H2020 – ICT-13-2018-2019

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

Grant No 824988

D3.1 Architecture Design – Initial Version

November 19

Imprint

Contractual Date of Delivery to the EC: **30 November 2019**

Author(s): **Mathieu Sinn (IBM), Mark Purcell (IBM), Minh Ngoc Tran (IBM), John Sheehan (IBM), Stefano Braghin (IBM)**
Participant(s): **TREE, IMP; ENG, UC3M; IDSA**
Reviewer(s): **Antoine Garnier (IDSA), Roberto Diaz Morales (TREE)**

Project: **Machine learning to augment shared knowledge in federated privacy-preserving scenarios (MUSKETEER)**

Work package: **WP3**
Dissemination level: Public
Version: **1.0**

Contact: **mathsinn@ie.ibm.com**
Website: **www.MUSKETEER.eu**

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

Executive Summary

This deliverable (D3.1 "Architecture Design") is a document describing the initial version of the MUSKETEER platform architecture. It addresses the previously delivered technical requirements and key performance indicators, takes into account legal and ethical requirements, and aligns with the algorithm library architecture and assessment framework. It informs the MUSKETEER platform development work and acts as counterpart of the client connectors' architecture, which describes the customization and end-to-end integration of the core platform capabilities for the industrial use cases.

Document History

Version	Date	Status	Author	Comment
1	01 November 2019	First version for internal review	Mathieu Sinn	First draft
2	20 November 2019	Final version for internal review	Mathieu Sinn	Draft for review
3		Review inputs	Antoine Garnier	Update
4		Review inputs	Roberto Diaz Morales	Update
5		Final Version		Update
6		Clean and submission	Gal Weiss	Final

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

Abbreviation	Definition
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
FaaS	Functions-as-a-Service
GQM	Goal/Question/Metric
IP	Internet Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
ML	Machine Learning
POM	Privacy Operation Mode
RAM	Random-Access Memory
REST	Representational State Transfer
SQL	Structured Query Language
TFIDF	Term Frequency – Inverse Document Frequency
TLS	Transport Layer Security
URL	Uniform Resource Locator
vCPU	Virtual Central Processing Unit
WP	Work Package
YAML	Yet Another Markup Language

1 Introduction

1.1 Purpose

The purpose of the MUSKETEER platform is to enable participants of the data economy to participate in Federated Machine Learning (ML) and thereby realize the value of their data assets, while preventing the leakage of information that is proprietary, confidential, personally sensitive, or that must not be shared because of other legal or regulatory requirements.

Functionally, the platform has to provide the infrastructure and implement the services that are required to enable the federated ML algorithms developed in WP4 and WP5 in end-to-end applications. It must also support the assessments to be carried it out in WP6 and provide interfaces which allow for the development of client connectors and end-to-end demonstration of the industrial use cases in WP7.

The purpose of this document is to describe the initial version of the MUSKETEER platform architecture. Particular emphasis is on:

- defining the scope of the core platform, particularly vis-à-vis the algorithmic library and the client connectors' architecture;
- explaining key design decisions in light of the envisioned scalability, security, trustworthiness and privacy-awareness of the platform;
- addressing the specific industrial, technical and legal requirements outlined in previous deliverables;
- documenting application programming interfaces (APIs) that expose core platform capabilities to the algorithmic library and client connectors' software;
- providing examples that illustrate how to use the platform APIs for federated learning and user/task management;
- discussing alignment of the architecture design with existing and emerging standards for industrial data platforms.

1.2 Related documents

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

- **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)** – in so far as the platform has to provide the core capabilities to support the development and deployment of the proposed client connectors' architecture.

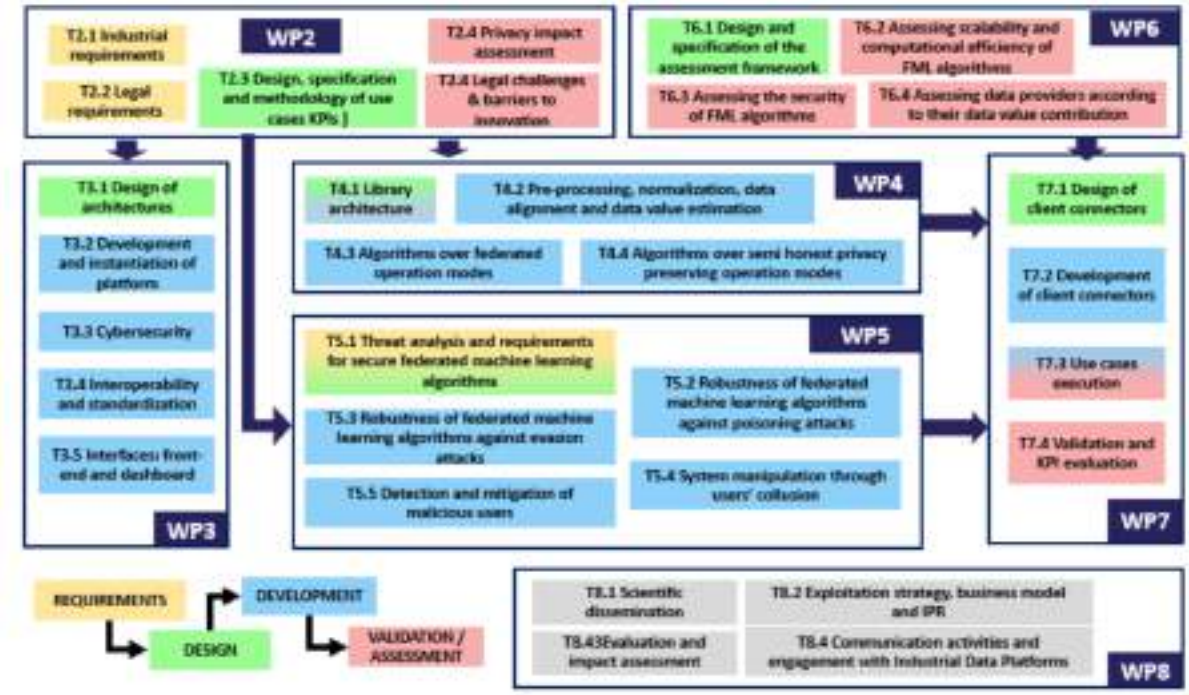

Figure 1: MUSKETEER's PERT diagram

1.3 Outline

The remainder of this document is structured as follows:

- Section 2 describes the scope of the MUSKETEER core platform (in particular vis-à-vis the algorithmic library and the client connectors software) and reviews the relevant functional and non-functional requirements outlined in the documents listed above.
- Section 3 describes the design of the platform architecture and provides detailed information on each of the platform's components as well as the underlying core technology.
- Section 4 outlines the proposed design of the API for utilizing the platform's services and shows an example how the API is intended to be used for federated learning algorithms and user/task management.
- Finally, Section 5 discusses possible extensions of the platform that were outside the scope of the initial version and may require further analysis in conjunction with other work packages for consideration in future versions to be developed under this project.

2 Requirements

2.1 Scope

When defining the scope of the MUSKETEER platform, it is important to draw distinctions between the core platform, the federated ML algorithm library, and the client connectors. This will become immanent when reviewing the technical requirements in Section 3 and understanding which of these three components they pertain to. At a high level, the scope of these three different components is defined along the following lines:

- The platform provides services (via an API) that allow new users to register to the platform.
- The platform provides registered users with the ability to create new federated ML tasks.
- The platform provides registered users with the ability to join existing federated ML tasks.
- The platform provides registered users with the ability to leave a federated ML task that they had previously joined.
- The platform provides registered users with the ability to cancel a task that they had previously created.
- The platform provides, during the execution of a federated ML algorithm, participants and the aggregator with the ability to send and receive messages in order to perform the federated ML training.
- The platform provides the aggregator with the ability to retrieve the number and status of participants in an ongoing task.

In its initial version, the platform does neither host nor start the aggregator training processes; the participants' training process are understood to be executed within the client connectors' software environments.

Any logic for performing the federated ML is implemented in the federated ML algorithms library. This includes logic for:

- Checking, on the aggregator side, whether the criterion for starting the federated ML training is met (e.g. quorum of participants, start time stamp etc.) and subsequently begin the actual training.
- Handling participants that either explicitly (via leaving a task) or implicitly (via disconnecting and not sending any further messages) cease to actually participate in an ongoing federated ML task.

- (De-)serialization of messages that are sent/received between the aggregator and task participants.

The core platform itself agnostic of specific elements of a Federated ML task definition. In particular, the platform does not make any assumptions about the actual ML backend (e.g. whether the training algorithms use Keras, Scikit-learn etc.). In principle, it doesn't even assume that the runtime is Python (with the only restriction that the API for interacting with the platform will be provided in Python, same as the sample code to be provided as well as scripts for running the aggregator and participants' processes).

The platform itself is agnostic, too, of whether messages are encrypted or not. It is unaware, too, of the working of local data connectors on the client side. Finally, it is unaware of whether/how trained models are deployed on the client side. (In the initial version of the platform, trained models are not persisted in a central location; in order to support local re-use and deployment of trained models, they would have to be stored locally as part of the participant training processes.)

On the other hand, the algorithms library will be agnostic of the actual protocol, backend and infrastructure that is used for sending and receiving messages. It is agnostic of where / how task and user information is stored.

2.2 Industrial and technical requirements

D2.1 (Industrial and technical requirements) comprises a detailed analysis of the MUSKETEER platform users and user stories, leading to an exhaustive list of technical requirements to drive the developments in the technical work packages (WP3-WP6) as well as the integration in WP7. In the following, we are going to review those technical requirements. We will discuss whether they fall under the responsibility of the core platform, the algorithms library or the client connectors software where applicable. We will refine them where needed and indicate their priority with regard to the first version of the platform architecture versus support in possible future extensions. We begin by examining and refining the platform user roles defined in Table 6 of D2.1.

2.2.1 User roles

A key aspect informing the design of the MUSKETEER platform architectures are the different roles of users that are interacting with the platform. A consolidated view of the user roles is provided in Table 1.

Table 1: MUSKETEER platform user roles

User role	Description
Platform administrator	A platform administrator has full access to the platform. He/she has the privilege to perform any action on the platform described in the following that any other user can perform in his/her role. Moreover, a platform administrator can register new users to the platform, provide them with initial usernames and passwords, change passwords of users, or delete users.
General user	A general user has access to the platform through a username and password provided by a platform administrator. General users can view and potentially join Federated ML tasks that have been created by other general users.
Task creator	A task creator user is a general user who has created a Federated ML task. Only the task creator is allowed to modify, stop or delete a task that they have created.
Task member	A task member user is a general user who has joined a Federated ML task created by a task creator. Task members have permission to participate in the training of the Federated ML task and potentially retrieve the trained model, depending on the Privacy Operation Mode (POM).
Aggregator	An aggregator is a task member whose role it is to coordinate Federated ML training and aggregate updates received from participants during the course of the training. In MUSKETEER, each Federated ML task involves exactly one aggregator.[1]
Participant	A participant is a task member whose role it is to contribute updates to Federated ML training based on their local data. Each Federated ML task involves at least one participant.

We note that, in D2.1, a few additional roles were described that we have subsumed here for consolidation and simplification purposes. In particular:

- We do not distinguish between **general** and **technical users**. In D2.1 this distinction was made to describe users who could register to the platform, view Federated ML tasks etc., who were however not allowed to

[1] There exist Federated ML protocols which consider multiple aggregators, e.g. for robustness and performance improvements, typically in compute environments where network connectivity is unstable. Since this is not an important requirement for MUSKETEER, we will be focusing on protocols that involve only one aggregator.

join tasks, create new tasks etc. In the initial version of the platform we do not see value of implementing this distinction; we will discuss, however, in Section 5 its relevance for possible future extensions of the platform.

- D2.1 also envisioned the role of a **group owner** who is a technical user with permission to facilitate the sharing of Federated ML tasks or models between members of the same organization or groups of organizations. We envision that, for the first version of the platform, a different instance will be deployed for each use case[2], thus, the separation between organizations or group will be enforced at the platform instance level. We will discuss the possible future importance of groups and group owners within a single platform instance in Section 5.
- Finally, D2.1 described the role of a **researcher** who is a general user aiming at benchmarking the performance of the platform and, towards that aim, needs the ability to run synthetic tasks involving multiple artificial users. We argue that the needs of this user role can be met by granting platform admin privileges on a dedicated platform instance.

On the other hand, D2.1 did not specify the roles of aggregators or participants. In some sense, while all the other roles typically correspond to human individuals exercising those roles, aggregators and participants are rather "algorithmic" roles.

2.2.2 Functional requirements

Next, we will provide a consolidated view of the functional requirements provided in Table 8 of D2.1. We will organize those requirements along three different categories:

- Managing platform users (Section 2.2.2.1)
- Managing Federated ML tasks (Section 2.2.2.2)
- Executing Federated ML tasks (Section 2.2.2.3)

2.2.2.1 Managing platform users

A consolidated view of the functional requirements for managing platform users is provided in Table 2: Functional requirements for managing platform users. At the end of the description

[2] Specifically, we plan to deploy one instance of the platform for the Manufacturing use case, a separate instance for the Healthcare instance, a separate instance for algorithmic research purposes on synthetic datasets and Federated ML tasks, and finally a separate instance for development and integration testing purposes.

of each requirement, we refer to the original identifier of the relevant functional requirement in D2.1.

Table 2: Functional requirements for managing platform users

ID	Description of the requirement
FR001	Ability for platform admin to grant username and password to new general user (**D2.1-FR034**).
FR002	Ability for platform admin to revoke username and password of existing general user (**D2.1-FR034**).
FR003	Ability for general user to avail of platform functionality through authentication with their username and password (**D2.1-FR001**).
FR004	Ability for general user to change their password (**D2.1-FR002**).

We emphasize a few requirements outlined in D2.1 that require further analysis before possible consideration in a future extension of the MUSKETEER platform design:

- D2.1 specified the ability for general users to provide and update personal profile information (**D2.1-FR002**), and to browse available information about other general users of the platform (**D2.1-FR005**). D2.1 also outlined the ability for general users to manage their own visibility (**D2.1-FR004**), i.e. to what extent their profile information would be accessible by other general users of the platform. We believe that, in order for those abilities to be considered as functional requirements of the MUSKETEER platform, an analysis of possible legal and ethical implications needs to be undertaken. Furthermore, it should be investigated whether such functionality would indeed help increase the value of the MUSKETEER platform for boosting the European data economy. A related effort – namely, creating a platform for professional networking and sharing assets among AI and Data Science practitioners – is currently undertaken with the AI4EU platform, so it could be worthwhile to explore its synergies with MUSKETEER to provide and complement such abilities. For the time being, we have assumed in our design of the initial version of the MUSKETEER platform a maximum degree of privacy protection, hence it is not possible for general users to browse information about other users, and also during the execution of Federated ML

training, the amount of information that is exposed about other participants is kept at minimum.

- D2.1 specifically indicated the ability for general users to provide and update information about datasets that they own, or even provide the datasets themselves (**D2.1-FR003**, **D2.1-FR036**, **D2.1-FR041**), along with the ability to browse datasets (or information about datasets) owned by other general users (**D2.1-FR006**). In some sense, this can be regarded as a special type of user profile information discussed in the previous paragraph. Same as before, we argue that a careful analysis of legal/ethical implications and the added value of such functionality is required, as well as a better understanding how the AI4EU platform could be leveraged for such purposes, in order to avoid duplication of efforts.
- D2.1 mentions the ability to manage access controls according to user groups, e.g. the visibility of information about datasets (**D2.1-FR003**) or the availability of trained ML models to third parties for downloading (**D2.1-FR018**, **D2.1-FR030**). Moreover D2.1 mentions the ability to manage groups by adding or removing general users (**D2.1-FR031**, **D2.1-FR035**). As discussed in Section 2.2.1, we see this ability as an important possible future extension of the platform to support multi-tenancy deployments and will discuss it in more detail in Section 5. The initial version of the platform is designed for single-tenancy deployments (as indicated in Section 2.2.1, we will deploy separate instances of the platform for the different MUSKETEER use cases) which alleviates the need for access controls via user groups.
- Finally, D2.1 outlines the ability to change the role of users (D2.1-FR035), specifically, grant general users admin privileges (D2.1-FR040)[3]. As will become clear in the following discussion of functional requirements pertaining to managing Federated ML tasks (Section 2.2.2.2), the change (or rather the addition) of roles such as task creator or task member occurs implicitly once a general user creates a new task a joins a task. The only other possible change of roles is for a general user to obtain admin privileges. We do not consider this, however, a functional

[3] As we had argued in Section 2.2.1, supporting the needs of a "researcher" general users essentially boils down to giving the researcher admin privileges so that he/she can create artificial users to study Federated ML training with as many participants as needed. Thus, **D2.1-FR032** relates to granting general users admin privileges, too.

requirement per say, but rather a process requirement, i.e. what is the organizational and approval process for general users to be issued the credentials that they need to have admin privileges.

2.2.2.2 Managing Federated ML tasks

A consolidated view of the functional requirements for managing Federated ML tasks is provided in Table 3. At the end of the description of each requirement, we refer to the original identifier of the relevant functional requirement in D2.1.

Table 3: Functional requirements for managing Federated ML tasks

ID	Description of the requirement
FR005	Ability for general users to create a new Federated ML task, including an unstructured description and all structured information that is required to define the task, such as the input data format, required mechanism for pre-processing the raw input data, the number of participants, starting/stopping criterions, etc. (**D2.1-FR016**, **D2.1-FR019**, **D2.1-FR043**).
FR006	Ability for a task creator to update the task description and information.
FR007	Ability for general users to list all the existing Federated ML tasks that have been created; view their description, definition and status; compute summary statistics, e.g., total number of tasks and participants (**D2.1-FR007**, **D2.1-FR008**, **D2.1-FR009**, **D2.1-FR010**, **D2.1-FR022**, **D2.1-FR027**, **D2.1-FR039**)
FR008	Ability for a general user to join a task that has already been created and that accepts new participants (**D2.1-FR012**).
FR009	Ability for a task member to actually participate in the training of that task's Federated ML model, either as aggregator or as participant (**D2.1-FR024**).
FR010	Ability for a task member to leave that task (**D2.1-FR029**).
FR011	Ability for a task creator to cancel that task (**D2.1-FR020**).
FR013	Ability for general users to list all the Federated ML models; view their description, definition, KPIs etc. if available (**D2.1-FR011**).
FR014	Ability for general users to download trained Federated ML models (**D2.1-FR013**, **D2.1-FR026**).

FR015	Ability for a task creator to delete the Federated ML models trained as part of that task (**D2.1-FR021**).

We emphasize a few requirements outlined in D2.1 that require further analysis before possible consideration in a future extension of the MUSKETEER platform design:

- D2.1 had suggested different mechanisms than **FR008** for a general user to join a Federated ML task, such as: selecting the tasks participants as part of the task creation process (**D2.1-FR016**), selecting which general users can join the Federated ML training "on-the-fly" (**D2.1-FR017**), or having potential task members seek for agreement by the task creator for them to join (**D2.1-FR023**). Similarly, D2.1 had suggested that task members would have to send a request to the task creator in order to leave a task that they had previously joined (**D2.1-FR029**), and the task creator would have to agree or disagree to such a request (**D2.1-FR023**). Some of those requirements contradict each other, and they may have legal, ethical or business implications that need to be further analysed. Therefore, for the initial version of the platform, we are considering the most basic functional requirements for joining/leaving a task as described in **FR008** and **FR010**.

- D2.1 had made different suggestions regarding the permission of general users for downloading trained Federated ML models, such as general users having to request permission (**D2.1-FR014**), general users having to pay for permissions (**D2.1-FR015**), the task creator deciding whether trained models would be available to any user, to specific groups of users, or kept privately (**D2.1-FR018**, **D2.1-FR030**), or trained models (intermediate and/or final) being accessible to task members. In light of those different, sometimes somewhat contradictory requirements, we decided to start with the most basic requirement described in **FR014**, however, we will discuss possible extensions and refinements in Section 5.

- D2.1 outlines requirements related to data monetization, such as the ability for task members be compensated for data that they contributed to Federated ML training (**D2.1-FR028**), or the ability to compute summary statistics of compensation and data value per user or per task (**D2.1-FR009**). While this could be important functionality to incentivize participation in Federated ML training, we believe that a further level of requirement analysis is needed before it can be envisioned to be supported in future versions of the platform; we will make a step in this direction in our discussion in Section 5.

- D2.1 includes two functional requirements (**D2.1-FR032**, **D2.1-FR033**) outlining the application of the MUSKETEER platform by researchers for

measuring and comparing performance on synthetic tasks/data with artificial users. In fact, this does not constitute any additional functional requirement, but – as discussed before – essentially requires the researcher to have platform admin privileges in order to efficiently perform such tasks.

When considering the requirements **FR005** and **FR009**, it is important to clearly distinguish between the responsibilities of the core platform, the algorithmic library and the client connections software (see the discussion in Section 2.1). Specifically, it is important to note that the core platform is not able to start or end the actual training processes on the client side (as required per **D2.1-FR019**). Those processes need to be initiated on the client side, either manually by the user, or automatically upon the user joining a Federated ML task.

In the following section, we are going to list the lower-level functional requirements that are required from an algorithmic viewpoint in order to support the higher-level requirement **FR009**.

For completeness' sake, we finally mention three functional requirements described in D2.1 which, from our viewpoint, are not relevant from the core platform perspective:

- Selecting datasets contributing to a Federated ML task (**D2.1-FR025**): This appears to be a manual process to be undertaken by task members. A possible functional requirement for the client connectors' software is that the selected datasets can be loaded in memory for the participation of the user in the actual Federated ML training.

- Pre-processing data by general users (**D2.1-FR043**): From an end-to-end platform perspective, this requirement pertains to defining appropriate data pre-processing steps as part of a Federated ML task definition and ensuring that the same pre-processing steps are performed by all task members, which has to be ensured by the algorithmic library in conjunction with the client connectors' software environment.

- Configuration of privacy-preserving data sharing methods (**D2.1-FR042**): This is a requirement for the algorithmic library to implement different privacy-preserving data sharing methods (e.g. POM1-POM6) and support the settings of those methods e.g. through configurable parameters of task definitions.

2.2.2.3 **Executing Federated ML tasks**

Finally, we provide a view of the functional requirements for executing Federated ML tasks in Table 4. Essentially, those are the platform capabilities that are required by algorithm developers to implement Federated ML algorithms in the algorithmic library. Requirements at this level had not been explicitly provided in D2.1, although they are implicitly needed to meet

requirement **FR009** in Table 3. Some of those requirements were described in the documentation of the prototype communications library in Section 7 of D4.2.

Table 4: Functional requirements for executing Federated ML tasks

ID	Description of the requirement
FR016	Ability for an aggregator or participant to retrieve the definition of a specific task.
FR017	Ability for an aggregator to retrieve the list of all participants of a specific task.
FR018	Ability for an aggregator to broadcast a message to all the participants.
FR019	Ability for an aggregator to send a message to a specific participant.
FR020	Ability for a participant to send a message to the aggregator.
FR021	Ability for a participant to route a message to the "next" participant (according to an underlying ring topology), without having to send it via the aggregator.
FR022	Ability for an aggregator to receive a message sent by a participant, together with an identifier of the participant who sent it.
FR023	Ability for a participant to receive a message sent by the aggregator.
FR024	Ability for a participant to receive a message routed from the "previous" participant (according to an underlying ring topology), including an identifier to distinguish from messages sent by the aggregator.
FR025	Ability for an aggregator to store task status updates.
FR026	Ability for an aggregator to store intermediate or final versions of the trained Federated ML model.
FR027	Ability for an aggregator to store information regarding the data value contributions per participants.

In Section 4 we will outline the design of a Python API that exposes the functional requirements described in **FR016-FR026** and that should thus allow algorithm developers to implement the methods described in D4.1 and D4.2.

2.2.3 Non-functional requirements

Next, we will provide a consolidated view of the non-functional requirements provided in Table 9 of D2.1.

Table 5: Non-functional requirements on the MUSKETEER platform

ID	Description of the requirement
NR001	High availability (**D2.1-NR001**).
NR002	Security, specifically regarding access control and adherence to industry security standards (**D2.1-NR002**).
NR003	Robustness of the overall platform with respect to software errors (**D2.1-NR016**).
NR004	Availability of appropriate logging mechanisms for all operations (**D2.1-NR010**).
NR005	Recoverability, specifically of the training of Federated ML models, from temporary system or component failures (**D2.1-NR003**, **D2.1-NR004**, **D2.1-NR005**, **D2.1-NR015**).
NR006	Scalability, specifically the efficient execution of Federated ML training algorithms (**D2.1-NR006**), and efficient handling of simultaneous requests (**D2.1-NR014**).
NR007	High usability, specifically regarding the ease of software installation for end users (**D2.1-NR009**) and the design of interfaces for interactions with the platform, including their documentation (**D2.1-NR008**).
NR008	Maintainability, specifically the availability of mechanisms to efficiently perform system or component updates with minimum downtime for the overall platform (**D2.1-NR007**, **D2.1-NR013**).

For completeness' sake we also mention **D2.1-NR007** which stipulates that the MUSKETEER platform should enable the interconnection and exchange of information among Federated ML task participants; since this essentially boils down to functional requirements described in Table 4, we exclude it from the list of non-functional requirements.

2.2.4 Technical requirements

Next, we provide a consolidated view of the technical requirements provided in Table 9 of D2.1, which are meant to be synthesis of the functional and non-functional requirements discussed before.

Table 6: Technical requirements on the MUSKETEER platform

ID	Description of the requirement	Related functional and non-functional requirements
TR001	The MUSKETEER platform requires users to authenticate with their unique username and a password in order to avail of the platform functionality, which includes exchange of information as part of Federated ML tasks (**D2.1-TR002**, **D2.1-TR003**, **D2.1-TR005**, **D2.1-TR006**).	**FR001**, **FR002**, **FR003**, **FR004**, **NR002**
TR002	The MUSKETEER platform allows general users to create one or more Federated ML tasks (**D2.1-TR007**, **D2.1-TR021**), the purpose of which is to train a machine learning according to the task definition on the task members' local data (**D2.1-TR008**).	**FR005**, **FR006**
TR003	Each task should be associated in the platform with a unique task identifier (**D2.1-TR009**).	**NR007**
TR004	Each task definition should include all the required information about the model to be trained such as hyperparameters, loss function etc. (**D2.1-TR031**).	**FR005**, **FR009**
TR005	Each task definition should include a general description of the task (**D2.1-TR010**).	**NR007**
TR006	Each task definition should include a description of the required input data features (**D2.1-TR010**)	**NR007**
TR007	Each task definition should include a definition of the input data pre-processing algorithms that are to be applied prior to the training of the Federated ML model (e.g. high pass filtering, edge detection, bag of words with TFIDF weighting …) (**D2.1-TR013**).	**D2.1-FR043**
TR008	A working implementation of input data pre-processing algorithms referred to in task definitions must be made available to task members (more specifically, to the participants) in the client connectors' software environment (**D2.1-TR014**, **D2.1-TR015**).	**D2.1-FR043**

TR009	Task members must have the ability to retrieve the task definition in order to configure the client connectors' software environment and contribute (either as aggregator or as participant) to the Federated ML training (**D2.1-TR016**).	**FR016**, **NR007**
TR010	The Privacy Operation Modes (POMs) implemented in the algorithm library must cover all the privacy restrictions that task members would want to apply to their data (**D2.1-TR017**, **D2.1-TR034**). This specifically includes the case where the task members want to collaborate to train a ML model without sharing or centralizing their local data (**D2.1-TR027**), thus no raw data must be transferred outside the task members' organizations client facilities and the ML model training is coordinated by an aggregator requesting and receiving model updates from the participants (**D2.1-TR026**).	**FR003**, **FR004**, **FR009**, **NR002**, **NR004**
TR011	In the task definitions, the privacy restrictions should be described in human-understandable terms (**D2.1-TR018**).	**NR007**
TR012	General users must have the ability to browse active Federated ML tasks (**D2.1-TR020**, **D2.1-TR021**).	**FR007**
TR013	The MUSKETEER platform must support the execution of Federated ML training among task members, comprising one aggregator and one or more participants (**D2.1-TR022**). This includes the transfer of information – such as sending and receiving models, model updates or gradients – among participants and the aggregator (**D2.1-TR011**, **D2.1-TR025**, **D2.1-TR032**, **D2.1-TR033**, **D2.1-TR035**, **D2.1-TR036**, **D2.1-TR037**). Depending on the POM, that information may or may not be encrypted (**D2.1-TR029**).	**FR009**, **FR016**-**FR025**, **NR006**
TR014	The MUSKETEER platform has to support potential re-encryption of information transferred among task members for POMs where task members use different private keys for the homomorphic encryption of their model updates (**D2.1-TR029**).	**FR009, D2.1-FR042**

TR015	Task members must have the ability to query the progress/status of Federated ML training tasks (**D2.1-TR023**).	**FR007**
TR016	Task members must have the ability to access the models trained as part of Federated ML tasks (**D2.1-TR024**).	**FR014**

The original list in D2.1 included a number of requirements pertaining to the management of datasets, such as providing or obeying access controls for datasets (**D2.1-TR001**, **D2.1-TR002**, **D2.1-TR003**, **D2.1-TR034**), or provisioning private cloud storage for users' encrypted data (**D2.1-TR030**). Considering the MUSKETEER platform in a broad sense where the platform comprises the end users' proprietary computational environments, those requirements could be considered in-scope, however, it would still remain the end users' responsibility to ensure the proper local access controls. From the core platforms' perspective, however, we consider those requirements to be out-of-scope, however, since the users' data is not supposed to enter the boundary of the core platform, and thus the required access controls must remain outside, too.

The technical requirements in D2.1 also consider the ability of the MUSKETEER platform to support the provision of monetary rewards (**D2.1-TR019**, **D2.1-TR023**). As discussed above, we do not consider requirements regarding the support of data value estimation and monetization in our design of the initial version of the MUSKETEER platform, however, we will discuss it as part of possible future extensions in Section 5.

Finally, we add some additional context and clarification on the technical requirements **TR007** and **TR008** pertaining to the input data pre-processing. We emphasize that the core platform is agnostic to the existence of input data pre-processing functions. The implementation of those functions is outside the scope of WP3, and the execution of those function outside the boundary of the core platform – same as the Federated ML algorithms. The responsibility of the core platform is only to store references to the pre-processing functions to be applied as part of Federated ML task definitions, and to make the information about which functions shall be applied available to any user of the platform, in particular to participants of Federated ML tasks who will ultimately have to execute those functions as part of the execution of the training algorithms within the client connectors' software environment on their premises.

2.3 Key performance indicators

D2.3 and D6.1 define key performance indicators (KPIs) for the MUSKETEER platform using the Goal/Question/Metric (GQM) methodology [1]. The four defined goals (G1-G4) comprise:

- **G1**: Evaluation of the platform architecture with respect to standardization and extensibility in the context of general use case validation (**G1.1**), of the healthcare use case (**G1.2**) and of the smart manufacturing use case (**G1.3**).
- **G2**: Evaluation of the privacy-preserving operation modes (POMs) with respect to privacy, computational/storage/communication requirements and data utility accountability, again the context of general use case validation (**G2.1**), the healthcare (**G2.2**) and the smart manufacturing (**G2.3**) use case.
- **G3**: Evaluation of the federated privacy-preserving ML algorithms in the context of WP6 evaluation scenarios, which are broken down with respect to performance (**G3.1.1**), reliability (**G3.1.2**), scalability (**G3.1.3**), computational efficiency (**G3.1.4**), and with respect to security (**G3.2**). Moreover, evaluation of those algorithms with respect to pre-processing, normalization, data alignment, supervised and unsupervised learning in the context of the healthcare (**G3.3**) and the smart manufacturing (**G3.4**) use cases.
- **G4**: Evaluation of rewarding models with respect to data value in the context of WP6 evaluation scenarios. (The G4 questions and metrics have not been defined yet; this will be part of the future deliverable D6.4.)

Table 7 lists a summary of the questions and metrics pertaining to core platform capabilities.

Table 7: GQM questions and metrics pertaining to core platform capabilities

IDs	Question	KPIs	Related requirements
G1.1_Q02	Does the MUSKETEER platform allow interoperability with ML frameworks?	Number of supported ML frameworks.	**FR005**, **FR007**, **FR016**, **TR002**, **TR004**, **TR009**
G1.1_Q04, **G1.2_Q01**,	Does it allow fast installation, deployment and use?	Effort to install/update client SW; effort to	**FR001**, **NR007**

G1.2_Q02, **G1.3_Q01**, **G1.3_Q02**		create and run a Federated ML task; effort to use a trained model; effort to onboard a new user.[4]	
G1.3_Q03	Are there different visibility constraints based on user permissions?	Different information for different user permissions (y/n).	**NR002**
G1.3_Q04	Is the architecture compliant with industry standard and production plant IT policies?	Compliance with such standards and policies (y/n).	**NR002**
G1.3_Q06	Is it possible to download the trained Federated ML models?	Possibility of downloading the ML model (y/n).	**TR016**
G1.3_Q07	Is the Federated ML model training fast enough?	Time for training (per sample), time for scoring	**FR018** – **FR026**
G1.3_Q08	When a new task is launched, what are the algorithm used and its parameters?	Possibility to access information about the algorithm used and its parameters (y/n).	**TR004**, **TR005**, **TR006**, **TR007**, **TR009**
G1.3_Q09	Is it possible to report a comment on an unexpected behavior of algorithms during a user session?	Ability to report an unexpected behavior (y/n).	
G2.3_Q06	How easy is it to verify if all the communications are working?	Possibility to verify if all the communication protocols are enabled (y/n).	**FR003**
G2.3_Q07	Which is the maximum dimension of messages supported by the platform?	Maximum dimension of messages (sent or received) supported by the platform.	**FR018** – **FR024**

[4] D2.1 also lists as a metric "Number of screens supported by help options". Since the development of screens, i.e. graphical user interfaces, is outside the scope of the core platform, we exclude this metric from this list.

G3.1.2_Q01	Does each ML algorithm give comparable output working on the same data and in the same conditions in different sessions (reliability)?	Standard deviation of normalized outputs in different sessions.	**FR018** – **FR026**
G3.1.3_Q01, **G3.1.3_Q02**, **G3.1.3_Q03**	Does the training algorithm scale up when the dimension of the application scenario grows in terms of the amount of data / users/ input features?	Trend profile of training time vs amount of data / users / input features.	**FR018** – **FR026**
G3.1.4_Q02	Are the message transmission costs reasonable?	Amount of information transmitted; fraction of training time dedicated to transmission.	**FR018** – **FR024**
G3.1.4_Q03	Is the memory usage during training reasonable?	Total memory usage by aggregator and participants normalized by size of dataset.	
G4.1_Q01	Is the task alignment procedure able to detect which are the most relevant data contributions to solve a given problem?	Error rate in experiments where the ground truth is known.	**FR022**
G4.1_Q02	the data value estimation method able to reward every participant according to the real data value of their data contribution?	Error in reward estimation in experiments where the ground truth is known.	**FR022**

The following goals and questions are not relevant for the evaluation of core platform capabilities, or will be discussed later, and have therefore not been included in Table 7:

- **G1.1_Q01** aims at evaluating the alignment of the MUSKETEER platform with the IDSA reference architecture, measured by the number of aligned artefacts. The alignment analysis will mostly pertain to the client connectors' software environment, nevertheless we will review this aspect in more detail in Section 2.7.
- **G1.1_Q03** aims at assessing the extensibility of the MUSKETEER platform in terms of whether it fosters the creation of a community of developers and researchers that can extend the platform with new algorithms and attack detection mechanisms; the principal KPI is the

amount of open source community interactions. Since this is not dependent on the architecture per say but on the licensing (including potentially open sourcing) of platform components, we excluded this question from Table 7.

- **G1.2_Q03** and **G1.3_Q05** aim at evaluating whether the MUSKETEER end-to-end platform requires the local deployment of special hardware. This will depend on the type of ML algorithms and the amount of data from the two use cases, thus we do not consider it here.
- **G1.2_Q04** aims at evaluating the interoperability of the MUSKETEER platform with Medical Imaging Systems standards, which is within the boundary of the client connectors' software environment, therefore we exclude it here.
- The questions for the evaluation of **G2.1**/**G2.2** and most of the questions for the evaluation of **G2.3** (with the exception of **G2.3_Q07** which is included in Table 7) pertain to the algorithmic library, therefore we did not include them here. Specifically, **G2.3_Q04** addresses the difficulty of encrypting/decrypting information as part of the Federated ML training, which is within the boundary of the algorithmic library[5], and **G2.3_Q05** the storage requirements, which is within the boundary of the client connectors' software.
- The questions related to **G3.2** pertain to the robustness of Federated ML training against evasion, poisoning and user collusion, which is within the scope of the algorithmic library framework.
- The questions related to **G3.3** and **G3.4** all pertain to the accuracy of ML models which falls within the scope of the algorithmic library and the development of the use cases.

2.4 Legal and ethical requirements

D2.2 outlines the legal requirements that are relevant to the scope of the MUSKETEER project and provides guidance in terms of their implementation. At this stage, the requirements pertaining to the core platform mainly concern the implementation of cybersecurity mechanisms and processes, such as

[5] Although we will discuss the relevance of key management solutions for possible future extensions of the platform in Section 5.

- Taking measures to ensure the security of the MUSKETEER platform as well as the relevant facilities;
- Availability of appropriate security incident handling processes;
- Measures to ensure business continuity in case of a security incident;
- Compliance with international security standards;
- Performing thorough monitoring, auditing and testing of the MUSKETEER system and facilities to ensure appropriate levels of security.

In terms of technical requirements, those implementation guidelines are reflected in **TR001** and the related functional and non-functional requirements (see Table 6), which specify e.g. the availability of access controls and user authentication.

2.5 Privacy operation modes and machine learning algorithms

In this section, we are going to review the technical requirements for supporting the different privacy operation modes (POMs) and Federated ML algorithms from the core platform perspective. At a high level, those requirements are described in **TR013**, and more detail is provided in the functional and non-functional requirements **FR009**, **FR016**-**FR025** and **NR006**. Here we discuss possible additional requirements.

Our analysis is based on two previous deliverables: **D4.1**, which provides a description of the targeted POMs and Federated ML algorithms, and **D4.2**, which describes an initial demonstrator of Federated ML algorithms, specifically of algorithms for aligning and estimation the value of participants' data. **D5.1**, which provides a threat analysis for Federated ML algorithms, did not contribute specific technical requirements at this point, however, future work in WP5 on defending Federated ML algorithms against the identified threats may lead to additional functional requirements on the core MUSKETEER platform.

2.5.1 Federated collaborative POMs (POM1-POM3)

POM1-POM3 all fall under the standard Federated ML training paradigm where raw data never leaves the participants' environment, instead the ML model is transferred among the participants who contributes by locally updating the model, using their data, and sending it to the aggregator who combines the model updates. A common feature of those POMs is that the participants have access, as part of the training process, to intermediate versions of the trained ML model (which is related to **TR016**), although the final model, after incorporation of the ultimate updates, is not necessarily shared with them by the aggregator.

The difference between POM1, POM2, POM3 is the approach for sharing and processing model updates in an encrypted domain:

- POM1 ("Aramis") handles model updates in the plain, unencrypted domain.
- POM2 ("Athos") works with (partial) homomorphic encryption of model updates where the same private key is used by all the participants.
- POM3 ("Porthos") envisions different private keys to be used by the participants to encrypt their model updates; in order for the aggregator to combine those updated in the encrypted domain, a proxy re-encryption scheme is required.

Fundamentally, **FR016**-**FR025** comprise all the non-encryption related functional requirements for performing Federated ML training under POM1-POM3. In particular, **F018**/**F019** and **F023** support the transfer (i.e. sending and receiving) of the ML model from the aggregator to participants, and **F020** and **F022** allow participants to transfer the model updated on their local data back to the aggregator. The transfer mechanisms are agnostic as to whether the transmitted information is encrypted or not.

Working with models and model updates in the encrypted domains requires the following:

- Encryption of model updates by the participants using their private key(s) before the updates are sent to the aggregator. The same private key is used among all participants in POM2, different private keys are used in POM3. We see this step as an integral functionality of the algorithmic library, thus, it does not inform technical requirements on the core platform. Also the generation and exchange of private keys (required in POM2) is outside the realm of the core platform.
- Applying model updates in the encrypted domain by the aggregator, possibly via a proxy re-encryption of updates encrypted using different private keys (in POM3). This requires knowledge of the public key(s) of the homomorphic encryption which can be transmitted to the aggregator in the same way as model updates. Again, those steps can be regarded as the integral functionality of the algorithmic library (and the core platform can stay completely agnostic to them).

Thus, we argue that the complete core platform functionality to support POM1-POM3 is comprised in the functional requirements **FR016**-**FR025**. From an end-to-end use case perspective, processes and mechanisms need to be defined for generating and managing keys (the most challenging requirement being the exchange of keys required for POM2). However, this this outside the realm of the technical requirements on the core platform. (We will revisit and

summarize our view, also with regard to possible future extensions of the platform, in Section 5).

2.5.2 Semi-honest scenarios (POM4-POM6)

POM4-POM6 consider semi-honest scenarios where Federated ML task members are honest-but-curious, i.e. while they follow the agreed-upon training protocol, they try to gather information about other members' inputs, intermediate results, or overall outputs. In particular, while in POM1-POM3 intermediate versions of the trained ML model are available to all task participants by default, POM4-POM6 provide different mechanisms to prevent the disclosure of this type of information.

- POM4 ("Rochefort") and POM5 ("deWinter)" deploy proxy-encryption of the users' data and, based on that, perform operations required for the ML model training either exploiting homomorphic properties of the cryptosystem, or protecting the privacy of the encrypted operands via cryptographic binding. POM5 specifically indicates sequential training protocols where participants' model updates are requested and incorporated sequentially by the aggregator.
- POM6 ("Richelieu") proposes to protect privacy by performing aggregation operations on the users' raw data (such as computing dot products, covariance matrices etc.) before sharing information with other task members. This way, no individual raw data is transferred outside the users' environment. POM6 aims at supporting different configurations where the ML model is either public to all participants (same as in POM1-POM3), or available only to the aggregator.

We note that a complete assessment of the functional requirements for POM4-POM6 will require the analysis of specific ML algorithms (at this stage, the descriptions of POM4-POM6 in D4.1 amount to a fairly general framework which is difficult to analyse in this regard). At a high level, we deem the functionality described in **FR016**-**FR025** to be sufficient to support POM4-POM6, not considering the requirements for supporting different cryptosystems which we believe – same as for POM1-POM3 – lies outside the realm of the core platform. POM4 potentially involves "private-cloud" services for secure data storage and re-encryption; since the core MUSKETEER platform will be hosted in the public cloud, such functionality – if critically required for the implementation of POM4 Federated ML algorithms – would have to reside outside the realm of the core platform. It is our understanding, anyhow, that such functionality may be beneficial to reduce the computational burden for the client connectors' computational environments, however, not critical from a strictly functional point of view.

2.5.3 Conventional ML scenarios (POM7-POM8)

Finally, D4.1 considers two conventional ML scenarios without specific privacy-preserving mechanisms:

- POM7 (Planchet) is a traditional cloud computing schema where all datasets are stored and the ML models are trained centrally in the cloud, with the possibility of selectively sharing with the users the resulting models.
- In the POM8 (D'Artagnan) schema, ML models are trained locally using local datasets.

We argue that neither of these two POMs require the MUSKETEER platform. Specifically, there exist a number of commercial cloud services already supporting POM7, and a number of standard software environments (commercial or open source) already supporting POM8. Thus, we do not derive any functional requirements on the MUSKETEER platform for supporting POM7 or POM8.

2.5.4 Algorithmic library assumptions

D4.2 describes a demonstrator of a first version of the algorithmic library, specifically of algorithms for data alignment and data value estimation. In the following we will analyze the assumptions that the demonstrator makes on available platform functionality and explain their relation to the technical requirements listed above.

1. A Federated ML task has been defined, i.e. the platform has to provide the ability for general users to create Federated ML tasks, including all the information that is required to define the task (**FR005**).
2. The platform has identified all the users participating in the training process, i.e. the platform provides the ability for general users to join a task that has already been created (**FR008**).
3. All task participants have access to the task description (**FR016**). A test has been performed to guarantee that the participants' input data has the required format[6].

[6] We envision that the code for performing such tests has to be available in the client connectors' computational environment, same as the algorithmic library. The task definition, accessible via the platform API, provides the configuration of the test (e.g., the required number and range of input features). For the execution of

4. The list of addresses of the participating nodes is available (**FR017**). We provide some clarifications on this point:
    a. "Addresses" correspond to pseudo-identifiers of participants, which allows the aggregator to send messages to any specific participant (**FR019**) and to identify which participant has sent a received message (**FR022**). For security and privacy purposes, neither the actual username nor an actual physical address (e.g. IP address or name of message queue) will be revealed.
    b. Only the aggregator / task creator is allowed to access the complete list of participants. Participants are oblivious of the total number of participants or (pseudo-)identifiers of specific participants. Thus, participants are not able to send messages to arbitrary other participants, but only to the aggregator (**FR020**) or to the "next" participant according to an underlying ring topology (**FR021**).
5. The local data for Federated ML training is available to the participant's training process via a data connector (this falls within the client connectors' software environment at the interface with the algorithmic library, outside the core platform's boundary).
6. Communication between the aggregator and participants during the training occurs via send and receive functions.
    a. The send functions allow an aggregator to broadcast a message to all the participants (**FR018**) or send it to a specific participant (**FR019**) where a pseudo-identifier is used to address that participant. On the other side, they allow a participant to send a message to the aggregator (**FR020**) or to the "next" participant according to an underlying ring topology (**FR021**).
    b. The receive functions allow an aggregator to receive messages from an arbitrary participant, together with an identifier of the participant who sent it (**FR022**).[7] Moreover, they allow a

the test, the client connectors need to implement the logic for retrieving the participants' data from their respective data sources.

[7] In D4.2, the design of the prototype communication library was such that the aggregator could indicate, as an argument of the receive function, from which specific participant it was waiting to a receive a message. **FR022** capture a more asynchronous design in which messages from different participants could arrive and be processed in arbitrary order.

participant to receive messages from the aggregator (**FR023**), or from the "previous" participant according to an underlying ring topology.

Platform functionality not assumed in D4.2 but essential for robust Federated ML training, data alignment and data value estimation, includes:

- Ability for an aggregator to store task status updates (**FR025**). This may include any information that is needed by the aggregator in order to resume a Federated ML training task in case the aggregator process is interrupted.
- Ability for an aggregator to store intermediate or final versions of the trained Federated ML model (**FR026**). This may be used for resuming training in case of an interruption, and to make the final trained model available to task members.
- Ability for an aggregator to store information regarding the data value contributions per participant (**FR027**). This information may be used to determine the appropriate compensation of task members according to the value of the data that they contributed to the training.

2.6 Client connectors

For the end-to-end demonstration of the industry use cases under the MUSKETEER project, the integration of services provided by the core platform with the client connectors' software to be installed within the clients' IT premises is critical. D7.1, which documents the initial design of the client connectors, is prepared and submitted concurrently with the present document. In the following, we describe central elements of this integration and key assumptions from the core platform's perspective.

- We assume that the end user will avail of the core platform functionality from within Python runtimes. This will allow for straight-forward usage of the Python API that we are planning to develop and package in to a Python library (see Section 4 for details on the proposed design of the API). The most basic designated use would be to perform interactive operations (like user and task management, see Table 2 and Table 3) in a Python notebook, and execute the actual training logic – either in a aggregator or a participant role – within a Python script. If desirable, a graphical user interface for the interactive operations could be developed in WP7 on top of the platform Python API.

- The execution of the training will require the algorithmic library (developed under WP4) to be available in the client connectors' computational environment. We envision that the task definition information will allow the training Python scripts to dynamically initialize, configure and run the appropriate algorithms Python object (we will show an example of this intended flow in Section 4). This applies to the core Federated ML algorithms, but also includes potential data preprocessing algorithms. In order to be able to apply the latter, they also have to be available, e.g. in a Python library, in the client connectors' computational environment.
- The core platform does not provide services for deploying trained ML models for production purposes. The trained models can be downloaded and either deployed in the local computational environment of the end user, or (by the end user) in a commercial cloud environment that supports the deployment of trained ML models.

2.7 Alignment with industrial data platform standards

An important consideration in the MUSKETEER project is the alignment of the end-to-end platform with existing and emerging standards for industrial data platforms, in particular with the Industrial Data Space Association (IDSA) reference architecture. Most elements of that reference architecture pertain to the client connectors. Certification of the client connectors' software by an independent third party is not an architectural but a procedural means to support the wide application in industrial data spaces. Abstraction from specific use cases in the platform architecture to make it broadly applicable across a variety of data and machine learning model types and application domains is paramount in this regard, too.

3 Platform architecture

3.1 Overview

This section describes the architecture of the MUSKETEER platform for providing centralized services. It is the culmination of Task 3.1 – "*Design of scalable, secure, trusted and privacy aware architectures*". There are two elements to this platform: a centralized server component for managing services, and a client package for interacting with these services. Figure 2 shows a diagram of the initial version of the architecture for the centralized server component.

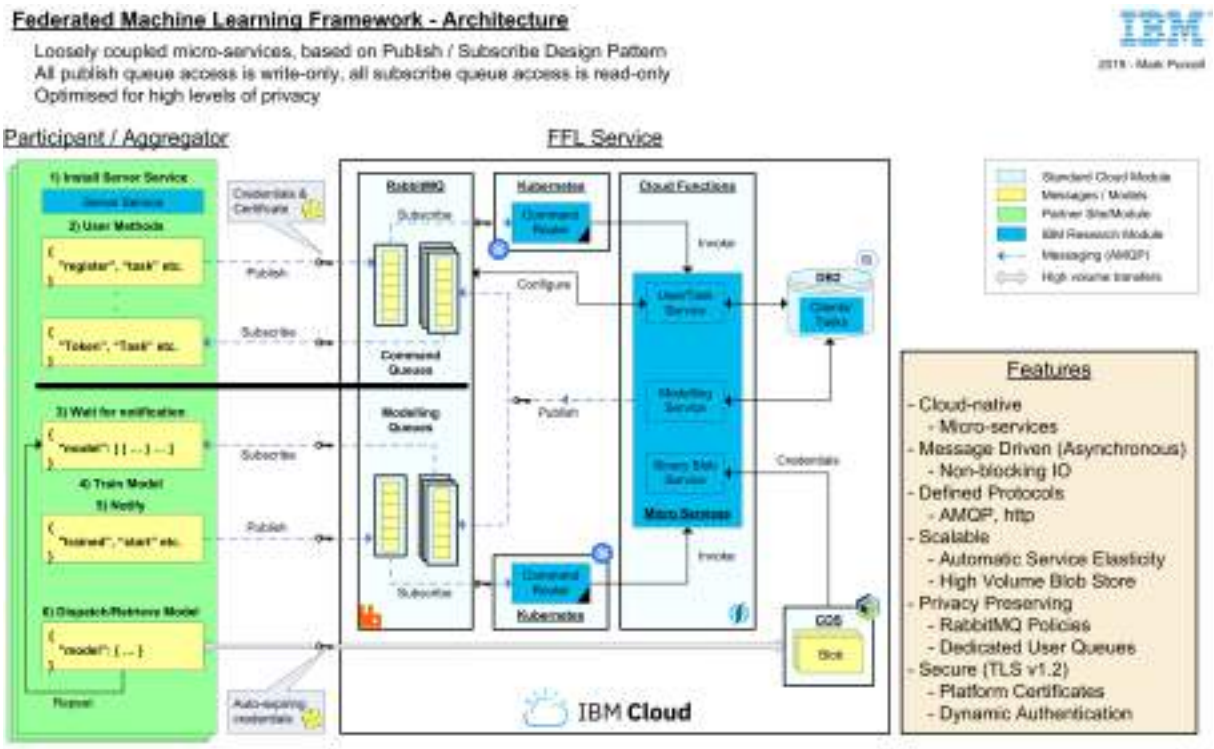

Figure 2: MUSKETEER platform architecture (initial version)

The architecture intends to utilise existing public cloud services, and as such, it is a cloud native system. Internally, it is based on a micro-services architecture [2]. The cloud infrastructure is provided by IBM, using the IBM® Cloud™ platform [3].

Micro-services are self-contained components, usually operating across a distributed system, interacting through well-defined interfaces. By adhering to these interfaces or "contracts", a given micro-service can be implemented in any runtime a developer wishes, e.g. Java, Python, NodeJS etc. For MUSKETEER, these contracts are in fact JavaScript Object Notation (JSON) based messages, and by using a message-based interface, MUSKETEER can now also use a messaging system to deliver the messages. This messaging system is based on the Publish / Subscribe Design Pattern [4], with each micro-service either subscribing to events of interest, publishing information, or both. In either case, the contents of information published or received is a JSON message.

Employing this design pattern enables asynchronous interactions between components, whereby a component can publish several messages in quick succession, and then subscribe to possibly receive replies, or messages from other sources. This asynchronous use case will be further described in the Aggregator section 3.3.2.

The messaging system used by MUSKETEER is RabbitMQ [5] and all interactions with the platform operate via this messaging system or gateway. This is instantiated as a public cloud,

internet addressable service, allowing remote clients to connect. Remote clients require appropriate credentials to connect, which will be discussed in the security section 3.3.

A second route for interactions with the platform is via the Binary service (Section 3.10). This mechanism is not directly accessible to remote parties, and in effect, is constrained by internal functionality that will be provided by the Client Package (Section 3.4) operating via the messaging gateway. The reason that this additional communication mechanism exists is to provide a scalable, high-volume data movement service, which is used for uploading and downloading potentially quite large models and model updates during the Federated ML training.

3.2 Cloud-hosted Services

The MUSKETEER architecture utilises a number of services available on the public IBM® Cloud™. Each of these will now be described.

3.2.1 IBM Cloud™ Messages for RabbitMQ

This is a fully managed instance of RabbitMQ, hosted on the public cloud. Its underlying disk, random-access memory (RAM), and optional virtual Central Processing Unit (vCPU) allocation, as well as backup storage usage are all factors in determining the price plan.

RabbitMQ [5] is open source message-queueing software. Effectively a messaging broker, it implements the Publish / Subscribe design pattern with the Advanced Message Queuing Protocol (AMQP). Clients publish messages to known RabbitMQ exchanges and queues. Subscribers listen for activity on known queues, and process the messages, which may result in some action, the result of which may also be published to an exchange/queue.

For MUSKETEER, RabbitMQ is used as the primary means of communication between both client applications and individual micro-services as well as between micro-services.

3.2.2 IBM® Db2® on Cloud

This is a fully managed SQL relational database, hosted on the public cloud, with several client runtimes supported. It is easily provisioned, with several plans available. For MUSKETEER, the Flex Plan is appropriate, whereby CPU, memory and storage resources can be scaled to match actual usage over time.

An SQL schema is deployed into this Db2 instance, and this schema details the required tables and indexes for representing users, tasks and models. An instance of this schema is called the MUSKETEER database.

3.2.3 IBM® Cloud Object Storage

Object storage, or object-based storage, is a data storage system which models the data for storage as an object. In object storage, an object consists of the data itself, metadata, and a unique identifier.

The architecture of object storage is flat, each object is stored in the same address space. This is in contrast to other storage systems such as block storage, where data is partitioned into blocks and stored in sectors, or file systems, where data is viewed as a file, or collection of files in a file hierarchy. Data stored in object storage is unstructured, and object storage places no constraints on the format of the data.

Metadata stored with the data in an object, usually takes the form of key/value pairs, is variable in size, and is generally user defined. Metadata is important as it describes the data contained in the object, without it the data is simply a sequence of bytes. Metadata can include details such as time of creation, access, revision, etc.

In order to identify the object for later retrieval, it is given an identifier, which can take any form. The only constraint on the identifier stems from the object storage flat architecture - it must be unique for each object.

3.2.4 IBM Cloud™ Functions

IBM Cloud™ Functions is an IBM Cloud™ instance of Apache OpenWhisk [6], which is a functions-as-a-service (FaaS) programming platform for developing lightweight code that scales on demand. Individual functions are billed on a per-execution scale, and the cost of an individual function execution is minimal. IBM Cloud™ Functions scales up parallel invocation requests on demand and also scales down to zero. At zero scale, the cost is also zero, which essentially means that applications pay for actual use rather that pre-determined capacity. This results in a very flexible and cost-effective platform, whereby application scaling is handled by the platform automatically in response to changes in workload.

For MUSKETEER, each individual deployment on IBM Cloud™ Functions is a MUSKETEER centralized platform micro-service.

3.2.5 IBM Cloud™ Kubernetes Service

This is a managed Kubernetes service [7] hosted on the public cloud. For the purposes of MUSKETEER, the Free Plan, which provides one cluster and one worker node is sufficient.

Kubernetes itself is an open source platform for managing containerized workloads. It is provided with declarative information in the form of YAML files and manages the state of the cluster and running containers (in so-called pods).

3.3 Security & Privacy

From inception, the MUSKETEER architecture has considered security and privacy as fundamental requirements for the platform. As the platform encompasses components running on physically different systems, some on cloud, some on premise, the overall architecture is a distributed system. Due to this, the network connections between these distributed systems use the latest available security, which, at the time of writing, is Transport Layer Security (TLS) v1.2. Connecting clients must also obtain user credentials (username/password) and a platform certificate to operate on the platform. These measures ensure that a user account is created, the legitimacy of the MUSKETEER platform messaging gateway server is established and that the contents of all network traffic over connections is encrypted.

3.3.1 User Accounts

User account management is backed by the RabbitMQ Management Console and API, and when a user registers with the platform, a centralized micro-service issues a RabbitMQ API call to create a user account on the RabbitMQ instance. There is no general mechanism to list the users registered on the system, and as such, any given user is unable to obtain the user account names of other registered users.

3.3.2 Task Aggregation/Participation

There is no direct interaction between task aggregators and any task participant. The aggregator dispatches federated models and training instructions to the centralized platform. A modelling micro-service then forwards this information to the relevant task participants. When a task participant completes a round of model training, it dispatches model updates to the centralized system. A modelling micro-service then forwards these updates to the aggregator. This separation ensures a high level of privacy and security for all users.

When a task is created, the task details are stored in the database and a RabbitMQ queue is also created for the task creator. The task creator is assumed to be the aggregator. This RabbitMQ queue is used to communicate modelling information to the aggregator, for example, the availability of model updates from task participants (via the centralized platform). Using a RabbitMQ policy, access to this queue is restricted to the task aggregator, which ensures that all model updates intended for the aggregator is a private exchange between the centralized platform and the aggregator. The aggregator is granted read-only permissions on this private queue.

When a user joins a task, a record of this activity is stored in the database and a RabbitMQ queue is created for the new task participant. This RabbitMQ queue is used to communicate modelling information to the task participant (via the centralized platform), for example, the availability of a new federated machine learning model and instructions to commence a new round of model training. Using a RabbitMQ policy, access to this queue is restricted to the task participant, which ensures that all modelling information intended for the participant is a private exchange between the participant and the centralized platform. The participant is granted read-only permissions on this private queue.

Using RabbitMQ policies, access to queues is restricted. In this way, no other user of the platform can access a queue to which they are not permitted. RabbitMQ policies ensure that queues are in effect, private queues.

In this way, by using dedicated private queues, the privacy of aggregators and participants is preserved.

3.3.3 Models

Models and model updates are communicated through the use of the Binary Service (see Section 3.10). Access to this service is managed internally through the Client Package (see Section 3.4) and is not directly visible to client applications. It is envisioned that the Binary Service will provide temporary credentials both for individual upload and download of models and model updates.

It is not intended that the contents of models or model updates stored in the Binary Service are inspected or processed in any way by components of the MUSKETEER centralized platform. This area is only used by the client package, to upload or download models or model updates, so that client applications can perform the appropriate processing.

3.4 Client Package

This is software installed locally by each potential user of the platform. It provides the necessary capabilities to interact with the MUSKETEER centralized platform. All interactions are through the MUSKETEER Messaging Gateway, meaning there is no direct invocation of the micro-services in the centralized platform.

Details of its envisioned use are discussed in Section 4. The client package will be the corner stone of the client connector's interaction with the MUSKETEER centralized cloud platform.

3.5 Messaging Gateway

All interactions between MUSKETEER clients and the centralized platform take place through the messaging gateway, which is an instance of RabbitMQ. As previously discussed, these interactions require the appropriate credentials.

There are two types of queues in the system. A single-command style queue, to which all requests for services (messages) are published. For example, if a user wishes to join a task, this request will be published to the single-command queue. All user accounts are granted write permissions on this queue, and therefore users cannot retrieve messages that were published by any user. The centralized platform is granted read-write permissions on this queue.

And secondly, multiple private read-only modelling queues, through which federated models and model updates are communicated between task aggregators and task participants. The centralized platform is granted write-only permissions to these private queues.

3.6 Command Router Service

MUSKETEER micro-services are not invoked directly by client applications, but rather, clients dispatch a message to a RabbitMQ exchange/queue. These messages are then examined, and the appropriate action taken to respond to the service request specified by the message.

The command router is the service which performs this action. It subscribes to the RabbitMQ single-command queue, receives messages, and determines which IBM Cloud™ Function (micro-service) should be invoked to handle each message received. As this service must promptly handle messages received, it in effect, must be an always-on service, with high availability. Therefore, it is intended that this service runs in a long-lived Kubernetes pod. If this pod exits for any reason it must be restarted as soon as possible. This will ensure that the latency between clients dispatching a message, and the appropriate micro-service handling the message is minimised.

This service was previously researched and developed as part of the GOFLEX H2020 project [8], and during that project, an open source contribution was made to the IBM Cloud™ Functions public github organisation, with a project called RabbitWhisker [9]. It is implemented as a multi-threaded Python application, allowing it to receive and route large numbers of concurrent service requests.

3.7 User Management Service

This is a micro-service based on IBM Cloud™ Functions which provides user account services through the RabbitMQ API and records user details in the database.

The service supports the following actions:

1. User registration: parameters - *username*, *password*
    a. Ensure *username* and *password* are non-empty strings
    b. Ensure that the *username* is unique
    c. Create a user account on the RabbitMQ instance
    d. Grant permission to *username* to the single-command queue
    e. Create a user entry for *username* in the database
    f. Ensure that collectively *c-d-e* above is an atomic operation
2. User removal: parameters - *username*
    a. Leave all tasks that *username* has previously joined
    b. Remove *username* from the RabbitMQ instance
    c. Remove the user entry for *username* from the database

3.8 Task Management Service

This is a micro-service based on IBM Cloud™ Functions which provides machine learning task management services through the RabbitMQ API and records task details in the database.

The service supports the following actions:

1. Task Create: parameters - task name, username, topology, definition
    a. Ensure *task name* and *username* are non-empty strings
    b. Create a queue for the task on the RabbitMQ instance
    c. Create a task entry for *task name* in the database
    d. Ensure that collectively *b-c* are atomic

2. Task List: parameters – None
    a. Retrieve the task entries from the database
3. Task Info: parameters – *task name, username*
    a. Retrieve the task entry for *task name* from the database
4. Task Participation: parameters – *task name, username*
    a. Retrieve the user entries for *task name* from the database
5. Task Start: parameters – *task name, username, model*
    a. Ensure *task name* has user participants
    b. Change the status of the task entry for *task name*
    c. Invoke Modelling Service – *Notify Participants* with *start, model*
6. Task Stop: parameters – *task name, username, task status*
    a. Ensure *task name* has user participants
    b. Invoke Modelling Service – *Notify Participants* with *stop*
    c. Change the status of the task entry for *task name*
    d. Remove the queue for the task on the RabbitMQ instance
7. Task Join: parameters – *task name, username*
    a. Ensure that *username* can participate in *task name*
    b. Create a queue for the user/task on the RabbitMQ instance
    c. Create a user entry for the task in the database
    d. Invoke Modelling Service – *Notify Aggregator*
    e. Ensure that collectively *b-c-d* are atomic
8. Task Leave: parameters – *task name, username*
    a. Ensure that *username* participates in *task name*
    b. Remove the queue for the user/task on the RabbitMQ instance
    c. Remove the user entry for the task in the database
    d. Invoke Modelling Service – *Notify Aggregator*
9. Task Update: parameters – *task name, username, status, model*
    a. Ensure that *username* participates in *task name*
    b. Update the user entry for the task in the database with *status*
    c. Invoke Modelling Service – *Notify Aggregator*

3.9 Modelling Service

This is a micro-service based on IBM Cloud™ Functions which provides the queue management services to handle the interactions between task aggregators and participants. As discussed in section 3.3.2, there is no direct interaction between task aggregators and any task participant, but rather the centralized platform routes the required information to the appropriate private queue.

The service supports the following actions:

1. Notify Participants: parameters - *task name, action, model*
    a. Ensure *task name* has user participants
    b. Retrieve the user entries for *task name* from the database
    c. For each user entry:
        i. Publish *action/model* to the user's private queue
2. Notify Aggregator: parameters - *task name, username, status, model*
    a. Ensure *task name* has user participants
    b. Ensure that *username* participates in *task name*
    c. Retrieve the aggregator entry for *task name* from the database
    d. Publish *status/model* to the aggregator's private queue

3.10 Binary Storage Service

This is a micro-service based on IBM Cloud™ Functions and the IBM® Cloud Object Storage Service. As discussed in section 3.3.3 access to this service is not directly available to client applications. It is however closely linked with, and invoked by, the Task Start and Task Update functions in the Task Management Service.

The IBM® Cloud Object Storage API provides a representational state transfer (REST) based API for reading and writing objects and supports a subset of the S3 API [10]. For MUSKETEER binary object storage (models), a means to upload and download these objects is required. It is envisaged that the S3 API's for creating pre-signed uniform resource locators (URLs) will be used to support this. These URLs will automatically expire after a pre-defined period, and it is intended that they should be used as soon as possible, hence the close integration with the calling functions.

The service supports the following actions:

1. Uploader: parameters - *username, object_name*

a. Ensure *username* has appropriate permissions
b. Generate an object name if *object name* is empty
c. Call *s3.generate_presigned_post* with *object name*
d. Return the URL generated

2. Uploader: parameters - *username, object name*
   a. Ensure *username* has appropriate permissions
   b. Call *s3.generate_presigned_url* with *object name*
   c. Return the URL generated

4 Example: proposed usage

This section introduces an end-to-end example for the envisioned usage of the MUSKETEER platform services. We first describe an example context motivating the utilization of the platform. Then we will illustrate, step by step, the envisioned useage of the platform to achieve the goal outlined in the motivation.

4.1 Motivation

Alice has a machine learning task for which she would like to train a model, but she has not data for the task. Therefore, she would like to harness the MUSKETEER platform to leverage training data provided by other parties. On the other hand, John and Jack possess available data that may be useful for Alice's task. Together, they can use the MUSKETEER platform to collaboratively train a machine learning in a federated fashion, without having to share or centralized the actual data. Thereby, they will be able to unlock additional value of their data and all benefit from the creating and training of the machine learning model.

4.1.1 Detailed steps

In order to use the MUSKETEER platform, Alice, John and Jack first must register their own user account (Figure 3). Once accounts are created, all subsequent interactions with the platform must be performed using those accounts.

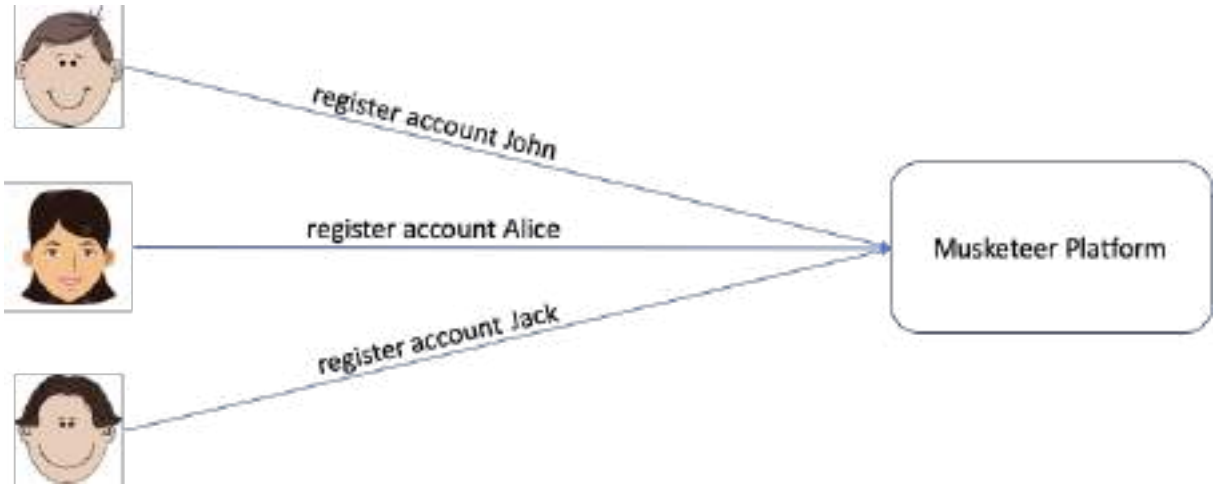

Figure 3: Account registration on the MUSKETEER platform

After registering with the platform, Alice will create a machine learning task and register that task with MUSKETEER so that it is stored in MUSKETEER's database. The task creation process will require Alice to define the machine learning task in detail as shown in Figure 4. The task definition may contain information such as the number of participants, number of training epochs, batch sizes, learning rates, etc. Upon successful creation, the task will be assigned with a name ("Task005" in this example). In the following, Alice will be playing the role of a task creator in the MUSKETEER platform.

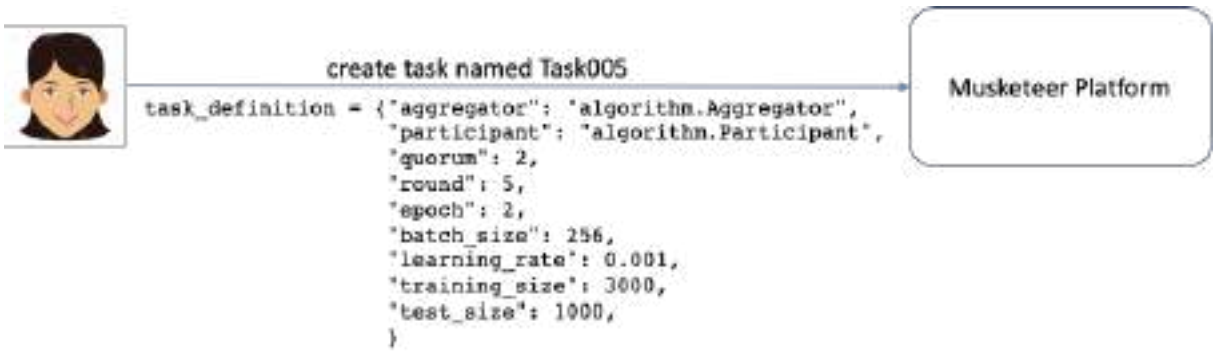

Figure 4: Create Federated ML task on the MUSKETEER platform

As task creator, Alice will also execute the aggregator side of the federated training process in her computational environment. In this example, the training quorum is 2 (see Figure 4), i.e. exactly two participants need to have joined the task before the training begins.

In parallel, John and Jack can avail of the MUSKETEER platform services to explore Federated ML tasks created by other users, including the task that was created by Alice. By inspecting the definitions of the task (and available meta descriptions), they can decide whether their available data may benefit a task and whether they want join that task (Figure 5).

Figure 5: List tasks on the MUSKETEER platform

Once they make the decision to participate in a specific task (typically independently and unbeknownst of each other), they can avail of the platform services to join that task (Figure 6) and assuming the role of task participants in the following.

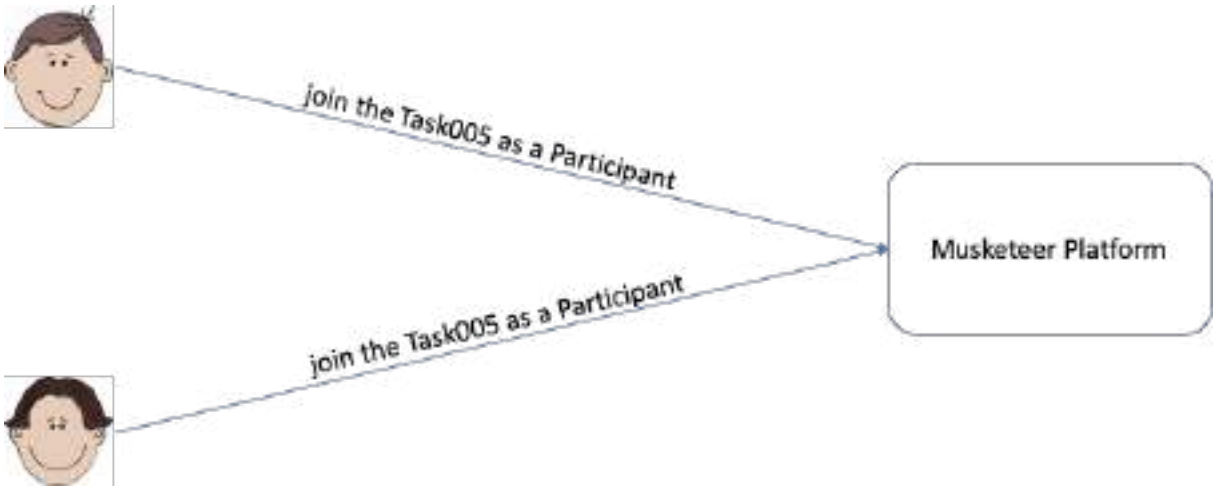

Figure 6: Join task on the MUSKETEER platform

With two participants having joined, the starting criterium of Alice's task has been satisfied, and so the training process of the machine learning task can start and run throughout the number of iterations specified in the task definition. The exact flow of information among participants and aggregators during the training process depends on the specific Privacy

Operation Mode (POM); in the following, we provide an example of standard Federated ML training as defined in POM1 (see Section 2.5.1).

Firstly, the aggregator sends an initial version of the ML model to both participants. This can be done via a broadcast function in order to save communication costs as illustrated in Figure 7. Without a broadcast function, the aggregator would have to send the same model repeatedly to the platform (once per participant) which will then relay the model to the specific designated sender. In practice, a model update could size up to tens of gigabytes, and therefore transferring such a large model several times through a cloud network would consume a lot of bandwidth. After broadcasting the model, the aggregator waits for incoming model updates from each of the participants.

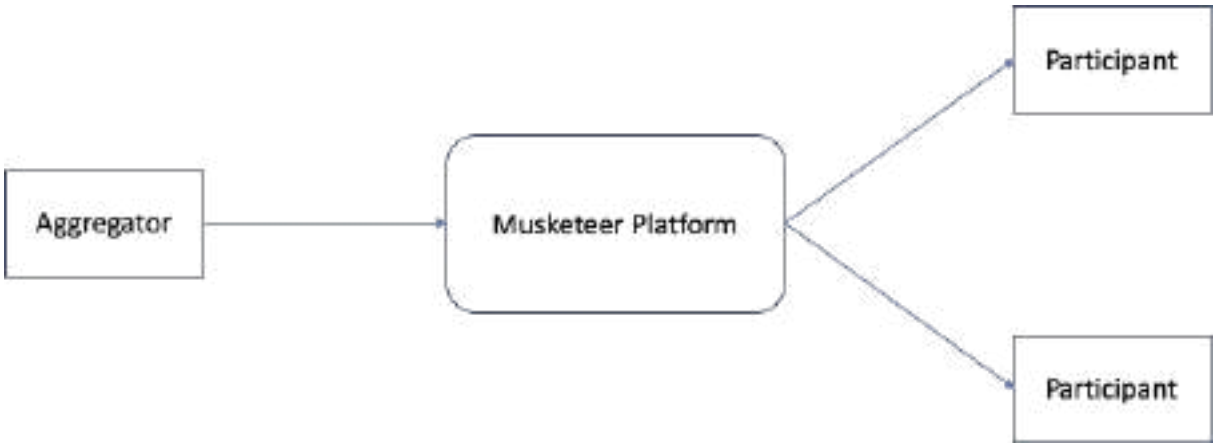

Figure 7: Communication from the aggregator to task participants

Secondly, each participant – after receiving the model from the aggregator – will update its local model, continue to train the model locally with their local data and obtain a new local model update. Then, this local model update will be transferred back to the aggregator through the MUSKETEER platform as shown in Figure 8. The aggregator will then collect these new model updates from all the participants, average them to produce a new model update, which then is broadcasted again to the participants for the next iteration. After a specified number of such iterations, the training will end and the aggregator will obtain a final version of the trained model, thus completing the Federated ML task created by Alice.

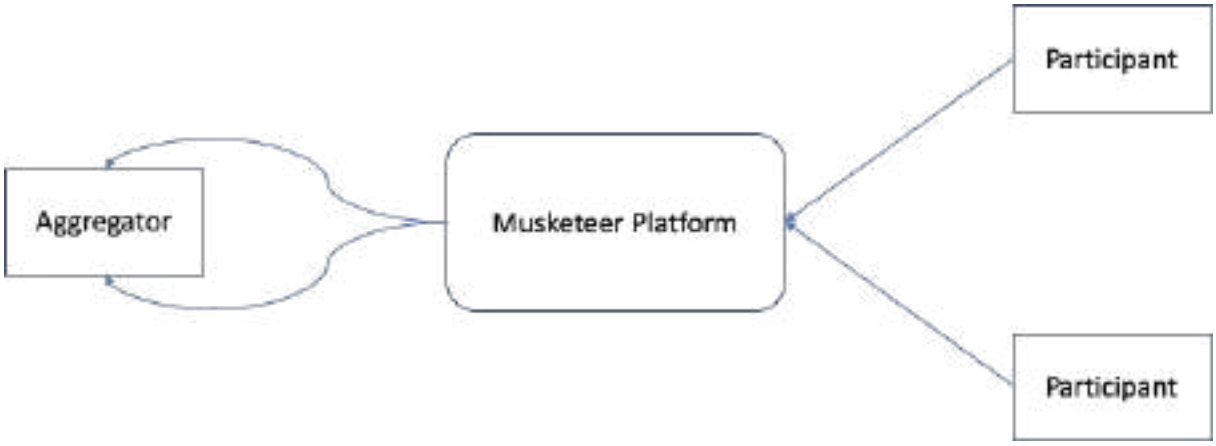

Figure 8: Communication from the task participants to the aggregator

5 Possible future extensions

To conclude this document, we give an outlook on possible future extensions of the platform. An analysis and consolidated view on the technical requirements stemming from these extensions will be provided in the documentation of the final version of the MUSKETEER platform architecture (D3.2).

Explore synergies and possible integration points with the AI4EU platform

As discussed in Section 2.2.2, the initial set of technical requirements provided in D2.1 envisioned the ability for users of the MUSKETEER platform to provide their own user profiles, explore the profiles of other users and, in conjunction with those profiles, share information about datasets that they own which could be leveraged for Federated ML tasks. We do not see such platform capabilities as central to the scope of the MUSKETEER project; moreover there may be potential legal/ethical implications related to storing this sort of information. However, we believe there may be synergies and possible integration points with the AI4EU platform [11] that are worthwhile exploring, perhaps less from a technical or architectural viewpoint, but certainly from a dissemination and exploitation perspective in this project.

Organizing platform user access permissions in groups

Another element of D2.1 was the envisioned platform capability to create groups of users in order to control, e.g., which users could see or join specific tasks. In the first version of the platform, this capability isn't integrated; as explained in Section 2.2.2, the initial version of the platform allows users to see or join any available tasks; we envision that different platforms

will be instantiated for the real-world use cases and for algorithmic performance assessment, thus this single-tenancy deployment mitigates the need for restricting access to tasks. In preparation of the final version of the MUSKETEER platform architecture (D3.2) we will conduct further analysis to understand the user stories and derive more precise technical requirements.

Permissions for downloading models

Currently, the platform does not implement a capability for persisting ML models that result from Federated ML tasks, but this is a feature that should be supported by the final version. In accordance with that, appropriate permissions should be defined as to who is allowed to retrieve the models persisted in the platform. In preparation of D3.2, we are planning to describe different options in more detail in order to arrive at an informed decision which mechanisms to implement; two possible approaches are that either only the task creator has access to the final model (restrictive), or that the task creator and all participants can download it (permissive).

Model serialization

A question related to the persistence and downstream use of trained Federated ML models is how the format that should be used for serializing and storing the actual models. One option would be to export and save the models in a standard format; another option would be to save them in framework-specific formats with meta information that is required to properly re-load and apply them. Decisions on the exact format should be made in conjunction with WP7 as the client connectors will ultimately provide the environment in which end users will retrieve the outcomes from Federated ML tasks.

Task lifecycles

Working towards the final version of the MUSKETEER platform, a further analysis needs to be performed in order to understand the full lifecycle of Federated ML tasks, how their status should be represented in the platform, and how the status informs different actions that can be performed on tasks. This is particularly important for handling, e.g. participants which temporarily disconnect from their training tasks, and aggregators temporarily disconnecting or even crashing. To handle such scenarios gracefully, checkpointing of Federated ML tasks maybe a required mechanism to be supported by the platform. A related question is how to allow users to monitor the status and execution of the Federated ML training. Finally, decisions need to be taken and mechanisms need to be implemented in order to control how long

task-related message queues should be kept up and running, and how long to persist task metadata, trained models etc.

User roles

Currently, the platform assumes that the creator of a task will also be taking the role of the aggregator during the actual training. In preparation of D3.2, it should be reviewed – in conjunction with WP7 – whether other cases could be foreseen (e.g. where the task creator would also act as training participant) and should be supported by the platform.

Data value estimation

The ability of the MUSKETEER platform to estimate the value of the data contributed by the different participants of a Federated ML task is a key functionality in order to unlock new value in the data economy. From the platform perspective, a further analysis needs to be conducted in order to understand which services should be provided (i) in order to allow algorithms for data value estimation to persist information about contributions from different users in the platform; (ii) in order to allow platform users to query, analyze and further process this information in order to determine potential rewards to the different participants.

Encryption / key management

Finally, in order to support Privacy Operation Modes (POMs) that rely on encryption mechanisms, the overall architecture of the MUSKETEER project needs to consider, e.g. how, where and by who encryption keys are going to be generated and managed, and how the services for key generation / management will interact with the core platform, the algorithmic library, and the client connectors.

6 References

[1] R. van Solingen, E. Berghout (1999). The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development, McGraw-Hill.
[2] S. Newman (2015). Building Microservices – Designing Fined-Grained Systems, O' Reilly.
[3] https://cloud.ibm.com/
[4] S. Tarkoma (2012). Publish/Subscribe Systems: Design and Principles, John Wiley & Sons, Ltd.
[5] https://www.rabbitmq.com/
[6] https://openwhisk.apache.org/
[7] https://kubernetes.io/

[8] https://www.goflex-project.eu/
[9] https://github.com/ibm-functions/package-rabbitmq
[10] https://docs.aws.amazon.com/s3/index.html
[11] https://www.ai4eu.eu/