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

D6.4 COMPOSITION Marketplace II

Date: 2019-06-27

Version 1.0

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:	D6.4 COMPOSITION Marketplace II_v1.0.docx
Document version:	1.0
Document owner:	LINKS
Work package:	WP6 – COMPOSITION Collaborative Ecosystem
Task:	T6.2 – Cloud Infrastructures for Inter-factory Data Exchange
Deliverable type:	OTHER
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	Jure Rosso (ISMB/LINKS)	2019-05-22	Initial content from D6.3
0.2	Alexandros Nizamis, Nikolaos Vakakis and Christos Ntinas (CERTH)	2019-06-06	Update Matchmaker section
0.3	Luca Tomaselli (NXW)	2019-06-12	Description of Marketplace Management (sec. 6.3)
0.4	Jure Rosso (ISMB/LINKS)	2019-06-12	Use case modified and integrated
0.5	Mathias Axling (CNET)	2019-06-15	Update of section 6
0.6	Nacho González (ATOS)	2019-06-20	Update security section
0.7	Jure Rosso (ISMB/LINKS)	2019-06-21	Finalization for internal review
1.0	Jure Rosso (ISMB/LINKS)	2019-06-26	Changes according to reviewers

Internal review history:

Reviewed by	Date	Summary of comments
Jannis Warnat (FIT)	2019-06-24	The document is fine with some minor comments.
Alexandros Nizamis and Vagia Rousopoulou (CERTH)	2019-06-25	The document is fine with some minor comments on the format. The reference table should be updated and the conclusions should be expanded.

Index:

1 Executive Summary

The present document named "D6.4 COMPOSITION Marketplace II" is a public deliverable of the COMPOSITION project, co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145. This deliverable presents the final results of the Task 6.2: "Cloud Infrastructures for Inter-factory Data Exchange". It aims to describe and analyse the COMPOSITION marketplace's final version.

The document owner is LINKS and this final version is a reiteration of the one submitted at M20, namely "D6.3 COMPOSITION Marketplace I".

COMPOSITION has two main goals:

1. The integration of data along the value chain from the inside of a factory into one integrated information management system (IIMS).
2. The creation of a (semi-)automatic ecosystem that extends the local IIMS concept to a holistic and collaborative system incorporating and interlinking both the supply and the value chains.

The purpose of this deliverable is to further describe the development process related to the generation of the (semi-)automatic ecosystem, tackling scalability and distribution issues.

Particularly, the analysis will focus on:

- The design of the different marketplace components grouped as follows:
    - Marketplace Agents, such as the Agent Management Service (AMS) and Matchmaker Agent.
    - Stakeholder Agents, such as Requester and Supplier agents
- The implementation of the components described above, alongside with a short description for every of them.
- The description of the marketplace infrastructure:
    - Design decisions and implementation regarding reliability
    - Design decisions and implementation regarding scalability
    - Security framework with focus on the marketplace components

The results of the analysis of all the aforementioned features have been implemented in the COMPOSITION Marketplace, a key component of the COMPOSITION ecosystem.

The Marketplace and his Agents have been developed in a full custom way to avoid constraints coming from frameworks usage. The existing technologies have been analysed as state of the art, then, due to the project needs, the COMPOSITION Marketplace has been developed without using any of these frameworks but taking inspiration from those ones.

The resulting architecture provides all the necessary elements to support the creation of new supply chains in dynamic way by offering a set of intelligent agents enabled to complete negotiations in a semi-automatic way. During the implementation of the Marketplace the security aspects have been strongly taken in account to ensure authentication, authorization and messages integrity exploiting the services provided by the COMPOSITION Security Framework and the COMPOSITION Blockchain.

It is worth to mention the language that has been developed to support Agents communication mechanism. It is called COMPOSITION eXchange Language and it is derived from FIPA ACL standard. This language is able to support many different actions and several different messages described by exploiting ad-hoc defined ontologies.

The tight connection between all the Marketplace components make the system reliable without sacrificing the modularity and the scalability properties. The Marketplace has been designed in a full modular way to allow the possibility to implement new features or replace some of the modules to support new feature, communication protocols or negotiation strategies.

The COMPOSITION Marketplace brings a new and dynamic way in supply chain creation. It has shown an easy and successful way in doing negotiation in a semi-automatic manner outperforming the old mechanism based on a full manual process that usually takes much more time.

2 Terminology

The currently adopted domain-specific terminology used in the remainder of the document is presented in Table 1 below.

Table 1: Terminology

Term	Definition
ACL	Agents Communication Language
AMQP	Advanced Message Queueing Protocol
AMS	Agent Management Service
API	Application Programming Interface
CRUD	Create Read Update Delete
CXL	COMPOSITION eXchange Language
FIPA	Foundations for Intelligent Physical Agents
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IIMS	Integrated Information Management System
JMS	Java Message Service
JSON	JavaScript Object Notation
MAS	Multi-Agent System
RMI	Remot Method Invocation
RDF	Resource Description Framework
REST	REpresentational State Transfer
SPARQL	Simple Protocol and RDF Query Language
XML	eXtensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol

3 Introduction

3.1 Purpose, Context and Scope of this Deliverable

This deliverable presents the actions performed on the design and implementation of the COMPOSITION Marketplace, focusing on the agent container technology, scalability and distribution issues. The work has been carried out mainly in Work Package 6 (WP6), “COMPOSITION Collaborative Ecosystem”. The main tasks involved are:

- Task 6.1 “Real-time event brokering for factory interoperability”
- Task 6.2 “Cloud Infrastructures for Inter-Factory Data Exchange”
- Task 6.3 “Connectors for Inter-Factory Interoperability and Logistics”
- Task 6.4 “Collaborative manufacturing services ontology and language”
- Task 6.5 “Brokering and Matchmaking for Efficient Management of Manufacturing Processes”

3.2 Content and Structure of this Deliverable

The structure of this deliverable is as follows:

Section 4 – Provides the analysis of the background and state of the art of the technologies used.

Section 5 – Describes the development status of the Agents on the marketplace.

Section 6 – Describes, from a high-level perspective, the marketplace infrastructure.

Section 7 – Presents a summary of the final development state with conclusions.

4 Background and State of the Art

Multi-agent systems (Lützenberger, et al., 2013) have been widely studied in literature, and some existing options have been evaluated before deciding to implement the marketplace. A multi-agent system is a computerized system composed of multiple interacting intelligent agents[1]. Agent platforms are typically designed with a container paradigm, where all agents live in a well-defined agent container. Among the many platforms available on the internet, a few subsets of candidates that might cover a good subset of composition features and requirements have been selected, they are listed in the following Table 2.

Table 2: Candidate Agent Platforms

Platform	Description	Pros	Cons	License
JIAC	JIAC (Java-based Intelligent Agent Componentware) is a Java-based agent architecture and framework that eases the development and the operation of large-scale, distributed applications and services. The framework supports the design, implementation, and deployment of software agent systems. The entire software development process, from conception to deployment of full software systems, is supported by JIAC. It also allows for the possibility of reusing applications and services, and even modifying them during runtime. The focal points of JIAC are distribution, scalability, adaptability and autonomy. JIAC V applications can be developed using extensive functionality that is provided in a library. This library consists of already-prepared services, components, and agents which can be integrated into an application in order to perform standard tasks. The individual agents are based on a component architecture which already provides the basic functionality for communication and process management. Application-specific functionality can be provided by the developer and be interactively integrated.	Exploits Apache ActiveMQ as transport Declarative agent definition through Spring	Not FIPA compliant	Apache v2.0
SARL	SARL is a general-purpose agent-oriented language. SARL aims at providing the fundamental abstractions for dealing with concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration. These high-level features are now considered as the major requirements for an easy and practical implementation of modern complex software applications. We are convinced that the agent-oriented paradigm holds the keys to effectively meet this challenge. Considering the variety of existing approaches and meta-models in the field of agent-oriented engineering and more generally multi-agent	Exploits ZeroMQ as transport	Not FIPA compliant	Apache v2.0

[1] https://en.wikipedia.org/wiki/Multi-agent_system

	systems, our approach remains as generic as possible and highly extensible to easily integrate new concepts and features. The language is platform- and architecture-independent.			
JADE	JADE (Java Agent DEvelopment Framework) is a software Framework fully implemented in the Java language. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications and through a set of graphical tools that support the debugging and deployment phases. A JADE-based system can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required. JADE is completely implemented in Java language and the minimal system requirement is the version 5 of JAVA (the run time environment or the JDK). (Bellifemine, et al., 2000)	FIPA compliant	Supports only RMI transport. Few extensions for JMS / XMPP transports	LGPL
SPADE[2]	SPADE (Smart Python multi-Agent Development Environment) is a Multiagent and Organizations Platform based on the XMPP/Jabber technology and written in the Python programming language. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML, just like FIPA-ACL. Many other agent platforms exist, but SPADE is the first to base its roots on the XMPP technology. (Bellifemine, et al., 2000)	FIPA compliant	Uses a custom XMPP server	LGPL

As it is possible to notice from Table 2, none of the platforms offers both a distributed approach and a general-purpose implementation for the communication back-end.

The COMPOSITION marketplace, therefore, represents a unique example of a marketplace based on semi-automatic multi-agent collaboration.

[2] Smart Python Agent Development Environment. Available: https://code.google.com/p/spade2/

5 Agent Container Technology

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

The marketplace includes the following elements: (a) a Marketplace management portal; (b) an easy to deploy Marketplace core based on Docker; (c) a set of "default" agent implementations ready to be adopted by interested stakeholders.

The main building blocks of the Marketplace are displayed in Figure 1 below.

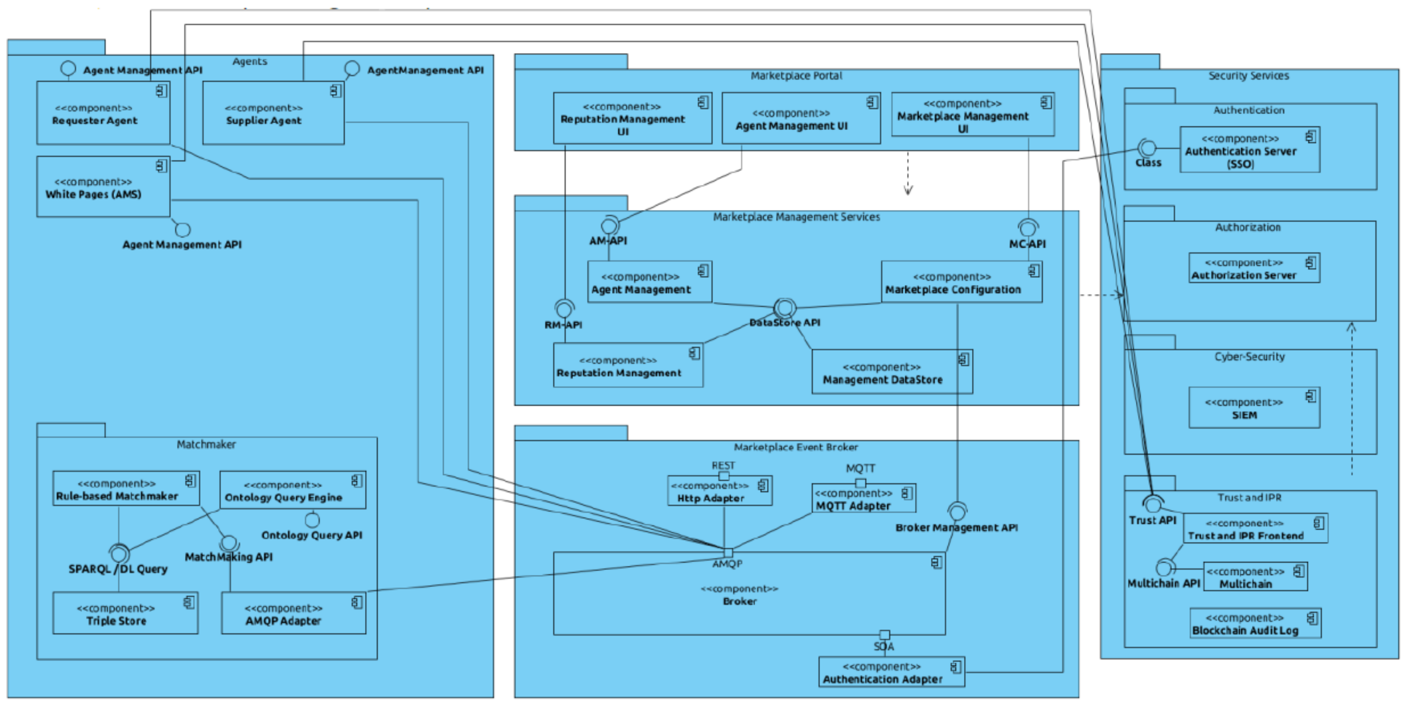

Agents
Agent Management API
AgentManagement API
<<component>> Requester Agent
<<component>> Supplier Agent
<<component>> White Pages (AMS)
Agent Management API
Matchmaker
<<component>> Rule-based Matchmaker
<<component>> Ontology Query Engine
Ontology Query API
SPARQL /DL Query
MatchMaking API
<<component>> Triple Store
<<component>> AMQP Adapter
Marketplace Portal
<<component>> Reputation Management UI
<<component>> Agent Management UI
<<component>> Marketplace Management UI
Marketplace Management Services
AM-API
MC-API
<<component>> Agent Management
<<component>> Marketplace Configuration
RM-API
DataStore API
<<component>> Reputation Management
<<component>> Management DataStore
Marketplace Event Broker
REST
MQTT
<<component>> Http Adapter
<<component>> MQTT Adapter
Broker Management API
AMQP
<<component>> Broker
SOA
<<component>> Authentication Adapter
Security Services
Authentication
Class
<<component>> Authentication Server (SSO)
Authorization
<<component>> Authorization Server
Cyber-Security
<<component>> SIEM
Trust and IPR
Trust API
<<component>> Trust and IPR Frontend
Multichain API
<<component>> Multichain
<<component>> Blockchain Audit Log

Figure 1: Marketplace Components

As stated in previous deliverable D2.3 there are two main categories of agents that can be defined a priori, depending on the kind of services provided:

- Marketplace agents
- 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.

Stakeholder agents can be divided in two different categories, Requester and Supplier. From an implementation point of view, they are very similar and share a large set of features, especially the communication protocols used for the interaction with stakeholders and other agents.

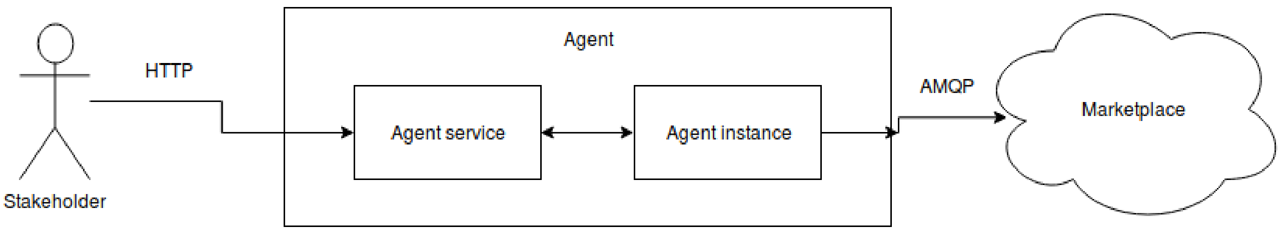


Figure 2: Simplified agents' communication logic

The communication protocols used by the agent for the interaction with the stakeholder on left side and with other agents on the marketplace on the right side are shown in Figure 2[3].

Any agent can be controlled by the stakeholder via RESTful APIs which will be listed and explained in the following sections. Agents on the marketplace can exchange messages using AMQP as transport layer.

5.1 AMS

According to FIPA specifications,[4] 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. Prior to any operation, every agent should register to the AMS to get a valid agent identifier, which will be unique within the agent platform.

In COMPOSITION the marketplace is the platform where agents operate.

In addition to this registration service the AMS has the role of collector of requests coming from the agents towards other useful services inside the platform. This is done to access additional features offered by the COMPOSITION Marketplace, as the ones offered by the Matchmaker.

5.1.1 White Pages Service

A White Pages service is a mandatory component of any MAS system. It 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[5], use MySQL as well for offering the White Pages service.

The table storing the agents has the following schema:

Table 3: Table schema

Agent_id	Agent_owner	Agent_role
The unique identifier for the agent. It is the primary key for the table, since it has the constraint of being unique.	The name of the company owning the agent.	The role of the agent on the Marketplace, can be either 'requester' or 'supplier'.

Since the directory service is offered by the AMS, it is the only agent allowed to directly interact with the database. When the AMS is executed, it needs the following configurations:

- IP address of the host running the MySQL instance.
- Username and password for CRUD operations on the database.

Name of the database to operate on, since the same instance may run different databases.

[3] More details are available in deliverable D6.5: Connectors for Inter-factory Interoperability and Logistics I.
[4] http://www.fipa.org/specs/fipa00023/SC00023K.pdf
[5] https://pypi.python.org/pypi/SPADE

MySQL database is implemented as NDB cluster to provide a set of additional feature as high availability and high redundancy, this solution improves system scalability and increase high load handling.

5.2 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 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 D6.6: Connectors for Inter-factory Interoperability and Logistics II.

Both kinds of agents share a set of features allowing them to respond in different ways to the same kind of solicitations (such as input from GUI, messages from other agents) according to the state of their current behaviour (when the solicitation is received). Agents support two different behaviours:

1. Authentication behaviour: This is triggered upon each agent activation. It consists of all the procedures that are necessary for the agent to correctly operate on the marketplace.
2. Negotiation behaviour: This is the behaviour each agent needs to support in order to be able to participate in marketplace negotiations. It is designed in a way that allows different negotiation protocols to be adopted.
3. Blockchain verification behaviour: This behaviour is implemented to allow agents to verify the message content and its provenance.

The authentication behaviour is the first one engaged by any agent since:

1. It takes care of getting the agent identifier from the AMS
2. It authenticates the agent with the messaging broker

Both these phases are mandatory for any agent to operate on the marketplace. After the authentication process, the negotiation behaviour comes to play. It allows the agent to perform transactions and interactions with the other agents on the marketplace. Its behaviour is constrained to the negotiation protocol that is being used.

Agents might or might not support a certain level of 'intelligence' in their decisions regarding certain negotiation protocols. For example, an agent might have the capability of self-evaluation for the received offers, whereas another agent might not have such capability and, therefore, it will involve the Matchmaker during the decision process. An agent having such level of intelligence will be described as 'smart', while it will be described as 'dumb' in absence of that.

The Blockchain layer is added on top to provide integrity and provenance properties. The Blockchain module provides the possibility to store public key of the agents and messages. Thanks to this feature it is possible to check and verify every single incoming message against the information stored in the Blockchain. This means that an agent will be sure about the message integrity (by checking the message with the one stored) and about the message provenance (by verifying it with the public key of the sender agent). The full description of the Blockchain's mechanisms is provided in deliverable "D4.3 The COMPOSITION Blockchain".

Use cases UC-KLE-4, UC-ELDIA-1[6] and UC-ATL-3 have been considered to drive the development and the prototyping of the agents which are described in the following sections.

5.2.1 Requester Agent

The Requester Agent is the agent used 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 baseline

[6] Description of these use cases are available in deliverable D2.1: Industrial use cases for an Integrated Information Management System

protocol, which must be supported by any COMPOSITION agent, is the so-called CONTRACT-NET[7]. In such protocol, a Requester agent plays the role of “Initiator”.

As described by Shoham (Shoham Y., 1993)), any agent can pass through a different set of states; the state of an agent consists of components such as beliefs, decisions, capabilities and obligations. In the CONTRACT-NET protocol Requester agents go through different sets of states, performing different actions upon receipt of a message (either from the UI or from other agents) according to their current state. In the following figure (Figure 3), the states for Requester agent are shown with arrows indicating the allowed transitions between one state and another.

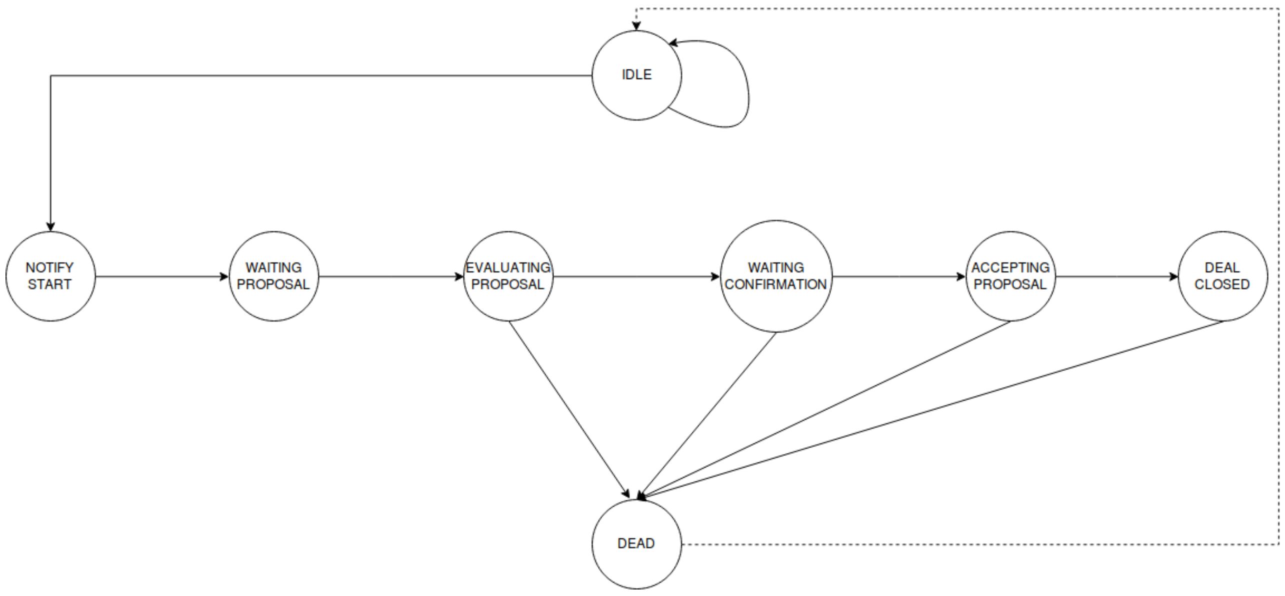


Figure 3: Requester states

5.2.1.1 IIMS Connection

One of COMPOSITION’s strengths is the tight connection existing between intra-factory and inter-factory components. This connection allows the agent on the marketplace to operate in the optimum way to fulfil the stakeholder’s needs.

Concerning the Requester agent, such connection is the starting point of a new negotiation protocol: the IIMS notifies, through the Learning Agent, that a certain good/service is needed. Upon receipt of such notification, the Requester agent can react accordingly, using the most suitable protocol that fulfils the request. An example is the use case UC-KLE-4, here described from the point of view of the Requester agent:

1. COMPOSITION system automatically notifies KLE about the scrap metal bins fill level, through a notification from the Learning Agent (which receives the data, as defined in the DFM, from the BMS collecting them from the sensors installed in the factory’s premises) to the Requester Agent
2. Requester agent has the logic for starting a new bidding process through default parameters (set by KLE)
3. Requester agent starts the bidding process, asking to the Matchmaker about the list of the Supplier agents currently available on the marketplace that are capable in replying to such offer
4. Offers come in to Requester agent, which can either evaluate them locally (smart agent behaviour) or send them out for evaluation towards the Matchmaker agent
5. Once the deadline for receiving proposals has expired, Requester agent notifies the UI about the best one(s) received from the Matchmaker
6. Stakeholder can now choose the preferred offer, select it and notify the Requester agent, that can finish the negotiation with the corresponding Supplier agent

[7] The CONTRACT-NET protocol has already been described in previous deliverable D2.3 and D6.5, therefore it will not be furtherly discussed in this deliverable.

Another relevant use case has been added to take advantage of the Marketplace architecture. This is called UC-ATL-3 and it is about searching for software solutions.

The company offers its software services through the composition marketplace, registering them at the matchmaker agent.

When another partner on the marketplace is looking for a certain software solution, it can contact the matchmaker on the marketplace which, based on the more opportune ontologies, will reply with the ATL agent identifier.
In this way, the two agents will eventually reach an agreement.

This use case is presented here from the Requester point of view only, simply because it's a common functionality that this type of agent shares with his counterpart.

1. An Agent receives an UI message with "search" as specified action and a service described inside
2. The Agent forward the message to the Matchmaker that will reply with a list of all possible suppliers for the requested solution

This is a very simple and flexible solution that will enable everyone to get a list of suppliers that provide the service needed.

5.2.2 Supplier Agent

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.

As described by Shoham (Shoham Y., 1993), any agent can pass through a different set of states; the state of an agent consists of components such as beliefs, decisions, capabilities and obligations. In the CONTRACT-NET protocol Supplier agents go through different sets of states, performing different actions upon receipt of a message (either from the UI or from other agents) according to their current state. In the following figure (Figure 4), the states for Supplier agent are shown with arrows indicating the allowed transitions between one state and another.

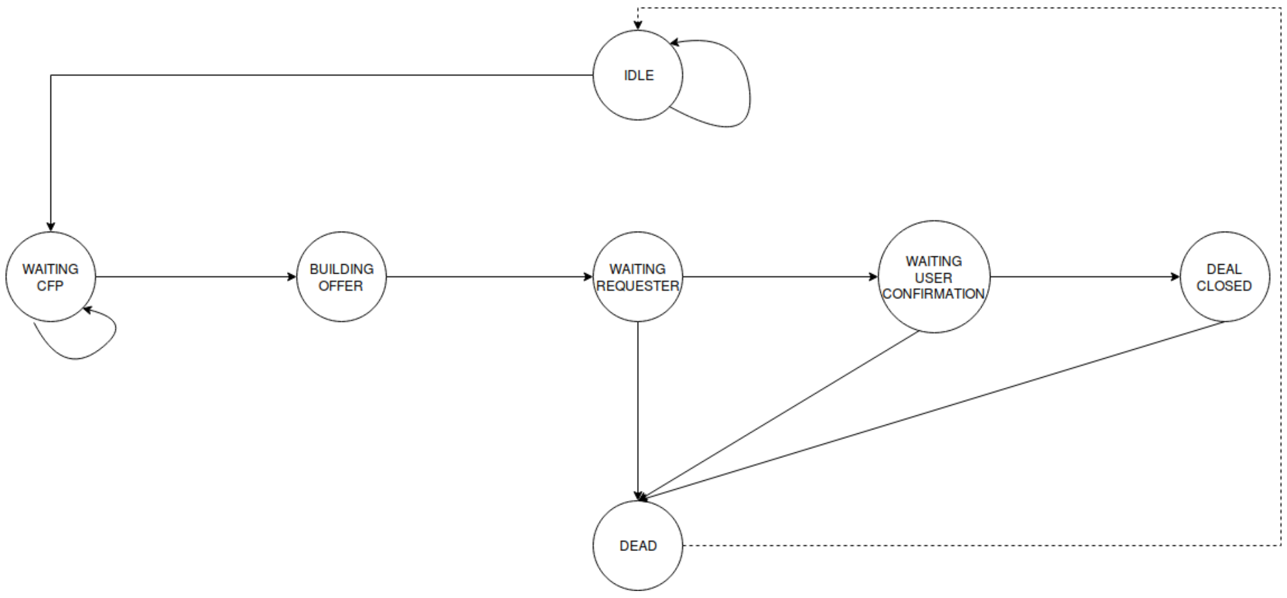


Figure 4: Supplier states

5.2.2.1 IIMS Connection

As mentioned in section 5.2.1.1, the tight connection between intra-factory and inter-factory systems is one of COMPOSITION's strengths. For the Supplier agent, this connection is exploited to dynamically adapt to the market needs.

Concerning the Supplier agent, it will be notified of a new request incoming in the system, about a good/service, that can be satisfied by the Supplier. Upon receipt of such notification, the Supplier agent can react accordingly, making the best offer he can afford. An example is the use case UC-KLE-4, here described from the point of view of the Supplier agent:

1. Supplier agent receives the new request from KLE Requester agent through the marketplace broker
2. Supplier agent will evaluate the request and will make an offer
3. The Supplier agent will wait the Requester selection and will be notified if the offer has been accepted or refused

For the Supplier agent, this connection has been also developed as part of the use case UC-ELDIA-1, here briefly described from the Supplier agent point of view:

1. The Supplier agent receive from the UI data about goods price
2. When such information arrives, the Supplier agent will send it to the Deep Learning Toolkit, to exploit it for predictions of future price of the considered good

This feature aims to have a reference for a possible bidding process by looking at the foreseen prices of a certain good.

5.3 Matchmaker

The COMPOSITION 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 fulfill a request for a service with raw materials or products involved in the supply chain. Different decision criteria for supplier selection are considered by the Matchmaker according to several qualitative and quantitative factors.

The Matchmaker component acts as a kind of special agent in the Marketplace. The matchmaking component is connected with the rest agents using REST protocol. The Matchmaker receives requests from the Agents, infers new knowledge by applying semantic rules to Collaborative Manufacturing Services Ontology and then responses back to agents.

The matchmaking module which is described at D2.4 - The COMPOSITION architecture specification II, comprises a complete semantic framework which offers to the Marketplace agents a high-performance ontology store with querying capabilities and a matchmaking engine which provides efficient matching in both agent and offer level.

The agents can access the ontology store using the Ontology API and its querying interfaces. The API provides to marketplace participants a catalogue of services for data access and management using SPARQL queries. This component is able to query the whole dataset with a satisfactory level of efficiency. Every marketplace agent is able to send a request for a service using HTTP protocol and the agent exchange language from the marketplace, which is described in JSON format. Every service translates the request into a SPARQL query and applies it to the ontology model in the store. The requests' types are GET and POST for retrieve, store or delete data.

The matchmaking services are available to Marketplace agents by the Rule-Based Matchmaker component. It is developed in Java language and it is offered through RESTful web services. The Rule-Based Matchmaker will be used by Marketplace's agents in order to match requests and evaluate offers between the agents. The Matchmaker's functionalities are exclusively depended on the Collaborative Manufacturing Ontology. The Matchmaker package is deployed as a Docker container to the COMPOSITION production server and offers to the agents the following functionalities:

- *Semantic Matchmaking between Marketplace agents – Agent Level matchmaking*: The engine provides the matching of an agent who sends a request for a service to the agents who provide services related to the request. The matchmaking is performed by using a generic terms dictionary that matches every vendor specific process and raw material to common terms in the ontology. Therefore, the Matchmaker is able to match agents that offer different types of services (waste management, sell raw materials, provide software solutions) in the Marketplace with the agents that request them. Furthermore, the matchmaker is able to match an agent who offers e.g. raw materials with possible customers by applying reasoning at the customers' manufacturing services and perform matching to resources or material level, which were using these services.

- *Offers Evaluation – Offer Level Matchmaking:* A marketplace requester agent is able to send a list of provided offers by the possible service providers to the matchmaker for evaluation. The Rule-based Matchmaking Engine applies set of rules in order to match the request with the best provided offer. Based on some predefined criteria, i.e. the requester prefers the cheapest solution over the quickest one, the matchmaker reorganize the list of applied rules and offers different results for each request. This kind of offers evaluation fits perfect to the needs of simple evaluations for scenarios such as KLE-4. In the case of KLE-7 and the evaluation of raw materials' offers the rule-based approach is enhanced with weighted-average algorithms in order to cover the needs of a more complex evaluation.

The figure below presents the information flow between the agents of the collaborative manufacturing eco-system and the Matchmaker:

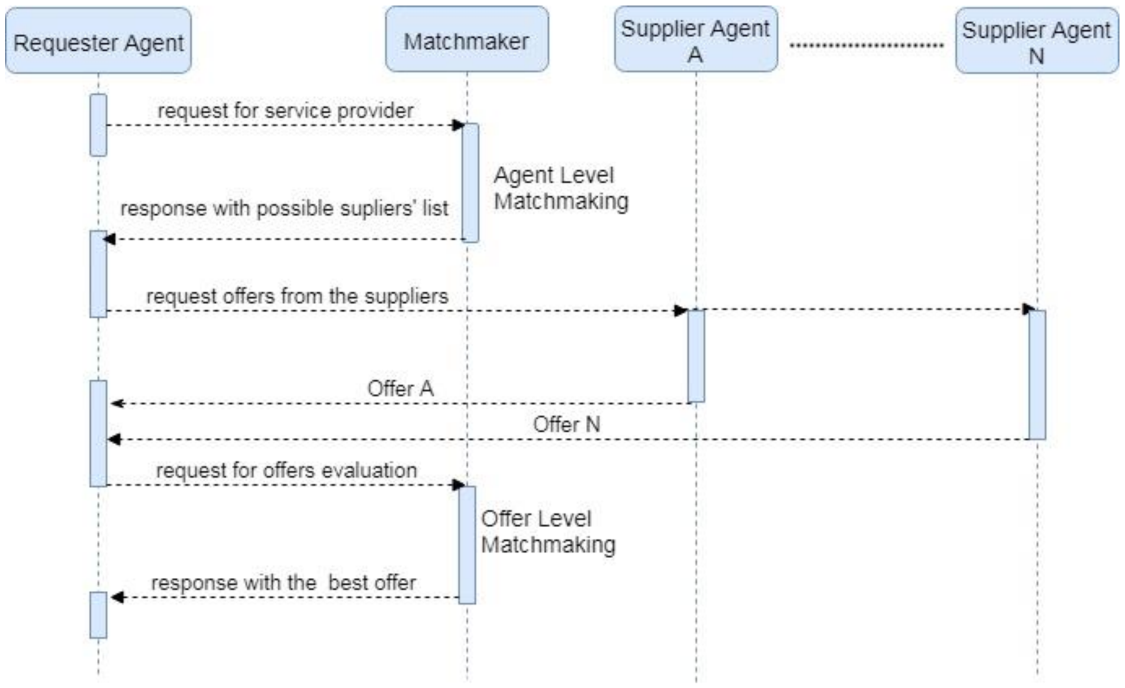

Requester Agent
Matchmaker
Supplier Agent A
Supplier Agent N
request for service provider
Agent Level Matchmaking
response with possible supliers' list
request offers from the suppliers
Offer A
Offer N
request for offers evaluation
Offer Level Matchmaking
response with the best offer

Figure 5: High Level Information Model of Matchmaker and Agents Collaboration

5.4 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 following 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 been changed a bit compared to the one defined in the previous deliverable D2.3 by adding a "message-id" field to keep track of the single message inside a conversation. It is, however, listed here for clarification purposes.

```
{
  "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"
    },
    "message-id": {
      "type": "string",
      "description": "Provides an identifier for the single message, to be used in conjuction with message-id to
have a way to uniquely address a certain message."
    },
    "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
}
```

Different ontologies have been developed in order to address the different scenarios and support the features needed by the Marketplace.

Here is reported one of the ontologies used by the agent to communicate. Is shown here the “proposal ontology” the most important one that enable agents to go through a negotiation process.

```
{
  "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"
        }
      }
    },
    "bid-ended": {
      "description": "True if the bid has ended, False otherwise",
      "type": "boolean"
    },
    "deal-closed": {
      "description": "True if a deal has been found, False otherwise",
      "type": "boolean"
    },
    "bid-winner": {
      "description": "If there's been a winner, details are here",
      "type": "object"
    },
    "message-id": {
      "description": "Identifiers of the message for which signature has been computed.",
      "type": "string"
    }
  }
}
```

An ontology will be specified inside the appropriate field of a CXL message and will enable the understanding of the “content” field.

6 Marketplace Infrastructure

In this section we'll detail the functional components of the infrastructure which will provide the different features stated in the design of the marketplace architecture:

- Design decisions and implementation regarding reliability, focusing also on the security aspects
- Design decisions and implementation regarding scalability
- Design decisions and implementation regarding the Marketplace Management Portal

6.1 Infrastructure and Reliability

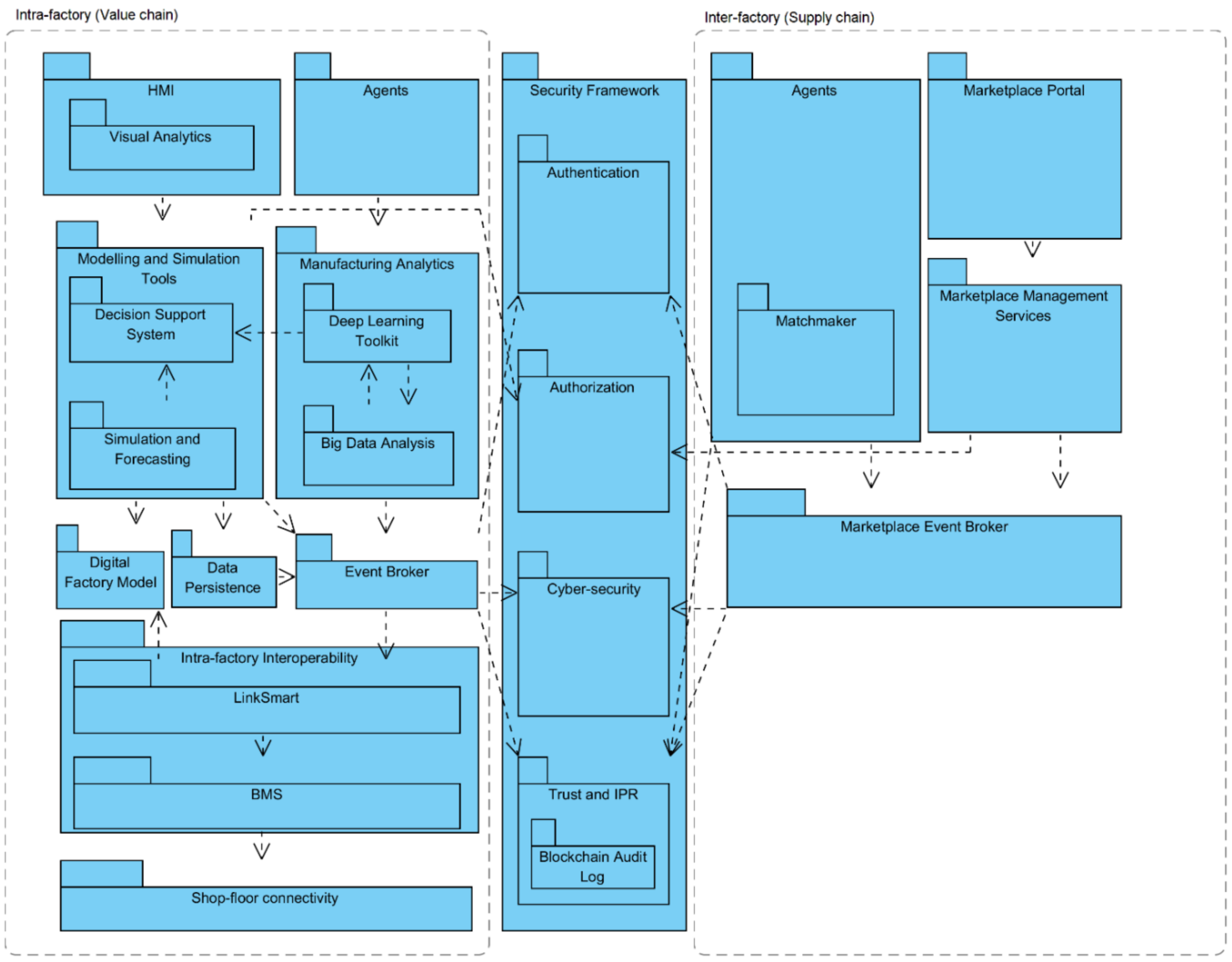


Figure 6 COMPOSITION infrastructure

There are different features of the Security Framework[8] aimed to achieve reliable delivery (messages are always delivered, even when there are failures in any component of the system). This reliability aspect is covered using RabbitMQ[9], a message broker that implements the following reliability functionalities:

- Acknowledgements and confirms
- Clustering and high availability
- Ensuring messages are routed
- Consumer cancel notification

[8] The design of the Security Framework is described in Deliverable D4.2: Design of the Security Framework II
[9] https://www.rabbitmq.com/

The scalability aspect is covered by the option of clustering the message broker. This operation will provide a collection of nodes, a logical grouping of one or several Erlang nodes[10], each one running an instance of the message broker, sharing users, virtual hosts, queues, exchanges, bindings and runtime parameters.

The Security Framework is designed as a transversal set of components, which allows to both approaches of the COMPOSITION platform (intra and inter factory) to use functionalities regarding authorization, authentication and cybersecurity. As well as the marketplace is totally fit into the initial design, it'll be fully covered with the Security Framework functionalities.

6.1.1 Message Broker

Message transport used in the marketplace has been kept agnostic with respect to the agents programming languages. This has been possible by using the CXL, as described in 5.4, and a message broker that will be described in this section.

The message broker system deployed in the Marketplace is integrated in the Security Framework architecture, it is one of the five pillars described in detail in D4.2 (authentication service, authorization service, message broker authorization and authentication service, XL-SIEM and reverse proxy).

Regarding the message management system implementation, it'll provide identity, access management and authorization policies centralised in COMPOSITION authorization and authentication services instead of having them in different components.

The implementation of the Message Broker has been done using RabbitMQ, an open source message broker (see 6.2.1). The decision was taken because of its performance, relatively small resource demands and ease of deployment. Furthermore, it supports high-scale and high-availability requirements by its deployment in distributed and federated configurations.

6.2 Scalability

The performance and scalability design described in the previous version of this deliverable has proven sufficient for the pilot installations, and the principles are well proven. Installations in the exploitation phase may have higher demand for scalability, and wider of the designs and techniques described below may be applied.

Attributes identified that may affect workload of system or components and thus the need for resources:

- The number of marketplaces
- The number of stakeholders in a marketplace
- The number of concurrent requests for Matchmaker services
- The number of concurrent agent negotiations taking place
- The number of participants in each negotiation.
- The number of data sharing agreements/links.

The strategies for scaling have been described in "D2.3 The COMPOSITION architecture specification I" and "D6.1 Real-time event broker I" and further updated in "D2.4 The COMPOSITION architecture specification II" and "D6.2 Real-time event broker II". A brief summary will be provided in this section.

We have defined scalability in earlier deliverables as the ability of a system to increase the maximum workload it can handle by expanding its quantity of consumed resources. The maximum workload of the system can then be increased by expanding its quantity of available resources. We can increase the quantity of consumed resources by increasing the amount of resources within existing execution nodes, or by adding more execution nodes. To scale up (or scale vertically) is to increase overall application capacity by increasing the resources within existing nodes. To scale out (or scale horizontally) is to increase overall application capacity by adding nodes, e.g., adding an additional message broker.

The scaling strategies discussed here concern the design decisions that may be taken to ensure that the available resources are used most efficiently and that additional resources may repetitively be added if the maximum workload of the system with the given resources is reached. Most components of the marketplace

[10] http://erlang.org/doc/reference_manual/distributed.html

are easy to scale out, transparently to other components, by standard strategies such as clustering and load balancing.

The Ontology Store is built using Jena and TDB, a fast and scalable persistent triple store for RDF storage and query. This can be scaled out as needed and load balanced. Matchmaker instances work on a copy of the store and so can be easily scaled out with several Matchmakers handling service requests.

The Marketplace Portal and Marketplace Services are web interfaces and web services that may also be scaled out using load balancing (e.g. using Nginx[11]).

The CXL messages between agents are exchanged via the message broker, both for data sharing and agent negotiation. The main concern for scalability is for the Message Broker, which is affected by all of the attributes above. The rest of this section will be dedicated to Message Broker scalability design.

6.2.1 RabbitMQ

For the message broker, there are two levels of design that will affect scalability which we will refer to as routing topology and broker topology. 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). Broker topology deals with the distribution of logical brokers on nodes, by clustering (on 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 (if not configured as mirrored queues). Queues are implemented as processes, whereas exchanges are just database entries. Thus, creating a queue for an agent will only create a new process in one broker in the cluster. Publishing and deleting of messages is replicated on all mirrored queues. A cluster without mirrored queues will have greater throughput than a single broker node.

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. The brokers may use different versions of RabbitMQ and be otherwise unsynchronized. (As noted in D6.1, 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.

The shovel moves messages from an exchange or queue in one logical broker to a destination exchange or queue in another logical broker.

RabbitMQ also 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 would not affect the primary exchange.

Routing topology design could e.g. favour many fanout exchanges or fewer exchanges and more use of routing. Fanout exchanges are slightly faster than topic and header exchanges. However, the difference is not a deciding factor in the choice of topology. There is no fixed limit to the number of exchanges and queues in a broker.

6.2.2 Scalability Design

This section will discuss examples of possible scaling strategies for the marketplace agent communication. 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.

[11] https://nginx.org/en/

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 7). 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 disconnect. 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.

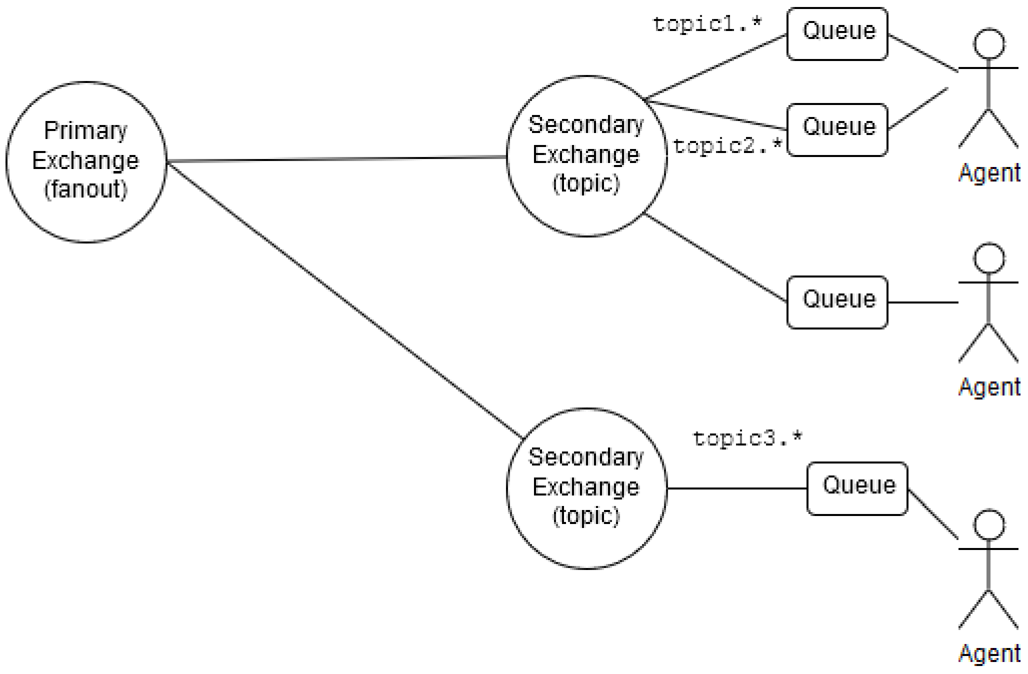

topic1.*
Queue
Primary Exchange (fanout)
Secondary Exchange (topic)
Queue
topic2.*
Agent
Queue
Agent
Secondary Exchange (topic)
topic3.*
Queue
Agent

Figure 7: 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 8). The number of participants in each negotiation will likely not be a limiting factor for the described topology.

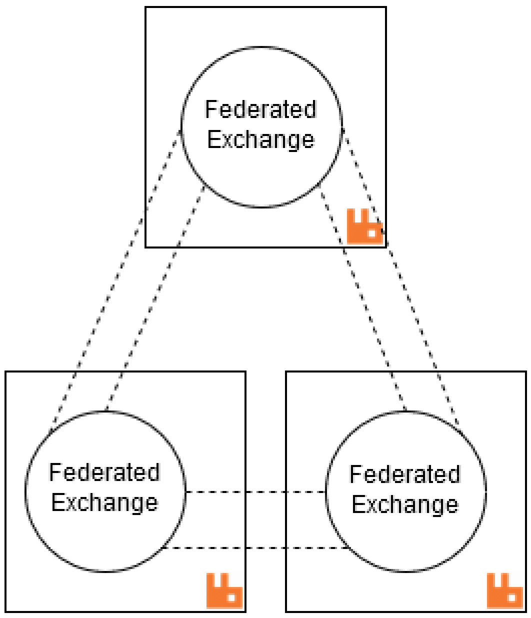


Figure 8: Federated exchanges broker topology

An exchange that only the involved parties can access can be set up for each data sharing agreement (Figure 9). 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 10), 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 will use a separate logical broker (cluster) in the marketplace depending on the load. This setup is currently under development in the pilot deployment.

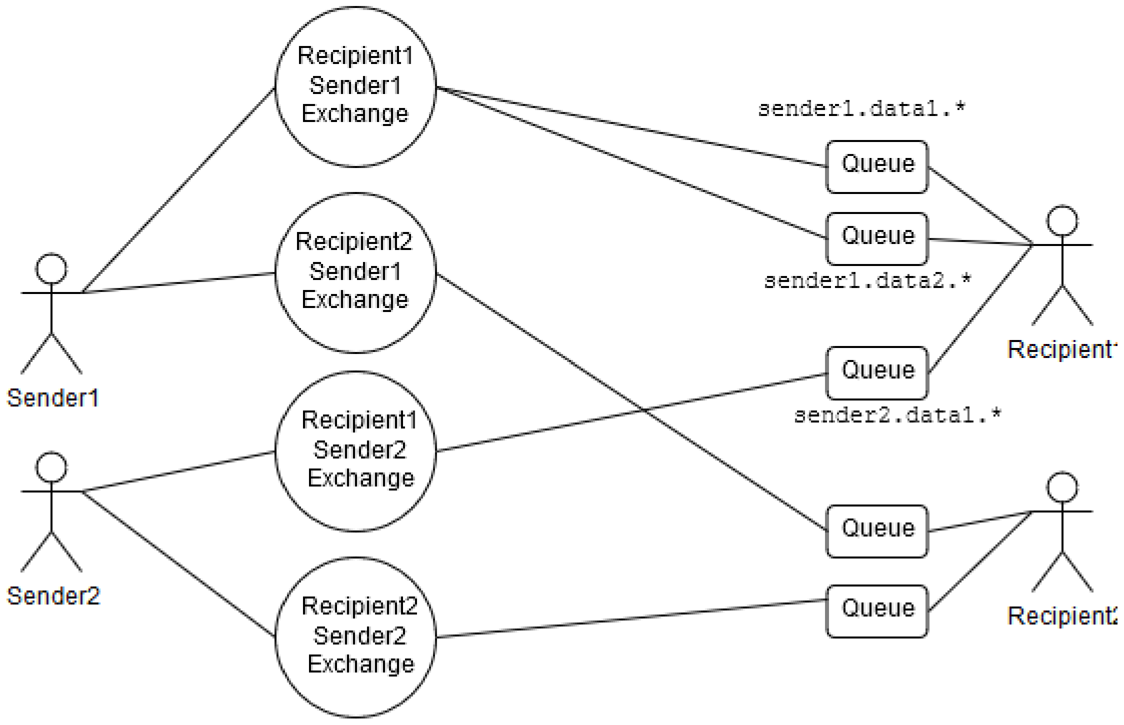

Recipient1 Sender1 Exchange
Recipient2 Sender1 Exchange
Recipient1 Sender2 Exchange
Recipient2 Sender2 Exchange
Sender1
Sender2
sender1.data1.*
Queue
Queue
sender1.data2.*
Queue
sender2.data1.*
Queue
Queue
Recipient
Recipient

Figure 9: Data sharing using one exchange per data sharing agreement

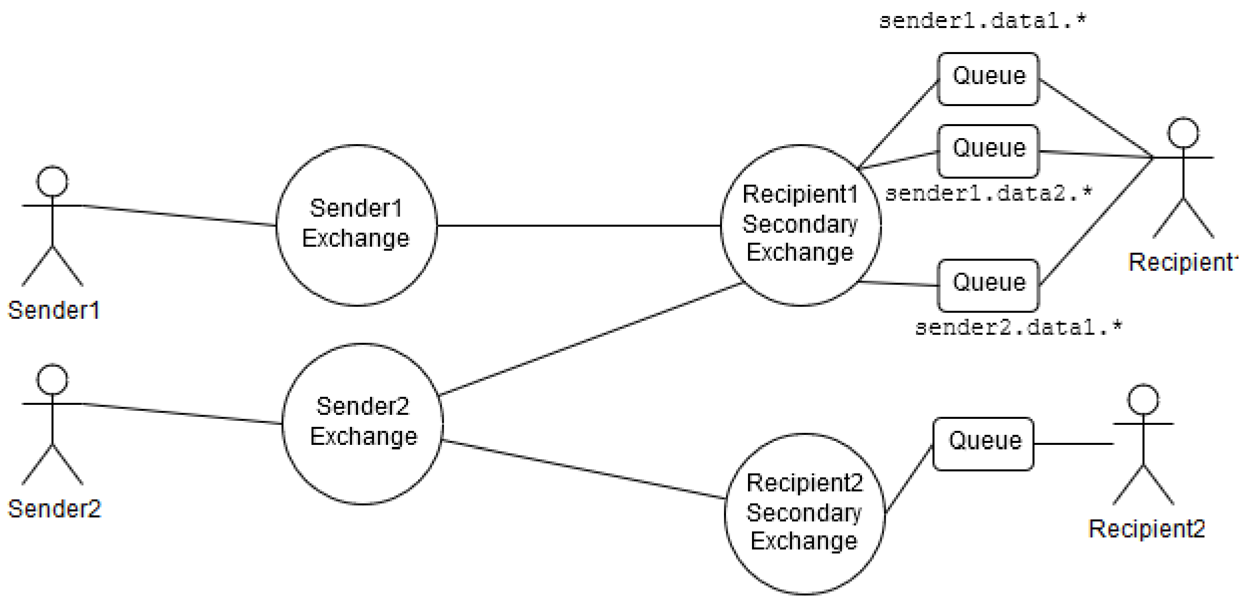


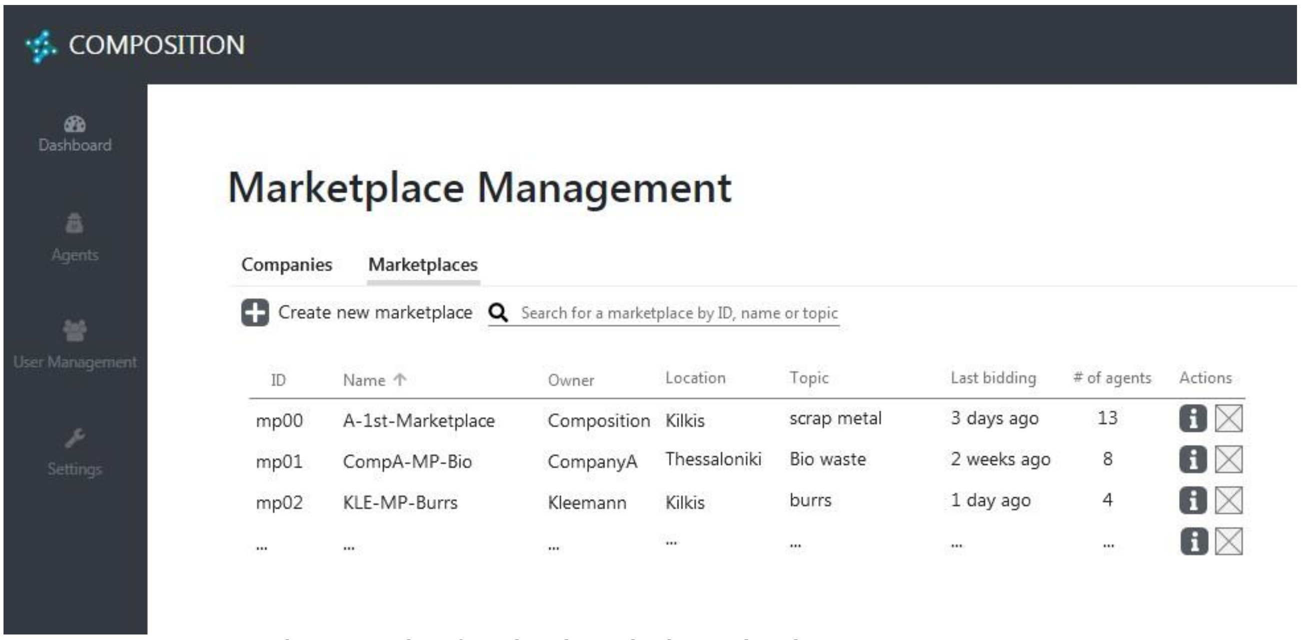
Figure 10: Data sharing using sender and recipient exchanges

6.3 Marketplace Management

The Marketplace Management component is composed by two main groups of elements respectively named:

- Marketplace Management Portal
- Marketplace Management Services

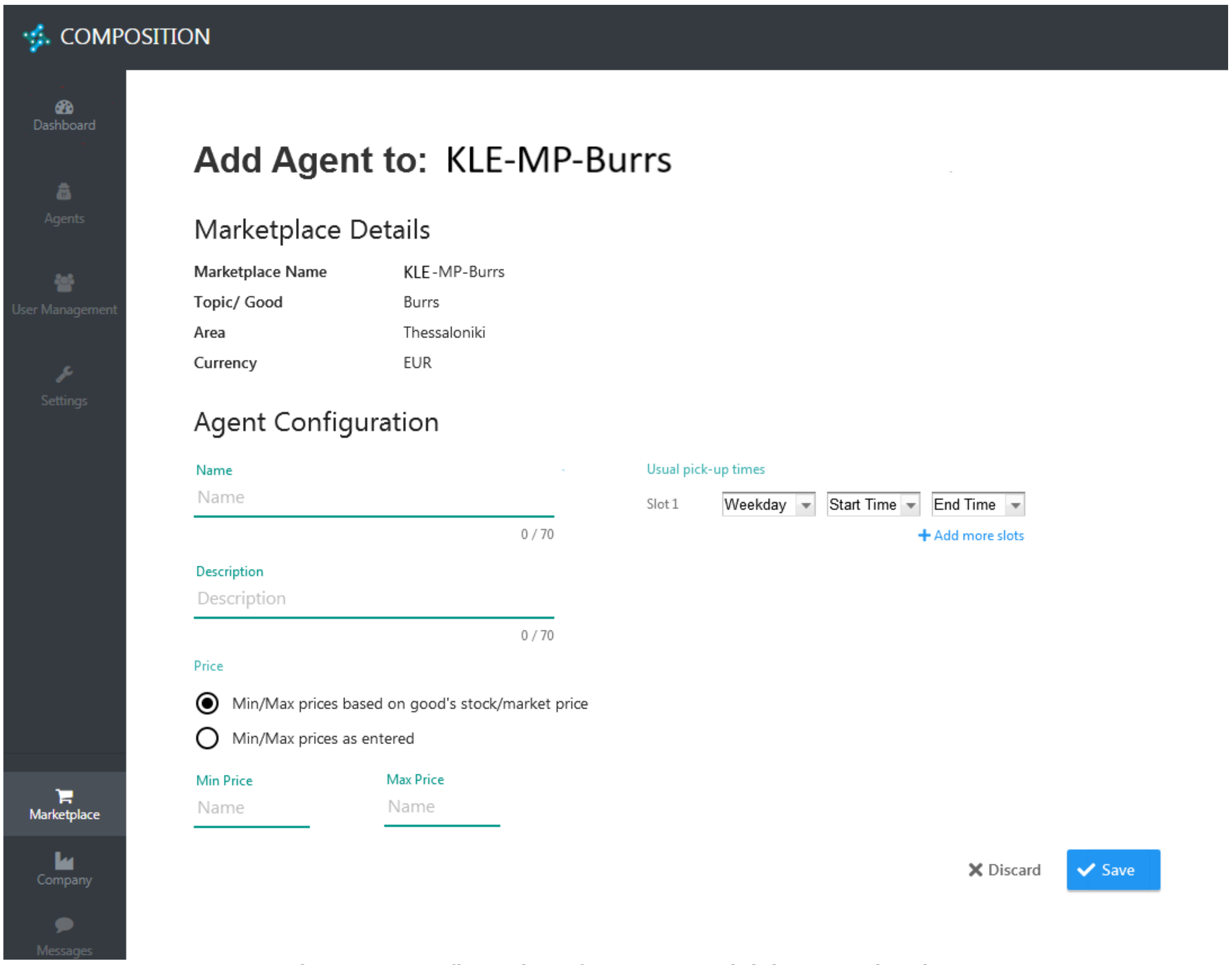
The former provides a web-based UI for managing a set of Composition Marketplaces, whereas the latter provides the backend, empowering the UI functions and allowing direct configuration and control of the marketplace event broker.



ID	Name	Owner	Location	Topic	Last bidding	# of agents	Actions
mp00	A-1st-Marketplace	Composition	Kilkis	scrap metal	3 days ago	13	
mp01	CompA-MP-Bio	CompanyA	Thessaloniki	Bio waste	2 weeks ago	8	
mp02	KLE-MP-Burrs	Kleemann	Kilkis	burrs	1 day ago	4	
...	...	...	...	...	...	...	

Figure 11: List of marketplaces in the Marketplace Management UI

Marketplace Management components are designed to support many operations that are crucial for the COMPOSITION ecosystem. For example, the Marketplace Management Portal allows stakeholders to join the COMPOSITION marketplaces and to receive the agent credentials and configuration parameters required by the corresponding agent containers to join. Figure 11 shows a list of available marketplaces with different topics and agents.



Add Agent to: KLE-MP-Burrs

Marketplace Details

Marketplace Name	KLE-MP-Burrs
Topic/ Good	Burrs
Area	Thessaloniki
Currency	EUR

Agent Configuration

Name: Name 0 / 70

Description: Description 0 / 70

Price

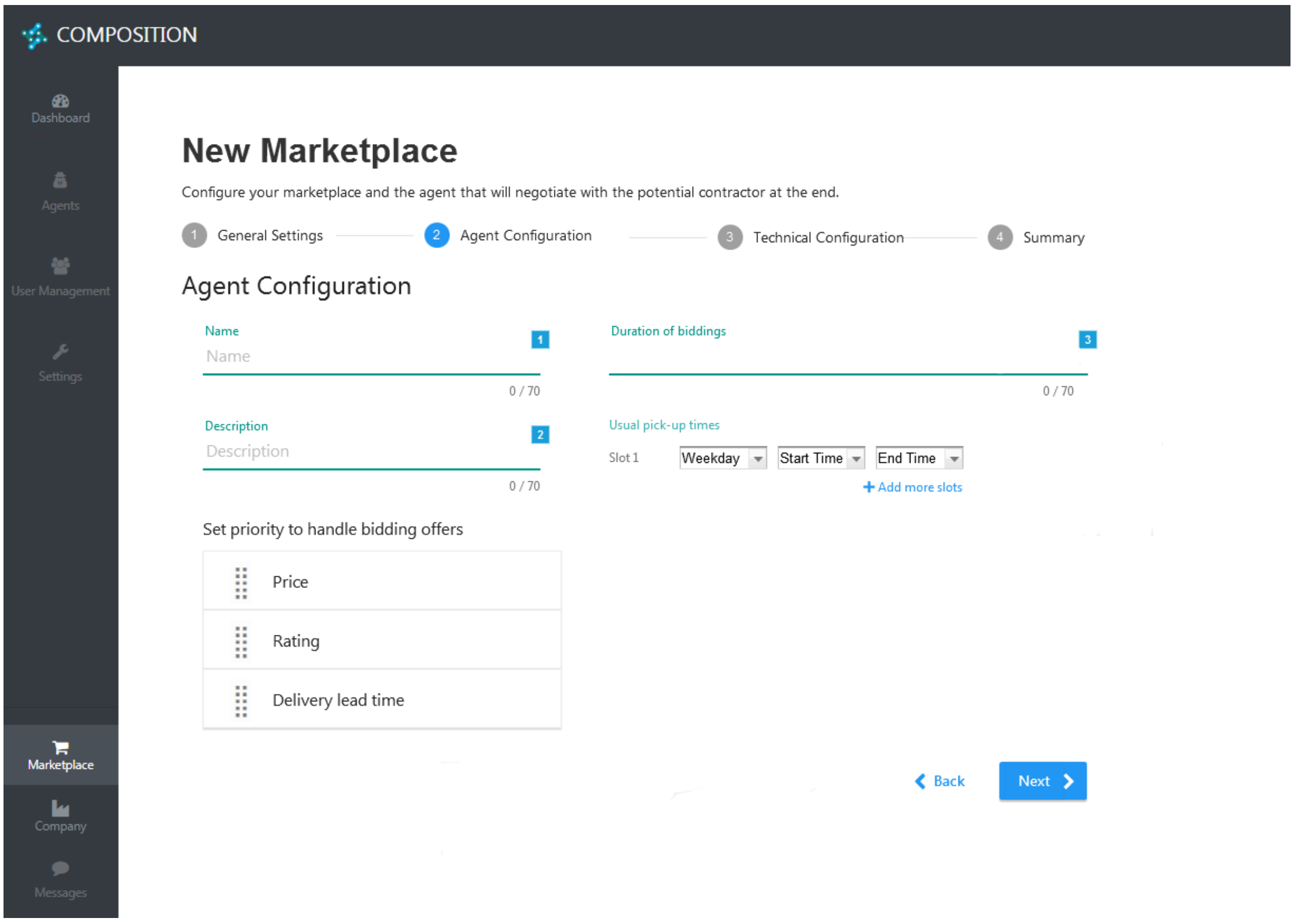
[✓ Min/Max prices based on good's stock/market price]
Min/Max prices as entered

Min Price: Name
Max Price: Name

Usual pick-up times
Slot 1: Weekday, Start Time, End Time
+ Add more slots

Figure 12: Configuration of a new agent joining a marketplace

Whenever a new stakeholder joins the COMPOSITION Open Marketplace, it must be classified by providing some specific data, such as the kind of business pursued, the category of goods provided and/or required, etc. Figure 12 shows the configuration of an agent trading in burrs in the area of Thessaloniki. The data provided by the agent is leveraged by the Management Portal to support complex search of available stakeholders, e.g., to conduct preliminary analysis of possible offers and/or possible actors to approach for selling services. Moreover, the same data is propagated to the Matchmaker agent which can take better decisions about possible matches between supply needs and registered suppliers. Coded in the agent is the behavior and trading strategies that will be applied, e.g. whether the lowest price will be used or stakeholders with high reputation scores will be favored. An agent implementation may be deployed in different configuration in different marketplaces.



New Marketplace

Configure your marketplace and the agent that will negotiate with the potential contractor at the end.

1 General Settings — 2 Agent Configuration — 3 Technical Configuration — 4 Summary

Agent Configuration

Name: Name 0 / 70

Description: Description 0 / 70

Duration of biddings 0 / 70

Usual pick-up times: Slot 1 Weekday Start Time End Time + Add more slots

Set priority to handle bidding offers
- Price
- Rating
- Delivery lead time

Back Next

Figure 13: Set-up of a new marketplace

The Marketplace Management supports setting up the configuration of new virtual and closed marketplaces. (Closed marketplaces require that the technical installation is complete.) If it is a closed marketplace, the endpoints of the message broker and matchmaker services and credentials to access must be entered. Some aspects of the Marketplace Management are at the time of writing still in development, however, the design of both the Marketplace Management Portal and the corresponding backend service are set. The full set of use cases describing the main functions that need to be provided at the end user level has already been defined and it is reported in Figure 13.

The Marketplace Management Services (MMS) provides a set of APIs that allow the user to create and to manage his company profile in the marketplace. The typical user of the MMS can be a sales manager or any other manager-level employee that signs-in the COMPOSITION ecosystem and that can exploit COMPOSITION services (e.g. agent-based supply chain management, company yellow pages, etc.).

After browsing the COMPOSITION Marketplace URL, the user can perform simple CRUD (Create-Read-Update-Delete) operations on a set of entities, that can be summarized in a few categories:

- Companies
- Agents
- Users

Thus, the user can create and manage company profile and its properties (i.e. name, full name, description). Moreover, he can manage agents and user. For the creation and management of users only some basic information is needed, such as username, first name, last name, description. The management of roles and the assignment of permissions (who can do what) is delegated to the responsible of authentication and authorization of users: the Security Framework. The Security Framework through Keycloak[12] add authentication to applications and secure services with minimum fuss and no need to deal with storing users or authenticating users. The operator can deploy its own intelligent agent on the marketplace specifying different parameters such as name, description, type (supplier or requester) and URL of image. Similarly to the other APIs, he/she can perform CRUD operations on the agents. In addition to this, agents are also

[12] https://www.keycloak.org/

linked to a docker container that is meant to run the actual agent instance. Indeed, while creating the agent, the backend triggers Docker APIs in order to initialize the container and start it. In turn, the agent will take care of calling the Agent Management System to register itself; the entire flow is shown in Figure 14.

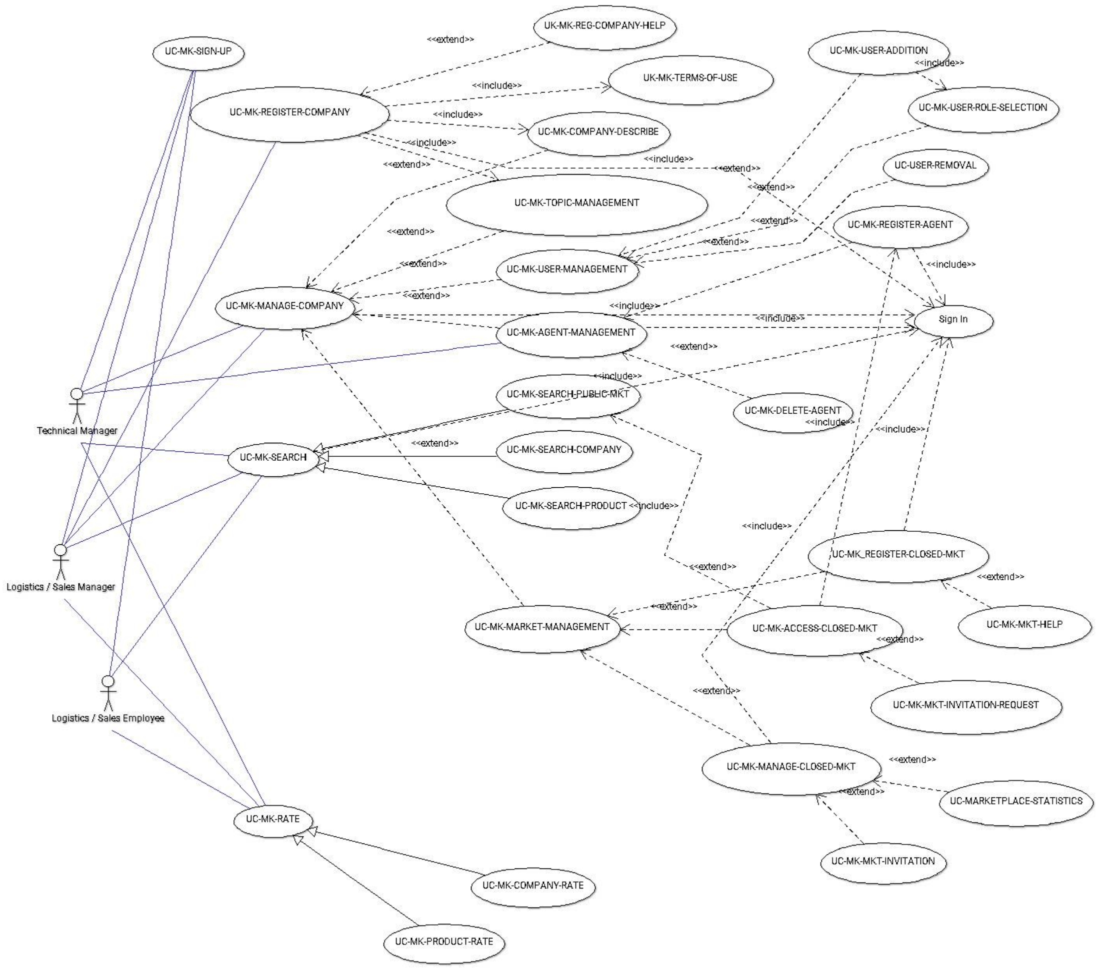
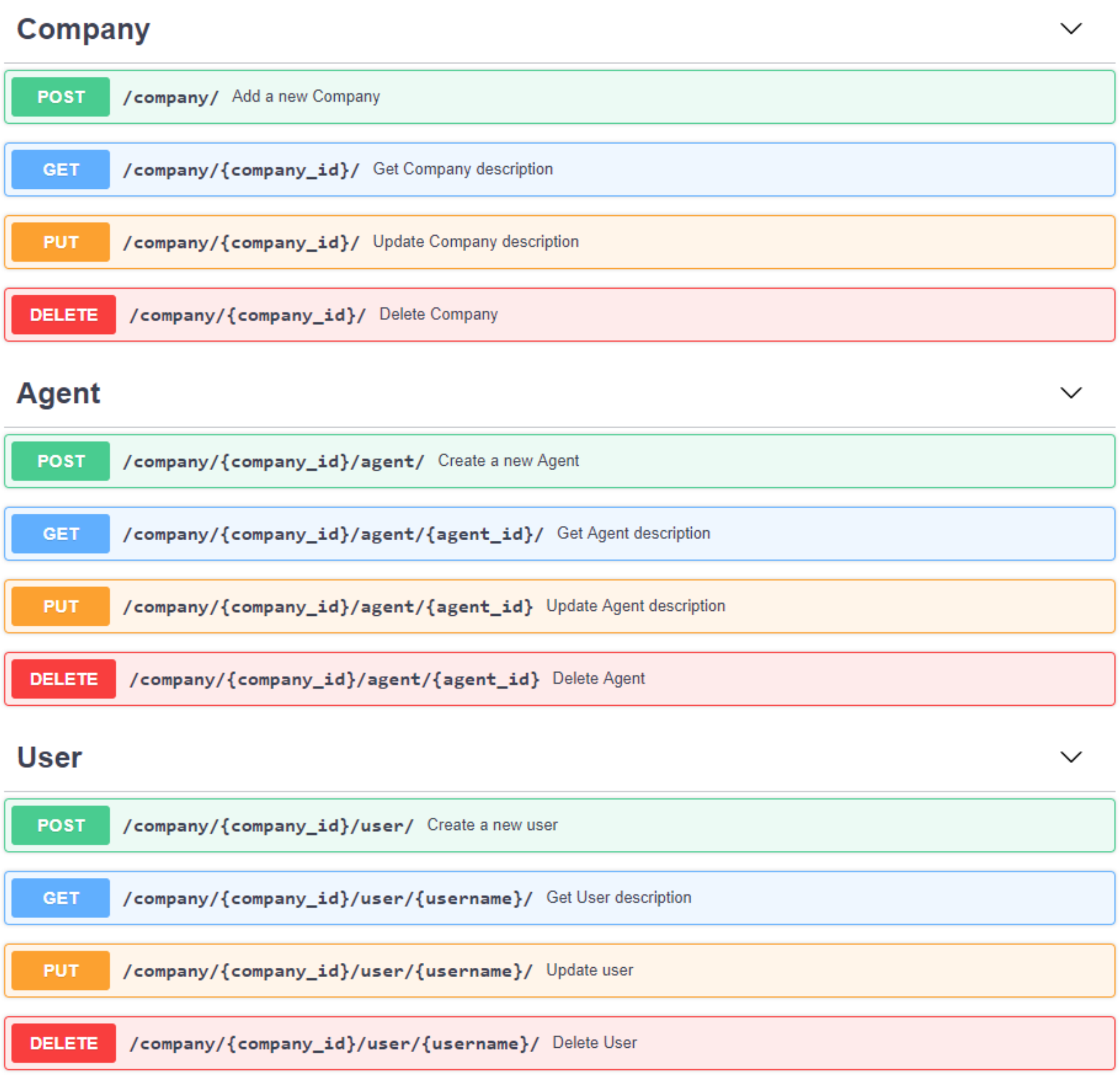


Figure 14: Marketplace management use cases

As for the creation, the user can update the agent by changing parameters or removing it and the backend takes care to call the Docker API for performing such operations directly on the container image.

A complete description of APIs is available in Figure 15.



Company

POST /company/ Add a new Company

GET /company/{company_id}/ Get Company description

PUT /company/{company_id}/ Update Company description

DELETE /company/{company_id}/ Delete Company

Agent

POST /company/{company_id}/agent/ Create a new Agent

GET /company/{company_id}/agent/{agent_id}/ Get Agent description

PUT /company/{company_id}/agent/{agent_id} Update Agent description

DELETE /company/{company_id}/agent/{agent_id} Delete Agent

User

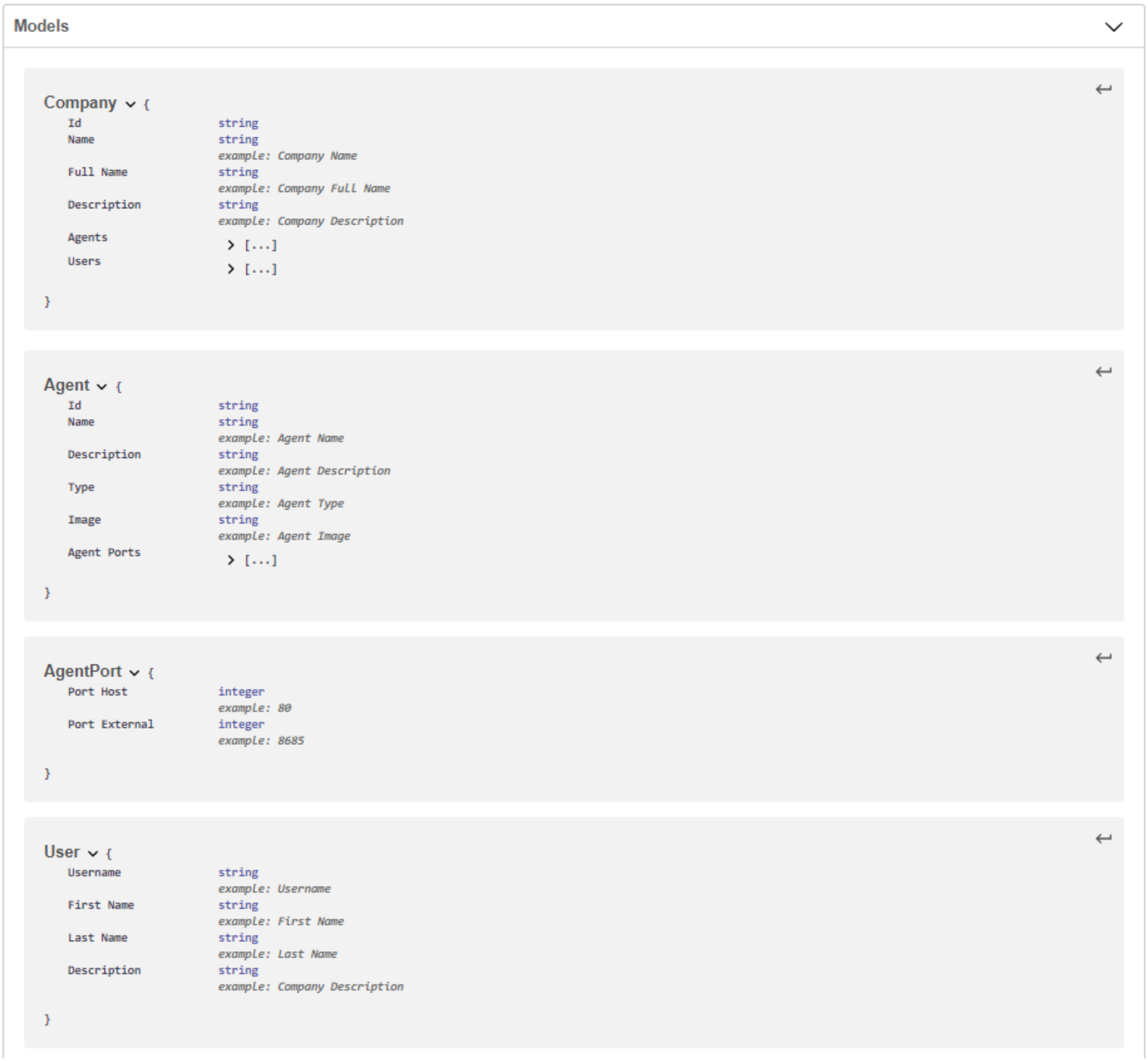
POST /company/{company_id}/user/ Create a new user

GET /company/{company_id}/user/{username}/ Get User description

PUT /company/{company_id}/user/{username}/ Update user

DELETE /company/{company_id}/user/{username}/ Delete User

Figure 15: API of Marketplace Management Service



```
Models

Company ⌄ {
    Id                  string
    Name                string
                        example: Company Name
    Full Name           string
                        example: Company Full Name
    Description         string
                        example: Company Description
    Agents               > [...]
    Users                > [...]
}

Agent ⌄ {
    Id                  string
    Name                string
                        example: Agent Name
    Description         string
                        example: Agent Description
    Type                string
                        example: Agent Type
    Image               string
                        example: Agent Image
    Agent Ports          > [...]
}

AgentPort ⌄ {
    Port Host           integer
                        example: 80
    Port External       integer
                        example: 8685
}

User ⌄ {
    Username            string
                        example: Username
    First Name          string
                        example: First Name
    Last Name           string
                        example: Last Name
    Description         string
                        example: Company Description
}
```

Figure 16: Model of the API of the Marketplace Management Service

7 Conclusions

The work that has been carried out in Task 6.2: “Cloud Infrastructures for Inter-factory Data Exchange” has been extensively described in this document. After an initial state of the art survey, the work has focused on the definition of the agent container technology and all the components required for the marketplace to be operative.

All the activities have been carried out by always keeping in mind scalability and reliability issues, for the marketplace to be as distributed and decentralized as possible.

During the months that have been passed from the first iteration of this deliverable the Marketplace has been continuously improved and tested by adding feature and enhancing software robustness. The development process has not been changed with respect to the initial guidelines defined and described in the first iteration of this deliverable. All the decisions made in the first part of the project have proved to be successful.

A final working version of the marketplace has been developed with following main components:

- Stakeholder agents (Requester agent, Supplier Agent) implementations
- Agent Management Service
- Matchmaker Agent
- Message broker (RabbitMQ)
- Blockchain layer

The integration between all the components has been carried out in conjunction with the work performed in Task 6.3, by defining and refining the COMPOSITION eXchange Language.

Also, the security issues have been addressed by tightly working with the different tasks of WP4.

The Marketplace has been successfully tested in the aforementioned use cases, achieving the objectives set and meeting all the expectations. With respect to the usual process of negotiation, the COMPOSITION Marketplace has shown an easy and successful way in doing this task in a semi-automatic manner outperforming the old mechanism that was much more time consuming.

8 List of Figures and Tables

8.1 Figures

8.2 Tables

9 References

(Shoham Y., 1993)	Shoham Y., 1993. Agent-oriented programming, Artificial intelligence.
(Lützenberger, et al., 2013)	Marco Lützenberger, Tobias Küster, Thomas Konnerth, Alexander Thiele, Nils Masuch, Axel Heßler, Michael Burkhardt, Jakob Tonn, Silvan Kaiser, Jan Keiser, Sahin Albayrak, 2013. JIAC V – A MAS Framework for Industrial Applications, AAMAS 2013.
(Rodriguez, et al., 2014)	Rodriguez, Sebastian & Gaud, Nicolas & Galland, Stéphane. (2014). SARL: A General-Purpose Agent-Oriented Programming Language. 10.1109/WI-IAT.2014.156.
(Bellifemine, et al., 2000)	F. Bellifemine, A. Poggi and G. Rimassa, 2000. Developing multi-agent systems with JADE. Accepted by Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000), Boston, MA, 2000.