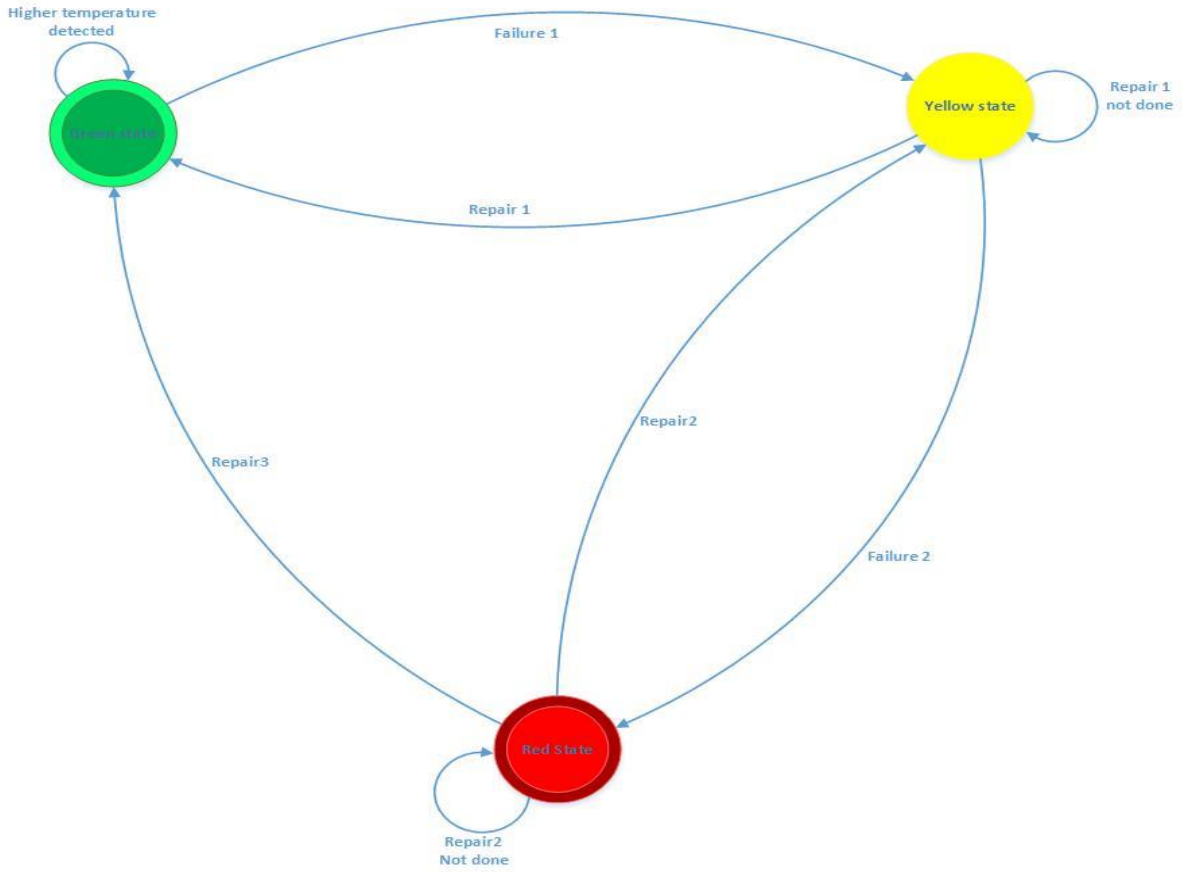

Figure 35: State Diagram for FSM Rule in the Rule Engine

5.4.1.3 Digital Factory Model

The Digital Factory Model or DFM was designed with the aim to describe in a common format, the data coming from heterogeneous resources with heterogeneous formats and to define this common format, which will be used by other COMPOSITION IIMS components. Both JSON and XML data formats were used in the definition of the DFM.

[34] D3.8 - Manufacturing Decision Support System I

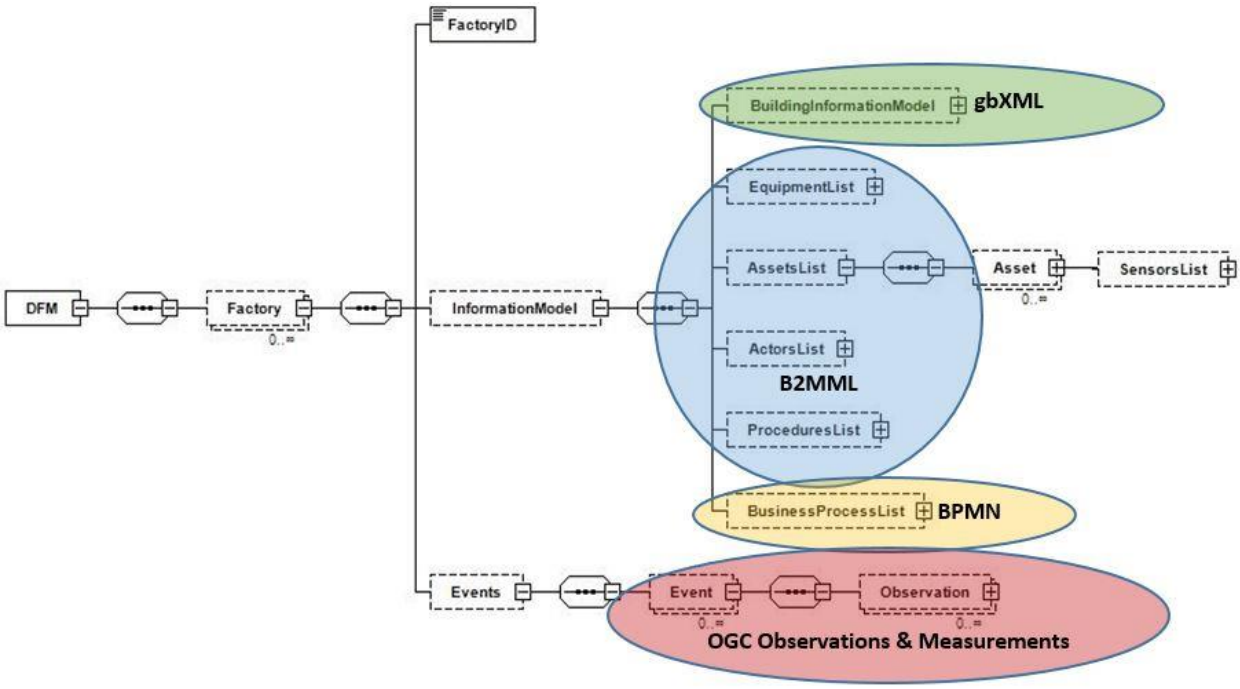

Figure 36: DFM Data Model

As depicted in the previous figure the DFM schema is divided in two parts:

- the Information model that contains all the static information related to a factory and
- the Events model that contains all the dynamic data related to a shop-floor

The B2MML[35] package covers the assets, equipment, procedures and actors concepts' descriptions. A sensors schema has been designed in a familiar format with the B2MML and was connected with assets. The description of the factory processes is covered by BPMN[36] package and the building information model is covered by gbXML[37]. The Events model which is related to dynamic data such as sensors' measurements and analytics tools' output is covered by OGC Observations and Measurements [38]JSON package. More details about the DFM are documented in D3.2 Digital Factory Model I and they will be updating in D3.3 Digital Factory Model II (M26).

5.4.1.4 COMPOSITION eXchange Language

Agents communicate through messages encoded in a dedicated language named COMPOSITION eXchange Language (CXL). Rather than defining yet another agent communication language, the consortium decided to stick to existing standards and to extend them wherever needed. CXL has therefore been designed as a dialect of the well-known FIPA ACL language specification, with a dedicated syntax ("codec" in the FIPA jargon) and with reference to a well-defined set of ontologies for representing the message payload data.
A CXL message is composed of:

- An almost fixed set of parameters, identifying the message purpose, sender and language
- A variable payload whose content depends on the message type, and typically is encoded according to an explicitly pre-defined ontology.

The CXL JSON schema listed in Appendix 3: CXL JSON Schema depicts the exact fields defined in CXL. Each of them has a 1-to-1 mapping to the corresponding FIPA ACL message parameter. The CXL schema has undergone minor changes compared to what defined in the previous deliverable D2.3.

At the current stage of development 3 main vocabularies, i.e., ontologies, have been identified and catalogued for use in the COMPOSITION CXL between stakeholder agents.

[35] http://www.mesa.org/en/B2MML.asp
[36] http://www.bpmn.org/
[37] http://www.gbxml.org/
[38] http://www.opengeospatial.org/standards/om

The first vocabulary is called COMPOSITION-negotiation-ontology, and it is used in order to prepare/reply to offers on the marketplace. Its definition is the following:

```
{
 "description":"The JSON syntax specification of the COMPOSITION-negotiation-ontology",
 "type": "object",
 "properties": {
  "offer-details": {
   "type": "object",
   "properties": {
    "good": {
     "type": "string",
     "description": "The good involved in the current bidding process"
    },
    "pickup-details": {
     "type": "object",
     "properties": {
      "start-date": {
       "type": "string",
       "format":"date-time",
       "description": "The earliest date for pickup"
      },
      "end-date": {
       "type": "string",
       "format":"date-time",
       "description": "The latest date for pickup"
      },
      "proposed-date": {
       "type": "string",
       "format":"date-time",
       "description": "The proposed date for pickup"
      }
     }
    },
    "currency": {
     "type": "string",
     "description": "The currency adopted for the bidding process"
    },
    "quantity-uom": {
     "type": "string",
     "description": "The unity of measure for the quantity",
     "enum":["kg", "q", "t"]
    },
    "quantity": {
     "type": "integer",
     "description": "The quantity of the good"
    },
    "price": {
     "type": "number",
     "description": "The offered price for the good, within the bidding process"
    },
```

```
      "rating": {
        "type": "number",
        "description": "The company rating"
      }
    }
   }
  }
}
```

The second vocabulary is called COMPOSITION-informative-ontology, and it is used in exchange of informative messages between stakeholder agents. It is still undergoing different changes, current implementation is the following:

```
{
 "description":"The JSON syntax specification of the COMPOSITION informative ontology",
 "type": "object",
 "properties": {
  "info": {
   "type": "object",
   "properties": {
    "information-type": {
     "type": "string",
     "description": "The type of informative message, either fill_level or price_forecast",
     "enum" : ["fill_level", "price_forecast"]
    },
    "details": {
     "type": "object",
     "properties": {
      "timestamp": {
       "type": "string",
       "format":"date-time",
       "description": "The timestamp related to the information"
      },
      "good": {
       "type": "string",
       "format":"date-time",
       "description": "The good involved in the current informative flow"
      },
      "price": {
       "type": "number",
       "description": "The (forecasted) price of the good"
      },
      "quantity-uom": {
       "type": "string",
       "description": "The quantity unity of measure",
       "enum":["kg", "q", "t"]
      },
      "quantity": {
       "type": "integer",
       "description": "The quantity of the good"
      },
      "fill-level": {
```

```
          "type": "number",
          "description": "The fill level for a certain container, containing the good"
         }
        }
       }
     }
   }
  }
}
```

The third vocabulary is called COMPOSITION-reputation-ontology, and it is used in order to support reputation values exchanges between stakeholder agents. Its definition is the following:

```
{
 "description": "The JSON syntax specification of the COMPOSITION reputation ontology",
 "type": "object",
 "properties": {
  "reputation-details": {
   "type": "object",
   "properties": {
    "agent-id": {
     "type": "string",
     "description": "Agent identifier"
    },
    "agent-owner": {
     "type": "string",
     "description": "Identifier for the company owning the agent"
    },
    "rating": {
     "type": "number",
     "description": "Rating for the company"
    },
    "timestamp": {
     "type": "string",
     "format": "date-time",
     "description": "Timestamp for the rating"
    }
   }
  }
 }
}
```

5.4.1.5 **Marketplace Ontology**

Collaborative Manufacturing Services Ontology is the knowledge base for the COMPOSITION Marketplace. It is used as a common vocabulary that offers interoperability and representation of both meanings and data in the Marketplace. The Collaborative Manufacturing Services Ontology enables:

- The description of manufacturing services, capabilities and resources for entities participate in the COMPOSITION Marketplace
- The description of supply and demand entities participate in the Marketplace

The Marketplace gents will be able to make transactions as the above information will be described using this common ontology. For example an agent who requests a service or a product will be able to find a matching agent who supports this service or product as they will be described using the ontology as a common vocabulary.

Collaborative Manufacturing Services Ontology should be able to represent manufacturing services and resources. For this reason, MSDL (Ameri, 2006) and MASON (Lemaignan, 2006) ontologies are imported to the Marketplace Ontology as they are manufacturing domain specific and they offer a large variety of classes and properties about this domain. Moreover, the COMPOSITION Marketplace should be able to support collaboration mechanism between business entities. This means that it should be able to describe relations and transactions between supply and demand entities which participate in the Marketplace. In order to fulfill this requirement the GoodRelations Language [39]ontology which is one of the most well-known and widely used ontologies in ecommerce domain is imported to the Collaborative Manufacturing Services Ontology as well. All the aforementioned ontological resources were imported and re-engineered using Neon Methodology (Suárez-Figueroa, 2010) in order to create a stable and consistent version of the Collaborative Manufacturing Services Ontology. The implemented ontology's main classes are presented in the next figure and they are presented in more details in the next page's table:

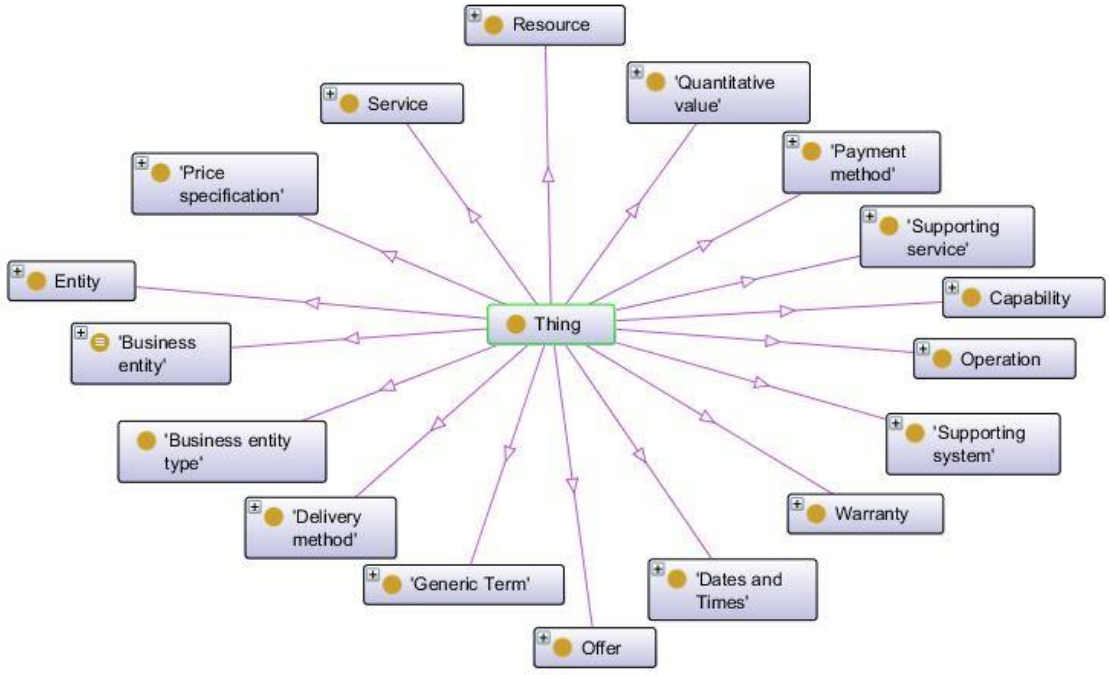

Figure 37: Collaborative Manufacturing Services Ontology Class Diagram

Table 4: Collaborative Manufacturing Services Ontology Main Classes

Class name	Description
Business entity	Represents an Ecosystem Agent who has a service (e.g. manufacturing service) and provides or seeks an offer
Business entity type	Represents the legal form, the size and the position of a business entity in value chain
Service	Conceptualizes all operations and processes related to a product in an abstract level
Operation	Represents the processes of a service

[39] http://www.heppnetz.de/ontologies/goodrelations/v1.html

Resource	Represents the total set of linked resources of a business entity
Supporting service	Represent services which are not basic services but are related to the basic one and support them
Supporting system	Represents some systems which support a business entity's services
Offer	Represents a public announcement of a business entity that provides or seeks a certain service or product
Warranty	Represents the duration and the scope of free services that will be provided to a customer in case of a possible malfunction or problem
Quantitative value	Represent the range of a certain property
Generic Term	Define common operations, materials and tools
Delivery method	Define the available delivery options for a service or product
Dates and Times	The days that a business entity has opening hours. Also represents the day of delivery or the day of availability of a service
Capability	Represents the capability of a service
Entity	Represents an entity as a result of a manufacturing process and describe its geometric flaw and entity, assembly entity and raw material
Price specification	Specifies the price of a unit, additional delivery costs and additional costs related to a payment method
Payment method	Describes the available procedures for transferring the requested amount for a purchase

5.4.1.6 **OGC SensorThings**

The SensorThings API[40] is an OGC[41] standard specification, part of the OGC Sensor Web Enablement standards[42]. This standard has been selected as the generic representation of data managed by the COMPOSITION system (see Figure 38 for the SensorThings data model). It is also used in the LinkSmart platform[43] and several implementations of persistent data stores are available.

As described in section 5.4.1.3, the project has defined mappings in the DFM between the data streams and observations in the OGC SensorThings Data Model and the factory assets and equipment.

[40] http://docs.opengeospatial.org/is/15-078r6/15-078r6.html
[41] http://www.opengeospatial.org/
[42] http://www.opengeospatial.org/ogc/markets-technologies/swe
[43] https://linksmart.eu/redmine/projects/iot-data-processing-agent/wiki/Usage_IoT_Data-Processing_Agent_

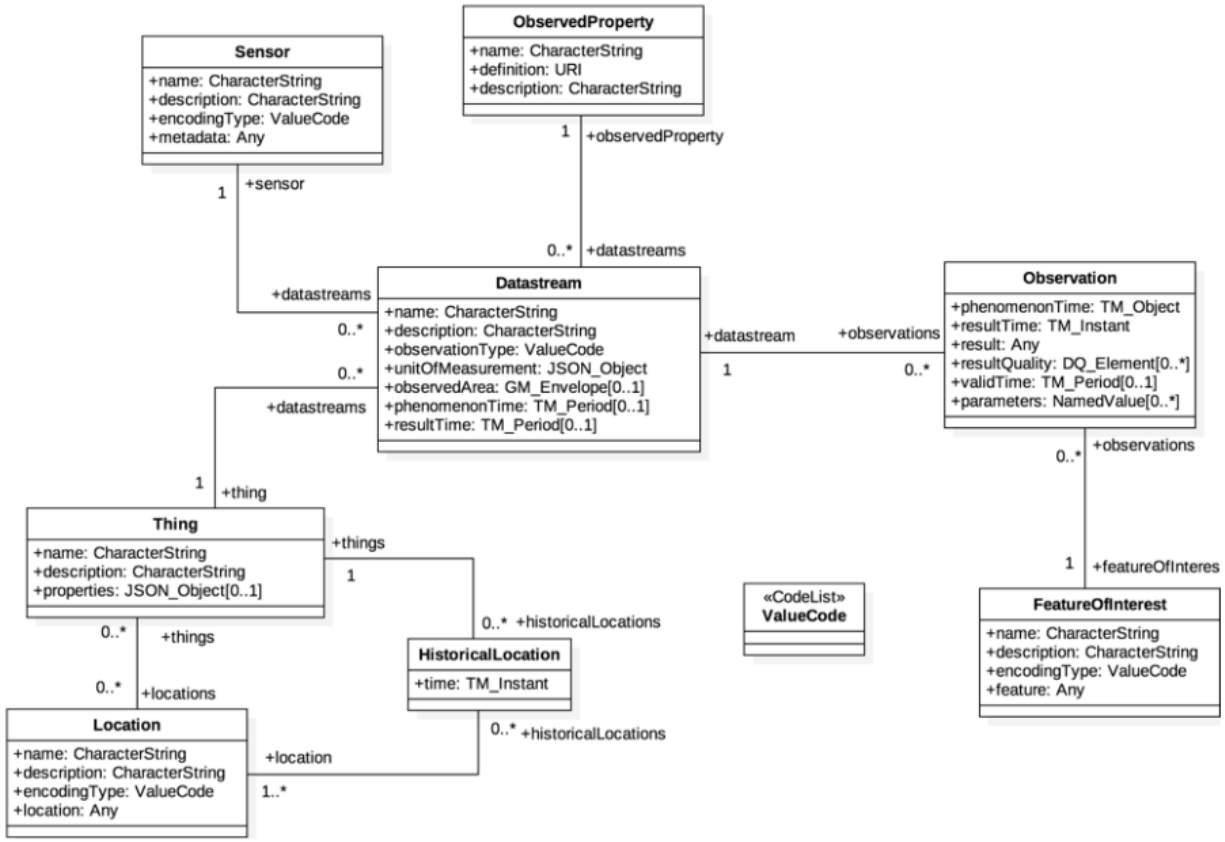

Figure 38: OGC SensorThings Data Model

The OGC SensorThings API consists of the Sensing and Tasking profiles.

The Sensing profile allows IoT devices and applications to CREATE, READ, UPDATE, and DELETE (i.e., HTTP POST, GET, PATCH, and DELETE) IoT data and metadata in a *Thing* service. Managing and retrieving observations and metadata from IoT sensor systems is one of the most common use cases. As a result, the Sensing profile is designed based on the ISO/OGC Observation and Measurement (O&M) model (OGC and ISO 19156:2011).

The key to the model is that an *Observation* is modelled as an act that produces a result whose value is an estimation of a property of the observation target or *FeatureOfInterest*. An Observation instance is classified by its event time (e.g., *resultTime* and *phenomenonTime*), FeatureOfInterest, ObservedProperty, and the procedure used (often corresponding to a *Sensor*). Things are also modeled in the SensorThings API, together with the historical set of their geographical positions

More specifically, in the Sensing profile, a *Thing* has *Location*s and *HistoricalLocation*s. It can also have multiple *Datastream*s associated. A Datastream is a collection of Observations grouped by the same *ObservedProperty* and *Sensor*. An Observation is an event performed by a Sensor that produces a result whose value is an estimate of an ObservedProperty of the FeatureOfInterest.

Following subsections better detail the single data model entries.

5.4.1.6.1 Thing

The OGC SensorThings API follows the ITU-T definition, i.e., with regard to the Internet of Things, a thing is an object of the physical world (physical things) or the information world (virtual things) that is capable of being identified and integrated into communication networks (Y.2060, 2012).

5.4.1.6.2 Location

The Location entity locates the Thing or the Things it is associated with. A Thing's Location entity is defined as the last known location of the Thing.

5.4.1.6.3 HistoricalLocation

A Thing's HistoricalLocation entity set provides the current (i.e. last known) and previous locations of the Thing with their time.

5.4.1.6.4 Datastream

A Datastream groups a collection of Observations and the Observations in a Datastream measure the same ObservedProperty and are produced by the same Sensor.

5.4.1.6.5 Sensor

A Sensor is an instrument that observes a property or phenomenon with the goal of producing an estimate of the value of the property.

5.4.1.6.6 ObservedProperty

An ObservedProperty specifies the phenomenon of an Observation.

5.4.1.6.7 Observation

An Observation is an act of measuring or otherwise determining the value of a property (ISO19156, 2011).

5.4.1.6.8 FeatureOfInterest

An Observation results in a value being assigned to a phenomenon. The phenomenon is a property of a feature, the latter being the FeatureOfInterest of the Observation (ISO19156, 2011). In the context of the Internet of Things, many Observations' FeatureOfInterest can be the Location of the Thing. For example, the FeatureOfInterest of a wifi-connect thermostat can be the Location of the thermostat (i.e. the living room where the thermostat is located in). In the case of remote sensing, the FeatureOfInterest can be the geographical area or volume that is being sensed.

5.4.2 Data Persistence

Data Persistence contains the COMPOSITION sub-components which are related to data storage and retrieval. These stored data are static information related to pilot cases, live data coming continuously from sensors or data related to predictions coming from the analytics tools. As decided the aforementioned data will be stored in two different components. The BMS will store all the real world data which are the sensors' measurements and the DFM together with an OGC SensorThings compliant data store will store all the COMPOSITION generated data (process models, predictions etc.)

5.4.2.1 Sensor data

The Deep Learning Toolkit needs to have historical data available to train the artificial neural networks, although this only has to be available as unstructured bulk data, without query capabilities. The Intrafactory Adaptation Layer leverages on Symphony BMS built-in storage for unprocessed shop-floor level data which can be used in this capacity and LinkSmart also provides capabilities for storing historical observation data.

Data persistence will to a significant extent be handled internal to the components and exposed through the component interfaces, in the case of component-specific data. However, there will still be a need to record and query both shop-floor data and data generated by the COMPOSITION system, common to all components. The Decision Support System and the Simulation and Forecasting Tool both need access to structured historical data generated by the system, with query capabilities.

The BMS provides a set of tools for collect and filter the real-time data incoming from the production facilities. This set of tools facilitate the possibility to build applications on top of real-time data. Secondly, through a component called Storage Handler, the BMS provides a repository for information valuable to be kept during the whole machine lifetime. These raw measurements can also be enhanced by providing additional metadata to be attached to them, in case it should become necessary.

In order to be as much as possible compliant with existing standards (actual or de facto), the design choice has been to implement a RESTful API that follows the specifications of "*FIWARE-NGSI v2 Specification API*[44]". This interface is used to manage data according to a very simple model using objects that are called *context entities* (applying this concept in COMPOSITION environment can turn into, e.g., an entity *Sensor* that have the type *vibration_sensor* and attributes such as *battery_level*, *amplitude* or *frequency*).

This data is transported in OGC SensorThings format inside the COMPOSITION system. The amount of data retrieved at every request coming from COMPOSITION components to the BMS storage could be very high and thus the risk of overloads must be taken into consideration. Therefore, it was necessary to use something less verbose than OGC Sensor Things Observation. In order to avoid the usage of completely different formats to represent the same information, OCG SensorThings dataArray[45] was the logical choice.

5.4.2.2 **COMPOSITION-generated data**

As COMPOSITION-generated data should be considered the data were produced from COMPOSITION components or models of the real world's objects:

- SFT and DLT predictions which are represented as OGC Observations in JSON format
- The designed BPMN diagrams that are exported in XML format
- Buildings information that are modelled in gbXML format
- Assets, sensors and actors modelled information in B2MML format

All these types of data will be stored in DFM instances using a developed and deployed DFM API. The use of the DFM API enables the creation of factory live instances stored in a MongoDB[46]. The COMPOSITION-generated data can be stored using the DFM API and its provided services. Furthermore, stored data related to a factory instance are able to be retrieved by other components such as decision support systems etc. using the DFM API services. The format of the data that are transferred through the API's services is the one that is defined by DFM schema. So, by using DFM schema and its corresponding API all the generated data can be available to other IIMS components or end users in a common format and in a common way.

The DFM API was implemented as a Java web application and it is offered through Restful web services. Its main functionality is to receive HTTP requests from IIMS components. Based on requests, the DFM API stores or retrieves data from MongoDB. After that, DFM API returns an HTTP response to the requested IIMS component. The response contains the requested resource from the MongoDB in the cases the requests are related to data retrieval. In the cases that the requests are about data storage or deletion, the response is just a simple message for successful operation. The next table summarizes the DFM API's provided services:

[44] https://orioncontextbroker.docs.apiary.io/
[45] http://docs.opengeospatial.org/is/15-078r6/15-078r6.html#79
[46] https://www.mongodb.com/

Table 5: DFM API Web Services

getActorByID	setActor	deleteActor
getActorList	setActorList	deleteActorList
getAssetByID	setAsset	deleteAsset
getAssetList	setAssetList	deleteAssetList
getEquipmentByID	setEquipment	deleteEquipment
getEquipmentList	setEquipmentList	deleteEquipmentList
getProcedureByID	setProcedure	deleteProcedure
getProcedureList	setProcedureList	deleteProcedureList
getSensorByID	setSensor	deleteSensor
getSensorList	setSensorList	deleteSensorList
getBuildingInformation	setBuildingInformation	deleteBuildingInformation
getBPMN	setBPMN	deleteBusinessProcess
getBusinessProcessList	setBusinessProcessList	deleteBusinessProcessList
getFactoryInformation	setObservation	deleteObservation
getObservationByID	setSample	deleteSample
getSampleByID		

How OGC SensorThings data is persisted and queried is specified in the OGC SensorThings Sensing Profile API (Liang, Huang , & Khalafbeigi, 2016).

5.4.3 Data Flow

5.4.3.1 High-level data flow

The Digital Factory Model (DFM) (described in D3.2 “Digital Factory Model I”) is the common source for information about the factory equipment and processes for all COMPOSITION components. Static and dynamic data provided from the COMPOSITION system are described in a common format using the DFM schema. The machines, devices and sensors in the factory instance are described in a Deployment Model; this also contains the mapping of these resources to a specific IoT data channel, such as a MQTT topic or REST endpoint. The DFM provides interfaces that other components use for reading and updating the models.

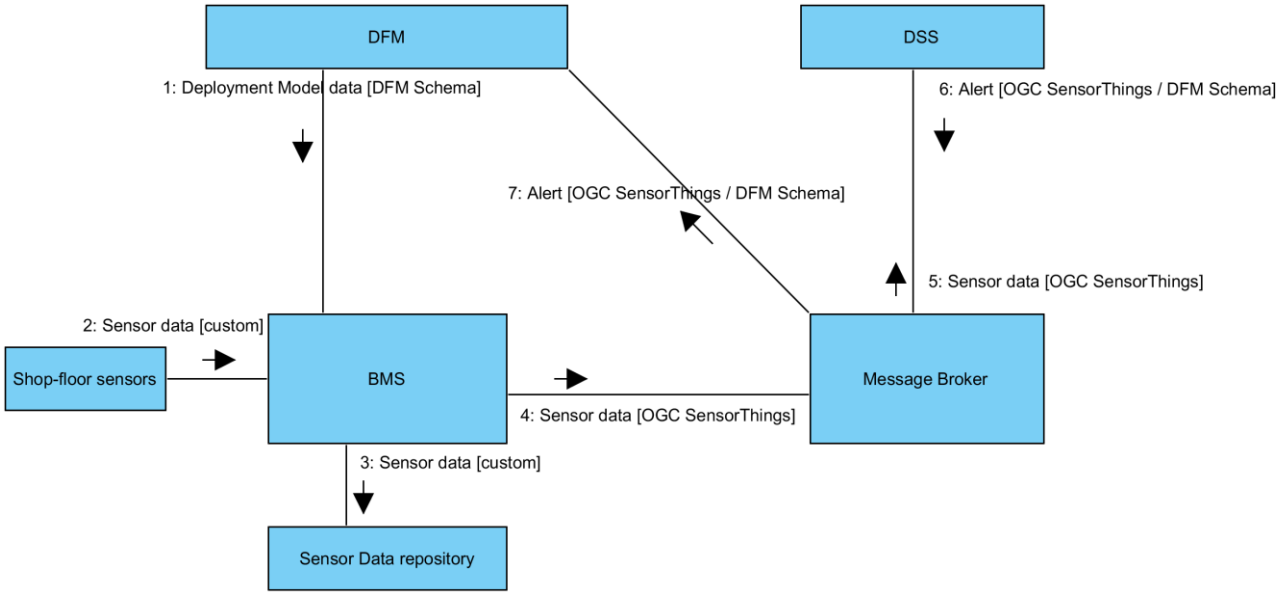

Figure 39: Example Intra-factory data flow

The format chosen for sensor data in COMPOSITION is SensorThings API Sensing Entities[47] JSON encoding. The BMS will deliver data from sensors and other shop-floor sources to the Message broker in this format. Information about the context of the data from the DFM will be added by the BMS Object Mapper. The data will be published on a MQTT topic structure adapted from the SensorThings Sensing MQTT Extension which allows subscribers to be notified when Observations are added to a Datastream or FeatureOfInterest.

Data consumers may subscribe to these topics to receive the sensor data. Components like the Deep Learning Toolkit (DLT) are configured at deployment to subscribe (mediated via the BDA) to specific data streams.

The Decision Support System (DSS) will dynamically visualize factory processes and will benefit from subscribing to annotated data from a topic where data on an entire process or asset is published. The IoT Agent in the Big Data Analytics (BDA) package may be used to annotate and re-publish data on a MQTT topic structure that includes information from the DFM on e.g. the process involved. Data generated microservices or other system components may also be published on such topics.

The data flow between the BDA and DLT has been integrated as one component and is no longer part of the external interfaces of these components.

5.4.3.2 Decision Support System

Data comes in the DSS component following the streaming process for the whole project. This data does not create any new data for the DSS and they are only used for predictions and KPIs. Data created by the rule engine stays only in the buffer as long as the conditions are valid, or the user decides against the suggested actions. Notifications mechanism only buffer internally the notifications, for as long as it is needed. DSS should be able to store tasks and users in the DFM following the schema provided and connecting with it using MQTT topics on the message broker.

JSON format is used for both incoming and outcoming data. The format follows the schema described for the project and is compatible for the whole project. Incoming data should also be stored in the DFM from the previous components.

Table 6: Data sources for DSS by use case

COMPONENTS	USE CASES
SFT + DFM + BMS	UC – KLE-1
WSN + BMS	UC – KLE-1
DLT + BMS	UC – BSL-2

[47] http://docs.opengeospatial.org/is/15-078r6/15-078r6.html

WSN + BMS	UC – BSL-2

5.4.3.3 Simulation and Forecasting Tool

As described in 5.3.8.2, the data comes to the Simulation and Forecasting Tool component from the Sensors, BMS and DFM results. The end user selects the input parameters based on decisions from DSS and visualizations from VA components. The SFT interacts with DFM, BMS and sensors so as to send results (DFM) and get new inputs (DFM, BMS and sensors) and with DSS/VA to provide the final results for the specified set of parameters and get new decisions. The interactions between DSS/VA, SFT, DFM, BMS, Sensors and end user are presented below:

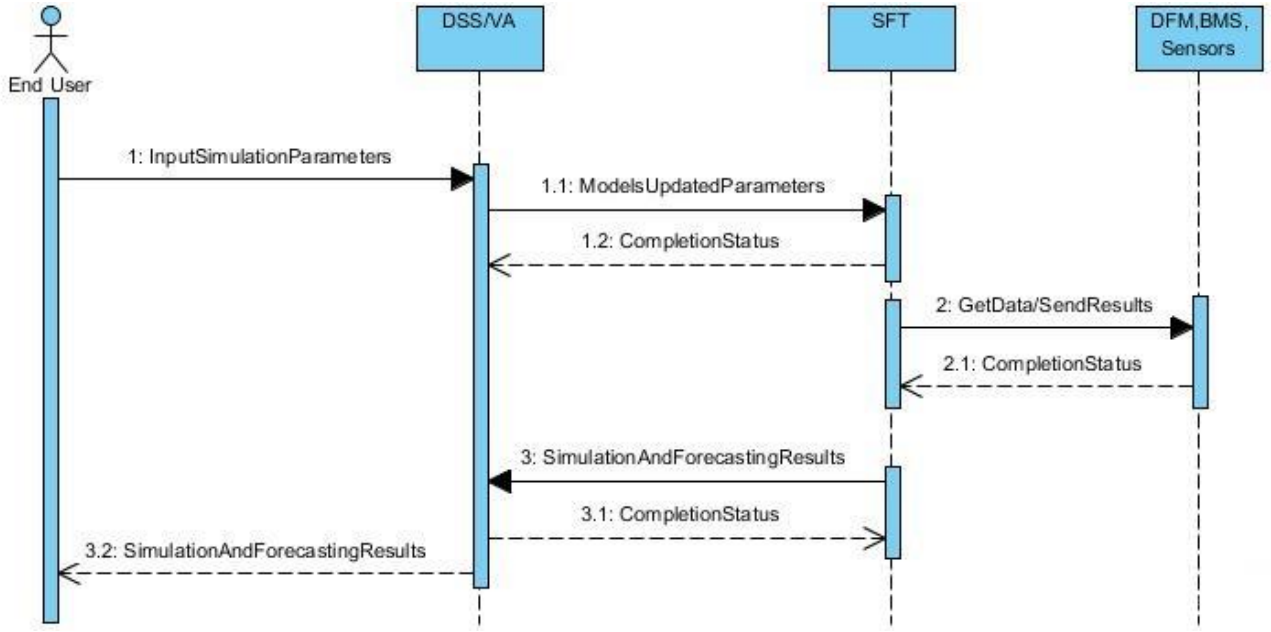

Figure 40: Sequence diagram of COMPOSITION Simulation and Forecasting Tool

5.4.3.4 Matchmaker

As described in 5.3.14.2, the data comes to the Matchmaker component from the Marketplace Agents' requests. The agents are responsible to add instances to the ontology and to trigger matchmaking processes by their provided requests. An agent request to Ontology Querying API is just a call and a response. However, the interaction of the agents and the Matchmaker during a bidding process in the Marketplace is more complex and it is presented below:

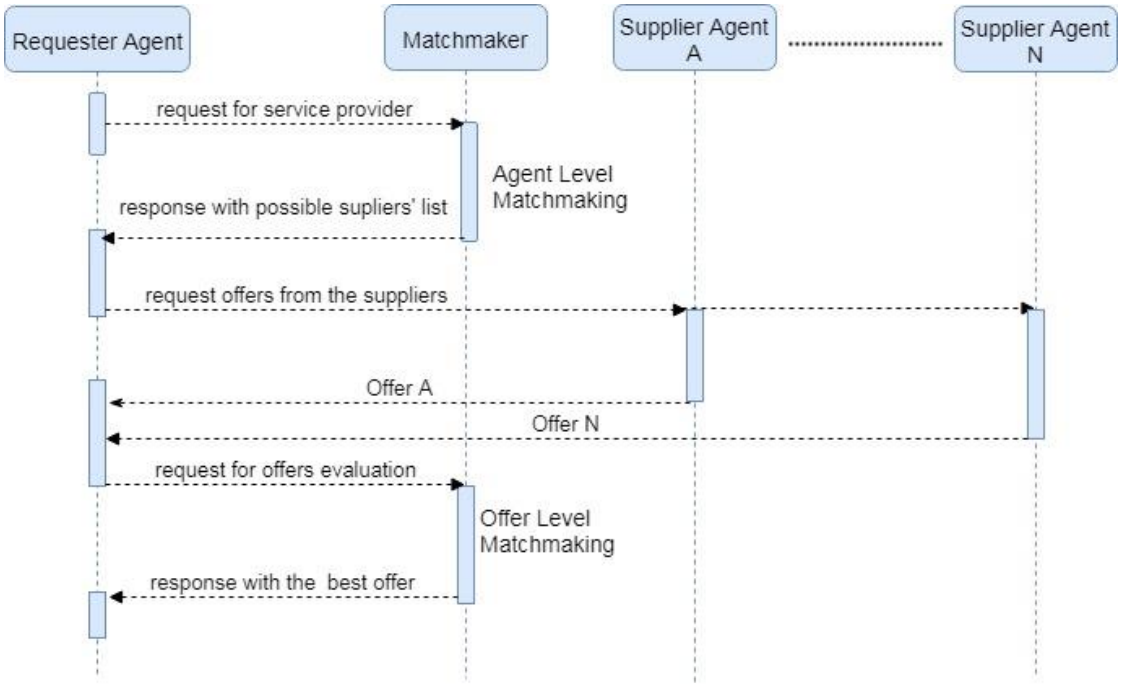

Figure 41: Sequence diagram of main interactions of COMPOSITION Matchmaker

All the data are exchanged in JSON format by using HTTP protocol.

5.4.3.5 Agent Management System

The data flow between the Agent Service component and the database underlying the White Pages Service is described in Figure 42 for the three main scenarios (insertion, deletion and update of an agent on the database).

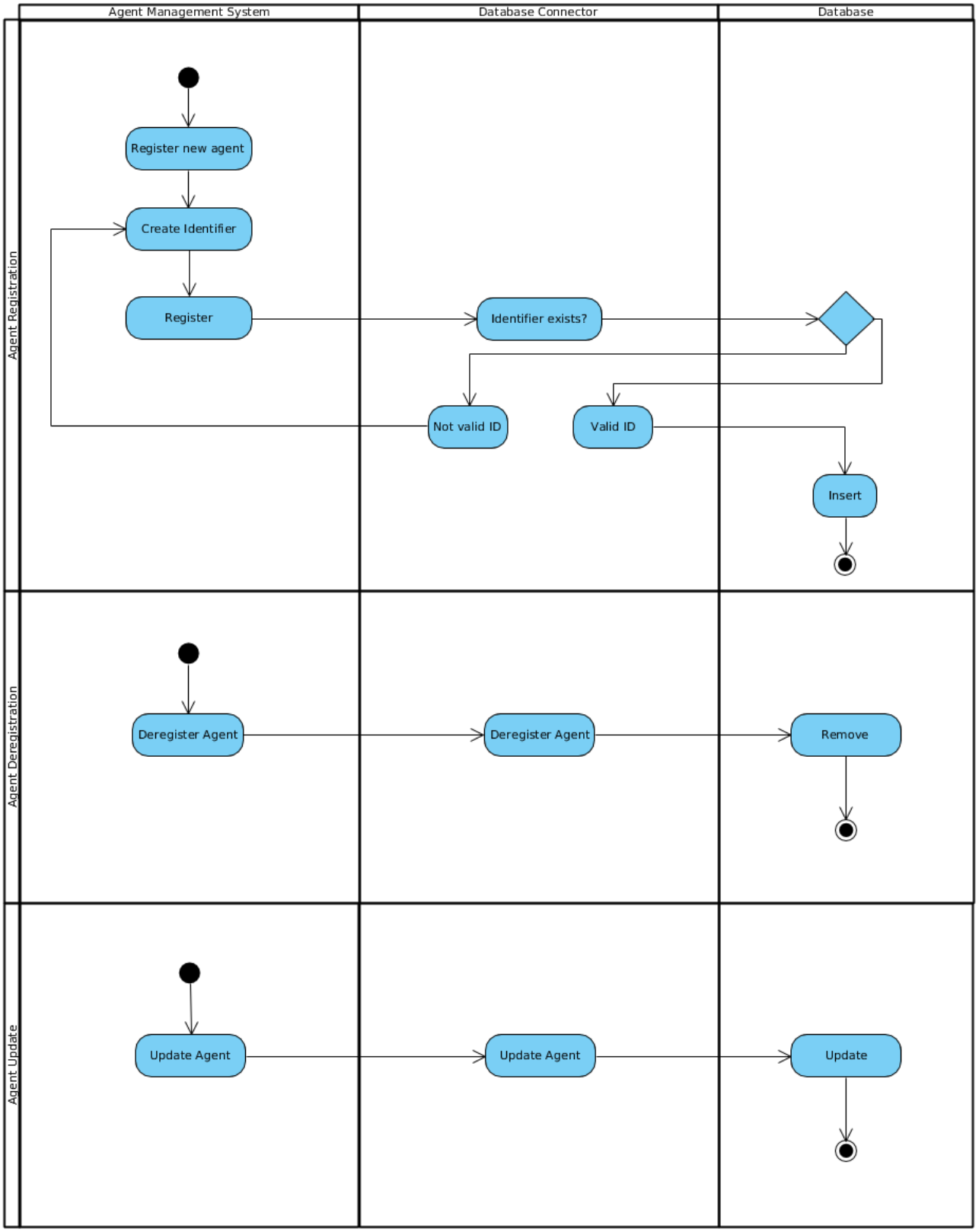

Figure 42: Data flow between AMS and underlying Database

The current deployment of the Agent Management System provides a proxy service towards the Matchmaker agent for the requests coming from the agents on the marketplace. This implementation allows to have a finer-grained control over the requests and having the Matchmaker agent deployed anywhere, without need for the

agents to know its address during the initialization phase. The data flow for a generic request for a Matchmaker service coming either from a Requester or Supplier agent is shown in Figure 43.

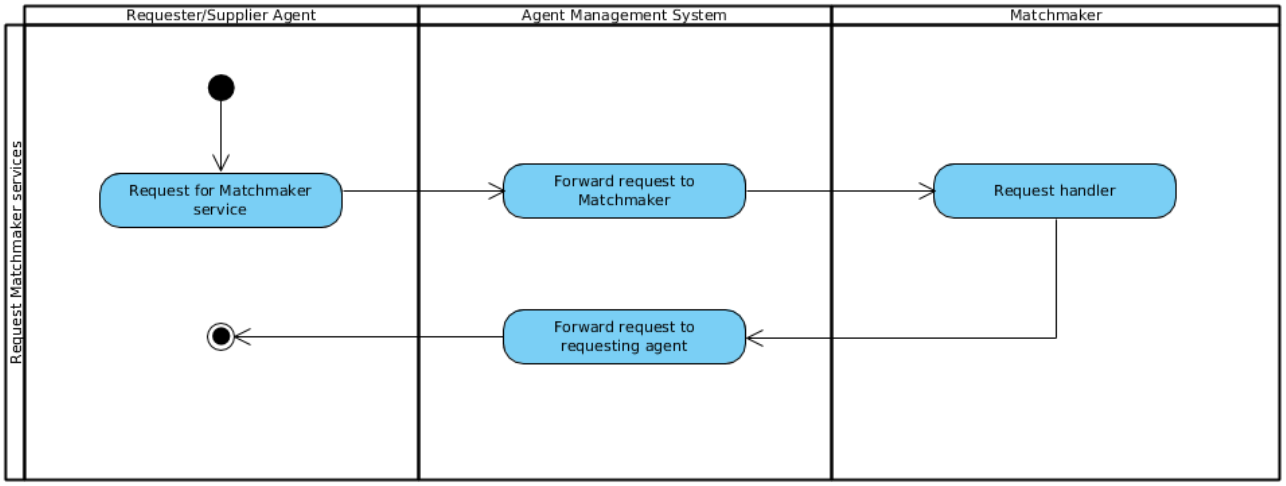

Figure 43: Data flow between Requester/Supplier agent, AMS and Matchmaker

5.4.3.6 Supplier Agent

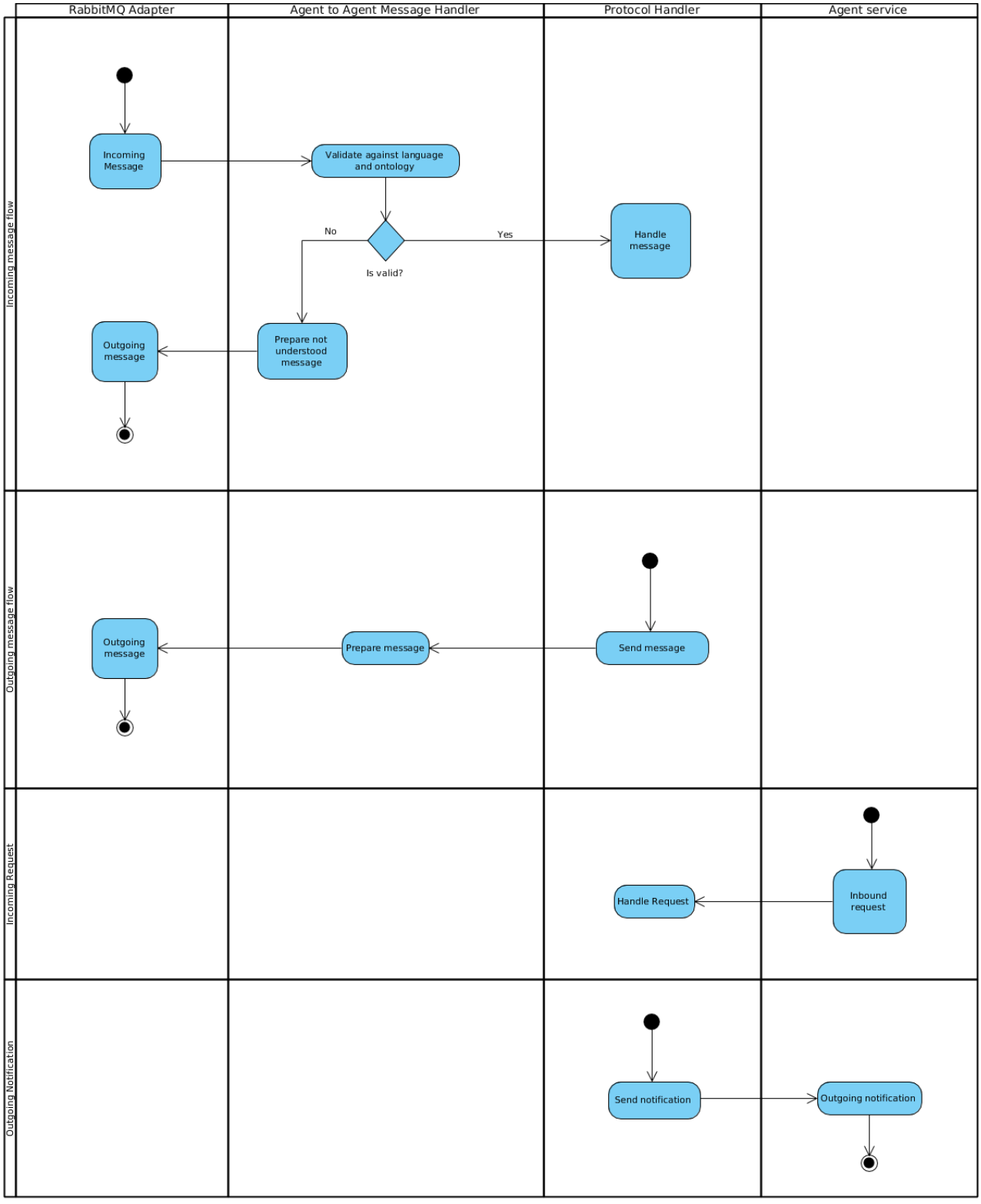

Figure 44: Internal Supplier Agent data flow

The communication between the agent and the GUI consists of two flows:

- Actions: commands sent from the GUI to the agent
- Notifications: notifications sent from the agents to the GUI

The schema for the actions is the following:

```
{
 "description":"An action request sent by a GUI to the associated agent",
 "type":"object",
 "properties":{
  "session_id":{
   "type":"string"
  },
  "agent_role":{
   "type":"string",
   "enum":["requester","supplier"]
  },
  "action":{
   "type":"string",
   "enum":["withdraw","selected_option","confirmed","rejected","start_bid"]
  },
  "payload":{
   "selected_option":{
    "type":"object",
    "properties":{
     "price":{
      "type":"number"
     },
     "currency":{
      "type":"string",
      "enum":["EUR","USD"]
     },
     "company":{
      "type":"string"
     },
     "rating":{
      "type":"number",
      "minimum":0,
      "maximum":5
     },
     "quantity":{
      "type":"number"
     },
     "quantity_uom":{
      "type":"string"
     },
     "good":{
      "type":"string"
     }
    }
   }
  }
 },
 "additionalProperties": false
}
```

The schema for the notifications is the following:

```
{
  "description":"Notification sent by an agent to the corresponding GUI, may be replied with a withdraw
action request",
  "type":"object",
  "properties":{
    "session_id":{
      "type":"string"
    },
    "agent_role":{
      "type":"string",
      "enum":["requester","supplier"]
    },
    "agent_owner":{
      "type":"string"
    },
    "sender_owner":{
      "type":"string"
    },
    "notification_type":{
      "type":"string",
      "enum":["withdrawable","confirmable","selection","info","ack"]
    },
    "status":{
      "type":"string"
    },
    "result":{
      "type":"string"
    },
    "payload":{
      "type":"object",
      "description": "Variable payload according to the notification"
      }
    }
  },
  "additionalProperties":false
}
```

COMPOSITION eXchange Language is used by the agent for communicating with other agents on the marketplace. Details about the language are in Section 5.4.1.4.

By default, the agent stores locally all the messages that have been sent or received, in plain format as they have been received. Future implementations will grant user an easier access to such logs through a proper GUI.

5.4.3.7 **Requester Agent**

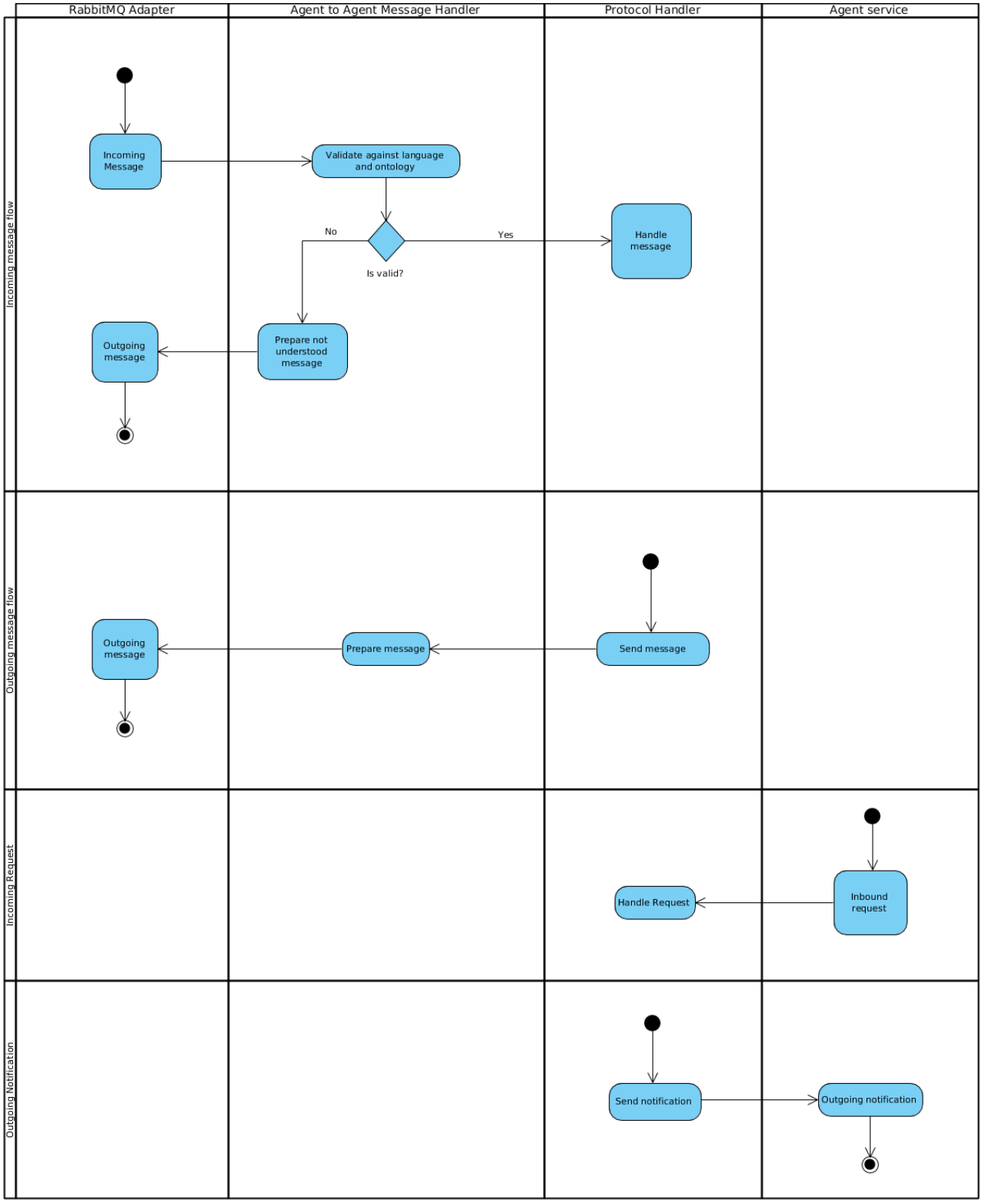

Figure 45: Internal Requester Agent data flow

In the typical protocol flow, Requester agent receives a request from IIMS to start a new negotiation session. The schema for such triggering message is the following:

```
{
  "description":"Trigger message sent by intra-factory toolchains to the requester agent",
```

```
  "type":"object",
  "properties":{
    "action":{
      "type":"string",
      "enum":["start_bid"]
    },
    "quantity":{
     "description" : "The quantity of the good to be traded"
      "type":"number"
    },
    "quantity_uom":{
     "description" : "The quantity unity of measure."
     "type":"string"
    }
  }
}
```

The communication between the agent and the GUI consists of two flows:

- Actions: commands sent from the GUI to the agent
- Notifications: notifications sent from the agents to the GUI

The schema for the actions is the following:

```
{
  "description":"An action request sent by a GUI to the associated agent",
  "type":"object",
  "properties":{
    "session_id":{
      "type":"string"
    },
    "agent_role":{
      "type":"string",
      "enum":["requester","supplier"]
    },
    "action":{
      "type":"string",
      "enum":["withdraw","selected_option","confirmed","rejected","start_bid"]
    },
    "payload":{
      "selected_option":{
        "type":"object",
        "properties":{
          "price":{
            "type":"number"
          },
          "currency":{
            "type":"string",
            "enum":["EUR","USD"]
          },
          "company":{
            "type":"string"
```

```
        },
        "rating":{
          "type":"number",
          "minimum":0,
          "maximum":5
        },
        "quantity":{
          "type":"number"
        },
        "quantity_uom":{
          "type":"string"
        },
        "good":{
          "type":"string"
        }
       }
     }
   }
  },
  "additionalProperties": false
}
```

The schema for the notifications is the following:

```
{
  "description":"Notification sent by an agent to the corresponding UI, may be replied with a withdraw
action request",
  "type":"object",
  "properties":{
    "session_id":{
      "type":"string"
    },
    "agent_role":{
      "type":"string",
      "enum":["requester","supplier"]
    },
    "agent_owner":{
      "type":"string"
    },
    "sender_owner":{
      "type":"string"
    },
    "notification_type":{
      "type":"string",
      "enum":["withdrawable","confirmable","selection","info","ack"]
    },
    "status":{
      "type":"string"
    },
    "result":{
      "type":"string"
```

```
    },
    "payload":{
      "type":"object",
      "description": "Variable payload according to the notification"
      }
    }
  },
  "additionalProperties":false
}
```

COMPOSITION eXchange Language is used by the agent for communicating with other agents on the marketplace. Details about the language are in Section 5.4.1.4.

By default, the agent stores locally all the messages that have been sent or received, in plain format as they have been received. Future implementations will grant user an easier access to such logs through a proper GUI.

5.4.3.8 **Marketplace Data Sharing**

COMPOSITION will provide mechanisms to share data from the intra-factory IIMS with other stakeholders in the marketplace. A factory may choose to share certain data with partners across the supply chain on a permanent basis or a single interaction, e.g., inventory data or scrap container fill levels. The sdata owner agent will route required information to the right recipient agents, through dedicated CXL messages. A sequence diagram illustrating the negotiation between agents using CXL to set up the data exchange can be seen in Figure 46.

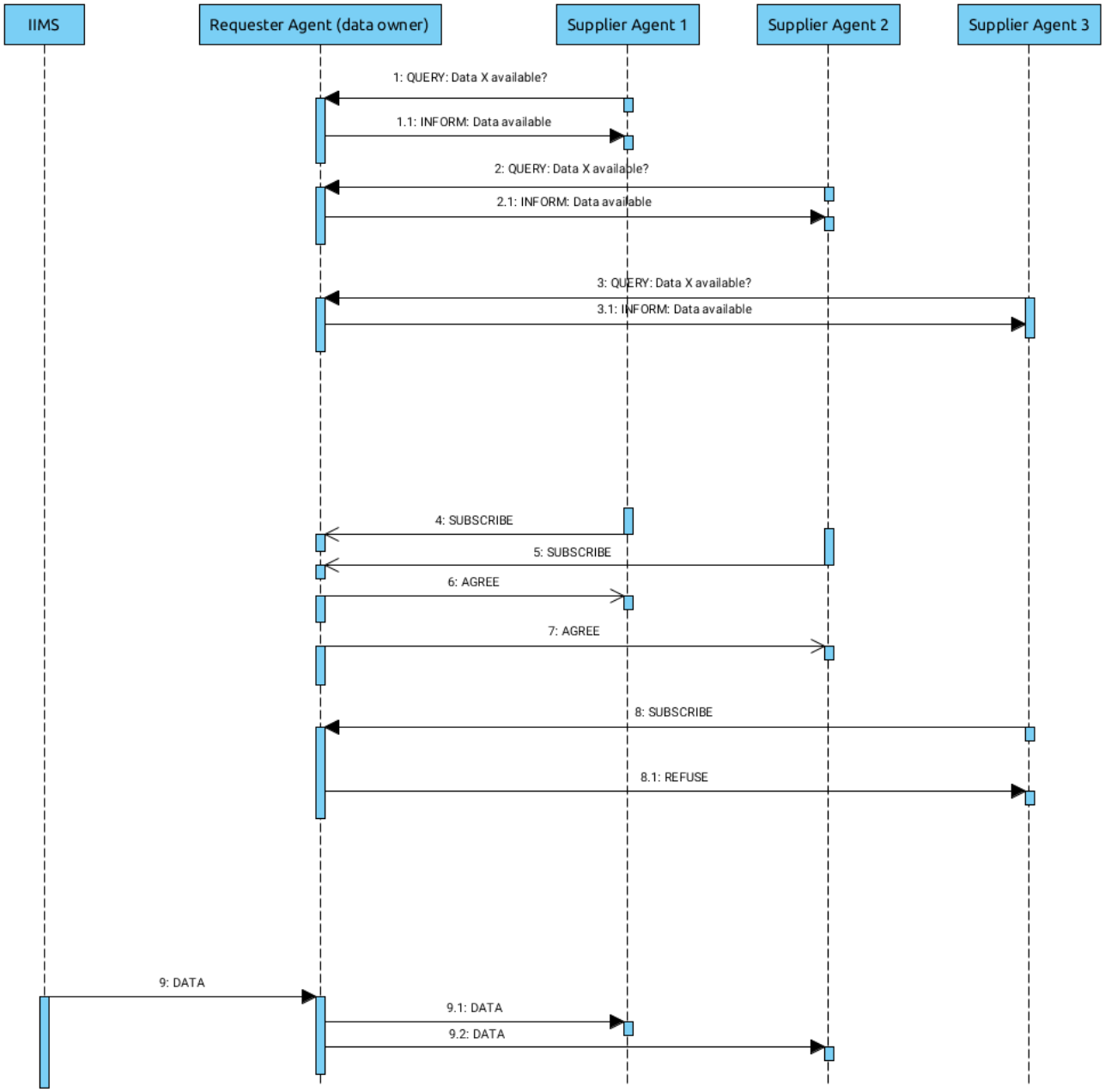

Figure 46: Data routing information flow

The data sharing mechanism is realized through the Message Broker (Figure 47). Integrated access control provided by the Security Framework makes it possible to set up an exclusive message queue for a business partner at the Marketplace Broker. Only the approved actors in the marketplace may publish and/or read data from the queue. The messages sent can also be secured by the possibility to store a hash of each message in the distributed blockchain ledger. As with all CXL messages, the agreement to share data itself may also be stored in the ledger to keep a non-repudiable audit trail of agreements.

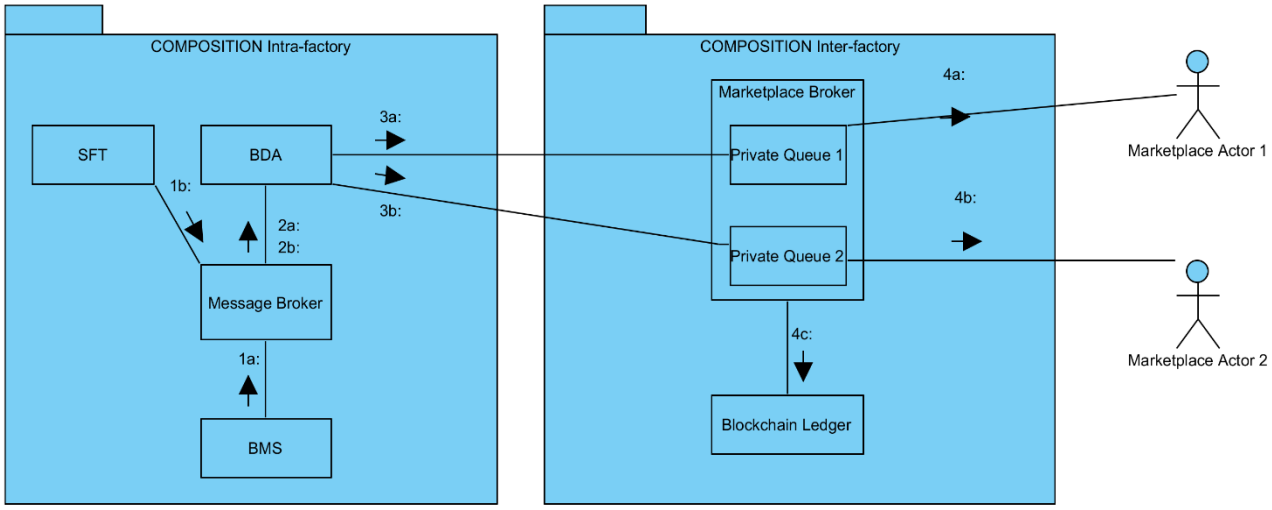

Figure 47: Simplified model of the marketplace data exchange design

5.5 Deployment View

The purpose of the deployment view is to describe the environment in which the system will be deployed, how components are mapped to deployment nodes, the requirements for each component, and the mapping of the software elements to the runtime environment that will execute them.

5.5.1 Docker

One of the critical points in adopting new systems in productive contexts is the need to perform specific hardware and software set-up, which are typically difficult to deploy, as companies have precise software deployment policies, rather strict options on operating systems and public access to company IT services. These restrictions are strongly dependent on company-level decisions and are the result of years of operation in real business.

In COMPOSITION, we have clear in mind that any particular technological requirement for the COMPOSITION IIMS and Marketplace may hamper or slow down adoption of the platforms in the real world. Therefore, after a careful evaluation of possible solutions, included PAAS and SAAS solutions (which on the other hand could be difficult to handle due to data ownership issues), the technical partners, in accordance with industrial stakeholders, identified Docker as a viable deployment infrastructure.

Docker is an open-source project aiming at automating the deployment of applications as portable, self-sufficient containers that can run virtually anywhere, on any kind of server. It can be considered as a lightweight alternative to full machine virtualization provided by hypervisors such as ESXi, Xen or KVM. While in the traditional hypervisor approaches each virtual machine (VM) needs its own operating system, in Docker applications operate inside a container that resides on a single host operating system that can serve many different containers at the same time.

Docker containers are designed to run on a wide range of platforms ranging from physical computers to bare-metal servers and up to cloud clusters, e.g., based on OpenStack. Technically speaking Docker extends the LinuX Containers (LXC) format designed to provide an isolated environment for applications, by enabling image management and deployment services. Among supported platforms, we can cite:

- Mac, Windows and Linux desktops
- AWS and Azure cloud services
- Windows, CentOS, Debian, Fedora, Oracle Linux, RHEL, SLES and Ubuntu servers.

This ensures the ability to deploy Docker-based COMPOSITION components on virtually all possible IT infrastructure available on site. Since deployment is a crucial part of the agile development process adopted in COMPOSITION, components are wrapped into Docker images since the very beginning. All continuous integration and testing processes in the project rely on Docker and act upon Docker images. This ensures full compatibility of systems under development with the targeted deployment tools.

Thanks to a dedicated web management tool, namely Portainer[48], also deployed as a Docker container, partners and in general all technical stakeholders have the ability to publish, run and test the single COMPOSITION components under their respective responsibility. Continuous monitoring and logging infrastructure allow deep analysis of the performances of deployed software that can both be carried before the final deployment inside factories and during real-world operation.

Docker natively supports distribution and replication of services. Moreover, it can easily be deployed on cloud-based platforms. This flexibility is a strong hint to the fact that such a deployment design choice will not generate issues when upscaling of performance will be required.

5.5.2 COMPOSITION Production Deployment

The original design of COMPOSITION architecture envisioned the deployment of an Inter-Factory server on the cloud and instances of Intra-Factory servers on premise infrastructures within use-case factories. Deployment of Intra-Factory components on premise provides full control of resources to data owners and significantly reduces centralized computational, networking, and storage requirements. Even though this remains the recommended approach, the consortium decided to offer Intra-Factory on the cloud for the following reasons:

- End-users often lack resources to setup and maintain server infrastructures.
- End-users are often reluctant to provide remote access to their server infrastructure due to security and privacy concerns.
- Changes to existing server infrastructure to satisfy the requirements for COMPOSITION pilots is usually infeasible because the effects can influence daily factory operations.

As of writing, the Inter-Factory server operate purely on the cloud and Intra-Factory ones offer infrastructure flexibility between cloud and on-premise to match different deployment requirements. Figure 48 illustrates the deployment view of COMPOSITION ecosystem.

[48] http://portainer.io/

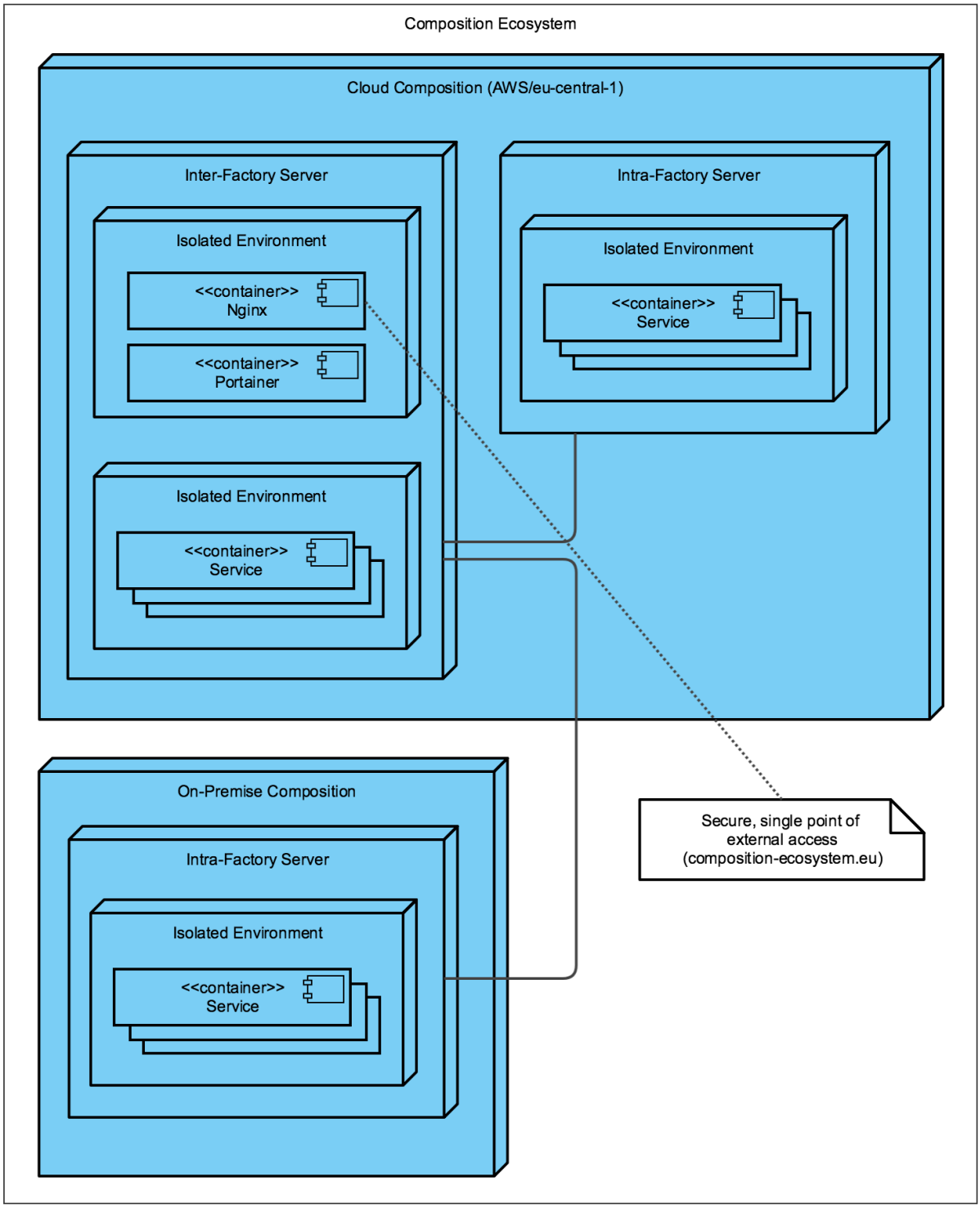

Figure 48: Current COMPOSITION production servers: all components are deployed as Docker containers, external traffic is secured by TLS

Cloud COMPOSITION

The COMPOSITION Ecosystem uses Amazon Web Services (AWS)[49] as the infrastructure for cloud components of the system. All selected computing, storage, and networking AWS resources operate in

[49] https://aws.amazon.com/

Frankfurt (eu-central-1) region, providing low latency across Europe. AWS guarantees data privacy[50] and is compliant to European Union's General Data Protection Regulation (GDPR)[51].

The current specifications of Inter- and Intra-Factory instances are described in Table 7. The resources are selected based on current requirements and can be easily expanded to support larger scale of deployments. The instance types are T2 Elastic Compute Cloud (EC2), general purpose instances with a monthly uptime percentage of at least 99.99%[52]. The CPU and RAM are bound to the instance type and can be changed with zero-migration efforts according to the demand. The secondary storage is Elastic Block Store (EBS)[53] that is automatically replicated within the availability zone (eu-central-1a) to protect the system from component failure, offering high availability and durability. In addition, there a backup system is set to take snapshots of data volumes, nightly kept for 7 days on S3[54], a highly-durable, available, and scalable storage system.

Table 7: Specifications of AWS resources for Inter- and Intra-Factory servers.

Name	Inter-Factory Server	Intra-Factory Server
Instance Type	t2.medium	t2.small
CPU	2 vCPU	1 vCPU
CPU Credits	24 per hour[55]	12 per hour
RAM	4GB	2GB
Secondary Storage	8GB SSD (root) 100GB SSD (data)	8GB SSD (root) 50GB SSD (data)
Snapshots	7 x data volume (nightly)	7 x data volume (nightly)
Static IPs	1	1
Operating System	Amazon Linux 2	Amazon Linux 2
Domain	inter.composition-ecosystem.eu	intra.composition-ecosystem.eu

The servers support network traffic of up to 5 Gbps for single-flow traffic or 25 Gbps for multi-flow traffic within the AWS region.[56]

All cloud software components are deployed as Docker containers to improve portability and isolation. The Inter-Factory server additionally hosts instances of Portainer[57] and Nginx[58]. Portainer provides an interface to manage Docker container across all servers. Nginx is the entry point to COMPOSITION Ecosystem, securing all the traffic by TLS (Let's Encrypt certificate) and proxying requests to appropriate servers and components based on subdomains and URL paths.

5.5.3 Digital Factory Model

The DFM API is deployed in a Glassfish[59] server. A Docker image for this server has been built with the DFM API deployed on it. The Docker container contains the aforementioned image communicates with a Docker container that contains a MongoDB Docker image in order to enable the connection between the API and the data base. Based on Glassfish documentation [60]the DFM API can receive over than 256 concurrent requests as it is depended on server's capabilities. Furthermore, the used MongoDB[61] is able to store about 32TB of data which is considered more than enough for static data and the continuously updated data (document type) of the prediction tools.

[50] https://aws.amazon.com/compliance/data-privacy-faq/
[51] https://aws.amazon.com/compliance/gdpr-center/
[52] https://aws.amazon.com/compute/sla/
[53] https://aws.amazon.com/ebs/
[54] https://aws.amazon.com/s3/
[55] One CPU credit is equal to one vCPU running at 100% utilization for one minute. When CPU credits are unused, they accumulate for up to 24-hours and can be consumed during CPU intensive burst operations.
[56] https://aws.amazon.com/blogs/aws/the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/
[57] https://portainer.io/
[58] https://www.nginx.com/
[59] http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html
[60] https://docs.oracle.com/cd/E19879-01/820-4343/abefk/index.html
[61] https://docs.mongodb.com/manual/reference/limits/

5.5.4 Agent Management System

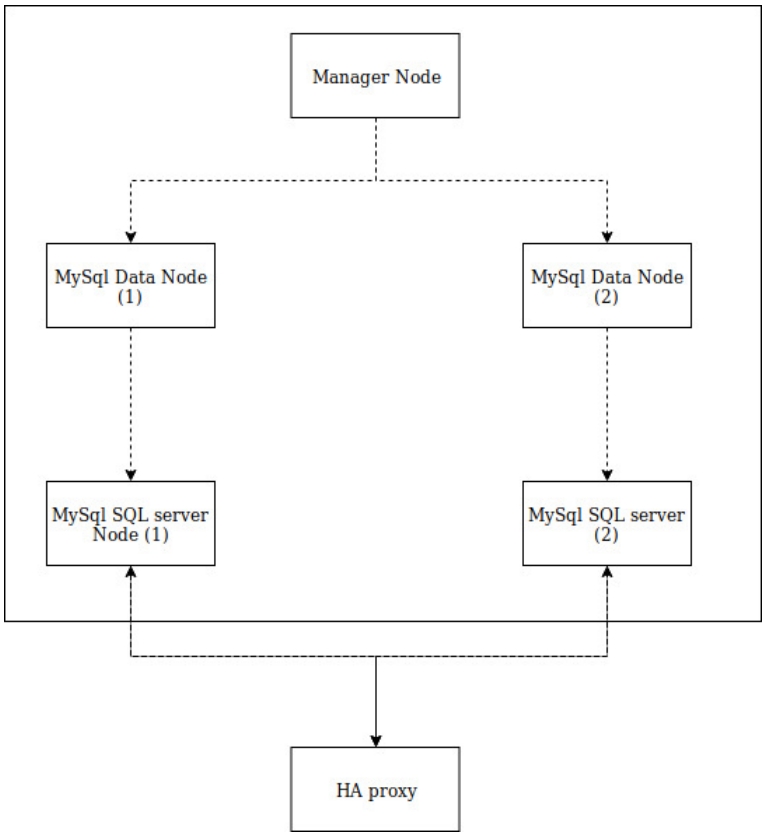

Figure 49: White Pages Deployment View

As shown in Figure 49, the current deployment foresees:

- A Manager Node in charge of managing the other nodes within the Cluster, performing such functions as providing configuration data, starting and stopping nodes, and running backups.
- Two MySql Data Nodes, storing the cluster data.
- Two MySql Server Nodes accessing the cluster data. They are actually specialized types of API node, which designate any application which accesses Cluster data
- An HAProxy to provide load balancing between incoming requests, in order not to overload a MySql Server Node.

Since it is recommended to have 2 (or more) replicas to provide redundancy (and thus high availability), the current deployment provide this minimal set. However, to support continuous operation, MySQL Cluster allows on-line addition of nodes and updates to live database schema to support rapidly evolving and highly dynamic workloads.

The Agent Management system can be deployed with a set of Docker containers:

- Agent Management System: ~300Mb
- 2 MySql data nodes: ~300Mb (150Mb each)
- 2 MySql Server nodes: ~300Mb (150Mb each)
- 1 MySql Manager node: ~150Mb
- 1 HAProxy: ~300Mb

The total amount of storage required with this configuration is ~1.5Gb, and it is important to notice that this is the smallest deployment guaranteeing high availability, scalability and reliability. If more data/server nodes are required, the amount of storage will increase accordingly.

No special requirements regarding CPU and RAM have emerged by analysing the statistics from the test server.

5.5.5 Supplier agent

A Supplier Agent can be deployed with a single Docker container, ~500 Mb.

No special needs regarding CPU and RAM have emerged by analysing the statistics from the test server, with an average use of 0% CPU, 40 Mb RAM.

A connection to the RabbitMQ (or other broker supporting AMQP) is needed to communicate with other agents on the marketplace. Further implementations might exploit an MQTT broker as well.

In order to store all the messages that have been sent or received by the agent, an appropriate quantity of storage must be allocated.

5.5.6 Requester Agent

A Requester Agent can be deployed with a single Docker container, ~500 Mb.

No special needs regarding CPU and RAM have emerged by analysing the statistics from the test server, with an average use of 0% CPU, 40 Mb RAM.

A connection to the RabbitMQ (or other broker supporting AMQP) is needed to communicate with other agents on the marketplace. Further implementations might exploit an MQTT broker as well.

In order to store all the messages that have been sent or received by the agent, an appropriate quantity of storage must be allocated.

5.5.7 Decision Support System

The DSS component is a dockerized web–based application which will be deployed in the intra-factory server (or on the shop floors to help decision – making process. Runtime requirements are a steady internet connection at 24Mbps. No dedicated server for the application.

- Browser compatibility
- WiFi network with speed at 24Mbps or Ethernet connection
- Physical obstructions arise connectivity issues
- Application on a heavy “noise” environment where connection is frequently lost
    - Experimental application
    - Security integration for shop floor data

Ease of use

5.5.8 Simulation and Forecasting Tool

The SFT component is a web-based component implemented in Python and deployed in Docker containers. Different Docker images are built for the different supported algorithms. The Docker containers of the SFT are deployed at COMPOSITION intra-factory production server.

5.5.9 Matchmaker

The Matchmaker component is a web-based component deployed in a Docker container. The Matchmaker contains the complete semantic framework as it is described in the functional view sub-section. The Docker container of the Matchmaker is deployed at COMPOSITION inter-factory production server.

The Matchmaker framework was developed as a Java EE application and was packaged as a .war file. The Docker image of the Matchmaker was created by using the Docker official image of the Apache Tomcat Server with the addition of the Matchmaker’s .war file. The application servers with stateless applications such as Tomcat is easy to be dockerized and scale easier as each new instance can receive requests without any

synchronization of state. There are no special network requirements as there are no other network bridges and dependences but only the default network requirements of Tomcat. Moreover, there are no third-party software requirements as they are all packaged in the Matchmaker image.

5.6 Operational view

The purpose of the Operational viewpoint is to identify a system-wide strategy for addressing the operational concerns of the system's stakeholders and to identify solutions that address these. For a product development project, the Operational view is more generic and illustrates the types of operational concerns that customers of the product are likely to encounter, rather than the concerns of a specific site. This view also identifies the solutions to be applied throughout the product implementation to resolve these concerns. These concerns pertain to system aspects not explicitly covered in the use cases; such as how the system is set up and administrated during its lifecycle. E.g. how can the functionalities demonstrated in the use cases in the two industrial pilots be applied to other customers. The operational view describes the architectural design to cover the gap between proof-of-concept and product.

In the pilot phase, all development partners monitor and manage the software and hardware installed. Manually monitoring systems is feasible in this environment by Portainer. In production, however, this task will need tool support and a unified management interface that could be managed by a small group of IT management staff. It is the aim of this viewpoint to illustrate how the system is prepared for such tools.

5.6.1 Configuration Management

The system components need to be installed, updated and versioned. In COMPOSITION, all components are kept as versioned docker repositories in Docker registries (Docker Hub[62] is used for all components except LinkSmart which has a dedicated registry). Portainer is used to manage the installed versions of components in the pilot deployment environment. Configuration files can be accessed through Portainer and exposed as Docker volumes on the docker host. This can be combined with a versioning system for the configuration files.

The life-cycles of connected equipment needs to be managed, e.g. sensor hardware providing a specific data stream may need replacing and new equipment will to be added and connected to the factory model. The DFM provides this functionality in COMPOSITION.

5.6.2 Monitoring

The status of components and logs can be inspected through Portainer in the pilot deployment environment. Developing operational monitoring tools is out of the scope of the project, however, the status of components can be reported in two ways that may be used by external tools. First, status messages (start, stop, errors) can be reported on a specific MQTT/AMQP topic to the broker. A system operator can monitor these using simple tools like MQTTfx. Second, components can expose REST endpoints reporting the status of the component simply by returning HTTP status codes in response to a GET request or respond with a more informative JSON payload. There is a wealth of free tools available to automate the monitoring of the status of endpoints, e.g. Postman[63] or PHP Server Monitor[64]. In the exploitation phase, the stakeholder operating the system is likely to already employ an operational monitoring toll (e.g. Microsoft System Center Operations Manager (SCOM)[65]) and the monitoring may be done through this.

5.6.3 Components

This section contains operational view documentation for specific components. This is ongoing work which is not completed at the time of writing and it will be further updated until the end of the COMPOSITION project.

5.6.3.1 Agent Management System

Like other COMPOSITION components, the component can be booted by launching the appropriate containers. Solutions (based on Docker Compose) are being studied in order to provide a smooth deploying

[62] https://hub.docker.com/u/composition/dashboard/
[63] https://www.getpostman.com/docs/v6/postman/monitors/intro_monitors
[64] https://www.phpservermonitor.org/
[65] https://docs.microsoft.com/en-us/system-center/scom/welcome?view=sc-om-1807

experience for the final end user, more details about these configurations will be provided in D6.6 “Connectors for Inter-factory Interoperability and Logistics II” (M34).

To support the monitoring of the component status, the current implementation provides a REST endpoint, located at *<Agent IP Address>/status*, that replies to GET calls. Future developments will provide more advanced and user-friendly solutions.

To guarantee the correct exchange of information about agents’ registrations over the marketplace, a mechanism for verifying the database connection status has been setup on AMS.

5.6.3.2 Requester Agent

Like other COMPOSITION components, the component can be booted by launching the appropriate container, providing an appropriate configuration file containing information such (but not limited to):

- Network configurations (e.g. GUI address, AMS address)
- Policies for market exchange (e.g. price, service, priorities)
- Languages and ontologies supported

More details about these configurations will be provided in D6.6 (M34).

To support the monitoring of the component status, the current implementation provides a REST endpoint, located at *<Agent IP Address>/status*, that replies to GET calls. Future developments will provide more advanced and user-friendly solutions.

5.6.3.3 Supplier agent

Like other COMPOSITION components, the component can be booted by launching the appropriate container, providing an appropriate configuration file containing information such (but not limited to):

- Network configurations (e.g. GUI address, AMS address)
- Policies for market exchange (e.g. price, service, priorities)
- Languages and ontologies supported

More details about these configurations will be provided in D6.6 (M34).

To support the monitoring of the component status, the current implementation provides a REST endpoint, located at *<Agent IP Address>/status*, that replies to GET calls. Future developments will provide more advanced and user-friendly solutions.

5.6.3.4 Matchmaker

Matchmaker component is part of the COMPOSITION Marketplace. It is deployed in Docker inter-factory production server and offers its operations as exposed end points (RESTful web services) to the Marketplace agents. It will be deployed, updated and supported by developers and production server administrators.

5.6.3.5 Decision Support System

DSS will run on the docker server and used as a web application on the shop floor. It will be run by maintenance personnel on the shop floor and maintained by project’s developers with the consent of docker’s administrators. RESTful web services in the backend can provide status information**.**

6 System Quality Perspectives

6.1 Security Perspective

This section describes how end-to-end security is realized in the COMPOSITION system by the COMPOSITION Security Framework, addressed in WP4. The details of the security framework are described in the deliverables D4.1 “Design of the Security Framework I”, D4.2 “Design of the Security Framework II” and D4.4 “Prototype of the Security Framework I”. This section will describe the integration and use of the Security Framework in COMPOSITION.

6.1.1 Authentication and Authorization

The prerequisites stated in the previous version of this document (D2.3 “The COMPOSITION architecture specification 1.1”) regarding authentication and authorization are still valid: two components have been deployed and integrated for the achievement of the authentication (Keycloak) and authorization services (EPICA). Figure 50 shows an overview of the architecture of how the the Authentication and Authorization framework is used.

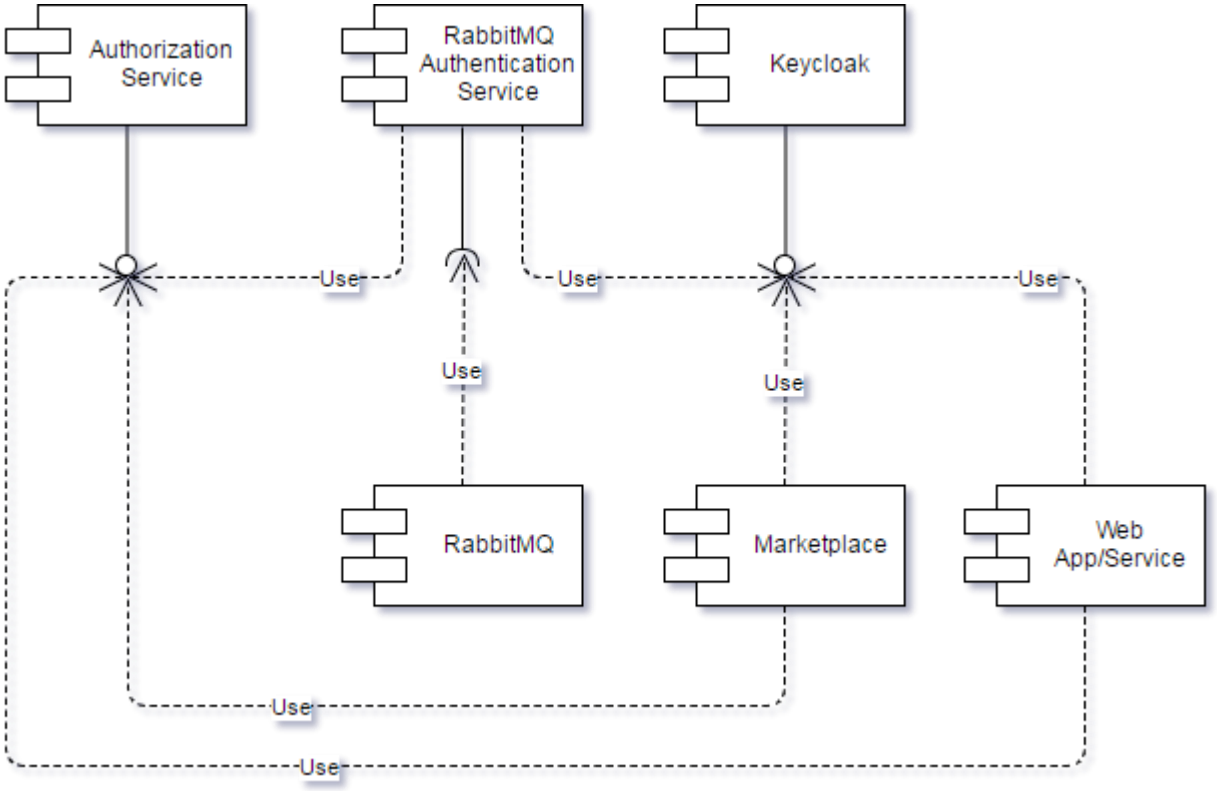

Figure 50: The Authentication and Authorization framework.

Any user or application needs to be identified through Keycloak before having access to the secured COMPOSITION applications and services. Once identification is successful Keycloak issue a token which is valid for a limited time and should be renewed after it expires. This token is also the one used by the Authorization Service to allow or deny access to data and resources based on the rights of the user/application and the rules stored in the Authorization Service.

Furthermore, the messages traffic is dispatched by the Message Broker, sending the appropriate requests and messages to the different components of the Security Framework, via the Reverse Proxy (Nginx component) which isolates the components that are part of the Security Framework from the rest of the COMPOSITION architecture, in order to prevent external intromissions or direct attacks.

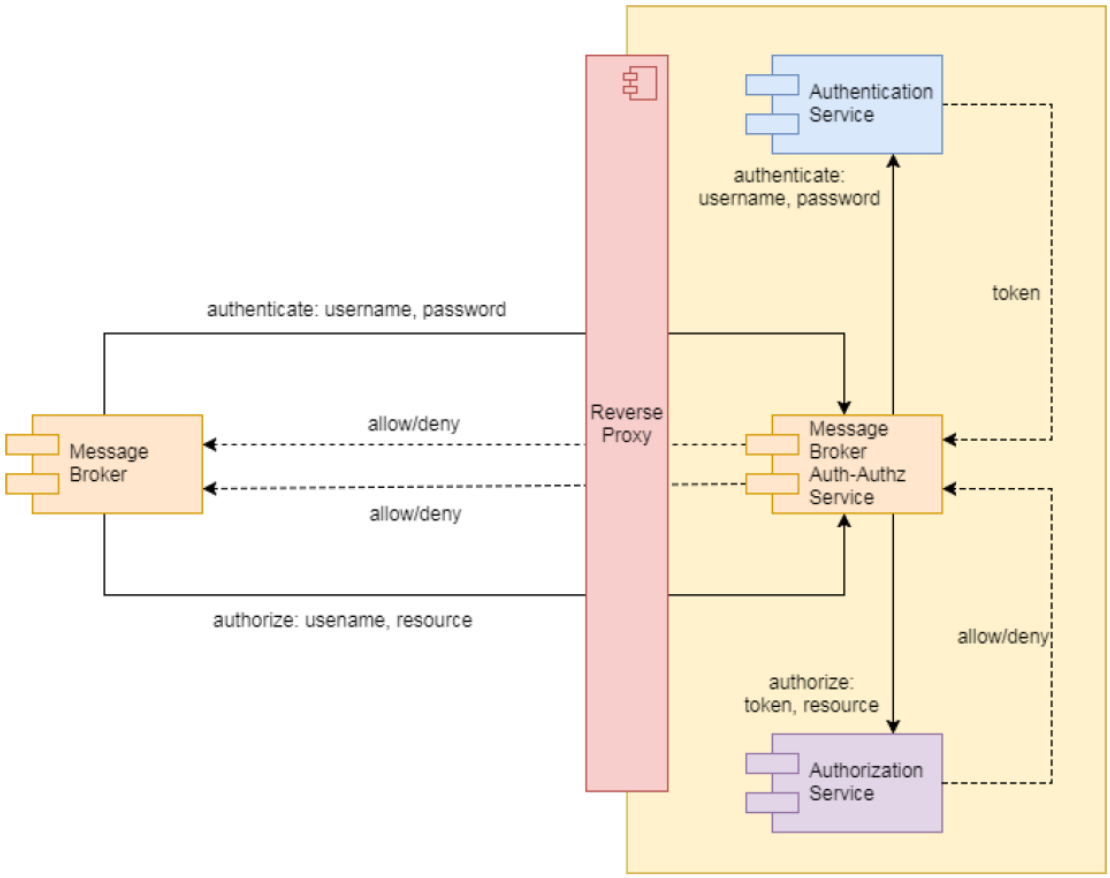

Figure 51: Authorization and authentication schema for the Message Broker

Regarding the technical deployment of the different components, at the moment of writing this document testing instances have been deployed on Atos' servers, planned to be moved to production environments in the later phases of the project in order to assure the integration and seamless aspects of the proposed solution for the securitization.

6.1.2 Blockchain Uses

Other areas where COMPOSITION aims to offer a high level of security are:

- IPR, Confidentiality and Data integrity
- Log and Traceability

For that it has been considered the use of a blockchain implementation; in this case Multichain, which is a private blockchain based on Bitcoin with interesting new features implemented like data streams and managed permissions. Since it´s a private blockchain there is no need for mining which is an important aspect to have into account, as transactions will have no cost if desired.

6.1.2.1 IPR, Confidentiality and Data integrity

In the case of protecting IPR, COMPOSITION proposal is to use the blockchain to get a digital certificate of authentication for any kind of digital document without storing the document itself in anyway in the blockchain. The next figure is an overview of the architecture.

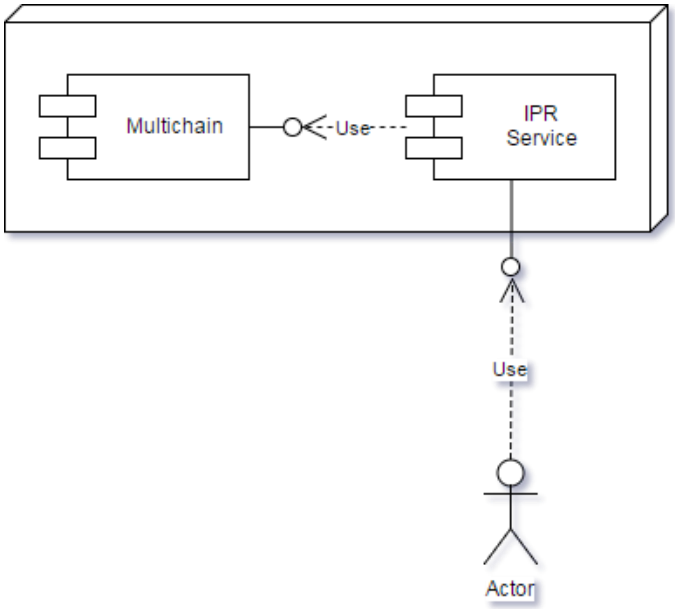

Figure 52: IPR Service

The method to obtain a certificate for a document is pretty simple:

1. Upload document
2. IPR service calculate hash and store in blockchain
3. Return hash

The following figure depicts the process in detail:

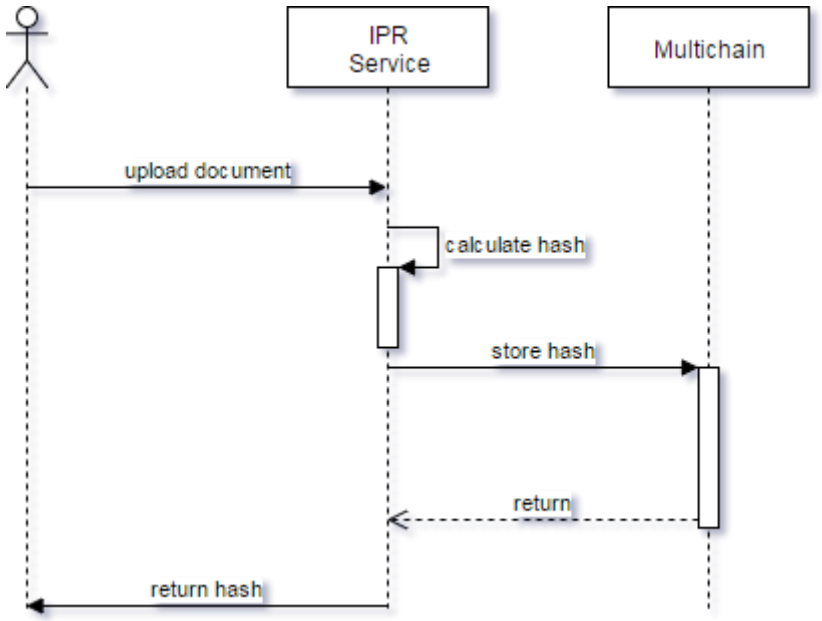

Figure 53: IPR Service sequence diagram

The method to check if a document existed at any given time is fairly simple also. The steps are the following:

1. Upload document
2. IPR service calculate hash

3. IPR service checks hash in block chain
4. Return date if found

To ensure Confidentiality, Data Integrity and also IPR of the messages/data sent across the platform using RabbitMQ message broker Multichain will be used in a similar way as with the certificate of authentication explained before. It´s important to note that, as in the previous case the message or data itself it´s not stored in the blockchain. The architecture can be depicted in the following figure:

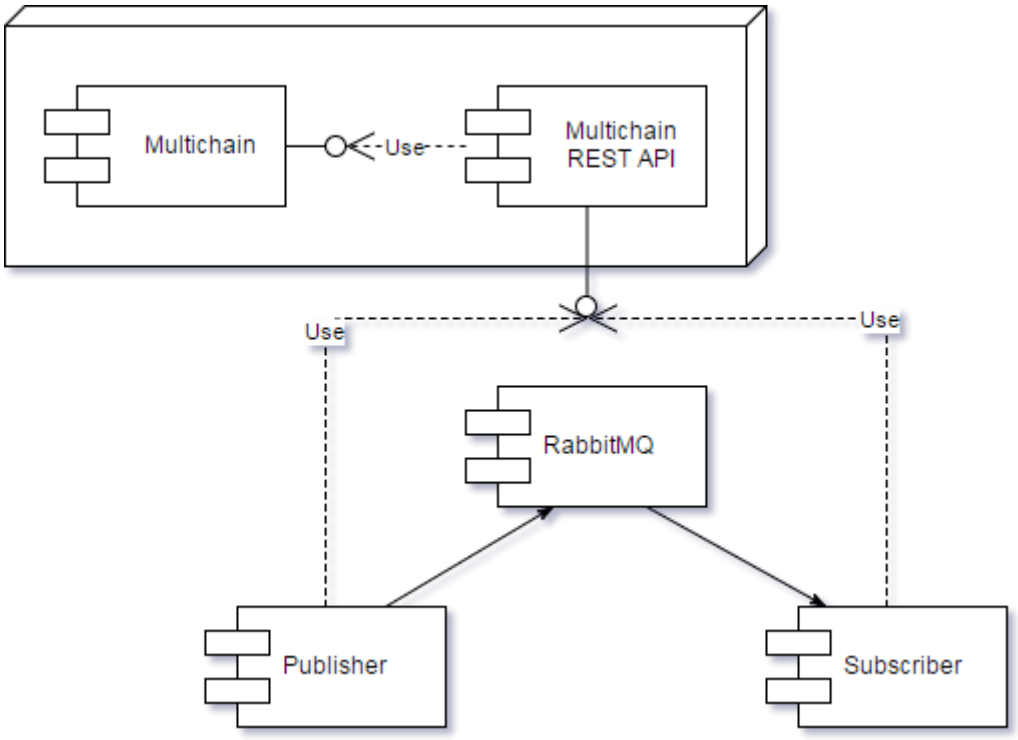

Figure 54: Blockchain used for distributed trust in messaging

Before sending any message/data a publisher must first sign the message/data using a service created for that purpose. Afterwards it can send the message using RabbitMQ message broker. Any subscriber receiving the message can check if the data has been modified in any way and ensure that is coming from where it is assumed. This is done by uploading the message/data in the service which will calculate the hash and will check if the same hash it´s already in the blockchain. The following figure depicts the whole procedure in detail:

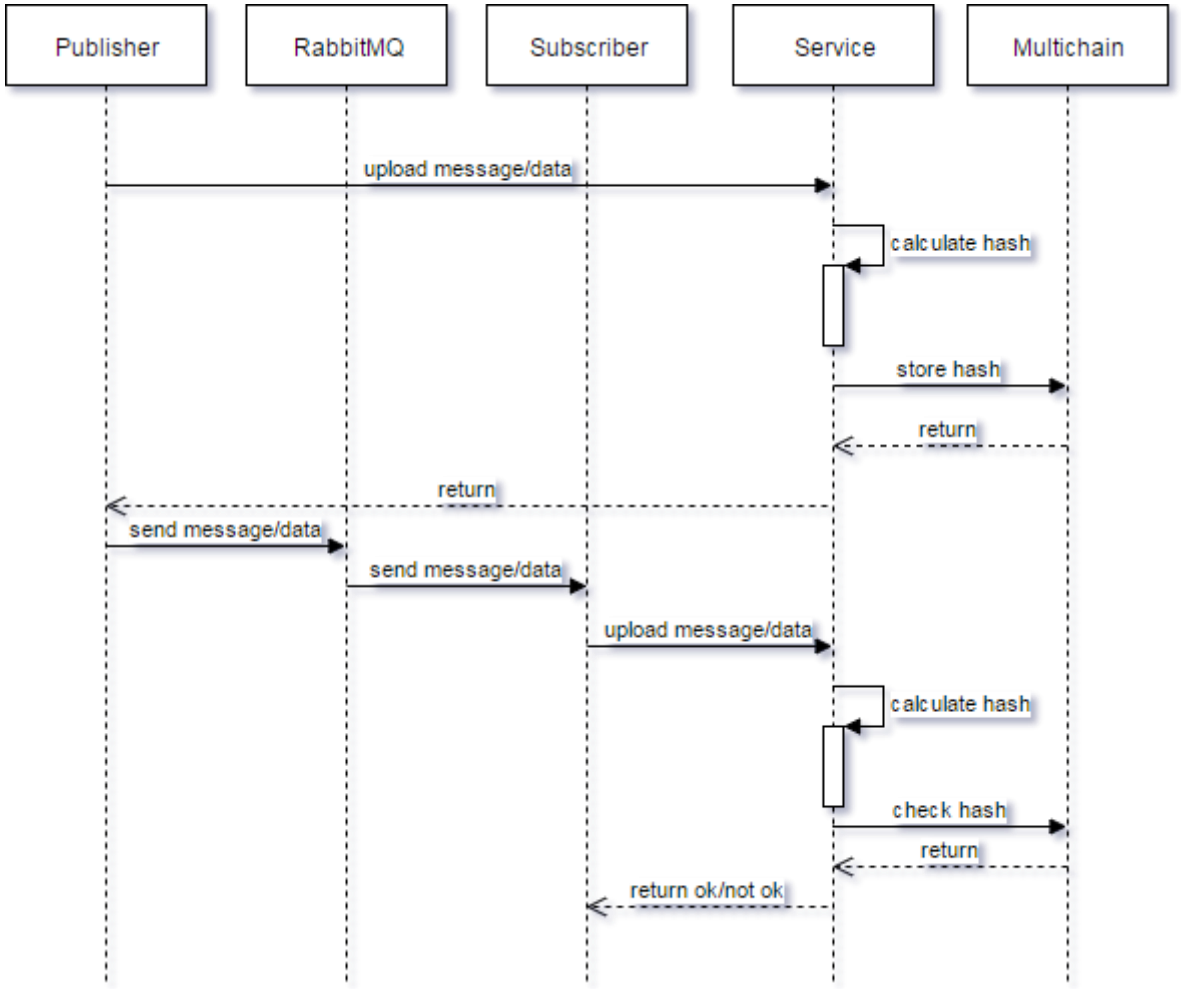

Figure 55: Sequence diagram of integrating blockchain in message sending

6.1.2.2 Log and Traceability

Multichain will also be used to provide an audit trail for manufacturing and supply chain data enabling both product data traceability and secure access for stakeholders. An approach of the architecture to be used is shown in the following figure.

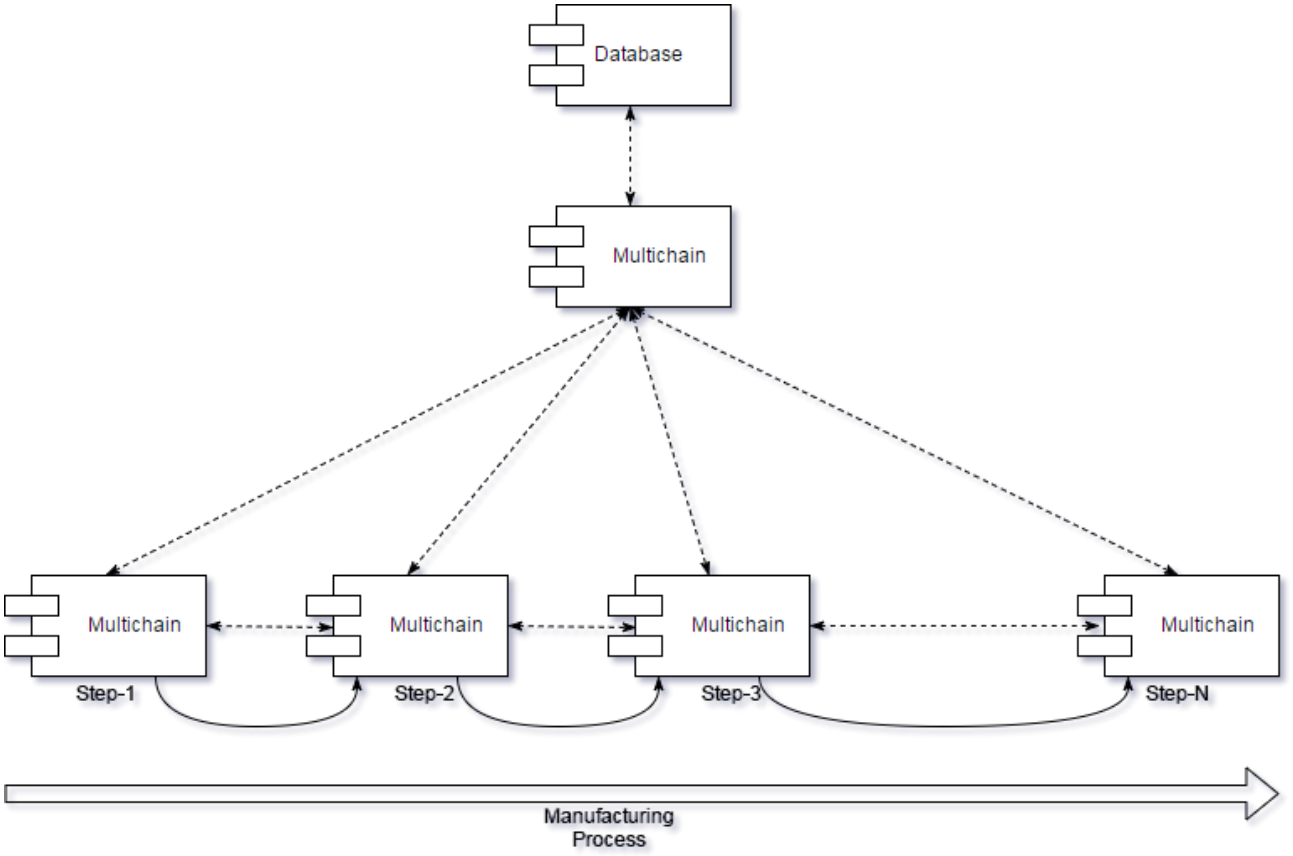

Figure 56: Blockchain in manufacturing process

The idea is to have multiple blockchain nodes along the whole manufacturing process with a central node. Each node in the chain will make a transaction to the next node with the data available at each stage of the process. Each node will add its own data to the one received from the previous node. As each transaction is stored in the blockchain by the end of the manufacturing process it will be possible to have a clear overview of what happened on each of the steps.

An advantage of this approach is that since the blockchain acts like a network of replicated databases, this means all nodes have exactly the same information; it's very difficult that a problem in the system may cause the loss of data. The failure of a node it´s not a big problem either, as replacing a node it´s really easy and as soon as it is connected to the network all data will be replicated on it.

It should be also considered only to store relevant data in each transaction while all other data is stored on an external database and linked to the data in the blockchain.

6.1.3 Cyber-Security

Following the approach stated in the previous deliverable D2.3 The COMPOSITION architecture specification I, the cybersecurity aspects of the Security Framework will be managed by Atos SIEM and the Cyber-Agents. This solution will monitor and protect the system against different kind of threads such as privileges abuse or DoS attacks.

Basically, XL-SIEM consists of a cross-layer security information and event management tool. The reason why it has been chosen is because of its main features that will provide a seamless protection against a wide set of threads:

- High-performance correlation engine

- Event collection, normalization and data transfer, managed by a set of distributed agents

Starting from the cybersecurity components described in the figure 62 of the previous deliverable D2.3 The COMPOSITION architecture specification I, the component in the top (SIEM) can be depicted in its architectural side in the diagram below, extracted from the XL-SIEM architecture paper (Gustavo González-Granadillo, 2017)

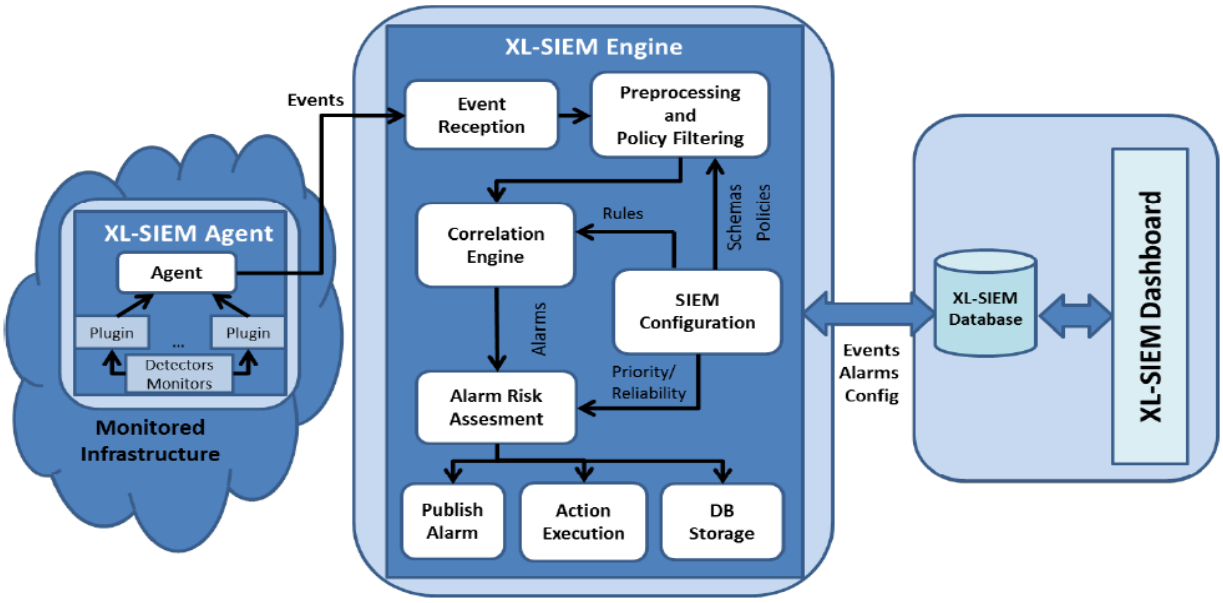

Figure 57: XL-SIEM Architecture

- XL-SIEM Agent

  The XL-SIEM Agent gathers all the events available in the network area where the XL-SIEM is deployed, then it transfers them to the XL-SIEM Engine for its processing.

- XL-SIEM Engine

  This component has two main purposes: analyze and process the events provided by the XL-SIEM Agents and the generation of alarms due to a predefined set of correlation rules and security directives.

- XL-SIEM Database

  Using some of the OSSIM database features and sharing key concepts as storage capabilities, provides persistent data storage assets. For example, data is stored in MySQL relational databases, the historical data is located in a separate database, does not support integration with cloud storage services, and the data storage can be in a different machine where the event processing is running.

- XL-SIEM Dashboard

  Is a web graphical interface with very useful visualization features: graphical charts reporting an overview of the monitored system status, alarms, security events and raw logs visualization.

For further information, see the Atos XL-SIEM paper (Gustavo González-Granadillo, 2017).

6.1.4 Transport Layer

All communication between COMPOSITION components should be encrypted using TLS/SSL where possible. In case of web applications and services its planned to use Nginx as a reverse proxy, with this approach all applications and services can run their own web servers and do not need to implement TLS/SSL on their own ad Nginx will take care of it. In the case of RabbitMQ message broker it needs to be configured to allow encrypted message transactions.

6.2 Scalability Perspective

This section describes scalability concerns for COMPOSITION and how the chosen design decisions and mechanisms can adopt measures to address these concerns. Basic concepts are introduced and scenarios that affect scalability in COMPOSITION are described. An overview of common design patterns that enhance scalability is followed by scalability design decisions for the individual components. Experience from the pilot installations is expected to further refine scalability design after the publication of this deliverable.

6.2.1 Basic Concepts and Terminology

6.2.1.1 Nodes, Resources and Scalability

As described in the Deployment View, each component will run in a Docker container; a virtual computational resource (node) with a certain specified computing and/or storage *capacity*. Other examples of nodes are physical servers, cloud services and execution containers in the cloud.

Computational resources are thus constrained by the amount allocated to the node with the limitations of the docker host being the upper limit, which means the physical specification of the hardware if this is a locally hosted deployment or in the case of cloud-based provisioning by the corresponding SLAs (Service Level Agreements).

For the sake of clarity, we would like to differentiate between performance and scalability. By performance we refer to the capability of a system to provide a certain response time *with a given set of nodes and resources*, e.g., to serve a defined number of users or processes a certain amount of data from a server with a certain capacity specification. Although no standard definition is available for these terms (Lehrig, Eikerling, & Becker, 2015), most of the available literature uses a similar definition for performance, e.g. (Wilder, 2012) where it is defined as "… an indication of the responsiveness of a system to execute any action within a given time interval".

Scalability we would like to define in analogous to the definition in (Lehrig, Eikerling, & Becker, 2015) as the ability of a system to increase the maximum workload it can handle by expanding its quantity of consumed resources. Similar definitions are "the ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased" (Wilder, 2012) or "the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth" (Bondi, 2000).

Scaling is thus about allocating more *resources* for an application, i.e., resource *provisioning*. In this discussion, we assume that the system has been designed to use the available resources as efficiently as possible i.e., by maximizing the performance with a given set of resources. Examples of resources needed by an application usually include CPU, memory, disk (capacity and throughput), and network bandwidth. An application or service is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added. Resources can be handled in *scalability units,* i.e., groups of resources that could be scaled together.

6.2.1.2 Vertical/Horizontal scaling

The scaling discussed here concerns the steps that may be taken when the available resources run out and the application does not fulfil its functional or non-functional requirements - the maximum workload of the system with the given resources is reached. We may then scale the system to increase the maximum workload it can handle by expanding its quantity of available resources. We can increase the quantity of consumed resources by increasing the amount of resources within existing nodes, or by adding more nodes.

To *scale up (or scale vertically)* is to increase overall application capacity by increasing the resources within existing nodes. In COMPOSITION, e.g., increasing the capacity of the node running the message broker in the IIMS. (For a Docker container, this can be achieved by using options such as "--cpus" and "--memory-reservation".).

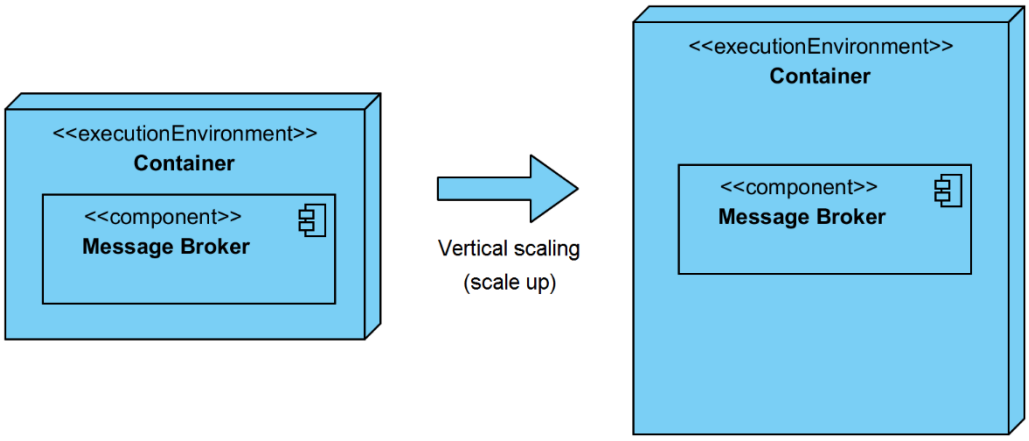

Figure 58: Boost capacity of node, scale up

Scaling up is usually the simplest and cheapest solution, as is does not require any changes to the design, code details or deployment of the application. While less complex (and sometimes cheaper compared to re-design or code improvements to increase performance) there are limitations to this approach compared to scaling out.

To *scale out (or scale horizontally)* is to increase overall application capacity by adding nodes, e.g., adding an additional message broker to the IIMS. (For Docker, this can be achieved by using options such as "--scale" or using docker swarm.)

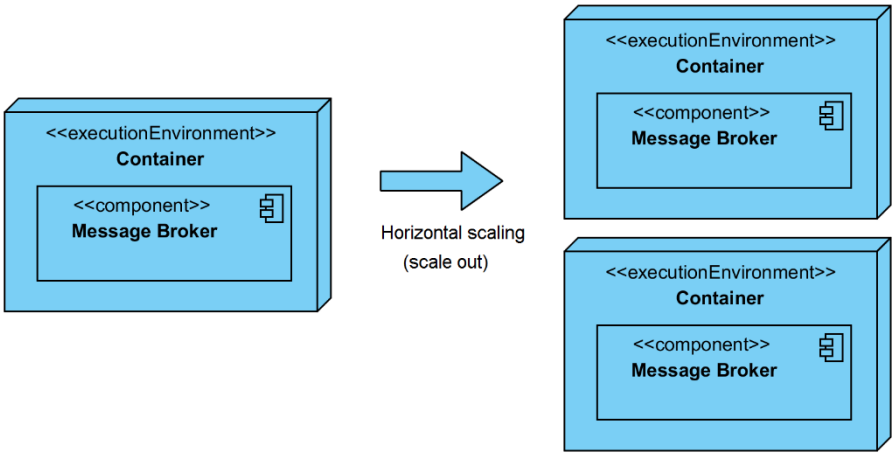

Figure 59: Scale out by adding nodes for component

Scaling out increases the overall application capacity by adding entire new computational nodes. Scaling out tends to be more complex than scaling up, and has more impact on the application architecture. We may scale out a COMPOSITION system instance by adding nodes for specific components (e.g., a Match Maker) and implement support for this at the component level. In the case of horizontal scaling, the system should also be able to adapt to shrinking demand for resources, to *scale in*. This property is often referred to as *elasticity* (Lehrig, Eikerling, & Becker, 2015).

When all the nodes supporting a specific function are configured identically - same hardware resources, same operating system, same function-specific software - we say these nodes are *homogeneous*[66]. We would add that components executing on different nodes may be homogenous with regards to functionality – all nodes support the same functions – and data or state – all nodes share the same data. This has implications on the design of horizontal scaling.

An *autonomous* node does not know about other nodes of the same type, similarly the same term can also be used for components.

In COMPOSITION, we have chosen is not to test scalability by creating a model of the system and performing simulations. The approach we have taken is to identify the scalability issues by analysis of deployed capacity,

[66] Wilder, Bill. Cloud Architecture Patterns: Using Microsoft Azure. Sebastapol, CA: O'Reilly Media, Inc., 2012

against application performance requirements, identifying scenarios where the maximum workload may exceed the capability of the system or components, investigate common design patterns for how these scenarios may be addressed, and, determine how the design of components deals with scaling up and out.

6.2.2 Issue identification and analysis

In this section, we list a number of scalability quality attribute scenarios where a high value of the attribute may cause the workload to exceed the maximum that the system or individual components can handle. Common design patterns to address these problems are described. The component designs and architectural decisions are described from a scalability viewpoint; the possible bottlenecks of each component, the possibility of scaling up or out, and the design implications.

6.2.3 Scenarios for scalability requirements of the system

6.2.3.1 Attributes that may affect workload of system or components

- Factory IIMS
    - The number of concurrently reporting sensors/field devices
    - The number of concurrently reporting BDA and ANN generated data streams
    - The number of generated data that should be persistently stored in the system (for future deep learning network training or in the blockchain
        - Number of generated data streams
        - Number of observations
    - The number of concurrent queries for stored data
    - The number of queries against the DFM for factory information
    - The number of concurrent users of the IIMS user interface
- Marketplace
    - The number of concurrent agent negotiations
    - The number of concurrent requests for Matchmaker services
    - The number of data sharing agreements between marketplace stakeholders
    - The number of concurrent requests for block chain storage
    - The number of users of marketplace user interfaces

6.2.3.2 Performance attributes affected

- Response time
    - Service time - how long it takes to do the work requested
    - Wait time - how long the request has to wait for requests queued ahead of it before it gets to run
    - Transmission time – How long it takes to move the request to the computer doing the work and the response back to the requestor
- Throughput
    - The amount of work accomplished in a given amount of time
- Resource usage
    - CPU usage
    - Memory usage
    - Storage usage

- Network usage - data sent and received

6.2.4 Performance and Scalability Design

Some examples of common design patterns used for performance and scalability are summarized here for convenience so that they may be referenced in the component scalability design section. More comprehensive descriptions may be found in e.g. (Wilder, 2012), (Homer, Sharp, Brader, & Swanson, 2014) or (Fowler, 2002).

6.2.4.1 Caching

When certain sets of data are frequently accessed, these may be copied to fast storage located close to the requesting application. E.g. since REST interfaces are employed for request response communication, HTTP caching may be used to avoid unnecessary load on the system by caching data at the HTTP client. Caching can also be performed in the Intra-factory Interoperability Layer or the Broker.

6.2.4.2 Materialized Views

HMI and other components may have need for views on data that is not stored or formatted in a way optimal for the query required to produce this view. The system may then generate prepopulated views over the data, possibly cached locally at the requesting node.

6.2.4.3 Throttling

To avoid that a single application or input source degrades the entire system, the services provided by the system may be temporarily limited. E.g. an agent sending a lot of requests may get a “503 Service Unavailable” response telling it to wait, or some functionality of the Marketplace or IIMS may be prioritized in case of insufficient resources.

6.2.4.3.1 Data partitioning

Data stored or processed in the system may be physically divided into separate nodes, so that they are not homogenous with respect to the data they manage. Using horizontal partitioning, the nodes may use the same schema but hold different parts of the data (e.g. different big data analytics nodes may process the same type of data but from different sources). With vertical partitioning, nodes will hold different parts of the schema, e.g. a broker instance may process only request-response type messaging or a storage node may only hold observation data. When different parts of the schema are handled by different nodes based on business or usage context, the term functional partitioning is sometimes used.

6.2.4.4 Load balancing

When the maximum workload of a single component is reached, redundant deployments of the component are created and a load balancing system dynamically distributes workloads. If the component works without state between calls and function calls are idempotent, this strategy is easier to implement.

6.2.4.5 Queue based load levelling and competing consumers

Instead of passing requests directly on to other components, a message queue can be used to implement the communication channel between the components. The sender component(s) post requests in the form of messages to the queue, and the consumer component(s) receive messages from the queue and process them, each at its own pace. This way, fluctuations in workload and differences in throughput between various parts of the system can be balanced, and individual components can be scaled out to optimize throughput.

6.2.4.6 Local hosting vs cloud hosting

The COMPOSITION system may be hosted, in whole or in parts, on physical or virtual hardware, in an environment owned and operated by a business (e.g. for a private marketplace) or in a cloud environment (e.g. Amazon, Azure).

Depending of the choice of hosting it may be possible to scale up or out automatically. In any case and at the very least, the components need to be able to indicate, when queried or by events, that the capacity limit is being reached. The systems administrator or an auto-scale component may use this information to start provisioning new resources. The design of components should be such that it is possible to scale them out by adding more nodes, and support able scaling elastically in if there is less demand for resources.

6.2.5 COMPOSITION Scalability Design

COMPOSITION addresses the scalability issues by scalable design of the components and of the architecture. Each instance, or deployment, of the intra factory system will face different scalability requirements.

A COMPOSITION component is deployed a docker container exposing a service, subscribing to data from the Message Broker and calling the services of other components. Docker supports control of both horizontal and vertical scaling of the services offered by a component. It also makes migration of containers to more capable hardware and re-configuration components to implement strategies such as queue-based load levelling or load-balancing easy compared to installations on virtual machines.

Docker Swarm, which is supported by the chosen Docker management tool Portainer, supports load-balancing and scaling up to 30000 containers[67].

6.2.5.1 Market Event Broker and Real-time Multi-Protocol Event Broker

The Message Broker is the central communication hub in both the intra- and inter-factory scenarios. This section builds on the scalability design reported in D6.3 “The COMPOSITION Marketplace I”. Choosing a scalability design for the message broker requires analysis of the usage pattern and how messages are distributed and utilizes on the design of the AMQP protocol. The message broker consists of one or several brokers distributed on one or more nodes. In a broker, exchanges receive and route messages to queues based on bindings with different filters. There is no fixed limit to the number of exchanges and queues in a broker. We have identified are two types of configuration which can be used to address scalability for the broker, which are referred to as routing topology and broker topology.

Broker topology deals with the distribution of logical brokers on nodes, by the built-in support for clustering (one logical broker on separate nodes) or federation (different logical brokers on separate nodes).

A RabbitMQ cluster connects multiple distributed nodes (all running the same version of RabbitMQ) together to form a single logical broker. Exchanges (and bindings) are replicated to all nodes in the cluster, while queues by default only exist only on the node where they are declared. It is possible to configure queues as mirrored, in which case publishing and deleting of messages is replicated on all mirrored queues. Thus, creating a queue for an agent will only create a new process in one broker in the cluster. A cluster - without mirrored queues - will have greater throughput than a single broker node. Queues are implemented as processes, whereas exchanges are just database entries. A cluster setup increases throughput and provides high availability and is the preferred setup of a logical broker for any full-scale installation of COMPOSITION and is the primary scalability strategy for the intra-factory system.

In a RabbitMQ federation, an exchange or queue on one broker can be set up to receive messages published to an exchange or queue on another, logically separate, broker. (Any federated logical broker may simultaneously be set up a cluster.) The brokers may use different versions of RabbitMQ and be otherwise unsynchronized. The integrated security provided by COMPOSITION Security Framework will facilitate the set-up of federated message brokers with shared user management. Unlike clusters, federations do not require all brokers in the federation to have direct connections. Only messages that need to be copied between federated brokers (due to declared bindings) will be copied over a link between federated brokers. In the Open marketplace, federations between brokers belonging to different stakeholders is a viable way to scale out the system and cater for differences in infrastructure.

Routing topology deals with the connections of exchanges and queues by bindings and the distribution of these on brokers. This topology can be set up dynamically on existing brokers by the AMQP protocol (and RabbitMQ extensions).

A RabbitMQ mechanism called “the shovel” moves messages from an exchange or queue in one logical broker to a destination exchange or queue in another logical broker.

RabbitMQ allows exchange-to-exchange bindings, routing messages from one exchange directly to a secondary exchange. Clients would then only bind to the secondary exchange, and the number of client queues and number of connects and disconnects at the secondary exchange would not affect the primary exchange. This is a viable way to scale out the system for a large number of agents in the marketplace. (A closed marketplace could require that stakeholders provide the resources for running a broker node.)

[67] https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/

Routing topology design could e.g. favour many fanout exchanges or fewer exchanges and more use of routing. Fanout exchanges are slightly faster than the other types of exchanges for multiple recipients, e.g. topic and header exchanges. However, the difference is not a deciding factor in the choice of topology.

6.2.5.2 Inter-factory

This section will discuss examples of possible scaling strategies for the marketplace. As the Message Broker manages agent CXL communication the design of broker topology is the primary way to ensure scalability for the marketplace.

Growth in the number of marketplaces is typically handled by adding nodes to the broker topology. A Closed Marketplace typically has a separate infrastructure from the Open Marketplace, whereas a Virtual Marketplace shares the infrastructure of the Open Marketplace. Marketplaces are logically separated; no messages are exchanged between marketplaces. Virtual marketplaces are set up by actors already in the Open Marketplace. Each Closed marketplace will be handled by a separate Message Broker. Open Marketplace and Virtual Marketplaces will use clustering.

In the cluster, load-balancing techniques may be used to distribute agents among the nodes so that the (non-mirrored) queues created by the agents is evenly distributed on the nodes,

Growth in the number of stakeholders in a marketplace may be handled by a routing topology which creates a secondary exchange for each specific stakeholder (Figure 60). The secondary exchange has an exchange binding to the primary exchange, which can be a fanout exchange. The consumers and producers (Agents) connected to the secondary exchange only create bindings and queues on one broker in the cluster when they connect. The secondary exchange may be a topic or header exchange.

The secondary stakeholder exchange will always exist, whether the stakeholder agents connect or disconnects. It will receive messages from all exchanges that the stakeholder has an interest in. Whenever a consumer (agent) connects it simply has to declare its queue and bind that queue to the stakeholder exchange using the desired topic filter.

A similar topology may be created by using either the shovel or federation with an upstream broker (primary) and a federated broker (secondary). These may be two separate broker nodes using different infrastructure. The messages to a queue declared in the federated broker are buffered in a queue created in broker the upstream exchange. If each connected stakeholder provides the infrastructure for the broker where the secondary exchange resides, the system can scale very well.

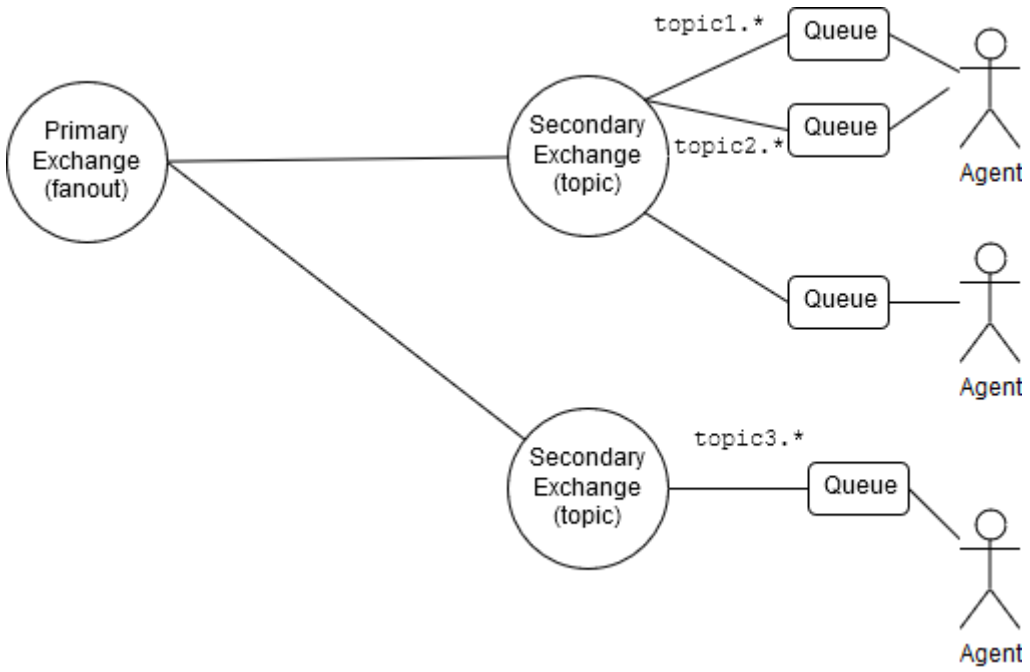

Figure 60: Primary and secondary exchange routing topology

The number of concurrent agent negotiations taking place will increase the number of messages being sent. In the above topology, the queues will be at the secondary exchanges and messages published to the exchange will be propagated to the primary and to all secondary exchanges. The primary/secondary broker topology deployed in a RabbitMQ cluster will handle a very large number of concurrent negotiations. Should the message flow require even more resources, a broker topology using a federation in a connected graph (each one a cluster), where an exchange for the negotiation will exist on one broker node in the federation only for the duration of the negotiation (Figure 61). The number of participants in each negotiation will likely not be a limiting factor for the described topology.

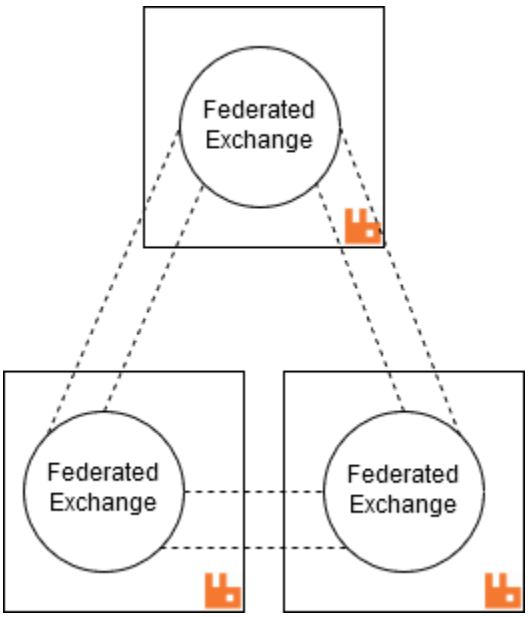

Figure 61: Federated exchanges broker topology

An exchange that only the involved parties can access can be set up for each data sharing agreement (Figure 62). At most this will result in a number of exchanges on the scale of $O(n^2)$ to the number of stakeholders. If one exchange is created for a stakeholder to publish to, and exchange to exchange bindings (or shovels) are defined for each recipient of data to the secondary exchanges described above (Figure 63), the number of exchanges will relate to the number of data sharing agreements by O(n). The sender will control the exchange to exchange bindings or shovels. The data sharing may need to use a separate logical broker (cluster) in the marketplace depending on the load.

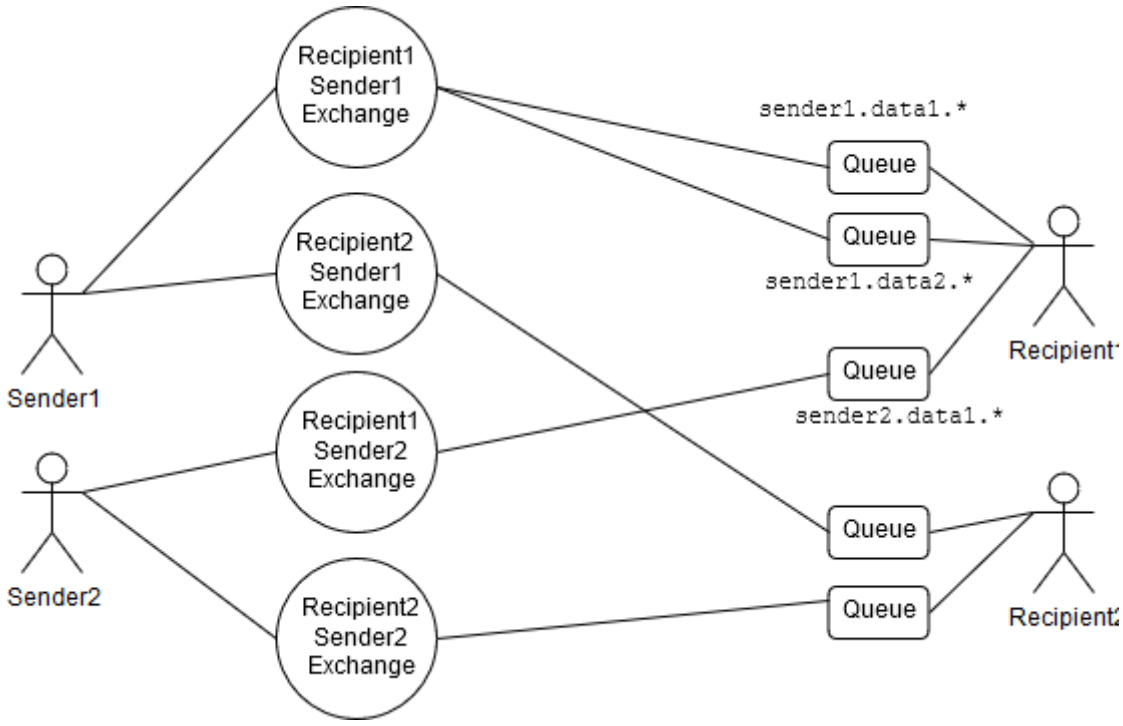

Figure 62: Data sharing using one exchange per data sharing agreement

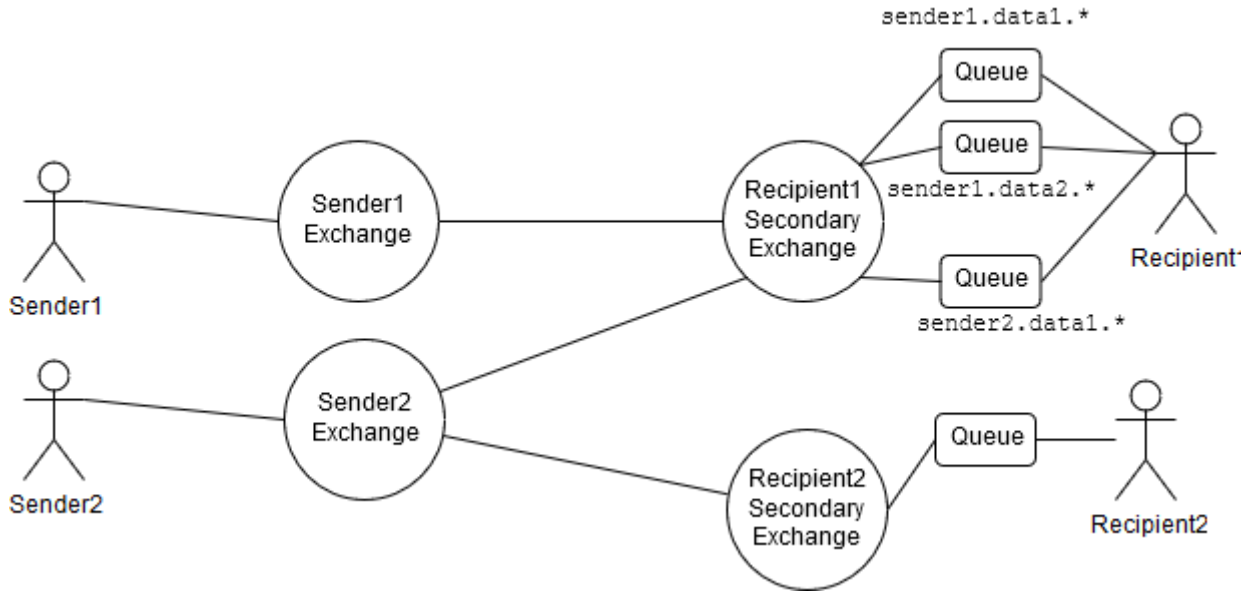

Figure 63: Data sharing using sender and recipient exchanges

6.2.5.2.1 Matchmaker

The COMPOSITION Matchmaker has been designed in order to offer high performance and support large Marketplaces with numerous of participants and services. It is designed after a thorough research for available tools, technologies, related works and methodologies.

As the Matchmaker component is packaged and deployed in an Apache Tomcat server, the maximum number of connections that this component can access and process depends on Tomcat web server configuration. Based on official Apache Tomcat 8 Configuration [68] the server is able to support over than 8000 connections.

[68] http://tomcat.apache.org/tomcat-8.5-doc/config/http.html

Furthermore, a RDF-triple store is used as the data store of the Marketplace. Based on the COMPOSITION project's pilot partners and use cases there was no need for a big data store for the Marketplace. However, in order to create a Marketplace that can be used beyond the project, triple-store was used. Two cases were examined based on Jena API. The first was the usage of SDB store which is a SQL database store. The second was the usage of TDB component for storing. The second approach was selected. As native triple store the TDB is faster, more scalable and better supported than SDB store. The SDB store is backed by SQL, so queries from SPARQL have to "turn" into SQL queries. This adds complexity and it is not as efficient as a native triple store. A native triple store is faster and supports the storage of millions of individuals. Using TDB every change at the ontology takes place at an ontology model stored in the file system leaving the original ontology immutable. This means that the original version of the ontology can be used in order to initialize new Marketplaces.

The performance of the Matchmaker and its included components was tested for the COMPOSITION use cases such as UC KLE-4 and the online bidding process. The Matchmaker responses in a reasonable time(less than 5 seconds). However, in order to examine the performance of some sub-components in large Marketplaces, automated JUnit tests were created and applied. Over 20.000 companies and services created and added to the Marketplace Ontology Store. Then some queries were applied and the responses were still in reasonable time (near 5 seconds). Only in the case that the instances were created simultaneously the required response were some minutes. But this is not consider as a serious problem as the Marketplaces was initialized ones and after that every new instance is added as soon as a new company arrives at the Marketplace or offers a new service etc.

6.2.5.3 Requester and Supplier Agent

The current implementation of the Requester and Supplier Agents is such that 1 single negotiation at a time is handled, due to its nature based on dynamic behaviour according to the current agent status. Different solutions are being studied and will be defined in D6.6 (M34).

6.2.5.4 Marketplace Management

The current deployment is capable of handling 1024 parallel connections to the interfaces exposed through the Agent Service. This can be increased either by vertically scaling the component to handle more simultaneous requests or by adding additional copies of it, properly tuning the routes towards the replicas.

Benchmarks for MySQL Cluster are available at the official page[i], some of them are shown in Figure 64, showing that the current deployment is capable of handling about 25.000.000 asynchronous reads.

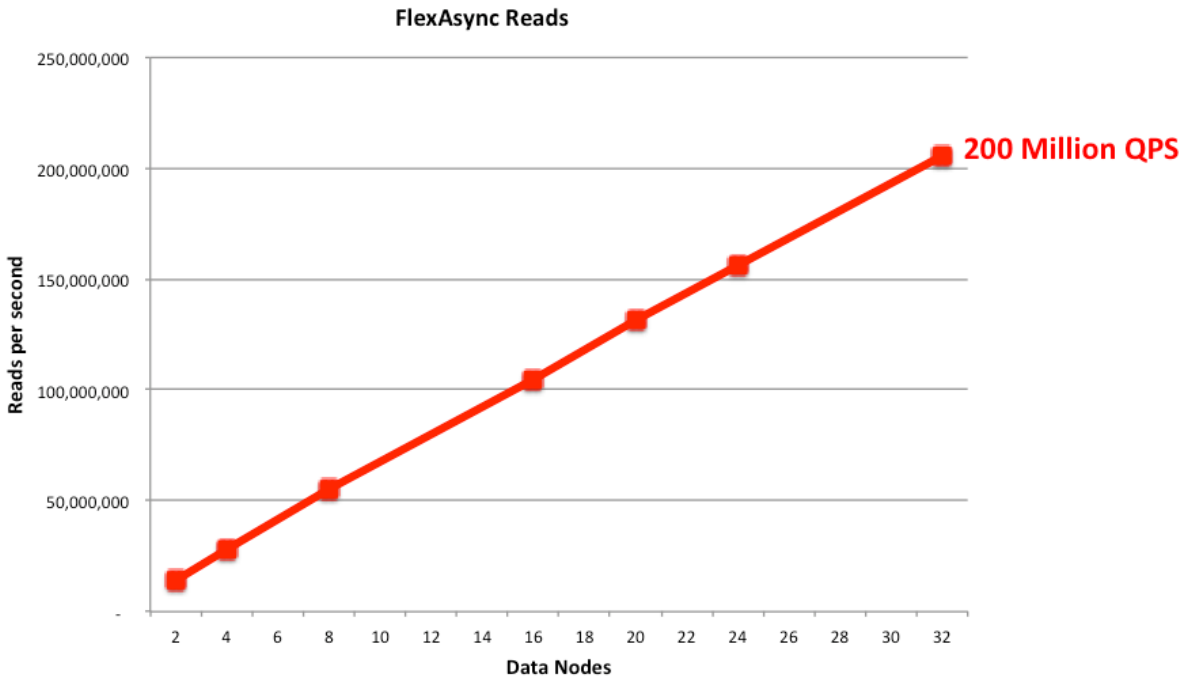

Figure 64: Async reads for MySQL Cluster

6.2.5.5 Intra-factory

6.2.5.5.1 BMS

Symphony BMS is a complex system that requires specific configurations to be put in place before running on a shop-floor. Internal mechanisms are implemented to ensure scalability functionality with respect on the number of sensors installed and the amount of data transmitted by these devices. Nevertheless, the system is targeted on building and factory environments, that can be big scopes, but somehow bounded on predictable scales. On the other hand, if a single instance (such as it is currently set up COMPOSITION demonstration) would not be enough for the intended purposes, there is no limit to the number of BMS instances that could be deployed, also because in a real application each factory has its own BMS instance. In this latter case, the usage of MQTT for streaming the data towards the other COMPOSITION components allows this change to be completely transparent to the rest of the system, since the broker itself decouples senders and receivers.

The Big Data Analytics and Deep Learning Toolkit components are deployed as a unit and only the Big Data Analytics communicate externally during runtime. It builds on a highly scalable CEP infrastructure but will manage scalability by internal configuration rather than by scaling out docker nodes.

The Decision Support System is easily scalable for the COMPOSITION project. It is stream process based on MQTT topics for communication and data retrieval from other components allows the program to scale up easily. Topics can be easily added for additional data sources, without further computational cost.

On the other hand, DSS Rule Engine is more difficult to scale up due to its complexity. When the rules increase, they create an exponential growth to the needed computational resources. Application of non – deterministic logic also increases the complexity of each rule and makes scalability tasks more complex to design. Though, taking into consideration, today's computational capabilities, the Rule Engine scale up problem would affect the system when the simultaneously applied rules where tens of thousands. This order of magnitude applies only when there is only one application for different shop floors and approaches.

Concerning the docker capabilities, DSS is easily scaled up both horizontally and vertically. The application is not expected to exceed certain levels of coding capabilities and limited data base requirements. Overall, the DSS applications can be dockerized without any concerns about scaling issues.

6.2.5.5.2 Simulation and Forecasting

The Simulation and Forecasting component should return its output to the Digital Factory Model and Visual Analytics in a reasonable time (e.g. less than 5 sec). Moreover, in this reasonable time Digital Factory Model and Visual Analytics will also provide input to Decision Support System. This is primarily a matter of increasing the performance of the component. This will be pursued by by conducting research and study related works for available tools, algorithms and best practices while optimizing the current design, e.g. by repeated processing and prioritized processing. The component can be scaled for a large number of requests by load-balancing identical instances.

6.2.5.6 HMI Framework

The design of the HMI framework with a single interface comprised of several micro frontends for independent components is suitable for load-balancing and scaling out on several levels, from back-end to data stores. It will also be able to be responsive even when individual parts experience high workload that affect performance.

The DSS HMI is designed to be independent of the data sources and the incoming data that the application should be able to visualise and show on the dashboard. As a result, DSS HMI is easily scalable to include various kinds of data sources, such as a new sensor network, different graphs for visualisation elements. Also, HMI can accommodate more than one instance on different shop floors for customised use. Since DSS is a web – based application, it is hosted on a server and can be easily reconfigured when new data sources are added on the application. Adding new data sources does not increase complexity of the application. One thing that maybe should be taken into consideration is that when dedicating different IP ports to different shop floor instances. Ports are a type of resource that is limited on a server. Concluding, the hosting capabilities of a server should be considered when deploying the application.

6.2.5.7 Security Framework

7 Summary and future work

No major changes to the architecture have been introduced since D2.3 and milestone MS2. There has been no reason to revise the major elements of the architecture laid out in the project description and the initial architecture workshops. Some components have been more tightly integrated, e.g. the BDA and DLT, and the Matchmaker and Marketplace Ontology. The design and development work have progressed from the bottom-up design laid out in the DOA and has been focused on a prioritized list of use cases and the design of common interoperability mechanisms.

The standards and interfaces that a COMPOSITION component need to adhere to has been specified. The components brought into the project have implemented the necessary adapters to comply with the common infrastructure (i.e. the ones that did not already have them) and the standards used in the Security Framework.

The design has focused on the selection and adoption of standards in each system perspective and providing interoperability between these. When new interfaces for existing components and adapters have been developed, standards from RAMI4.0 and FIWARE such as OPC-UA and FIWARE-NGSI v2 Specification API have been applied.

The HMI framework has adopted a modular micro frontend design using web components that will seamless integration of new functionality in the HMI. Menu and login are shared, but any language, framework or platform can be used to build additional functionality for the marketplace or factory user interfaces.

The Operational viewpoint has been of lower priority relative to other viewpoints and has at the time of writing not yet been addressed for all parts of the system. This is however of importance to the exploitation of the COMPOSITION system and the current description will be complemented when this work is completed.

Scalability design for the system has been laid out and delegated to individual components. The components implement internal saleability strategies for potential bottlenecks The dockerized component nodes can be managed by tools like Docker Swarm or Kubernetes. It is expected that the full-scale pilot deployment will provide further feedback to the scalability design and serve as a pointer to which scalability design patters are most relevant to COMPOSITION.

8 Appendix 1: The RAMI4.0 Model

8.1 IT Layers

The six layers on the vertical axis represent a layered IT system structure, with loose coupling between the layers and high cohesion within each layer. The layering is strict; i.e. components in a layer may only communicate internally or with adjacent layers.

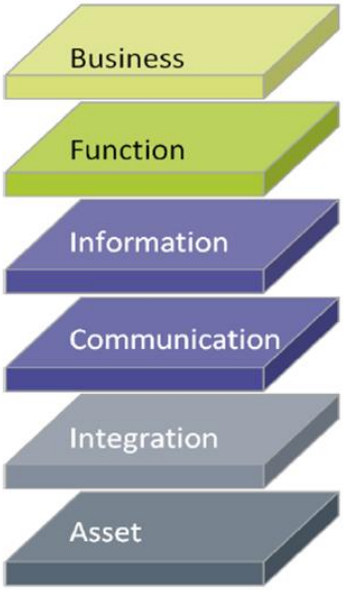

Figure 65: The IT Layers of RAMI 4.0

8.1.1 Asset Layer

The asset layer spans primarily the physical components of a system; physical things in the real world. E.g. production lines, manufacturing machinery, field devices, products and also the humans involved. However, other business assets e.g. software or information may also be regarded as assets. An asset is “a physical or logical object which is owned or managed by an organisation and which has an actual or perceived value for the organisation” (Plattform Industrie 4.0, 2016). In COMPOSITION, the trained artificial neural networks may be regarded as an asset.

8.1.2 Integration Layer

The mapping from the physical world to the digital is performed by the Integration layer, which performs provisioning of information on the assets in a form which can be processed by computer. This involves all digitization of assets, such as connected sensors and other field devices, but also Human Machine Interfaces (HMI).

8.1.3 Communication Layer

The Communication Layer performs transmission of data and files. It standardizes the communication from the Integration Layer, providing uniform data formats, protocols and interfaces in the direction of the Information Layer. It also provisions the services for controlling the Integration Layer.

8.1.4 Information Layer

In the Information Layer, data and events are processed, integrated and persisted. This layer ensures the integrity of data, performs message translation and annotation and manages data persistence. It provides the service interfaces to access structured data from the Functional Layer and also applies event rules and transformation of event to the models and formats used in that layer. This is the run-time environment for Complex Event Processing (CEP), data APIs and data persistence mechanisms.

8.1.5 Function Layer

The Function Layer is the primary location of rules and decision-making logic and contains the formal descriptions of functions and service models. It is the run time environment for applications and services that support the business processes.

8.1.6 Business Layer

The services provided by the Functional Layer are orchestrated by the Business Layer. It maps the services to the business (domain) model and the business process models. It also models the business rules, legal and regulatory constraints of the system. The Business Layers receives events that advance, link and integrate the business processes.

8.1.7 Hierarchy Levels

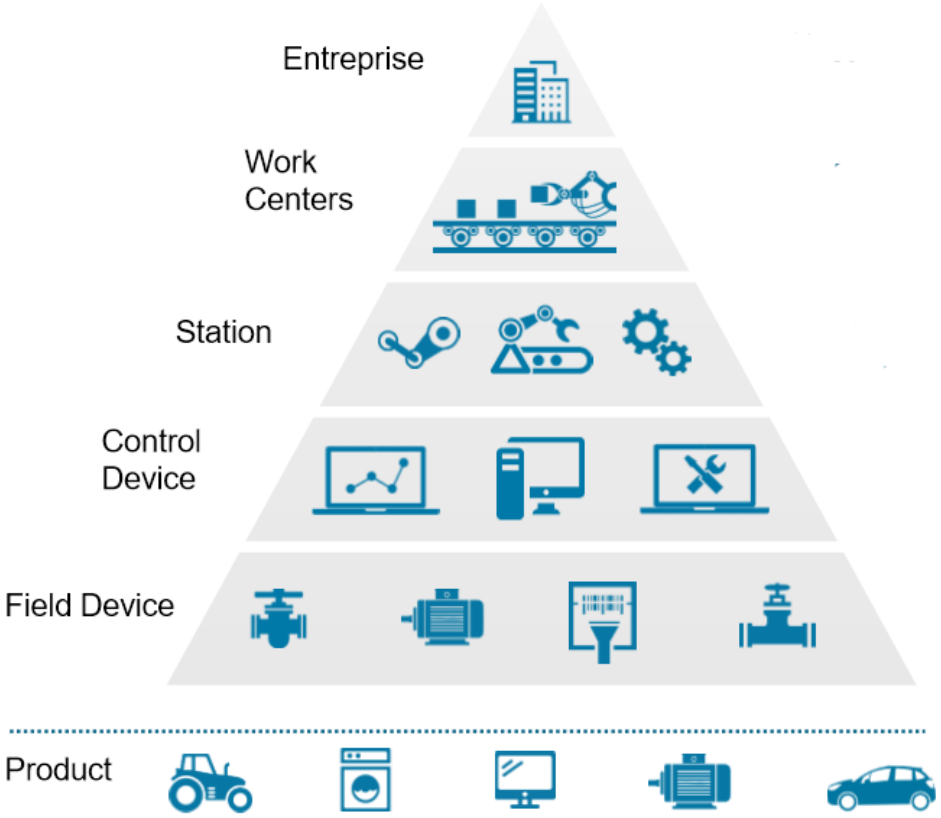

Figure 66: Hierarchy Levels of RAMI 4.0 (Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0), 2015)

The right horizontal axis represents a hierarchy of different functionalities within factories or facilities. The ones shown in the pyramid in Figure 66, from "Field device" to "Enterprise" are derived from the IEC 62264 (IEC62264, 2013) international standards series for enterprise IT and control systems. The standard originated by modelling "wired" connections between functions performed by hardware in the factory, but today the functions are implemented in software. To represent the Industry 4.0 environment, the functionalities of IEC 62264 have been expanded to include workpieces, labelled “Product” (both the type and the instance, through the entire lifecycle), and the connection to the Internet of Things and Services, labelled “Connected World”. The "Connected World" involves the manufacturing ecosystem: groups of factories, collaborations with external engineering firms, component suppliers and customers.

8.2 Life Cycle and Value Stream

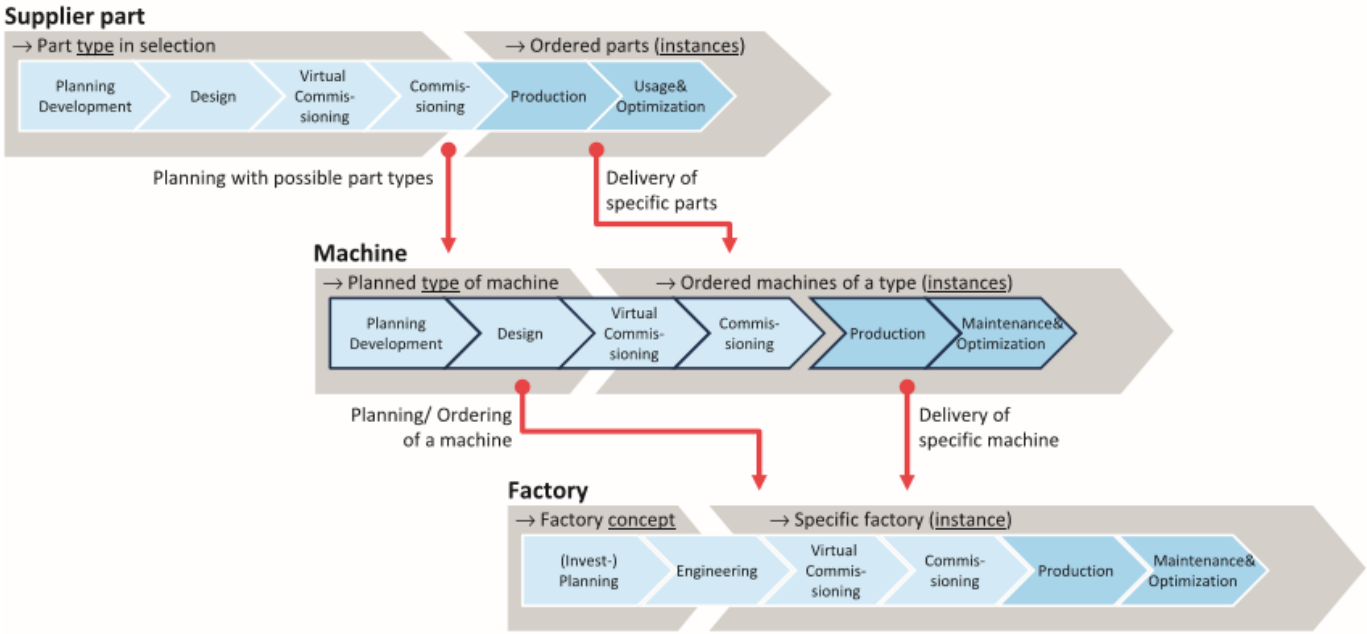

Figure 67: Type and instance lifecycles in RAMI 4.0 (Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0), 2015)

The left horizontal axis in RAMI 4.0 represents the life cycle of facilities and products, based on the IEC 62890 (IEC, 2013). Distinction is made between types and instances; design and prototyping involve types, and each actual product being manufactured is an instance of this type.

As illustrated by Figure 67, this life cycle and value stream does not only cover the planning, design, production and maintenance of parts and products, but also types and instances of production equipment and factories. RAMI4.0 spans both processes and workflows internal to the company and the services and products offered to clients.

8.3 Industrie 4.0 Component Administrative shell

An I4.0 component is the digitization of assets in the manufacturing process: it can be a factory, a production system, an individual station, or an assembly inside a machine. It consists of one or more assets and an administrative shell. The administrative shell is the virtual representation of an asset. The manifest of the administration shell describes the data provided by the asset and the resource manager provides access to the data and functionality of the asset. The I4.0 component is located within the layers of RAMI 4.0, up to the Functional Layer. It can adopt various positions in the life cycle and value stream, and occupy various hierarchical levels.

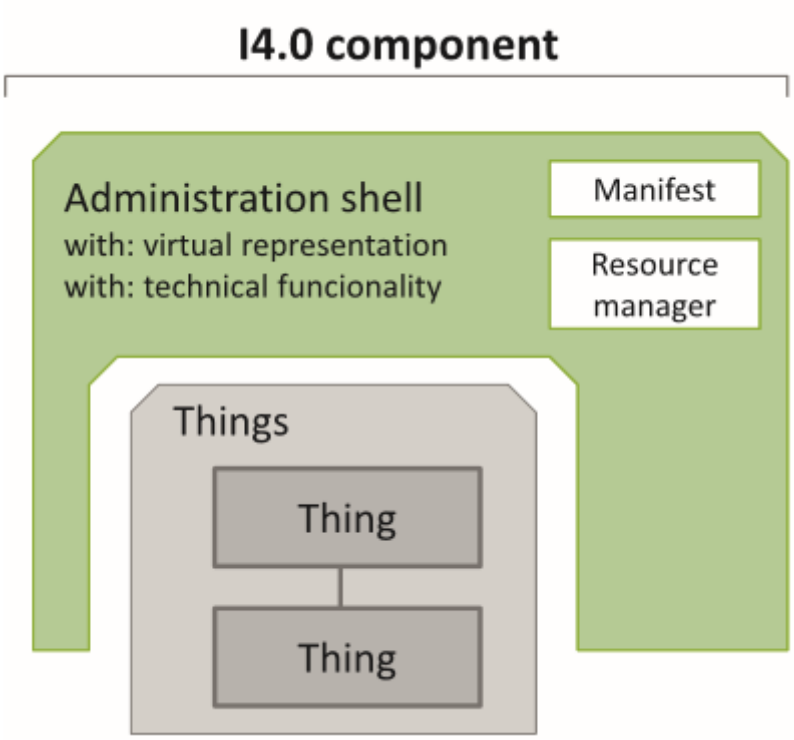

Figure 68: The I4.0 component (Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0), 2015)

An asset may have several administration shells for different purposes and aspects of the manufacturing process. I4.0 components may be nested and accessed directly of as part of the implementation of the services of another I4.0 component. The administrative shell may be deployed in the run-time environment of the asset – if it possesses the necessary computational capabilities – or remotely, e.g. in a cloud environment.

9 Appendix 2: Deep Learning Toolkit REST service interface


```
{
 "info": {
  "description": "Deep Learning Toolkit Inter-factory interfaces",
  "version": "1.0.0",
  "title": "Deep Learning Toolkit",
  "contact": {
   "email": [
    "vergori@ismb.it",
    "raimondo@ismb.it"
   ]
  }
 },
 "host": "not.defined.yet",
 "basePath": "/goods",
 "schemes": [
  "http"
 ],
 "paths": {
  "/goods": {
   "get": {
    "tags": [
     "good"
    ],
    "summary": "Get the list of all the available goods",
    "responses": {
     "200": {
      "description": "List of goods",
      "schema": {
       "$ref": "#/definitions/GoodId"
      }
     }
    }
   },
   "post": {
    "tags": [
     "good"
    ],
    "summary": "Add a new good to the store. If the good doesn't exist a new ANN will be created",
    "consumes": [
     "application/json"
    ],
```

```
    "produces": [
     "application/json"
    ],
    "parameters": [
     {
      "in": "body",
      "name": "body",
      "description": "Good object that needs to be added to the store",
      "required": true,
      "schema": {
       "$ref": "#/definitions/Good"
      }
     }
    ],
    "responses": {
     "201": {
      "description": "Created"
     },
     "400": {
      "description": "Bad request",
      "schema": {
       "$ref": "#/definitions/Error"
      }
     },
     "503": {
      "description": "Service unavailable when the ANNs number limit has been reached (server overloaded)"
     }
    }
   }
  },
  "/goods/{id}": {
   "get": {
    "tags": [
     "good"
    ],
    "summary": "Get good by type",
    "description": "",
    "produces": [
     "application/json"
    ],
    "parameters": [
     {
      "name": "id",
```

```
      "in": "path",
      "description": "The id of the good to be fetched.",
      "required": true,
      "type": "string"
     }
    ],
    "responses": {
     "200": {
      "description": "Successful operation",
      "schema": {
       "$ref": "#/definitions/Details"
      }
     },
     "404": {
      "description": "Good not found"
     }
    }
   },
   "delete": {
    "tags": [
     "good"
    ],
    "summary": "Delete good by id. The related ANN will also be deleted",
    "description": "",
    "produces": [
     "application/json"
    ],
    "parameters": [
     {
      "name": "id",
      "in": "path",
      "description": "The name of the good that needs to be deleted",
      "required": true,
      "type": "string"
     }
    ],
    "responses": {
     "200": {
      "description": "Successful operation"
     },
     "404": {
      "description": "Good not found"
     }
```

```
    }
   }
  },
  "/goods/{id}/predictions": {
   "get": {
    "tags": [
     "good"
    ],
    "summary": "Get predictions for a specific good",
    "produces": [
     "application/json"
    ],
    "parameters": [
     {
      "name": "id",
      "in": "path",
      "description": "The name that good to be fetched.",
      "required": true,
      "type": "string"
     },
     {
      "in": "query",
      "name": "from",
      "type": "integer",
      "description": "Query start epoch"
     },
     {
      "in": "query",
      "name": "to",
      "type": "integer",
      "description": "Query end epoch"
     }
    ],
    "responses": {
     "200": {
      "description": "Successful operation",
      "schema": {
       "$ref": "#/definitions/PredictionArray"
      }
     },
     "404": {
      "description": "Good not found"
     }
```

```
    }
   }
  },
  "/goods/{id}/predictions/last": {
   "get": {
    "tags": [
     "good"
    ],
    "summary": "Get the last prediction for a specific good",
    "produces": [
     "application/json"
    ],
    "parameters": [
     {
      "name": "id",
      "in": "path",
      "description": "The name that good to be fetched.",
      "required": true,
      "type": "string"
     }
    ],
    "responses": {
     "200": {
      "description": "Successful operation",
      "schema": {
       "$ref": "#/definitions/Prediction"
      }
     },
     "404": {
      "description": "Good not found"
     }
    }
   }
  },
  "/goods/{id}/values": {
   "get": {
    "tags": [
     "good"
    ],
    "summary": "Get the values uploaded for a specific good",
    "produces": [
     "application/json"
    ],
```

```
    "parameters": [
     {
       "name": "id",
       "in": "path",
       "description": "The name that good to be fetched.",
       "required": true,
       "type": "string"
     },
     {
       "in": "query",
       "name": "from",
       "type": "integer",
       "description": "Query start epoch"
     },
     {
       "in": "query",
       "name": "to",
       "type": "integer",
       "description": "Query end epoch"
     }
    ],
    "responses": {
     "200": {
       "description": "Successful operation",
       "schema": {
         "$ref": "#/definitions/ValueArray"
       }
     },
     "404": {
       "description": "Good not found"
     }
    }
   },
   "post": {
    "tags": [
     "good"
    ],
    "summary": "Add a new value to the good store",
    "description": "",
    "consumes": [
     "application/json"
    ],
    "produces": [
```

```
        "application/json"
      ],
      "parameters": [
       {
         "name": "id",
         "in": "path",
         "description": "The id of the good",
         "required": true,
         "type": "string"
       },
       {
         "in": "body",
         "name": "body",
         "description": "Good object that needs to be added to the store",
         "required": true,
         "schema": {
           "$ref": "#/definitions/Value"
         }
       }
      ],
      "responses": {
       "201": {
         "description": "Created"
       },
       "400": {
         "description": "Bad request",
         "schema": {
           "$ref": "#/definitions/Error"
         }
       }
      }
    }
   }
},
"definitions": {
  "Error": {
    "type": "object",
    "required": [
      "code"
    ],
    "properties": {
      "code": {
        "type": "integer",
```

```
      "format": "int32",
      "example": 4
     },
     "message": {
      "type": "string",
      "example": "Invalid input data"
     }
    }
   },
   "GoodId": {
    "type": "array",
    "items": {
     "type": "string"
    },
    "example": [
     "paper",
     "scrap_metal"
    ]
   },
   "Value": {
    "type": "object",
    "required": [
     "price",
     "quantity",
     "start_date",
     "end_date"
    ],
    "properties": {
     "price": {
      "type": "number",
      "description": "The price to be updated, in floating point",
      "example": 25.8
     },
     "quantity": {
      "type": "number",
      "description": "The quantity to be updated, in floating point",
      "example": 30.4
     },
     "start_date": {
      "type": "integer",
      "description": "The start date quantity and price refer to, in epoch time",
      "example": 1524491482
     },
```

```
    "end_date": {
     "type": "integer",
     "description": "The end date quantity and price refer to, in epoch time",
     "example": 1524494482
    }
   }
  },
  "Details": {
   "type": "object",
   "required": [
    "id_good",
    "values",
    "predictions"
   ],
   "properties": {
    "id_good": {
     "type": "string",
     "description": "good type",
     "example": "paper"
    },
    "values": {
     "type": "array",
     "items": {
      "$ref": "#/definitions/Value"
     }
    },
    "predictions": {
     "type": "array",
     "items": {
      "$ref": "#/definitions/Prediction"
     }
    }
   }
  },
  "Good": {
   "type": "object",
   "required": [
    "id_good"
   ],
   "properties": {
    "id_good": {
     "type": "string",
     "description": "good type",
```

```
      "example": "paper"
    },
    "price": {
      "type": "number",
      "description": "The price to be updated",
      "example": 25.8
    },
    "quantity": {
      "type": "number",
      "description": "The quantity to be updated",
      "example": 30.4
    },
    "start_date": {
      "type": "integer",
      "description": "The start date quantity and price refer to, in epoch time",
      "example": 1524491482
    },
    "end_date": {
      "type": "integer",
      "description": "The end date quantity and price refer to, in epoch time",
      "example": 1524494482
    }
  }
},
"Prediction": {
  "type": "object",
  "required": [
    "value",
    "date",
    "accuracy"
  ],
  "properties": {
    "value": {
      "type": "number",
      "description": "The price prediction",
      "example": 120.3
    },
    "date": {
      "type": "integer",
      "description": "The date prediction refer to, in epoch time",
      "example": 1524491482
    },
    "accuracy": {
```

```
        "type": "number",
        "description": "The accuracy of the prediction, from 0 to 1"
       }
      }
    },
    "PredictionArray": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/Prediction"
      }
    },
    "ValueArray": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/Value"
      }
    }
  }
}
```

10 Appendix 3: CXL JSON Schema

```
{
 "description": "The JSON syntax specification of the COMPOSITION CXL language, mainly focus on the message envelope",
 "type": "object",
 "properties": {
  "act": {
   "type": "string",
   "enum": [
    "accept-proposal",
    "agree",
    "cancel",
    "cfp",
    "confirm",
    "disconfirm",
    "failure",
    "inform",
    "inform-if",
    "inform-ref",
    "not-understood",
    "propagate",
    "propose",
    "proxy",
    "query-if",
    "query-ref",
    "refuse",
    "reject-proposal",
    "request",
    "request-when",
    "request-whenever",
    "subscribe"
   ]
  },
  "sender": {
   "type": "object",
   "description": "the message originator",
   "properties": {
    "name": {
     "type": "string"
    },
    "addresses": {
     "type": "array",
     "items": {
      "type": "object"
     }
    },
    "user-defined": {
     "type": "object"
    }
   }
```

```
  },
  "receiver": {
   "type": "array",
   "description": "The set of recipients for this message",
   "items": {
    "type": "object",
    "description": "the message recipient",
    "properties": {
     "name": {
      "type": "string"
     },
     "addresses": {
      "type": "array",
      "items": {
       "type": "object"
      }
     },
     "user-defined": {
      "type": "object"
     }
    }
   }
  },
  "reply-to": {
   "type": "object",
   "description": "The agent to which replies for this message shall be sent",
   "properties": {
    "name": {
     "type": "string"
    },
    "addresses": {
     "type": "array",
     "items": {
      "type": "object"
     }
    },
    "user-defined": {
     "type": "object"
    }
   }
  },
  "language": {
   "type": "string",
   "description": "The language used for encoding the message content"
  },
  "encoding": {
   "type": "string",
   "description": "The specific encoding used for language expressions, typically a mime type"
  },
  "ontology": {
   "type": "array",
```

```
    "description": "The set of ontologies defining the primitives that are valid within the message content",
    "items": {
     "type": "string",
     "format": "url"
    }
   },
   "protocol": {
    "type": "string",
    "description": "Identifies the agent communication protocol to which the message adheres"
   },
   "content": {
    "type": "object",
    "description": "The actual payload of the message"
   },
   "conversation-id": {
    "type": "string",
    "description": "Provides an identifier for the sequence of communicative acts (messages) that together
form a conversation"
   },
   "reply-with": {
    "type": "string",
    "description": "Provides an expression that the message recipient shall include in the answer, exploiting
the in-reply-to field. This allows following a conversation when multiple dialogues occur simultaneously."
   },
   "in-reply-to": {
    "type": "string",
    "description": "Denotes an expression that references and earlier action to which this message is a
reply"
   },
   "reply-by": {
    "type": "string",
    "format": "date-time"
   }
  },
  "additionalProperties": false
}
```

11 References

(den 19 07 2018). Hämtat från The Hydra Project: https://linksmart.in-jet.dk/news.php
(2018). Hämtat från Portainer: https://portainer.io/
(2018). Hämtat från PMD: https://pmd.github.io/
(2018). Hämtat från Keycloak: https://www.keycloak.org/
(2018). Hämtat från IMPACT: https://www.cs.umd.edu/projects/impact/
(2018). Hämtat från GoodRelations Language: http://www.heppnetz.de/projects/goodrelations
(2018). Hämtat från FIWARE: http://www._ware4industry.com/
(2018). Hämtat från Docker: https://www.docker.com/
(2018). Hämtat från Apache Tomcat: http://tomcat.apache.org/
(2018). Hämtat från Apache Jena: https://jena.apache.org/documentation/inference/
Ameri. (2006). *Manufacturing Service Description Language.* Hämtat från https://www.researchgate.net/publication/267486591_An_Upper_Ontology_for_Manufacturing_Service_Description
Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. *Machine learning*, 5-43.
Apache Maven. (2018). Hämtat från Apache Maven: https://maven.apache.org/
Axling, M., Bonino, D., Ioannidis, D., Kaklanis, N., Nizamis, A., Vergori, P., . . . Rosengren, P. (2017). *D2.3 The COMPOSITION Architecture Specification I.* COMPOSITION Consortium.
Bondi, A. (2000). *Characteristics of scalability and their impact on performance.* Proceedings of the second international workshop on Software and performance - WOSP '00.
Bonino, D., Carvajal Soto, J. A., Delgado Alizo, M. T., Alapetite, A., Gilbert, T., Axling, M., . . . Spirito, M. (2015). Almanac: Internet of things for smart cities. *2015 3rd IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, 309-316.
Carvajal Soto, J. Á., Jentsch, M., Preuveneers, D., & Ilie-Zudor, E. (2016). CEML: Mixing and Moving Complex Event Processing and Machine Learning to the Edge of the Network for IoT Applications. *Proceedings of the 6th International Conference on the Internet of Things*, 103-110.
COMPOSITION. (2016). *GRANT AGREEMENT 723145 — COMPOSITION: Annex 1 Research and innovation action.*
Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 15.
Dawid, A. P. (1984). Present position and potential developments: Some personal views: Statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, 278-292.
ECLIPSE EGit. (2018). Hämtat från ECLIPSE: http://www.eclipse.org/egit/
ECLIPSE IDE. (2018). Hämtat från ECLIPSE: https://www.eclipse.org/ide/
Fernandes, J. L. (2013). FernPerformance evaluation of RESTful web services and AMQP protocol. *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on.* IEEE.
FIPA. (2004). *FIPA Agent Management Specification.* Hämtat från Foundation for Intelligent Physical Agents : http://www.fipa.org/specs/fipa00023/SC00023K.pdf
FITMAN-SeMa. (2018). Hämtat från http://www.ware4industry.com/?portfolio=metadata-and-ontologies-semantic-matching-sema
Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison Wesley.
Gaber, M. M. (Advanced Methods for Knowledge Discovery from Complex Data). Gaber, Mohamed Medhat; Krishnaswamy, Shonali; Zaslavsky, Arkady. *On-board Mining of Data Streams in Sensor Networks*, 307-335.
Gustavo González-Granadillo, S. G.-Z. (2017). *Towards an Enhanced Security Data Analytic Platform.* Atos Research and Innovation, Cyber Security Department.
Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns.* Addison-Wesley Professional.
Homer, A., Sharp, J., Brader, L. N., & Swanson, T. (2014). *Cloud Design Patterns.* Microsoft patterns & practices.
IEC. (2013). *IEC 62890: IEC Project: Life Cycle Management for Systems and Products used in Industrial-Process Measurement, Control, and Automation.* IEC.
IEC62264. (2013). *IEC 62264-1: Enterprise-control system integration Part 1: Models and Terminology.* IEC.
IEEE. (2000). *IEEE 1471 Recommended Practice for Architectural Description for Software Intensive Systems.* IEEE.
ISO/IEC/IEEE42010. (2011). *ISO/IEC 42010: Systems Engineering – Architecture description.* ISO/IEC/IEEE.
ISO19156. (2011). *Geographic information -- Observations and measurements.* ISO.

Kruchten, P. (2004). *The Rational Unified Process: An Introduction.* Addison-Wesley Professional.

Lehrig, S., Eikerling, H., & Becker, S. (2015). *Scalability, Elasticity, and Efficiency in Cloud Computing: a Systematic Literature Review of Definitions and Metrics.* Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15), Montreal, QC, Canada, May 4–7.

Lemaignan. (2006). *Manufacturing's Semantics Ontology or MASON is a manufacturing ontology, aimed to provide a common semantic net in manufacturing domain.* Hämtat från http://ieeexplore.ieee.org/document/1633441/

Liang, S., Huang , C.-Y., & Khalafbeigi, T. (2016). *OGC SensorThings API Part 1: Sensing.* Open Geospatial Consortium.

Maciej, R., Krzysztof, G., & Aleksander, S. (2014). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on.* IEEE.

Milagro, F. A. (2008). SOAP tunnel through a P2P network of physical devices. *Internet of Things Workshop.* Sophia Antopolis: Internet of Things Workshop, Sophia Antopolis.

OPC Foundation. (2018). *OPC Foundation*. Hämtat från https://opcfoundation.org/

Plattform Industrie 4.0. (2016). *Structure of the Administration Shell - Continuation of the Development of the Reference Model for the Industrie 4.0 Component.* Federal Ministry for Economic Affairs and Energy (BMWi).

Pyro - Python Remote Objects. (den 18 07 2018). Hämtat från https://pythonhosted.org/Pyro4/

Rozanski, N., & Woods, E. (2012). *Software Systems Architecture,: working with stakeholders using viewpoints and perspectives.* Addison-Wesley.

(2015). *Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0).* Düsseldorf: VDI e.V.

Stehman, S. V. (1997). Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment*, 77 - 89.

Suárez-Figueroa, M. C. (2010). *NeOn Methodology for Building Ontology Networks: Specification, Scheduling and Reuse.* Hämtat från https://www.researchgate.net/publication/47900862_NeOn_Methodology_for_Building_Ontology_NetworksSpecification_Scheduling_and_Reuse

Syed, N. A., Huan, S., Kah, L., & Sung, K. (1999). Incremental learning with support vector machines. *Citeseer*.

Wilder, B. (2012). *Cloud Architecture Patterns.* O'Reilly.

Y.2060, I.-T. (2012). *ITU-T Y.2060 : Overview of the Internet of things.* ITU.

12 List of Figures and Tables

12.1 Figures

12.2 Tables

[i] https://www.mysql.com/why-mysql/benchmarks/mysql-cluster/