

H2020 – ICT-13-2018-2019



# MUSKETEER



**Machine Learning to Augment Shared Knowledge in  
Federated Privacy-Preserving Scenarios (MUSKETEER)**

**Grant No 824988**

**D7.2 Client connectors' architecture design –  
Final version**

**November 20**

## Imprint

<b>Contractual Date of Delivery to the EC:</b>	<b>30 November 2020</b>
<b>Author(s):</b>	<b>Susanna Bonura, Davie Profeta and Domenico Messina (ENG)</b>
<b>Participant(s):</b>	<b>ENG, IBM, IDSA</b>
<b>Reviewer(s):</b>	<b>Lucrezia Morabito (COMAU), Mark Purcell (IBM)</b>
<b>Project:</b>	<b>Machine learning to augment shared knowledge in federated privacy-preserving scenarios (MUSKETEER)</b>
<b>Work package:</b>	<b>WP7</b>
<b>Dissemination level:</b>	<b>Public</b>
<b>Version:</b>	<b>1.0</b>
<b>Contact:</b>	<b>Susanna Bonura – susanna.bonura@eng.it</b>
<b>Website:</b>	<b>www.MUSKETEER.eu</b>

## Legal disclaimer

The project Machine Learning to Augment Shared Knowledge in Federated Privacy-Preserving Scenarios (MUSKETEER) has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 824988. The sole responsibility for the content of this publication lies with the authors.

## Copyright

© MUSKETEER Consortium. Copies of this publication – also of extracts thereof – may only be made with reference to the publisher.

## Executive Summary

The client connector is the component required for a participant to join the MUSKETEER Platform. It is the software application supporting MUSKETEER platform participant in his data exchange, share and process, so to guarantee that he is sovereign of his data.

The client-side connectors have to support the set of privacy operation modes (POMs) made available throughout the project according to the architecture defined in T3.1 and meet the requirements of the federated and privacy-preserving machine learning services designed in WP4. Moreover, the client component provides services for locally combining model updates into one consistent, up-to-date model instance. The client component serves as adaptor for the integration and industrial validation of the MUSKETEER platform in WP7.

The client connector, running on a secure and private space, provides the interface (Client Connector External APIs) for receiving a set of instructions from a master controller of the MUSKETEER server, related to the transferring of the required datasets and/or models (according to the POM chosen) from/to the MUSKETEER core to the secure and private space for the training of an MUSKETEER ML model.

Two different architectures of the client connector are presented. The Desktop Client Connector can be used when data is collected in a non-centralized way and there is no need to use a cluster to distribute the workload, both in terms of computing and big data storage. Anyway, the Desktop version could also leverage GPUs for the training process, enabling the processing of a large amount of data in terms of volume. Finally, the Desktop Client Connector can be easily deployed in any environment thanks to the use of Docker in order to containerize the Client Connector application. The Cluster Client Connector is devised to meet big data processing and federated machine learning needs. From the user perspective, there are not many changes related to the user interface and the user experience because the frontend consistency is kept in both versions. Contrarywise, there are deep differences in the backend side of the architecture due to the distributed nature of the system.

The client connector component is comprised of two local connectors. One is external, to allow users to share their (encrypted) data and/or to receive (encrypted) model updates generated on the server side, by exposing an endpoint to upload/download information and/or to retrieve trained models or (encrypted) model updates, depending on the POM. The second one is local and implements a set of interfaces to access and, if needed, pre-process data stored in local databases or file systems.

## Document History

Version	Date	Status	Author	Comment
0.1	1/10/2020	Table of Content	ENG	First draft
0.2	5/10/2020	Architecture images updated	ENG	Update
0.3	9/10/2020	Inputs for desktop client connector general description	ENG	Update
0.4	16/10/2020	Inputs for cluster client connector general description	ENG	Update
0.5	23/10/2020	Reference to the MUSKETEER Platform architecture added	ENG	Update
0.6	2/11/2020	Completed. Ready for internal review	ENG	Second draft
0.7	5/11/2020	Review inputs from IBM	ENG	Update
0.8	6/11/2020	Review inputs from FCA	ENG	Update
0.9	9/11/2020	Finalization		Cleaning to be ready for submission
1.0	9/11/2020	Final Version	IBM	Final

## Table of Contents

<b>LIST OF FIGURES.....</b>	<b>5</b>
<b>LIST OF ACRONYMS AND ABBREVIATIONS.....</b>	<b>6</b>
<b>1 INTRODUCTION.....</b>	<b>7</b>
1.1 Purpose .....	7
1.2 Related Documents.....	7
1.3 Document Structure.....	8
<b>2 CLIENT CONNECTOR: A GENERAL DESCRIPTION.....</b>	<b>9</b>
<b>3 MUSKETEER TECHNICAL REQUIREMENTS.....</b>	<b>11</b>
<b>4 MUSKETEER PLATFORM ARCHITECTURE .....</b>	<b>13</b>
4.1 MUSKETEER ML Task entity .....	16
4.2 Client Connector entity.....	18
<b>5 MUSKETEER CLUSTER CLIENT CONNECTOR ARCHITECTURE .....</b>	<b>21</b>
<b>5.1 MUSKETEER Cluster Client Connector .....</b>	<b>22</b>
5.1.1 Cluster Client Connector: Proposed APIs.....	28
5.1.2 Cluster Client Connector: Workflows.....	33
<b>5.2 MUSKETEER Desktop Client Connector .....</b>	<b>37</b>
5.2.1 Desktop Client Connector: Proposed APIs.....	39
5.2.2 Desktop Client Connector: Workflows.....	48
<b>6 CONCLUSION.....</b>	<b>51</b>
<b>7 REFERENCES.....</b>	<b>52</b>

## List of Figures

Figure 1 - MUSKETEER's PERT diagram .....	8
Figure 2 – Connector concept [1] .....	9
Figure 3 – Interaction among connectors according to IDSA RAM [1] .....	10
Figure 4 - MUSKETEER centralized server platform architecture .....	15
Figure 5 – Connector system layer architecture according to IDSA RAM [1] .....	19
Figure 6 - Client Connector Modes .....	22
Figure 7 – MUSKETEER Client Connector Architecture .....	23
Figure 8 – ‘List tasks’ sequence diagram .....	34
Figure 9 - “create a task” sequence diagram .....	35
Figure 10 – “Task participation” sequence diagram .....	36
Figure 11 - Desktop Client Connector Architecture .....	37
Figure 12 - Desktop Client Connector Sequence Diagram – Listing tasks.....	48
Figure 13 - Desktop Client Connector Sequence Diagram – Task participation .....	49
Figure 14 - Desktop Client Connector Sequence Diagram – Task aggregation.....	50

## List of Acronyms and Abbreviations

Abbreviation	Definition
API	Application Programming Interface
CA	Consortium Agreement
DC	Data Connector
DP	Differential Privacy
DV	Data Value
FS	Feature Selection
FSM	Finite State Machine
GA	Grant Agreement
IDR	Intermediate Data Representation
IDS	Industrial Data Space
LC	Logistic Classifier
LGFS	Linear Greedy Feature Selection
MK	Master Key
ML	Machine Learning
MLP	Multi-Layer Perceptron
MN	Master Node
OS	Operating System
PERT	Program evaluation and review technique
PK	Public Key
POM	Privacy Operation Mode
PP	Privacy Preserving
PPML	Privacy Preserving Machine Learning
RAM	Reference Architecture Model
ROC	Receiver Operating Characteristics
SQL	Structured Query Language
TA	Task Alignment
UI	User Interface
WN	Worker Node

## 1 Introduction

### 1.1 Purpose

This document presents the final version of the MUSKETEER client connector architecture. It derives from technical requirements for MUSKETEER platform and from user needs of the two industrial scenarios considered within the project. The MUSKETEER client connector architecture is compliant with the general MUSKETEER platform architecture, presented in the deliverable D3.2 - Architecture design – final version.

### 1.2 Related Documents

This deliverable is the document describing the final version of main functionalities of the client connector. It contains the design of two types of Client Connector architectures (cluster-based and desktop-based) to meet two different sets of user needs and requirements.

This deliverable is related to the following documents (also see Figure 1):

- **D3.1 Architecture Design – Initial Version** – detailing the first version of the MUSKETEER architecture.
- **D3.2 Architecture Design – Final Version** – detailing the final version of the MUSKETEER architecture.
- **D2.1 Industrial and technical requirements** – in so far as the platform architecture has to address functional and non-functional technical requirements described in that document.
- **D2.2 Legal requirements and implementation guidelines** – in so far as the design of the platform architecture should follow the implementation guidelines arising in the context of the applicable legal and ethical framework.
- **D2.3 Key performance indicators selection and definition** – in so far as the platform has to either provide the core capabilities that other functional components (e.g. the algorithmic library or the client connectors) require to meet their goals, or to meet specific goals itself.
- **D4.1 Investigative overview of targeted architecture and algorithms** – in so far as the platform has to provide the core capabilities to support and enable the targeted architecture and algorithms.
- **D4.2 Pre-processing, normalization, data alignment and data value estimation algorithms (initial version)** – in so far as the platform has to provide the core capabilities to support the deployment of the proposed algorithms.

- **D5.1 Threat analysis for federated machine learning algorithms** – in so far as the platform has to provide the core capabilities to support the deployment of the proposed algorithms.
- **D6.1 Assessment framework design and specification** – in so far as the platform has to provide the core capabilities to support the application of the proposed framework and meet relevant key performance indicators (KPIs).
- **D7.1. - Client connectors’ architecture design (initial version)** – the precursor to this document.

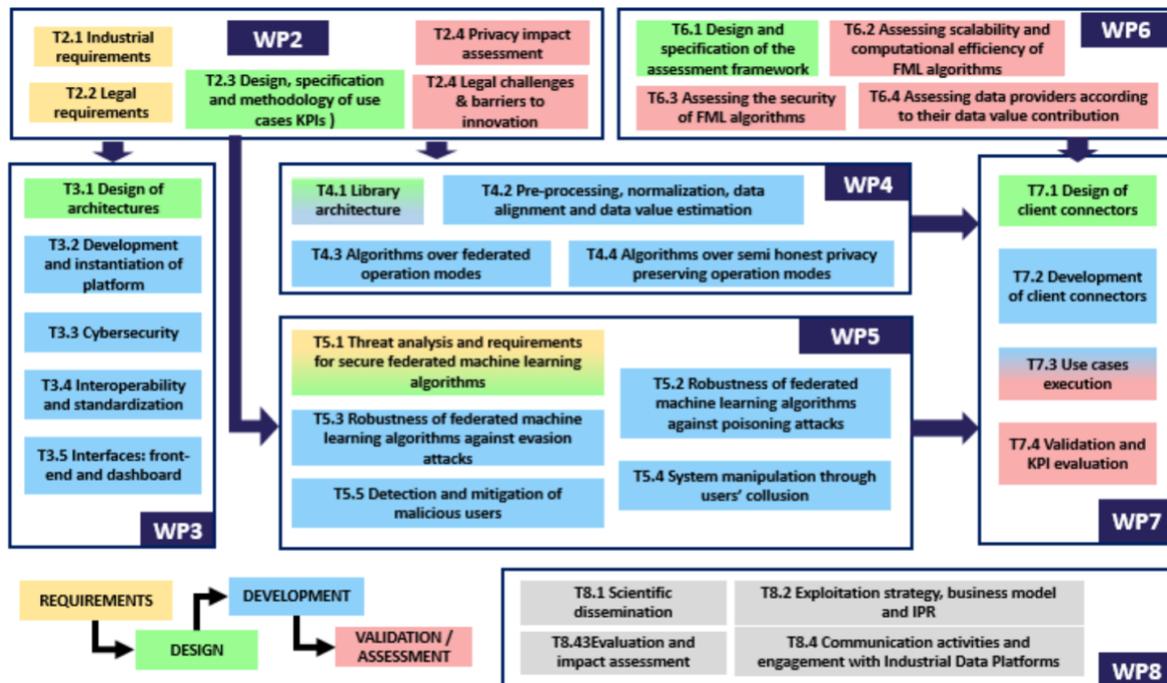


Figure 1 - MUSKETEER’s PERT diagram

### 1.3 Document Structure

In the next section (Section 2), the general description of client connectors is presented according to the IDSA connector specifications.

In Section 3, MUSKETEER platform technical requirements are recapped in order to cross check that the requirements involving end user software side, are met in the client connector architecture design.

In Section 4, a summary of the MUSKETEER platform architecture is presented, so to have a comprehensive picture before detailing client connectors.

Section 5 presents the final version of the MUSKETEER client connector architecture, according to two flavours: desktop based one and cluster based one, showing for each of them proposed APIs and workflows among components.

Finally, Section 6 concludes the deliverable. It outlines the main findings of the deliverable and takes into account the chance for further analysis in conjunction with other work packages.

## 2 Client Connector: a general description

The client connector is the component required for a user who wants to join the MUSKETEER Platform. In general, connectors are the central technological building block of the International Data Spaces [1]. They are software components supporting participants in their data exchange, share and training of machine learning models. At the same time, connectors guarantee that the Data Owner is sovereign of his data.

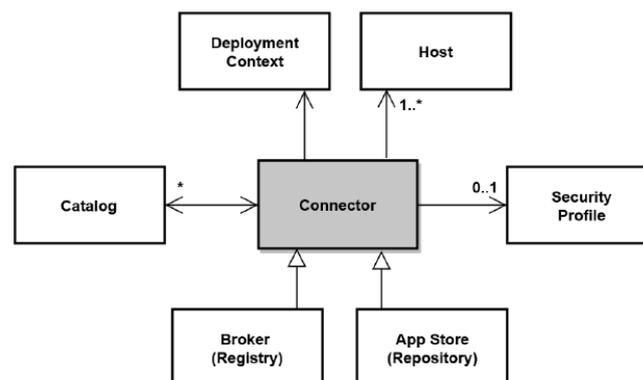


Figure 2 – Connector concept [1]

Figure 2 shows the main elements composing a connector.

- The deployment context of a connector records the connector's location, as for example the data centre and coordinates, the type of its deployment (on-premises or cloud-based), and the name of the Participant.
- The security profile indicates the capabilities of a connector to maintain a controlled, secure and trusted environment for exchanging, sharing and processing data, through remote integrity verification, application isolation, usage control support.
- The catalogue represents the repository of the metadata of resources, constructed in accordance with the IDS Ontology, through which connectors provide or consume data.
- Optionally, the catalogue, or individual sets of resource metadata (about functions and interfaces, pricing models, licenses, etc.), may be advertised via intermediary nodes,

such as the broker service provider, which is an intermediary that stores and manages information about the data sources available in the International Data Spaces, or in the app store.

- Each host represents an individual communication capability of the connector, a server that exposes resources via endpoints (HTTPS URLs, MQTT topics, etc.) according to the communication protocol supported [1].

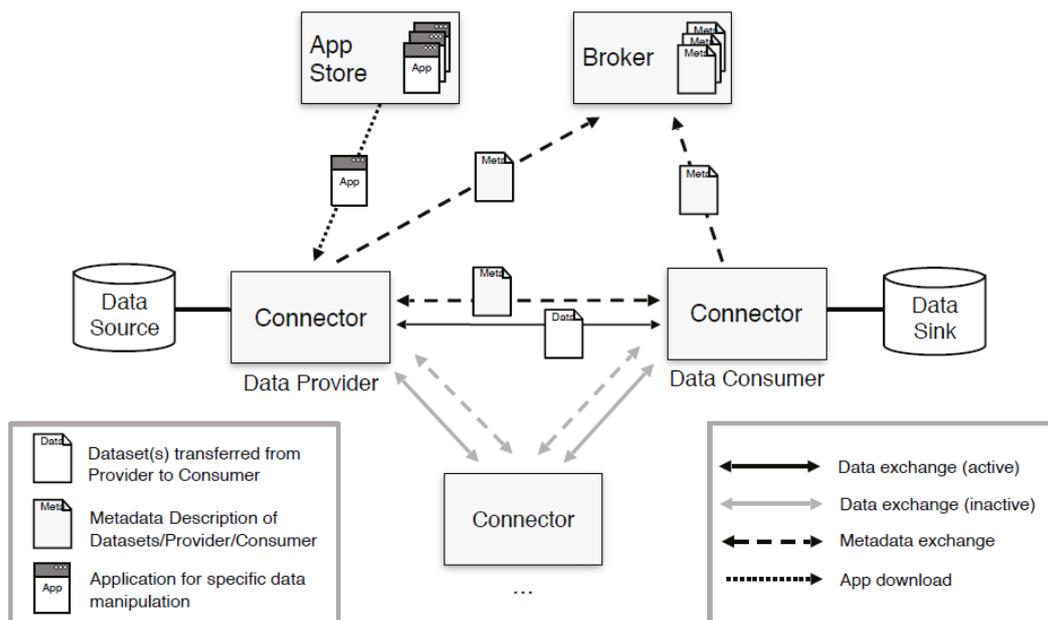


Figure 3 – Interaction among connectors according to IDSA RAM [1]

According the IDSA Reference Architecture Model (RAM), in general, a set of applications (Apps) are deployed inside the connector, to facilitate data processing workflows.

The Figure 3 shows interaction among connectors, one for each user participating to the federation, according to IDSA RAM, thus each participant should be able to run the Connector software in their own IT environment.

Every connector participating in the IDS Platform must have a unique identifier and a valid certificate. In addition, it must check and verify the identity of other connectors and provide a valid certificate, so that each participant in the platform is able to verify the identity of any other participant. The connector serving as the data source must be able to verify the receiving connector’s capabilities and security features as well as its identity.

Communications among connectors can be encrypted and integrity protected. In addition, connectors must be able to ensure that IDS participants’ data is handled according to the usage policies specified: otherwise the data will not be exchanged.

Data providers and data consumers must be able to decide which level of security they intend to apply for their respective connectors by deploying connectors supporting the selected security profile.

Participants must have the opportunity to describe, publish, maintain and manage different versions of metadata, which describe the syntax, serialization and semantics of data sources.

Participant must be able to provide an interface for data and metadata access, to transmit metadata of its data sources to one or more brokers, and to browse and search metadata in the metadata repository, provided the participant has the right to access the metadata.

To create and structure metadata, participant may use vocabularies, that can be already existing or created ad hoc by the operator.

Vocabulary hubs are central servers that store vocabularies and enable collaboration, searching, selection, matching, updating, requests for changes, version management, deletion, duplicate identification, and unused vocabularies.

Alternatively, they can run a Connector on mobile or embedded devices. The operator of the Connector must be able to define the data workflow inside the Connector. Participants must be identifiable and manageable. Passwords and key storage must be protected. Every action, data access, data transmission, incident, etc. should be logged. Using this logging data, it should be possible to draw up statistical evaluations on data usage etc. Notifications about incidents should be sent automatically.

The Connector receives data from an enterprise backend system, either through a push-mechanism or a pull-mechanism. The data can be provided via an interface or pushed directly to other participants.

Finally, according the IDSA RAM, other Connectors can subscribe to data sources or pull data from these sources. Data can be written into the backend system of other participants.

### 3 MUSKETEER Technical Requirements

This section shows the list of the technical requirements that were elicited starting from the business requirements coming from the description of the user stories in Smart Manufacturing and Health scenarios (for more detail please see the deliverable D2.1 - Industrial and Technical Requirements).

The list of such technical requirements is shown in the following table.

For each requirement, the ID is highlighted in **green** text if the current prototype described in D7.3 satisfies the requirement. A requirement may also be highlighted in **orange** text if the

current prototype partially satisfies the requirement. If a requirement is not currently satisfied by the D7.3 prototype, it is not highlighted. These requirements are still subject to ongoing development.

ID	Description
TR001	The MUSKETEER platform shall ensure that access control over datasets is applied according to the data policies and the terms of relevant active valid data sharing contracts.
TR002	The MUSKETEER platform shall forbid unauthorised user access to the platform and the datasets.
TR003	The MUSKETEER platform ensures different authorisation levels for accessing datasets.
TR005	MUSKETEER end user must have a unique identification that will be used in all the data exchange/communications.
TR006	Registration into the MUSKETEER Platform with username a password.
TR007	MUSKETEER end user could create one or more new tasks.
TR008	In MUSKETEER a task must be defined as a problem statement that feeds from data and produces a trained machine learning model.
TR009	New tasks should obtain unique task identifier.
TR010	In MUSKETEER a task should have a general description.
TR011	The MUSKETEER Platform must, for each task, uniquely identify every input data from every end user.
TR012	Description of the input features. The meaning of every field must be explicitly described.
TR013	The MUSKETEER Platform must share a set of pre-processing algorithms such that every end user pre-processes its own raw data to obtain a common representation (e.g. high pass filtering, edge detection, bag of words with TFIDF weighting ...).
TR014	MUSKETEER pre-processing modules should always produce an output vector with the expected content and format.
TR015	MUSKETEER must make any ad hoc pre-processing algorithms (defined and implemented by the end users) shared with other users contributing to the task.

TR016	In MUSKETEER, for each task, definition and nature of the problem to be solved, must be shared among all the participants in a task such that they can contribute with new data to the training process.
TR017	MUSKETEER Privacy Operation Modes (POMs) must cover all kinds of privacy restrictions that end users would apply to his/her data.
TR018	Privacy restriction should be described in natural language to facilitate the specification of the task to the end user.
TR019	MUSKETEER Platform should envisage monetary rewards as well as collaborative results.
TR020	Browsing for published active tasks by MUSKETEER end users.
TR021	Creation and/or access to published active tasks by MUSKETEER end users.
TR022	Running of the training procedure associated to a given MUSKETEER ML task.
TR023	Monitoring of the progress of MUSKETEER ML tasks until completion.
TR024	MUSKETEER must provide the outcome of a task (reward, trained model, etc.).
TR025	MUSKETEER must allow data to be transferred and joined either in the server or in a given user.
TR026	MUSKETEER must support the case where no raw data is transferred outside the client facilities (the ML model training must take place in the server by using the aggregated information from the clients).

It is worth to noticing that the fulfilment of requirements that are not highlighted, highly depends on the design and implementation of the server-side and MMLL library. They can only really be evaluated over time. As such, at this point in time, they cannot be considered complete. The requirements highlighted in orange, are deemed partially complete already, but a more thorough review over a longer period of time is also preferable.

## 4 MUSKETEER Platform Architecture

The MUSKETEER platform must provide the infrastructure and implement the services that are required (gathered in WP3) to enable the distributed machine learning capabilities developed in WP4 and WP5, along with interfaces supporting the use case integration in WP7.

The final version of the MUSKETEER platform architecture is described in the deliverable D3.2.

This architecture is based on micro-services and places a significant emphasis on open standards. Many of the underlying components used are open source. The use of open

standards and services avoids vendor lock-in to a significant extent, thereby enhancing the prospect of utilising alternative cloud providers or on-premise deployments in the future, if that is so desired.

The MUSKETEER platform architecture is client-server (see Figure 4): the server component coordinates the secure transportation of information and models across the participants, the client components allow use case participants to share information (the kind of data to be exchanged will depend on the Privacy Operation Mode (POM)), retrieve model updates, incorporate those locally, and ultimately retrieve the trained machine learning models for the deployment in their local business processes.

The server infrastructure is provided by IBM, using the IBM® Cloud™ platform. It will host micro-services for data management, machine learning and internal/external data exchange (i.e. IBM Cloud™ Messages for RabbitMQ, IBM® Db2® on Cloud, IBM® Cloud Object Storage, IBM Cloud™ Functions, IBM Cloud™ Kubernetes Service).

For data management, different database persistence layers are envisioned to support different types of data, like Object Storage for large unstructured data (such as images), or Relational Database Management Systems for relational data (such as numerical sensor measurements and associated metadata).

With regard to the machine learning capabilities, algorithms provided will be based on Python with deep learning frameworks such as TensorFlow [5], Caffe [6], PyTorch [7], Keras [19], to support efficient execution of model training on CPU and GPU processing.

An external connector implements the features that allow users to upload/download information and/or to retrieve trained models or (encrypted) model updates, depending on the POM.

Interoperability between components (cloud-based and remote) is through a messaging system, based on the Publish / Subscribe Design Pattern [3]. This is backed by RabbitMQ [4]. Messages are published to RabbitMQ and routed to subscribed parties. RabbitMQ is instantiated in the public cloud and is an internet addressable service, allowing remote clients to connect. Remote clients require appropriate credentials which are obtained through the registration process.

Using this messaging system, the initiation of all network connections is outbound only. This means that no remote component (aggregator or participant system) accepts an incoming connection with no network ports openly addressable to the internet.

**Federated Machine Learning Framework - Architecture**

Loosely coupled micro-services, based on Publish / Subscribe Design Pattern  
All publish queue access is write-only, all subscribe queue access is read-only  
Optimised for high levels of privacy enforced by RabbitMQ policies

2020 - Mark Purcell

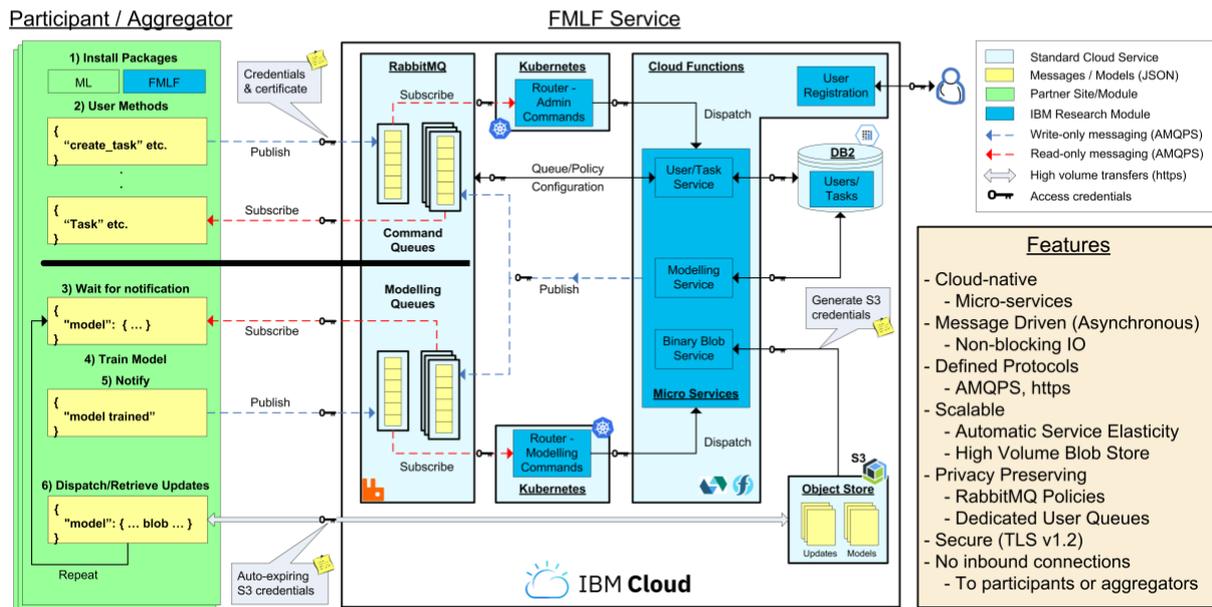


Figure 4 - MUSKETEER centralized server platform architecture

All access to platform services from remote components (provided by other work packages) is through the *Federated Machine Learning Framework* (FMLF) package. This contains APIs to simplify access to the platform and is installed at remote sites.

Users of these APIs must be authenticated, but first, the *User Registration* service allows users to register with the platform. This service creates user accounts on the RabbitMQ instance. These registration details allow users to subsequently authenticate with the platform, providing access to the APIs and platform services. Access to individual APIs is also controlled through an authorisation layer. Further details on the MUSKETEER cloud server architecture are described in D3.2.

As already mentioned, the MUSKETEER platform is a client-server architecture, where the client is intended as a software application that in general is installed on-premise and run at every end user side. Such a software application is named client connector in the MUSKETEER taxonomy.

On the other hand, the MUSKETEER server is the central part of the platform that communicates with all the client connectors and acts as a coordinator for all the operations. Users federated to MUSKETEER, interact with the client connector installed on their side and that client will communicate with the server to perform several actions on the platform.

For the aims of the present document, only the technical requirements involved in the client side will be reported. They will be mentioned in round brackets in section 5.1.

The interactions between the MUSKETEER server and the client connector aim at supporting two types of activities: the first one is operative and regards interactions to exchange messages and pieces of information for the general operation of the platform (create a new user, 'log-in' as a user, define a task, declare privacy preferences, describe data attributes, feedback reporting, etc.); the second one regards the actual model training, that is the interaction to exchange the messages and pieces of information during the training phase of the ML algorithm. For both activities, the knowledge of the 'task' entity is crucial. It is detailed in the next section.

#### 4.1 MUSKETEER ML Task entity

A ML task may be as a problem statement that feeds from data and produces a trained machine learning model as an outcome. Any MUSKETEER end user can create one or more new tasks thereby obtaining a unique task identifier (task\_id) from the platform and run them even in parallel.

The definition step requires the specification of the task characteristics:

- General description: a high-level description of the task (the problem to be solved) is necessary for a rapid identification of existing tasks by other users. In a first version of MUSKETEER, this part can be reduced to a minimum, since the end users and their tasks are already defined.
- Data: it is recommended to facilitate an initial dataset, i.e., some data illustrating the task to be solved. Data comprises input feature vectors ( $\underline{x}$ ) (for non-supervised tasks) and pairs of input feature vectors and target values ( $\underline{x}, t$ ) (for supervised tasks).
- Features description: a general description of the input features (like defining the fields in a Table) is necessary to unify the data representation among users and finally being able to combine all the contributed data during the learning stage. In the above-mentioned general case where input data is represented as a vector, the meaning of every field in such a vector must be explicitly described, for compatibility purposes.
- "Ad hoc" preprocessing algorithm (optional): in some cases, input data is not so easily represented in terms of individual meaningful features and the feature vector may be the result of applying some (possibly complex) preprocessing to a raw piece of information (for instance an image, a text, a voice recording, etc.). In those cases, when the raw data cannot be transmitted to the MUSKETEER server and processed in the same place, it is necessary to share the preprocessing algorithm such that every end user preprocess its own raw data to obtain a common representation into the feature vector  $x$ . For instance, in the image case, some transformations may need to be applied to an image before feeding it into a machine learning model, such as high pass filtering followed by an edge detection, a specific feature extraction, etc.; in the case of texts, the preprocessing could be a bag of words with TFIDF weighting, etc. The casuistic can be extremely large and problem dependent, so it is important to guarantee that the preprocessing module always

produces an output vector with the expected content and format and it is recommended that the “ad hoc” (non-standard) preprocessing algorithms be defined and implemented by the end users defining a specific task, such that it can be shared with other users contributing to the task as a “preprocessing object”.

- Target values: the problem to be solved is defined by the target values (in supervised tasks). For instance, given the above described features, the target could be to estimate the annual income in euros, or to estimate if that person is unemployed or not. The definition and nature of the target must also be shared among all the participants in a task such that they can contribute with new pairs  $(x, t)$  to the training process.
- Privacy requirements: it is important that the end user determines which are the privacy restrictions that apply to his/her data, because those restrictions will determine the POMs that can be used. It is more operative that the user declares the privacy restrictions that apply to the data and then the platform offers the available POMs to solve the ML task. The ‘data\_privacy’ parameter can be chosen among several options, described in natural language to facilitate the specification of the task to the end user, for example, one end user may adhere to some of the following statements:
  - my data is open and can be freely distributed;
  - my data can be shared after anonymization;
  - my data can be shared only with the MUSKETEER platform under some confidentiality agreement with the platform;
  - my data can be shared with other end users under some confidentiality agreement with the end users;
  - my raw data cannot leave my facilities (only the MUSKETEER client can see it and obtain some operations on it: gradients, dot products, etc., but never reveal individual data points.)
  - my data can be used for a given task, but not for other tasks.
- Reward: this is the motivation for other users to join the task and offer their own data. After completion of the task, the reward is shared among the participants. In the context of this project the motivation by the end users is taken for granted, but it is important not to forget its existence, specially to motivate the data value estimation procedures to be developed in WP6.

The expected reward could be, for example, among other possible options are:

- monetary: a user wants to improve a ML model, he/she deploys a MUSKETEER task and offers a monetary reward upon successful completion of the task. Other users owning data of potential interest for the task can join the initiative. After training is completed, the contribution of every participant will be (hopefully) determined by the data value estimation modules, and the reward will be distributed according to the value of every contribution. The initial user retains the trained model for its own use.

- collaborative results: all participants contribute with a portion of data to the task and the resulting learned model is shared among all participants in the task (well, possibly only among those identified as positively contributing to the task solution).

Hence, once in the platform, any end user will have mechanisms to:

- browse for published active tasks, and join one or more of them
- create his/her own task as explained above
- run the training procedure associated to a given ML task and follow the progress until completion
- receive the outcome of a task (reward, trained model, etc.).

## 4.2 Client Connector entity

Before presenting the MUSKETEER Client Connector design, it is considered useful to recall the conceptual architecture of the client connector entity, as presented by IDSA.

From a technical point of view, the client connector provides metadata as specified in the connector self-description, e.g. technical interface description, authentication mechanism, exposed data sources, and associated data usage policies. Thus, metadata is a fundamental building block for the deployment and composition of several ML model definitions inside a client connector: all operations are defined in terms of input and output parameters, bound protocols, and endpoints. Preconditions and postconditions need to be made explicit, and effects on the environment must be outlined.

Indeed, according the IDSA Reference Architecture Model (RAM), for each client connector, it is possible to specify pre- or post-conditions that have to hold before (as the integrity check of the environment) and after (as the data item is deleted after usage) decision-making. In addition, it is possible to define on-conditions that have to hold during usage (e.g., only during business hours). These conditions usually specify constraints and permissions that have to be fulfilled before, during, and after using data [1].

The client connector must allow participant to share information (the kind of data to be exchanged will depend on the POM), retrieve model updates, incorporate those locally, and ultimately retrieve the trained machine learning models for the deployment in the local business processes.

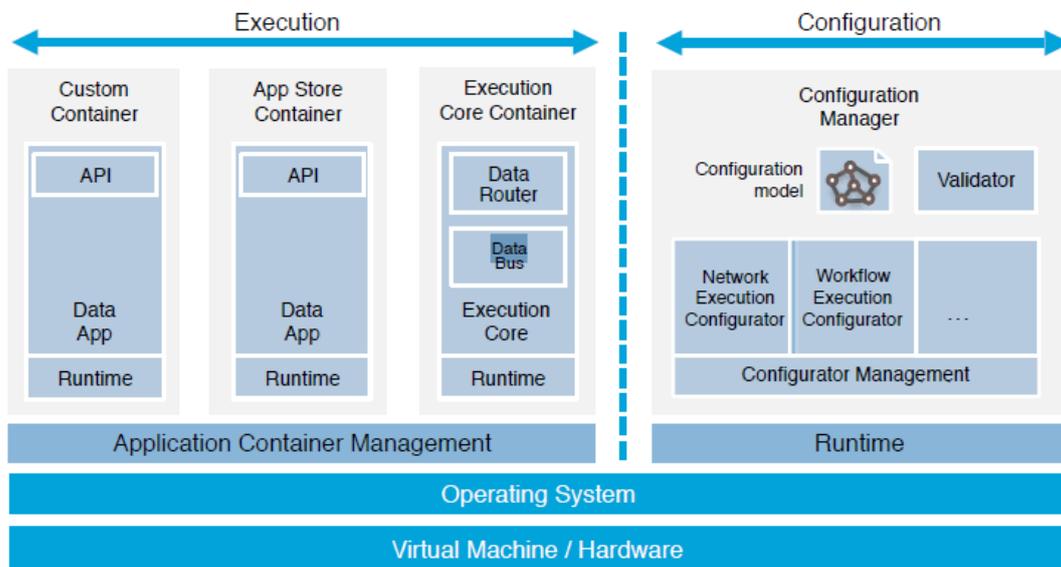


Figure 5 – Connector system layer architecture according to IDSA RAM [1]

The connector architecture, as described within the IDSA RAM, uses application container management technology to ensure an isolated and secure environment for individual data services.

The Figure 5 shows the connector system layer architecture presented in the IDSA RAM as splitted in two phases: execution and configuration.

- Within the execution phase, we have six entities involved. The first one is the Application Container Management component. It supports the deployment of an Execution Core Container and selected Data Services. Thereby, Data Services are isolated from each other by containers so to prevent unwanted interdependencies. Using Application Container Management, it is possible to apply extended control of Data Services and containers.

The second element involved in the execution phase is the Execution Core Container, which provides components for interfacing with Data Services and supporting communication. More in detail, within an Execution Core Container, a Data Router handles communication with Data Services to be invoked according to predefined configuration parameters. In this regard, it is responsible for the way the data is sent (and received) to (and from) the data bus by (and to) Data Services. Participants have the option to replace the Data Router component by alternative implementations of various vendors. If a connector in a limited or built-in platform consists of a single data service or a fixed connection configuration (eg. On a sensor device), the data router can be replaced by hard-coded software or the data service can be exposed directly. The Data Router invokes relevant components for the enforcement of Usage Policies, as configured in the connector or specified in the Usage Policy.

In addition, within an Execution Core Container, the Data Bus is present, to exchange data with Data Services and Data Bus components of other Connectors and also to store data within a Connector. In general, the Data Bus provides the method to exchange data among Connectors. Like the Data Router, the Data Bus can be replaced by alternative implementations in order to meet the requirements of the operator. Like the data router, the data bus can be replaced by alternative implementations in order to meet the operator's requirements. The third element involved in the execution phase is the App Store Container (one for each Data Service), which is a certified container downloaded from the App Store and provides a specific Data Service to the Connector.

The fourth element involved in the execution phase is the Custom Container, which provides a self-developed Data Service. Custom containers usually require no certification. The fifth element involved in the execution phase is the Data Service, which defines a public API, which is invoked from a Data Router. A meta-description specifies this API and is imported into the configuration model. Data Services can be implemented in any programming language and target different runtime environments. The tasks to be executed by Data Services may be different. Existing components can be reused to simplify migration from other integration platforms. Finally, the Runtime of a Data Service is found in the execution phase. It depends on the selected technology and programming language. The Runtime and the Data Service represent the main part of a container. Different containers may use different runtimes. What runtimes are available depends only on the base operating system of the host computer. From the runtimes available, a service architect may select the one deemed most suitable.

- With regard to the configuration phase, the connector architecture envisages five elements:
  1. the configuration manager, which represents the administrative part of a Connector and it is in charge of managing and validating the Configuration Model, followed by deployment of the Connector. Deployment is delegated to a collection of Execution Configurators by the Configurator Management.
  2. the configuration model, which is an extensible domain model to describe the configuration of a connector. It consists of configuration aspects that are interconnected and independent of technology.
  3. the configurator management. Its main task is to load and manage an exchangeable set of Execution Configurators. When a Connector is deployed, the Configurator Management component delegates each task to a special Execution Configurator.
  4. The execution configurators are interchangeable plug-ins that perform or translate individual aspects of the configuration model into a specific technology. The procedure for performing a configuration depends on the technology used. Common examples

could be the generation of configuration files or the use of a configuration API. By using different execution configurators, new or alternative technologies can be adopted and integrated into a connector. Therefore, every technology (operating system, application container management, etc.) gets its own execution configurator.

5. The Validator, which is in charge of checking if the Configuration Model complies with self-defined rules and with general rules specified by the International Data Spaces, respectively. The violation of the rules can be considered as a warning or an error. If such warnings or errors occur, the deployment may fail or be rejected.

Since the configuration phase and the execution phase are separated from each other, it is possible to develop and subsequently operate these components independently of each other.

According to the IDSA RAM, different implementations of the connector can use various types of communication and encryption technologies, depending on the requirements indicated.

As already mentioned, when defining the scope of the MUSKETEER platform, it is important to keep in mind the distinction between (i) the server-side platform, which enables the creation and execution of data sharing and federated machine learning among geographically distributed participants and (ii) the client connectors, in charge of starting and/or participating to ML training processes.

On the server component, detailed information are available in the deliverable D3.2 - Architecture design – Final version. In short, the server is the cloud platform which uses message queues for asynchronous exchange of information required for federated learning, such as the latest version of the central model computed by the aggregator, or model updates computed by the participants on their local data. The platform itself is agnostic to the semantics of this information (generally it will not even be aware whether or not the information is encrypted); it is parsed and interpreted in the context of the federated learning algorithm processes running on the aggregator and participants' sides, respectively.

Besides the exchange of information for the execution of the actual federated learning tasks, the server side also provides services to manage tasks throughout their lifecycle, such as: creating new tasks, browsing created tasks, aggregating tasks, joining tasks as a participant or deleting tasks. The meta-information that is required for task management is stored in a cloud database.

## 5 MUSKETEER Cluster Client Connector Architecture

Concerning the Client Connector, two types of architectures have been designed: a Cluster mode, the first version of which has been described in D7.1, and a Desktop mode, whose first

prototype is released along with this documentation as deliverable D7.3. The following figure shows the main differences between the desktop and cluster modes.

The Cluster Client Connector supports the storage and the processing of Big Data, through horizontal scalability and workload distribution on multiple nodes of the cluster (more details are provided in the next section).

The Desktop Client Connector can be used when data is collected in a non-centralized way and there is no need to use a cluster to distribute the workload, both in terms of computing and big data storage. Anyway, the Desktop version could also leverage GPUs for the training process, enabling the processing of a large amount of data in terms of volume. Finally, the Desktop Client Connector can be easily deployed in any environment thanks to the use of Docker [8] in order to containerize the Client Connector application. Docker containers ensure us a lightweight, standalone and executable package of the software that includes everything needed to run the Desktop Client Connector: operating system, code, runtime, system tools, libraries and settings. In this way the whole Desktop Client Connector application can be easily deployed in a sandbox to run on the host operating system of the user.

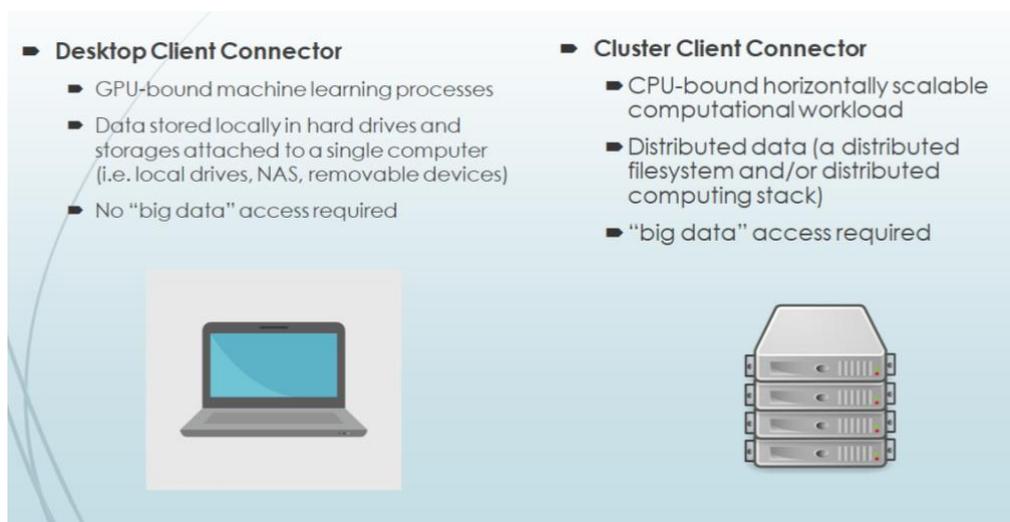


Figure 6 - Client Connector Modes

## 5.1 MUSKETEER Cluster Client Connector

As mentioned in the previous paragraph, the MUSKETEER Cluster Client Connector is devised to meet big data processing and federated machine learning needs. From the user perspective, there are not many changes related to the user interface and the user experience because the frontend consistency is kept by design. Contrarywise, there are deep differences in the backend side of the architecture due to the distributed nature of the system.

The functional architecture of the MUSKETEER client connector is described in Figure 7.

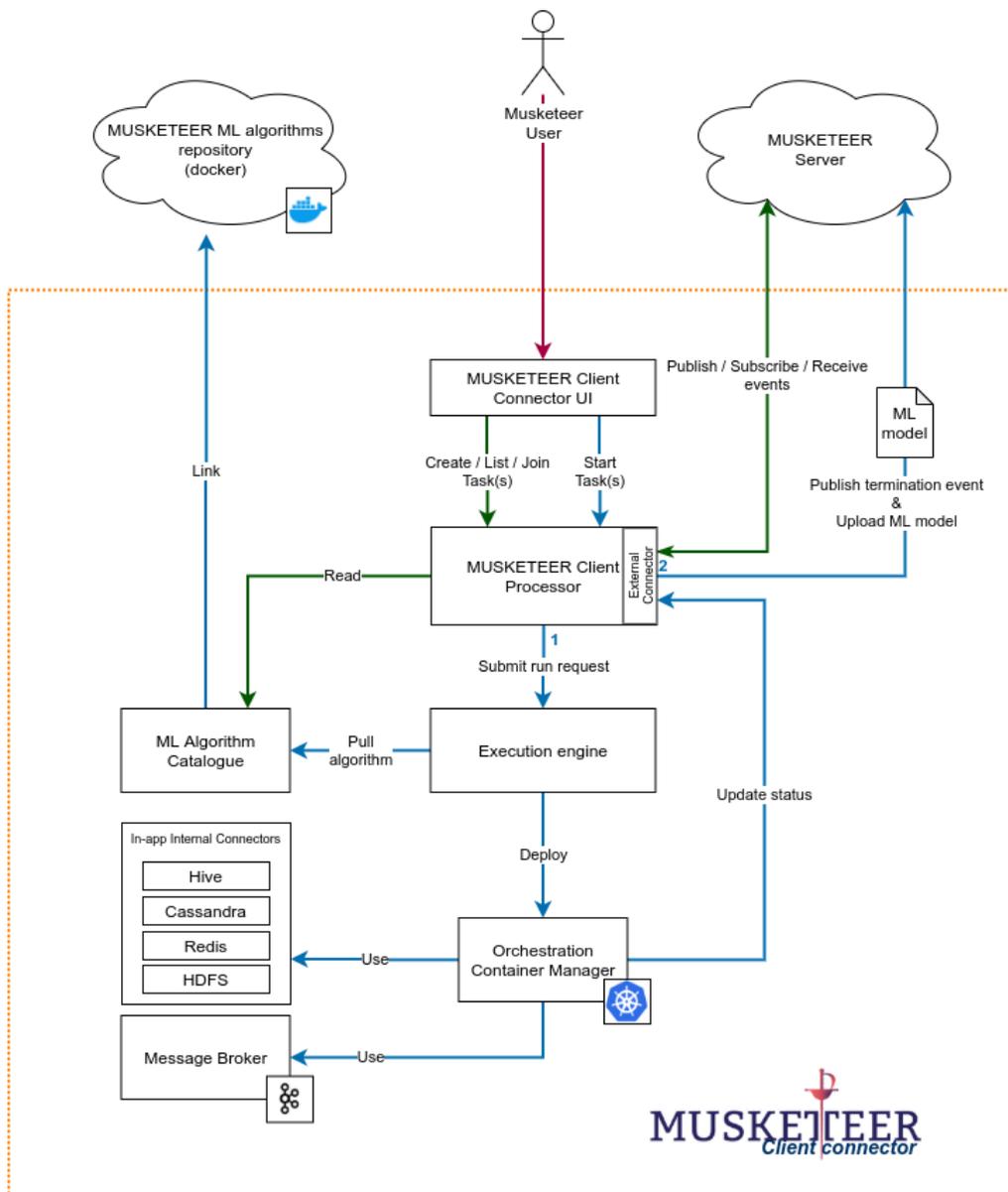


Figure 7 – MUSKETEER Client Connector Architecture

The boundary between the MUSKETEER user and the Cluster Client Connector is represented by the MUSKETEER Client Connector UI, which is a web application based on Angular [17] that communicates with a core microservice called MUSKETEER Client Processor.

The MUSKETEER Client Processor has two main coordination functionalities: it manages the in-cluster big data processing workloads (local jobs) that produce Machine Learning Models and dispatches federated machine learning events with the MUSKETEER Server the Cluster Client Connector is connected to. The MUSKETEER Client Processor exposes a RESTful API in order to receive the requests from the UI and includes an external connector sub-component that implements the MUSKETEER Server Communication Library. This component is implemented using the Spring Boot Framework [9], a java-based set of libraries used to implement lightweight and production-ready microservices in a fast and reliable manner.

The Execution Engine is the backbone of all the distributed processing that occurs in the Cluster Client Connector. In a nutshell, it handles the lifecycle of each local job, notifies their status to the Client Processor and all the related information, and communicates to the Orchestration Container Manager to request the allocation of the resources the job need to be executed as expected. The Execution Engine adopted in the Cluster Client Connector is Spring Cloud Dataflow [10], a microservice-based streaming and batch data processing component which belongs to the Spring ecosystem and implemented in Java. Spring Cloud Dataflow supports a lot of pre-built apps that allow matching a wide variety of scenarios and that can be used to compose MUSKETEER's local jobs without difficulties.

The Orchestration Container Manager allows by one side to instantiate every MUSKETEER Cluster Client Connector component, by the other side to deploy the local job on-demand at scale. The Container Manager is aware of the available resources, Memory, Volume, and CPUs in the cluster and distributes them according to the workload generated by the local jobs. Also, the Container Manager provides a sandboxed execution environment in the form of a container in order to run the processes with a high level of isolation, in this way it reaches the fault tolerance and reliability requirements a production system should have. Kubernetes has been chosen as a container orchestrator [11]. It belongs to the Anthos product family by Google and it is supported by a big community of developers worldwide.

To launch the local jobs, the container manager pulls the docker images registered in the ML Algorithm Catalogue, a Spring Application, which is accessed and referenced by the execution engine. The Catalogue contains all the metadata related to the model of the Algorithms and components used to manage the local jobs.

The Container Manager can manage also the hosting of some persistence infrastructures such as HDFS and Cassandra, which can be accessed via the internal connectors available, and bundled into some application registered in a docker registry that can take part in the composition of a local job. These applications should be in line with the Spring App specifications and implemented using the guidelines of Spring Cloud Dataflow.

For internal communication, a reliable and high performing message broker is adopted, in the specific case of the Cluster Client Connector, Apache Kafka is the solid pick. It is used by several internal components such as the Client Processor, the Execution Engine, and the applications that compose the local jobs in order to communicate by events.

The MUSKETEER Client Connector enables the user to interact with the MUSKETEER Server so to take part to the Federated Machine Learning processes, according to the specifications defined in the project. In order to be compliant with the IDSA RAM [1] shown so far, the client connector will be released as a multi-container application that will support both on-premise distributed environments as well as cloud providers.

Containers allow to run an application and all of its dependencies in isolated processes. The goal of containerization is to allow to easily package everything is needed to run software reliably when moved from one environment to another. There are a number of benefits that moving to containerization provides. Some of the main benefits companies can see include increased portability, simple and fast deployment, enhanced productivity, possible lower cost, improved scalability, improved security (**TR002**).

For the correct usage of the access control mechanism the design of an effective user management process is envisaged (**TR001**, **TR003**) so that each client connector will be univocally identified (**TR005**). In MUSKETEER platform, a potential approach will be to include two main subprocesses: a) the registration and subsequently user creation (**TR006**), and b) the user authentication (login) that enables the access to the platform as a whole. In MUSKETEER platform, the users appear under the concept of organisations. The registration process is handled by a component and includes the following steps:

- a) The organisation manager submits the organisation signup form.
- b) The MUSKETEER administrator receives the request, checks and approves it.
- c) The organisation manager creates the invitations for the organisation members. Each member receives the invitation link via email accompanied with an invitation token.
- d) Each organisation member fills-in the member signup form providing also the invitation token and, upon successful registration, access is granted to the MUSKETEER platform.

The client connector architecture implements an intuitive user interface through which the user will be able to perform the canonical operations of the MUSKETEER platform, such as browsing published active tasks (**TR020**), joining one or more of them, creating her own task (**TR007**, **TR021**), running the training procedure associated to a given ML task (**TR022**) and following the progress until completion (**TR023**), receiving the outcome of a task (reward, trained model, etc) (**TR019**, **TR024**).

The user interface will allow end users to straightforwardly define a task (**TR008**, **TR010**, **TR012**) that will be univocally determined (**TR009**, **TR011**)

The communication with the MUSKETEER server will occur through the MUSKETEER Client Processor module that implements the external connector exploiting the MUSKETEER-Client python APIs provided by IBM<sup>1</sup>.

---

<sup>1</sup> <https://github.com/IBM/Musketeer-Client>

Since the client processor is shipped as a docker image, it will run in a sandboxed execution context and will securely communicate through the external connector with the server, so that the compliancy with the IDSA RAM [1] will be preserved in all the parts of the Client Connector.

In order to run the algorithms in a safe and isolated environment, the execution engine provides all the capabilities to manage the lifecycle of the running jobs locally. Such an execution engine has to support the deployment, execution, monitoring and orchestration of algorithms as micro-service both in streaming and batch mode.

The chosen ML algorithm micro-services are retrieved from the ML algorithm catalogue and are instantiated according to the resources available on the machines in which the Client Connector runs.

The ML algorithm catalogue gathers all the machine learning models created in the project to cover a variety of privacy-preserving scenarios and ensure security and robustness against external and internal threats (**TR017, TR018**).

More in detail, the library will contain a complete set of algorithms for data pre-processing, normalization and alignment of horizontal and vertical distributed datasets (**TR013, TR014, TR015**); models for data value estimation; supervised learning algorithms to solve regression and classification tasks (Linear models like Logistic regression or ElasticNet, Kernel Methods such as semiparametric SVMs, Tree Based Algorithms such as Random Forest and Deep Neural Networks such as MLPs or CNNs); unsupervised learning to perform clustering or topic modelling (methods like K-means or LDA). Such pre-processing and training algorithms will run under different POMs (**TR016, TR025, TR026**) in which the platform can operate.

It is worth mentioning that in order to cover the largest possible number of industrial scenarios, MUSKETEER has to support several POMs. The main features to compare these POMs are the following ones:

- Privacy level: This is possibly the most obvious requirement in any IDP where data is to be shared.
- Computational local overload: Some problems require standard computational means, while in other cases, special computational resources might be needed: a Spark cluster or GPU units, for instance.
- Central Storage requirements: This requirement is mainly to be fulfilled by the central platform; it is needed if the users' data is collected and stored in a single place (a cloud service, for instance).
- Communication requirements: Depending on the volume of the datasets and the type of machine learning algorithm, large communication resources may be needed.

- **Data Utility Accountability:** It is important to correctly evaluate the relevance/contribution of the provided data for the resulting final machine learning model.

Each ML algorithm micro-service will be packed as Docker image. A Docker image is an artifact used to execute some software in a Docker container. An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container.

Docker is an open source OS-level virtualization software platform primarily designed for Linux and Windows. Docker uses resource isolation features of the OS kernel, such as c-groups in Linux, to run multiple independent containers on the same OS. A container that moves from one Docker environment to another with the same OS will work without changes, because the image includes all of the dependencies needed to execute the code [8].

A container differs from a virtual machine (VM), which encapsulates an entire OS with the executable code atop an abstraction layer from the physical hardware resources.

Within the end user's virtual machines dedicated to run the client connector, resources are supervised by the Orchestration Container Manager which is the component that provisions the runtime environments for each ML algorithm.

As Orchestration Container Manager, Kubernetes was chosen. It is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery [11].

It is worth noticing that there's a perfect match with the concept of Custom Container and/or App Store Container in the IDSA's RAM and, as an internal communication mechanism, a message broker will be used so that the Client Connector can coordinate properly the workflows and the dataflows.

Such a message broker may be based on Kafka, which allow to publish and subscribe to streams of records, similar to a message queue or enterprise messaging system, store streams of records in a fault-tolerant durable way and process streams of records as they occur.

Kafka is generally used for two broad classes of applications: *(i)* building real-time streaming data pipelines that reliably get data between systems or applications; *(ii)* building real-time streaming applications that transform or react to the streams of data [12].

The ML algorithm micro-services, as mentioned before, must be wrapped so that they include an internal connector to obtain the training sets supporting multifarious sources as well as an

external connector that sends the trained model, once the job is finished, to the MUSKETEER server.

More in detail, in order to make client connector able to use data which is stored in different storages, it has to be made available a set of internal connectors to user's data sets, like:

- HIVE connector, to read data from or write data to Hive data sources. The Apache Hive data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL. Structure can be projected onto data already in storage [13].
- Cassandra connector to read data from or write data to Cassandra data sources, and enable the ingestion of temporal data in real time and maintain these records with a long retention period. The Apache Cassandra database is the right choice when scalability and high availability are needed without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it suitable for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages [14].
- Redis to read data from or write data to REDIS data sources. Redis is an open source, in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams [15].
- HDFS to read data from or write data to HDFS systems. It is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject [16].

Each ML algorithm will also send status update messages to the server using the MUSKETEER-Client python libraries.

### 5.1.1 Cluster Client Connector: Proposed APIs

In this section the messages for each individual service/API are described. This is not intended as a definitive API guide, but rather a synthesis of functionality required to build a full end-end API.

In the following tables the interfaces of the MUSKETEER Cluster Client Connector are defined.

<b>MUSKETEER Client Processor: Create task</b>	
<b>Technical interface ID</b>	MCP_CT
<b>Endpoint name</b>	Create Task
<b>Endpoint description</b>	Used by the UI to create a new Task
<b>Component</b>	MUSKETEER Client Processor
<b>Endpoint URL</b>	/create_task
<b>HTTP method</b>	POST
<b>Request parameters</b>	Task name, Task definition
<b>Request body</b>	User defined task entity attributes
<b>Response body</b>	Task entity

<b>MUSKETEER Client Processor: Get tasks</b>	
<b>Technical interface ID</b>	MCP_GTs
<b>Endpoint name</b>	Get Tasks
<b>Endpoint description</b>	Get the list of the available tasks
<b>Component</b>	MUSKETEER Client Processor
<b>Endpoint URL</b>	/get_tasks
<b>HTTP method</b>	GET
<b>Request parameters</b>	None
<b>Request body</b>	None
<b>Response body</b>	Collection of task entities

<b>MUSKETEER Client Processor: Get task info</b>	
<b>Technical interface ID</b>	MCP_GTI
<b>Endpoint name</b>	Task Info
<b>Endpoint description</b>	Provides the details of the tasks
<b>Component</b>	MUSKETEER Client Processor
<b>Endpoint URL</b>	/task_info
<b>HTTP method</b>	GET
<b>Request parameters</b>	None
<b>Request body</b>	None
<b>Response body</b>	Collection of task info entities

<b>MUSKETEER Client Processor: Get joined tasks</b>	
<b>Technical interface ID</b>	MCP_GJT
<b>Endpoint name</b>	Get joined tasks
<b>Endpoint description</b>	Provides a list of the tasks user has joined to
<b>Component</b>	MUSKETEER Client Processor
<b>Endpoint URL</b>	/get_joined_tasks
<b>HTTP method</b>	GET
<b>Request parameters</b>	User ID
<b>Request body</b>	None
<b>Response body</b>	Collection of task entities

MUSKETEER Client Processor: Aggregate	
Technical interface ID	MCP_Ag
Endpoint name	Aggregate
Endpoint description	Starts the FMM aggregation of a certain task
Component	MUSKETEER Client Processor
Endpoint URL	/aggregate
HTTP method	POST
Request parameters	None
Request body	Task entity, dataset
Response body	Success json

MUSKETEER Client Processor: Get result task image	
Technical interface ID	MCP_GRTI
Endpoint name	Get result task image
Endpoint description	Provides a metrics evaluation chart resulting from a task execution
Component	MUSKETEER Client Processor
Endpoint URL	/results/image/<task>
HTTP method	GET
Request parameters	Task name
Request body	None
Response body	Image representing the metrics evaluation chart of the task selected [B64 IMAGE]

<b>MUSKETEER Client Processor: Participate</b>	
<b>Technical interface ID</b>	MCP_Pa
<b>Endpoint name</b>	Participate
<b>Endpoint description</b>	Forwards the request for participation related to a task
<b>Component</b>	MUSKETEER Client Processor
<b>Endpoint URL</b>	/participate
<b>HTTP method</b>	POST
<b>Request parameters</b>	None
<b>Request body</b>	Task entity, datasets
<b>Response body</b>	Success json

<b>ML Algorithm Catalogue: Get algorithms</b>	
<b>Technical interface ID</b>	MLAC_GAs
<b>Endpoint name</b>	Get algorithms
<b>Endpoint description</b>	Provides a list of all the registered algorithms
<b>Component</b>	ML Algorithm Catalogue
<b>Endpoint URL</b>	/algorithms
<b>HTTP method</b>	GET
<b>Request parameters</b>	None
<b>Request body</b>	None
<b>Response body</b>	A collection of algorithm entities

ML Algorithm Catalogue: Get algorithm	
Technical interface ID	MLAC_GA
Endpoint name	Get algorithm
Endpoint description	Provides an algorithm by ID
Component	ML Algorithm Catalogue
Endpoint URL	/algorithms/<id>
HTTP method	GET
Request parameters	Id
Request body	None
Response body	Algorithm entity

ML Algorithm Catalogue: Create algorithm	
Technical interface ID	MLAC_CA
Endpoint name	Create algorithm
Endpoint description	Registers a new algorithm to the catalogue
Component	ML Algorithm Catalogue
Endpoint URL	/algorithms
HTTP method	POST
Request parameters	None
Request body	User defined algorithm entity attributes
Response body	Algorithm entity [JSON]

### 5.1.2 Cluster Client Connector: Workflows

The pictures below, show the interactions that occur among the components of the MUSKETEER Cluster Client Connector during the execution of the main scenarios of “list tasks”, “create a task”, “join a task” and “aggregate”.

Concerning the tasks listing, once the MUSKETEER user lands on the main screen of the Client Connector UI, the list tasks event is triggered and the client connector performs a request to the processor, that forwards the same request to the MUSKETEER Server using the communication library, in order to retrieve the list of the tasks (Figure 8).

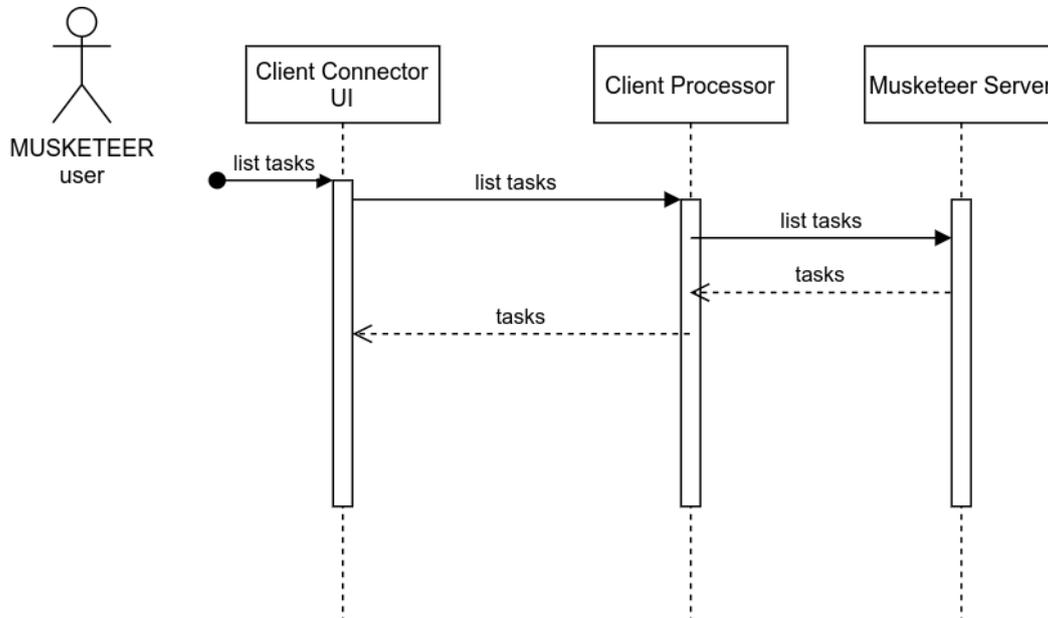


Figure 8 – ‘List tasks’ sequence diagram

The second important workflow, concerns the creation of a new task (Figure 9). The user is able to configure the new task picking a set of choices related to the algorithm she wants to use, the POM level among the available ones and metadata. The client processor retrieves all the algorithms and POMs from the ML Algorithm Catalogue in order to allow the user to build her new task. Once algorithm and POM are selected and all the forms are filled, the user uploads a json file descriptor that contains the description of the data used by the algorithm. At the end of the activity, the user clicks on the create task button, all the information are sent to the processor which communicates with the Musketeer Server submitting the new request.

The other key scenario is about the participation to a task (Figure 10). The user selects a task she wants to participate to, and the Client Processor forward the request to the Musketeer Server. Once the participation is approved, the user fills a form with the reference to the dataset and then starts the participation. The Client Processor triggers an initialization event on the Execution Engine which pulls the involved algorithm from the Machine Learning Algorithm Catalogue and asks to the Orchestration Container Manager to deploy the local job.

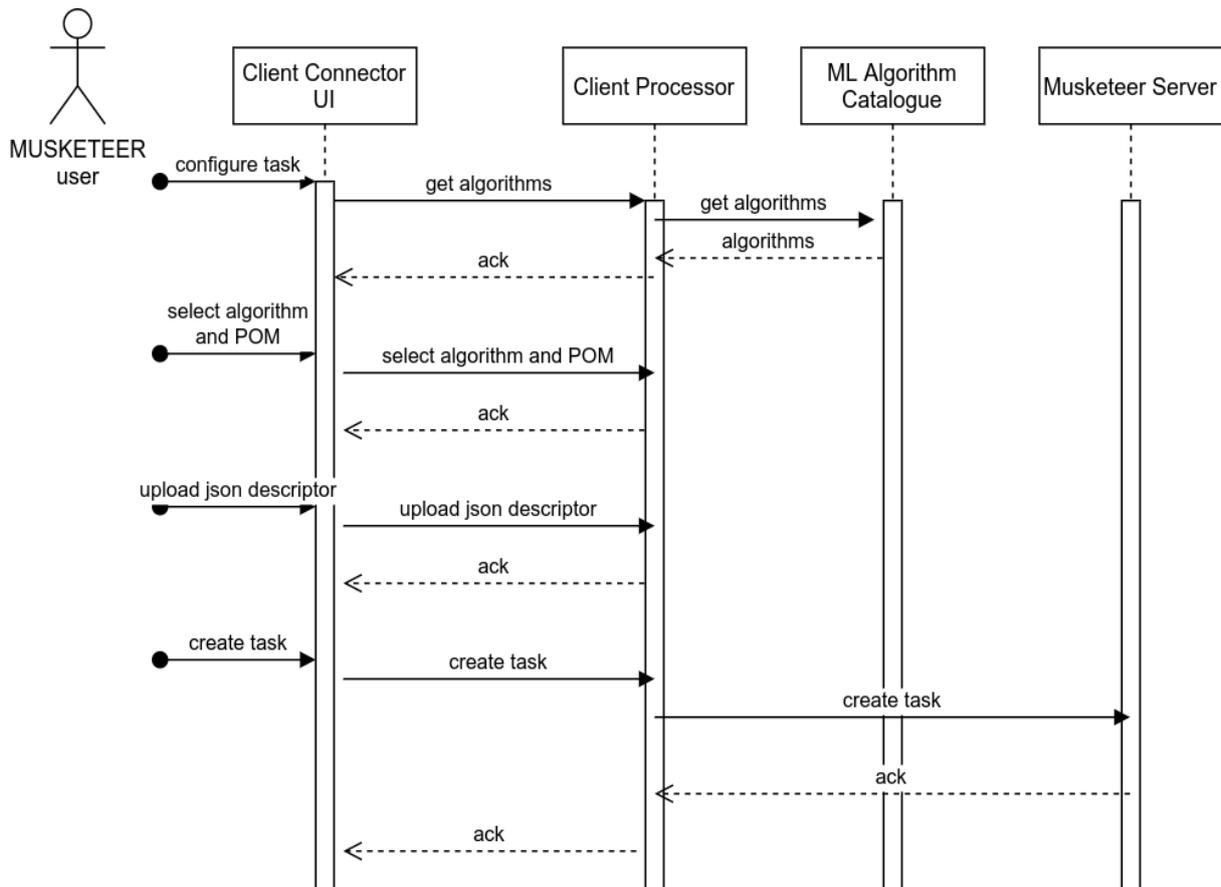


Figure 9 - “create a task” sequence diagram

A new Local Job gets instantiated and it interacts with the Storage in order to get access to the data the user has previously selected, then, it processes the data according to the algorithm. Once done, the Local Job produces a Machine Learning model which is sent to persistence, the local job notifies the Client Processor and then terminates. At that point, the Client Processor sends the model and the related metadata to the Musketeer Server in order to deliver the contribution of the federated model. If the user is also an aggregator, in the same way as the Desktop Client Connector, an aggregation process has to be started. The responsible component for this activity is the Client Processor that runs the aggregation algorithm in a dedicated process. The Processor keeps listening on the model contributions coming from the participants and, iteratively, sum them all. At the end of the loop, the Client Processor sends the final aggregator model to the MUSKETEER server and generates an evaluation metrics chart for assessment purposes.

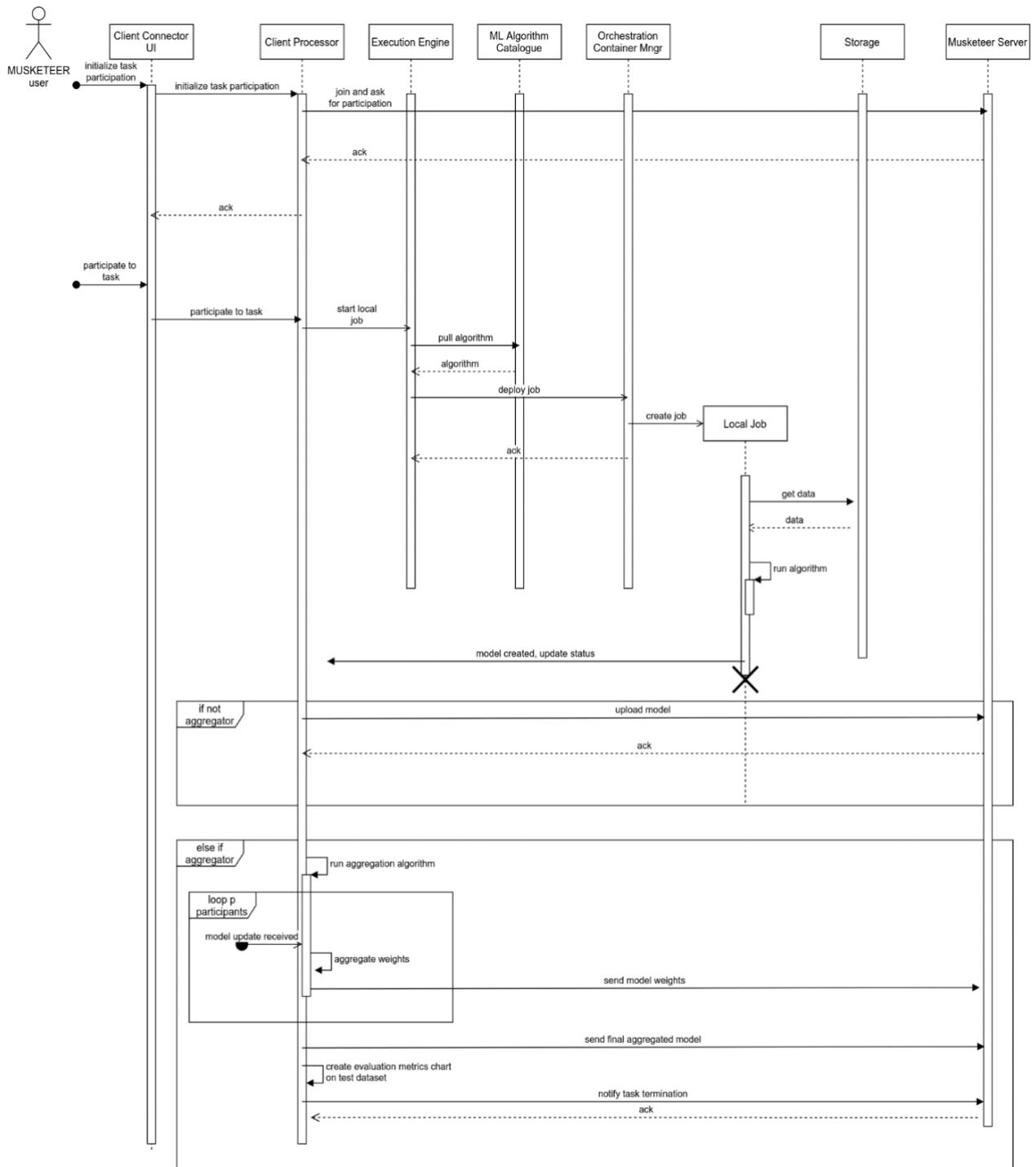


Figure 10 – “Task participation” sequence diagram

## 5.2 MUSKETEER Desktop Client Connector

The Desktop Client Connector architecture is shown in Figure 11. The application is mainly composed by 5 components that will be described in detail. There are also two external components that are loaded inside the Client Connector after the application is *up and running*: the communication messenger and the federated machine learning (MMLL) library. This solution produces a modular application with respect to those components, reusable in any context and independent from the central server and federated machine learning library used.

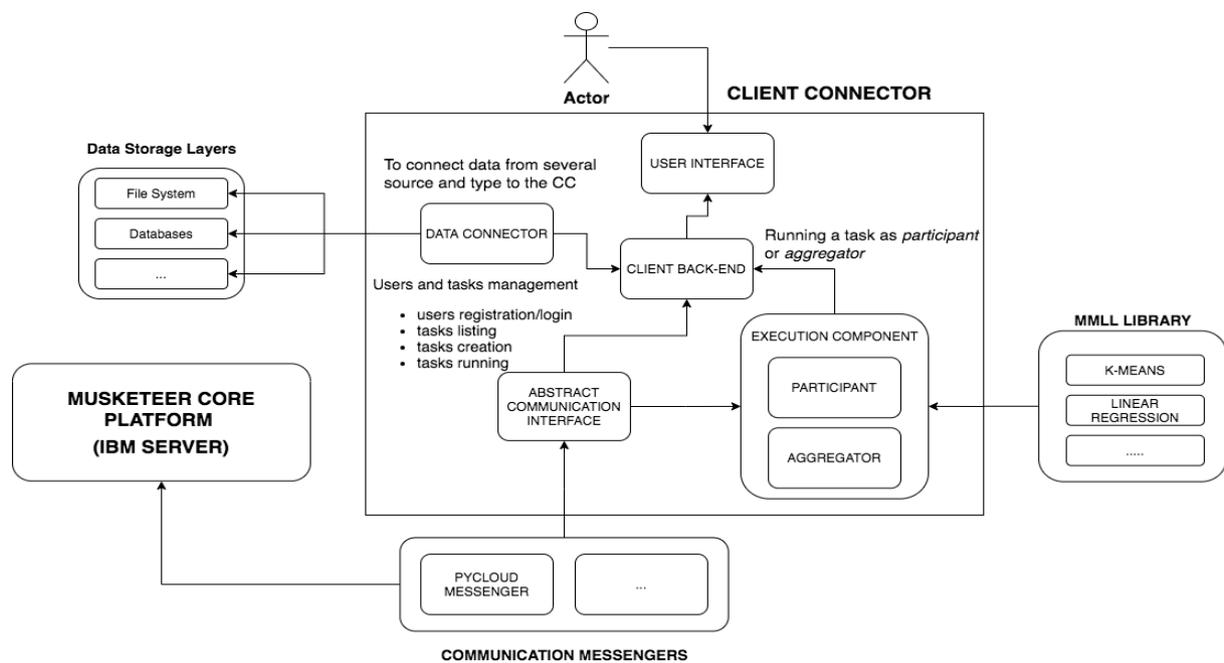


Figure 11 - Desktop Client Connector Architecture

At higher level, the *Actor*, through a *User Interface* component, that is a local web application, performs a set of functionalities that are described on Section 5. These functionalities range from the access to the target server platform, with which the *Communication Messenger* library communicates, to the binding of local data to the Client Connector, up to obtaining the results produced by the completed tasks. The *User Interface* is developed as a web application using Angular CLI version 8.3.8 [17]. This component represents the frontend part of the Client Connector, in accordance with the specifications described in D3.2.

The core *Client Back-End* component acts as a RESTful Web Service that handles all user requests, ranging from local operations (e.g. to connect user data to the Client Connector) to server operations (e.g. tasks and users management); these operations need to use a *Communication Messenger* library to communicate toward a target external server. In particular, Flask, a lightweight WSGI (web server gateway interface) web application framework, has been used [18]. WSGI is basically a protocol defined so that Python application

can communicate with a web-server and thus be used as web application outside of CGI (common gateway interface).

The *Data Connector* component connects user data, which may come from different sources or storage layers, to the Client Connector. In addition to connecting data from different sources, the component can manage and support different kinds of data: in fact, a user can load a CSV tabular data from the File System, images files, binary data, a table from a database and so on. Depending on the source from which the datasets are retrieved, and their data format, different libraries may be used. For example, the current version of the Desktop Client Connector retrieves datasets in CSV format through the Pandas library (<https://pandas.pydata.org>). Pandas is an open-source library, which allows you to read and process structured data providing high-performance.

The *Abstract Communication Interface* component allows to import and use an implementation of the communication library. In the MUSKETEER project the *Communication Messenger* library used is the *pycloudmessenger* library developed by IBM, and it is available at the following URL: <https://github.com/IBM/pycloudmessenger>. After the *pycloudmessenger* library is configured and installed, the Client Connector can use the APIs to communicate toward the MUSKETEER core platform. As a result, this component integrates all the user and task management parts: the listing task functionality, login and registration step, task creation and so on. This component is also connected and used by the *Execution* component, since during the training process the weights are sent and received to and from the central server (Musketeer Core Platform) using the *Communication Messenger* (*pycloudmessenger*) API.

On the other hand, the execution of tasks as a participant or aggregator is handled by the *Execution* macro-component. This component instantiates and runs a federated machine learning algorithm according to an interface that has been defined in WP4 by UC3M and TREE; which algorithm to be used and with which parameters are defined in the task definition and stored in the central server during the task execution. As well as the *Communication Messenger* library, the *Federated Machine Learning* library is an external library imported into the Client Connector. The imports of these libraries can be fully performed through *User Interface* in an initial configuration step after the first start of the Desktop Client Connector application.

### 5.2.1 Desktop Client Connector: Proposed APIs

In this section the messages for each individual service/API are described. This is not intended as a definitive API guide, but rather a synthesis of functionality required to build a full end-end API.

In the following tables the interfaces of the MUSKETEER Desktop Client Connector are defined.

Client Back-End: Get algorithms	
Technical interface ID	CBE_GAs
Endpoint name	Get algorithms
Endpoint description	Provides a list of all the registered algorithms
Component	Client Back-End
Endpoint URL	/cc/catalogue/algorithms
HTTP method	GET
Request parameters	None
Request body	None
Response body	A collection of algorithms

Client Back-End: Get POMs	
Technical interface ID	CBE_GPs
Endpoint name	Get POMs
Endpoint description	Provides a POMs metamodel
Component	Client Back-End
Endpoint URL	/cc/catalogue/poms
HTTP method	GET
Request parameters	None
Request body	None
Response body	A POMs (Privacy Operation Modes) metamodel

Client Back-End: Get step configuration	
Technical interface ID	CBE_GSC
Endpoint name	Get step configuration
Endpoint description	Provides a number pointing the configuration step to complete (-1 if all configuration steps have been completed)
Component	Client Back-End
Endpoint URL	/cc/configurations/step
HTTP method	GET
Request parameters	None
Request body	None
Response body	The configuration step to complete [JSON]

Client Back-End: Set communication configuration	
Technical interface ID	CBE_SCC
Endpoint name	Set comm configuration
Endpoint description	It configures and downloads the messenger communication library
Component	Client Back-End
Endpoint URL	/cc/configurations/comm
HTTP method	POST
Request parameters	None
Request body	The information to download and set the messenger communication library for the MUSKETEER server
Response body	Success JSON

<b>Client Back-End: Get communication configuration</b>	
<b>Technical interface ID</b>	CBE_GCC
<b>Endpoint name</b>	Get comm configuration
<b>Endpoint description</b>	Provides the communication library configuration stored into the Client Connector
<b>Component</b>	Client Back-End
<b>Endpoint URL</b>	/cc/configurations/comm
<b>HTTP method</b>	GET
<b>Request parameters</b>	None
<b>Request body</b>	None
<b>Response body</b>	The communication library configuration [JSON]

<b>Client Back-End: Set Machine Learning library configuration</b>	
<b>Technical interface ID</b>	CBE_SMLLC
<b>Endpoint name</b>	Set MMLL configuration
<b>Endpoint description</b>	It configures and downloads the Machine Learning library
<b>Component</b>	Client Back-End
<b>Endpoint URL</b>	/cc/configurations/mml
<b>HTTP method</b>	POST
<b>Request parameters</b>	None
<b>Request body</b>	The information to download and set the Machine Learning library
<b>Response body</b>	Success JSON

Client Back-End: Get Machine Learning library configuration	
Technical interface ID	CBE_GMLLC
Endpoint name	Get MMLL configuration
Endpoint description	Provides the Machine Learning library configuration stored into the Client Connector
Component	Client Back-End
Endpoint URL	/cc/configurations/mml
HTTP method	GET
Request parameters	None
Request body	None
Response body	The Machine Learning library configuration [JSON]

Client Back-End: Get datasets	
Technical interface ID	CBE_GD
Endpoint name	/cc/datasets
Endpoint description	Provides metamodels of the datasets connected to the Client Connector
Component	Client Back-End
Endpoint URL	/cc/datasets
HTTP method	GET
Request parameters	None
Request body	None
Response body	Metamodels of the datasets connected to the Client Connector [JSON]

Client Back-End: Get result task image	
Technical interface ID	CBE_GRTI
Endpoint name	Get result task image
Endpoint description	Provides a metrics evaluation chart resulting from a task execution
Component	Client Back-End
Endpoint URL	/cc/results/image/<task>
HTTP method	GET
Request parameters	Task name
Request body	None
Response body	The metrics evaluation chart of the task selected [B64 IMAGE]

Client Back-End: Get task logs stream	
Technical interface ID	CBE_GTLS
Endpoint name	Get task logs stream
Endpoint description	Provides the logs produced by a task you have run as a participant or aggregator
Component	Client Back-End
Endpoint URL	/cc/results/stream/logs/<task&mode>
HTTP method	GET
Request parameters	Task name and execution mode (participant/aggregator)
Request body	None
Response body	The logs produced by a task execution [JSON]

Client Back-End: Add dataset	
Technical interface ID	CBE_AD
Endpoint name	Add dataset
Endpoint description	It stores a dataset metamodel containing the information needed by the Data Connector component to retrieve that dataset
Component	Client Back-End
Endpoint URL	/cc/datasets
HTTP method	POST
Request parameters	None
Request body	Information about the source and type of dataset, and other metadata
Response body	Success JSON

Client Back-End: User login	
Technical interface ID	CBE_ULI
Endpoint name	Login user
Endpoint description	Provides authentication to the MUSKETEER server through the communication messenger library configured into the Client Connector; creates a local session into the Client Connector
Component	Client Back-End
Endpoint URL	/cc/comms/login
HTTP method	POST
Request parameters	None
Request body	User credentials: username and password
Response body	Success JSON

Client Back-End: User logout	
Technical interface ID	CBE_ULO
Endpoint name	Logout user
Endpoint description	Terminate the local authentication session
Component	Client Back-End
Endpoint URL	/cc/comms/logout
HTTP method	POST
Request parameters	None
Request body	None
Response body	Success JSON

Client Back-End: User registration	
Technical interface ID	CBE_UR
Endpoint name	Register user
Endpoint description	Provides registration to the MUSKETEER server through the communication messenger library configured into the Client Connector
Component	Client Back-End
Endpoint URL	/cc/comms/registration
HTTP method	POST
Request parameters	None
Request body	User credentials: username, password and organization name
Response body	Success JSON

Client Back-End: Get tasks	
Technical interface ID	CBE_GTs
Endpoint name	Get tasks
Endpoint description	Get the list of all the available tasks registered in the MUSKETEER server
Component	Client Back-End
Endpoint URL	/cc/comms/tasks
HTTP method	GET
Request parameters	None
Request body	None
Response body	The list of the available tasks registered to the Musketeer server [JSON]

Client Back-End: Get tasks joined	
Technical interface ID	CBE_GTJ
Endpoint name	Get user assignments
Endpoint description	/cc/comms/tasks/assigned
Component	Client Back-End
Endpoint URL	Get the list of all the tasks the user is participating to
HTTP method	GET
Request parameters	None
Request body	None
Response body	The list of all the tasks the user is participating in [JSON]

Client Back-End: Aggregate task	
Technical interface ID	CBE_AT
Endpoint name	Aggregate task
Endpoint description	/cc/fml/aggregate
Component	Client Back-End
Endpoint URL	Start a task as aggregator (only the task creator can run a task as aggregator)
HTTP method	POST
Request parameters	None
Request body	The task name and metamodels of the datasets to process during the aggregation
Response body	Success JSON

Client Back-End: Participate task	
Technical interface ID	CBE_PT
Endpoint name	Participate task
Endpoint description	Start a task as participant
Component	Client Back-End
Endpoint URL	/cc/fml/participate
HTTP method	POST
Request parameters	None
Request body	The task name and metamodels of the datasets to process during the task execution
Response body	Success JSON

### 5.2.2 Desktop Client Connector: Workflows

This section presents the main interaction workflows between the MUSKETEER user and the Desktop Client Connector. To show how the components of the Desktop Client Connector work together, sequence diagrams have been used. The following Figure 12 shows the sequence diagram related to the "list tasks" event performed by the user when accesses to the main page of the Desktop Client Connector. When the user triggers the "list tasks" event it starts a request from the Client Connector UI (user interface) to the Client Back-End.

This component will then invoke the logical Communication Interface component. This component makes use of the communication library imported during the configuration steps of the Desktop Client Connector to request and retrieve the list of tasks registered to the MUSKETEER platform.

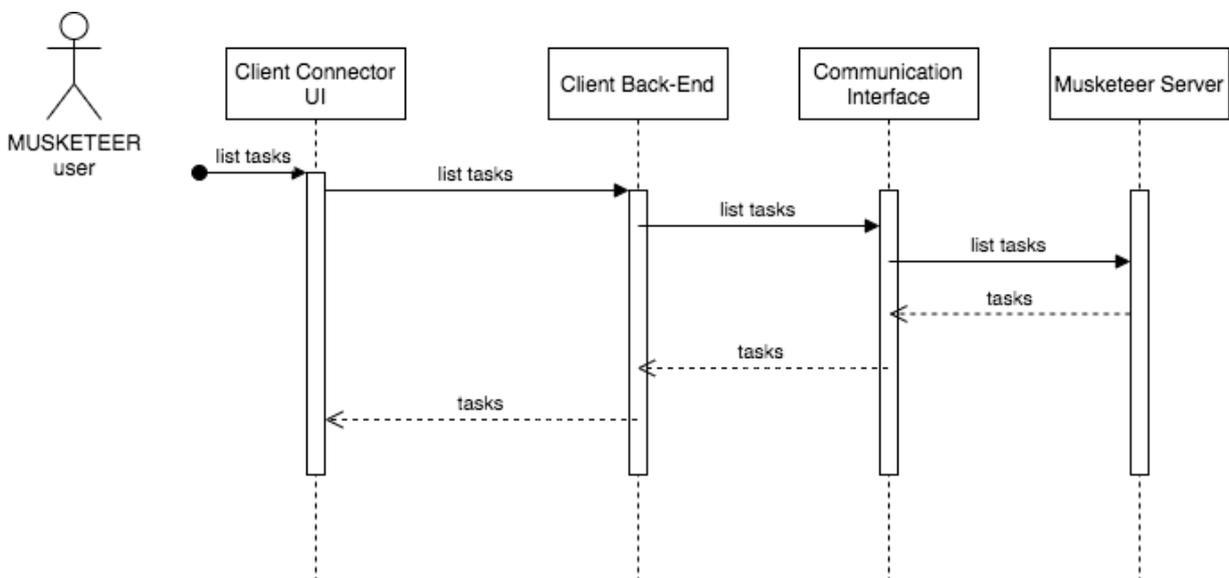


Figure 12 - Desktop Client Connector Sequence Diagram – Listing tasks

The second main event concerns the participation of a task by a MUSKETEER user. The related sequence diagram is shown in Figure 13. The user, from the UI, selecting the properly task she wants to join, can drag-and-drop the training dataset, and optionally also the validation and test datasets. The start of a task triggers a request to the Client Back-End which is responsible for handling the event.

The Client Back-End notifies the MUSKETEER Server that the user is going to participate to the selected task via the Communication Interface component. Once the participation has been validated by the MUSKETEER server, the Execution Component is delegated to start the task by instantiating an asynchronous process (local job). Once the local job has been successfully started, a confirmation ack is given back to the Client Connector, confirming the task job has been executed. The local job has all the information related to the task: algorithm type and POM to run, and the datasets to process. The datasets will be read through the Data Connector component, and then the chosen algorithm is instantiated and started. The algorithm is part



At a deferred time, the user who created the task can start it as an aggregator, selecting at most a validation dataset and a test dataset. The aggregator mainly handles the aggregation of the model weights received from the participating nodes, and at the end returns an aggregated model. The aggregator workflow, similarly to the participant’s workflow, is in charge of the Execution Component that instantiates a local job. As aggregation result, a metrics evaluation chart is created so that the user can assess the goodness of the model.

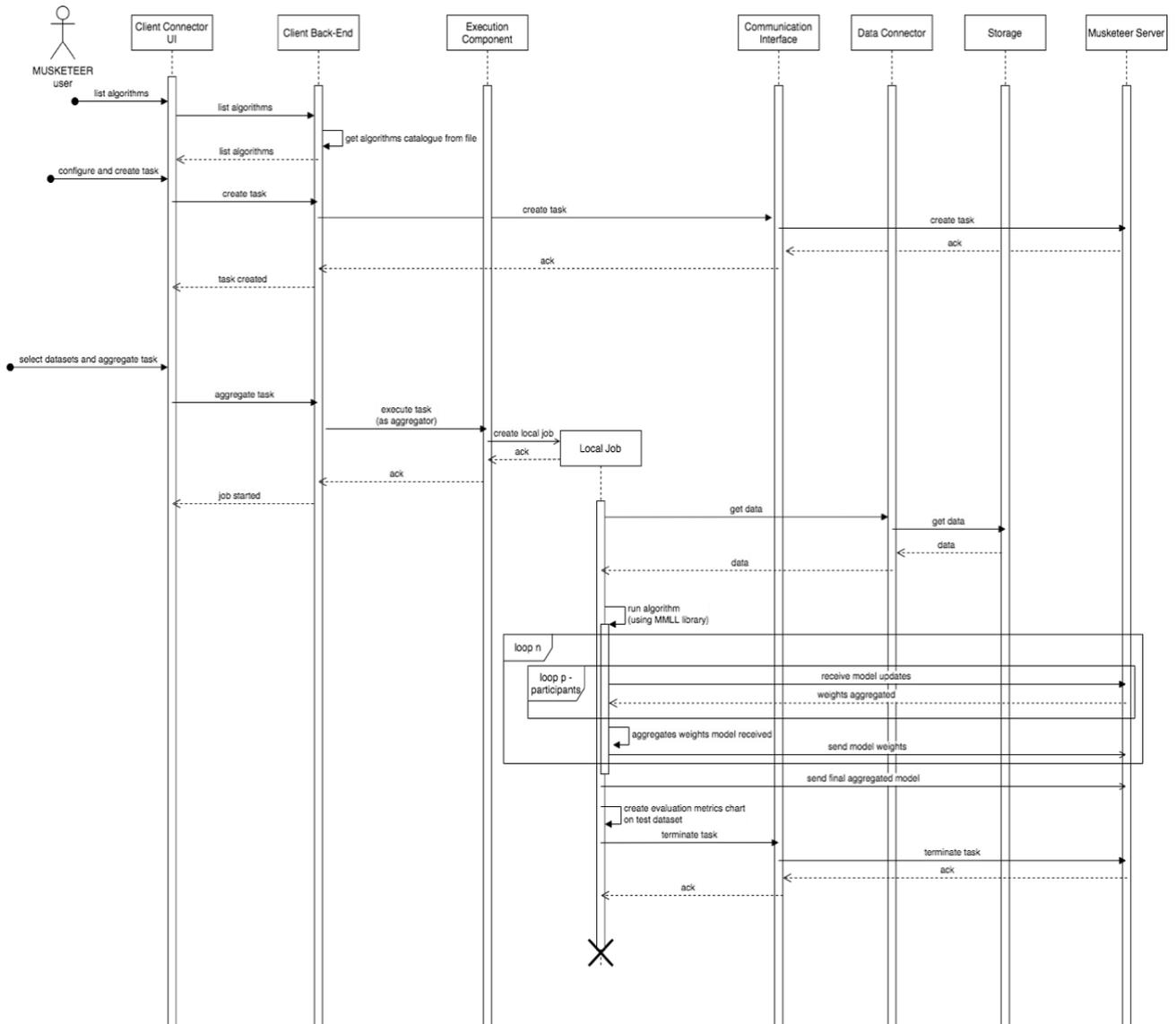


Figure 14 - Desktop Client Connector Sequence Diagram – Task aggregation

## 6 Conclusion

The purpose of this deliverable entitled *D7.2 - Client connectors' architecture design – Final version*, was to deliver the design specifications of the client connectors in MUSKETEER. This deliverable provides an update to the D7.1 content.

It is built directly on top of the first version of list of technical requirements presented in D2.1 and the knowledge extracted from deliverable D3.1 on the MUSKETEER general architecture, in order to deliver the details of the design of the client side of the integrated MUSKETEER platform.

It describes two different architecture of the client connector to meet two sets of user needs and requirements, while sharing the same user interactions: the desktop client connector and the cluster client connector.

The Desktop Client Connector can be used when data is collected in a non-centralized way and there is no need to use a cluster to distribute the workload, both in terms of computing and big data storage. Anyway, the Desktop version could also leverage GPUs for the training process, enabling the processing of a large amount of data in terms of volume. Finally, the Desktop Client Connector can be easily deployed in any environment thanks to the use of Docker in order to containerize the Client Connector application.

The Cluster Client Connector is devised to meet big data processing and federated machine learning needs. From the user perspective, there are not many changes related to the user interface and the user experience because the frontend consistency is kept by design. Contrarywise, there are deep differences in the backend side of the architecture due to the distributed nature of the system.

It is worth noticing that the architectural design was done always having in mind the IDSA reference architecture. The alignment with the Industrial Data Platform standards brought forward by the Industry Data Space (IDS) Association guarantees that the MUSKETEER project outcomes will be interoperable with any other asset building on the IDSA standards. However, it is difficult at this stage to talk about actual compliance with IDSA standards.

However, as the project development activities evolve, this initial design of the described services composing the client connectors will receive the necessary updates and optimisations in order to encapsulate all the project's advancements, as well as the new technical requirements that will be extracted from the feedback that will be collected from the platform's evaluation.

## 7 References

- [1] Reference Architecture Model. Version 3.0. April 2019. IDSA. <https://www.internationaldataspaces.org/wp-content/uploads/2019/03/IDS-Reference-Architecture-Model-3.0.pdf>
- [2] S. Newman (2015). Building Microservices – Designing Fined-Grained Systems, O’ Reilly.
- [3] S. Tarkoma (2012). Publish/Subscribe Systems: Design and Principles, John Wiley & Sons, Ltd.
- [4] <https://www.rabbitmq.com/>
- [5] <https://www.tensorflow.org/>
- [6] <http://caffe.berkeleyvision.org/>
- [7] <https://pytorch.org/>
- [8] <https://www.docker.com/>
- [9] <https://spring.io/projects/spring-boot>
- [10] <https://spring.io/projects/spring-cloud-dataflow>
- [11] <https://kubernetes.io/>
- [12] <https://kafka.apache.org/>
- [13] <https://hive.apache.org/>
- [14] <http://cassandra.apache.org/>
- [15] <https://redis.io/>
- [16] <https://hadoop.apache.org/hdfs/>
- [17] <https://cli.angular.io/>
- [18] <https://flask.palletsprojects.com>
- [19] <https://keras.io/>