

H2020 – ICT-13-2018-2019



**Machine Learning to Augment Shared Knowledge in  
Federated Privacy-Preserving Scenarios (MUSKETEER)  
Grant No 824988**

**D3.3 First prototype of the MUSKETEER  
platform**

## Imprint

**Contractual Date of Delivery to the EC:** 31 May 2020

**Author(s):** Mathieu Sinn (IBM), Mark Purcell (IBM), Minh Ngoc Tran (IBM), Susanna Bonura (ENG)  
**Participant(s):** TREE; IMP; ENG; UC3M; IDSA; COMAU; FCA  
**Reviewer(s):** Antoine Garnier (IDSA), Marcos Fernández (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](mailto:mathsinn@ie.ibm.com)  
**Website:** [www.MUSKETEER.eu](http://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

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

## Executive Summary

In this document we provide a report which describes the demonstration of a first prototype of the MUSKETEER platform. The demonstration involves the end-to-end execution of data sharing and federated machine learning on synthetic data and one real-world use case. It supports different privacy operating modes and uses a basic dashboard to support user interactions with the platform.

## Document History

Version	Date	Status	Author	Comment
1	08 May 2020	First version for internal review	Mathieu Sinn	First draft
2	15 May 2020	Final version for internal review	Mathieu Sinn	Draft for review
3		Review inputs	Antoine Garnier	Update
4		Review inputs	Marcos Fernández	Update
5		Final Version		Update
6	23 May 2020	Clean and submission	Gal Weiss	Final

## Table of Contents

<b>LIST OF FIGURES .....</b>	<b>4</b>
<b>LIST OF TABLES .....</b>	<b>4</b>
<b>LIST OF ACRONYMS AND ABBREVIATIONS .....</b>	<b>6</b>
<b>1 INTRODUCTION .....</b>	<b>7</b>
1.1 Background .....	7
1.2 Federated learning .....	8
1.3 Related documents .....	8
1.4 Outline.....	8
<b>2 PROTOTYPE COMPONENTS .....</b>	<b>9</b>
2.1 Cloud platform.....	9
2.2 Local platform.....	10
2.3 Platform APIs.....	11
2.4 Client package .....	11
<b>3 EXAMPLES OF END-TO-END EXECUTIONS.....</b>	<b>12</b>
3.1 Motivation.....	12
3.2 Installation instructions.....	15
3.3 Synthetic data .....	17
3.3.1 Terminal windows.....	17
3.3.2 Jupyter notebooks.....	23
3.3.3 Graphical user interface mock-up .....	28
3.4 <b>Use case: Smart manufacturing</b> .....	<b>32</b>
3.4.1 Data Science workflow .....	32
3.4.2 Integration with the MUSKETEER platform.....	36
<b>4 CONCLUSIONS .....</b>	<b>39</b>
<b>5 REFERENCES.....</b>	<b>39</b>

## List of Figures

Figure 1: MUSKETEER’s PERT diagram.....	7
Figure 2: MUSKETEER platform architecture (final version).....	10
Figure 3: User registration on the MUSKETEER platform.....	12
Figure 4: Create Federated ML task on the MUSKETEER platform.....	13
Figure 5: List tasks on the MUSKETEER platform.....	13
Figure 6: Join task on the MUSKETEER platform.....	14
Figure 7: Communication from the aggregator to task participants.....	14
Figure 8: Communication from the task participants to the aggregator.....	15
Figure 9: Registering the aggregator user via terminal.....	18
Figure 10: Registering the participant-1 user via terminal.....	18
Figure 11: Registering the participant-2 user via terminal.....	19
Figure 12: Creating a federated learning task via terminal.....	19
Figure 13: Starting the aggregator process.....	20
Figure 14: Listing federated learning tasks.....	21
Figure 15: Joining a task as participant.....	21
Figure 16: Performing the federated learning on the aggregator side (start).....	21
Figure 17: Performing the federated learning on the participant side (start).....	22
Figure 18: Termination of the federated learning on the aggregator side.....	22
Figure 19: Termination of the federated learning on the participant 1 side.....	23
Figure 20: Termination of the federated learning on the participant 2 side.....	23
Figure 21: Jupyter notebook tree view.....	24
Figure 22: Demonstrator notebooks.....	24
Figure 23: Task creator notebook - loading prerequisites.....	24
Figure 24: Task creator notebook - registering user.....	25
Figure 25: Task creator notebook - listing existing tasks.....	25
Figure 26: Task creator notebook - creating a new task.....	26
Figure 27: Task creator notebook - aggregator process (start).....	26
Figure 28: Task participant notebook - loading prerequisites.....	27
Figure 29: Task participant notebook – registering user.....	27
Figure 30: Task participant notebook - listing tasks.....	27
Figure 31: Task participant notebook - joining a task.....	28
Figure 32: Task participant notebook - participant process (start).....	28
Figure 33: Graphical user interface - login page.....	29
Figure 34: Graphical user interface - user registration.....	29
Figure 35: Graphical user interface - task listing.....	30
Figure 36: Graphical user interface - task creation.....	30
Figure 37: Graphical user interface - task operations.....	31
Figure 38: Graphical user interface - task execution settings.....	31
Figure 39: Graphical user interface - task result.....	32
Figure 40: Use case task creator notebook – task definition.....	37
Figure 41: Use case task participant notebook – training execution (start).....	38

## List of Tables

Table 1: Confusion matrix when training on FPS2 and testing on FPS2.....	34
Table 2: Confusion matrix when training on FPS2 and testing on FPD2.....	34

Table 3: Confusion matrix when training on FPS2 and testing on FPS2 + FPD2 ..... 35  
Table 4: Confusion matrix when training on FPD2 and testing on FPS2 ..... 35  
Table 5: Confusion matrix when training on FPD2 and testing on FPD2 ..... 35  
Table 6: Confusion matrix when training on FPD2 and testing on FPS2 + FPD2 ..... 35  
Table 7: Confusion matrix when training on FPS2 + FPD2 and testing on FPS2 ..... 35  
Table 8: Confusion matrix when training on FPD2 + FPS2 and testing on FPD2 ..... 35  
Table 9: Confusion matrix when training on FPS2 + FPD2 and testing on FPS2 + FPD2 ..... 36  
Table 10: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPS2 ..... 38  
Table 11: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPD2 ..... 39  
Table 12: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPS2 +  
FPD2..... 39

## List of Acronyms and Abbreviations

<b>Abbreviation</b>	<b>Definition</b>
<b>AMQP</b>	Advanced Message Queuing Protocol
<b>API</b>	Application Programming Interface
<b>CNN</b>	Convolutional Neural Network
<b>HTTP</b>	HyperText Transfer Protocol
<b>ML</b>	Machine Learning
<b>POM</b>	Privacy Operation Mode
<b>VM</b>	Virtual Machine
<b>WP</b>	Work Package

# 1 Introduction

## 1.1 Background

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 secure and privacy-preserving federated ML algorithms developed in WP4 and WP5 (see Figure 1 for an overview of the MUSKETEER work packages) 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.

In this document we provide a report which describes the demonstration of a first prototype of the MUSKETEER platform. The purpose of this document is to give readers a view on the main interactions with the MUSKETEER platform in order to perform end-to-end execution of data sharing and federated machine learning; the workflow is demonstrated on synthetic data and one real-world use case. Moreover the document provides instructions for setting up local instances and connectors to the platform, so that the reader can set up their own environment for experimenting with the platform functionality. Finally, the document presents a set of high-fidelity mock-ups of the platform’s user interface as a result of the task T3.5 - Interfaces: development of front-end / dashboards for standard reporting, showing the user interaction with the MUSKETEER server through the client connector.

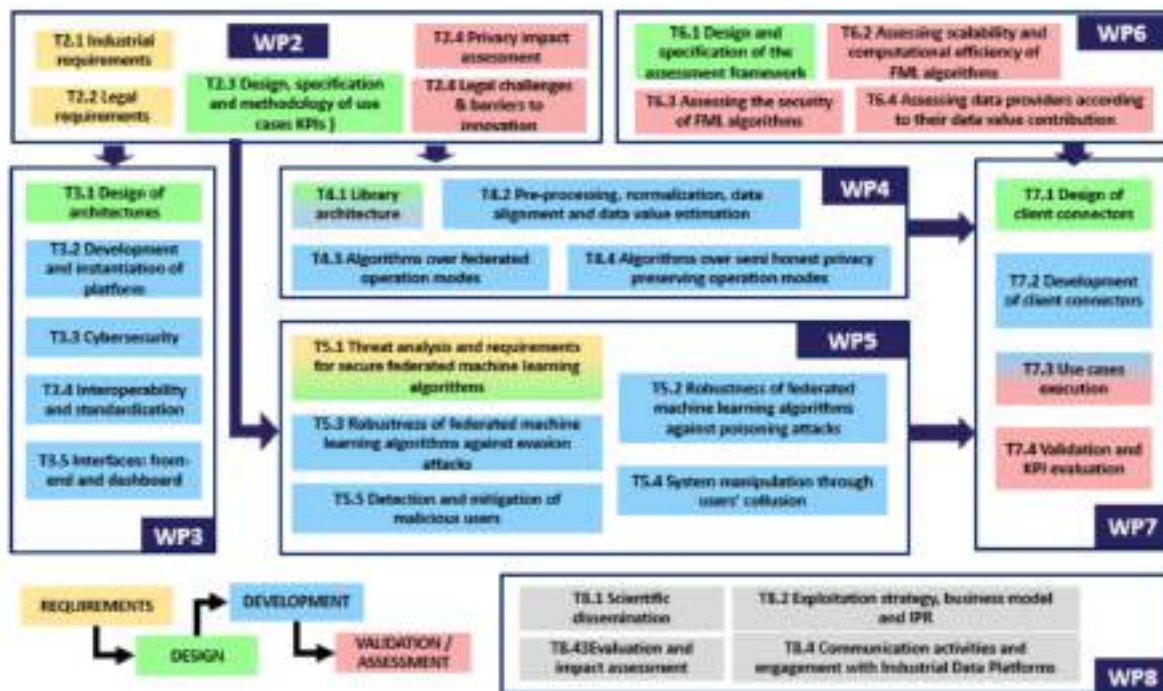


Figure 1: MUSKETEER’s PERT diagram



## 1.2 Federated learning

Here we briefly recapitulate the key concepts of federated ML. For a more detailed review, we refer to D4.1. The goal of federated ML is to create a ML model, leveraging distributed data sets without having to centralize those. In the MUSKETEER project, federated ML is extended to support different **Privacy Operation Modes** (POMs), which control the amount of information that the data owners share during the model training and validation process. In POMs 1-3 (which closely follow conventional federated ML protocols), the model training is coordinated by a central instance, called **aggregator**, while the data owners act as **participants**. Model training is typically performed iteratively throughout a number of **rounds** which is either determined a priori, or dynamically, e.g. by considering a model convergence criterion. In each round, the aggregator dispatches the current central version of the model to all the participants. The participants then compute updates to that model based on their local data, and send the updates back to the aggregator. Model updates can either be in the form of **gradients**, or in the form of new versions of the model. Upon having received the updates from all participants, the aggregator incorporates them (e.g. by taking an average of all the updates) into the new version of the central model. After the training rounds have completed, the aggregator holds the final version of the model (which under certain POMs may be encrypted), which can then be centrally stored for later use and/or deployed by the participants in their local production environments.

## 1.3 Related documents

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

- **D3.1** and **D3.2**, describing the initial and final version of the MUSKETEER platform architecture, respectively. We will refer to these deliverables for detailed documentation of the technical requirements which drove the development of the platform, background on the cloud-based architecture, and a complete documentation of the Application Programming Interfaces (APIs) for interacting with the platform.
- **D2.1**, describing the industrial and technical requirements for the MUSKETEER platform. We will refer to this deliverable for more background on the Smart Manufacturing use case which will be part of the demonstration described here.
- **D4.1** and **D4.2**, describing and demonstrating the types of federated ML algorithms to be developed during the project and to be supported by the MUSKETEER platform.
- **D7.1**, describing the initial version of the client connectors' architecture design. We will refer to this deliverable for more information about the envisioned packaging and deployment of the MUSKETEER platform client connectors in end users' computational environments.

## 1.4 Outline

The remainder of this document is structured as follows:

- In Section 2 we describe the components of the first prototype of the platform: the cloud platform, the alternative local platform, the APIs for interacting with the platform, the simple placeholder prototype for

client connectors (the full client connectors will be developed in WP7), and the mock-ups that were produced along with a brief description of the depicted functionalities and the user's interactions with the platform. We note that the local platform, the APIs and the client connectors prototype are all available open source under an Apache 2.0 license [1][2].

- Section 3 provides a walk-through of the two exemplary demonstrations on synthetic and real-world data; this section also recapitulates the steps for creating a local compute environment in which the reader can experiment with the open source software and the platform functionality.
- In Section 4 we provide conclusions and an outlook on future work towards the final prototype of the MUSKETEER platform.

## 2 Prototype components

### 2.1 Cloud platform

The MUSKETEER cloud platform is the central component enabling the creation and execution of data sharing and federated machine learning tasks among geographically dispersed participants. A diagram of the platform architecture is shown in Figure 2.

For detailed information we refer to deliverable D3.2. Here we only note at a high level:

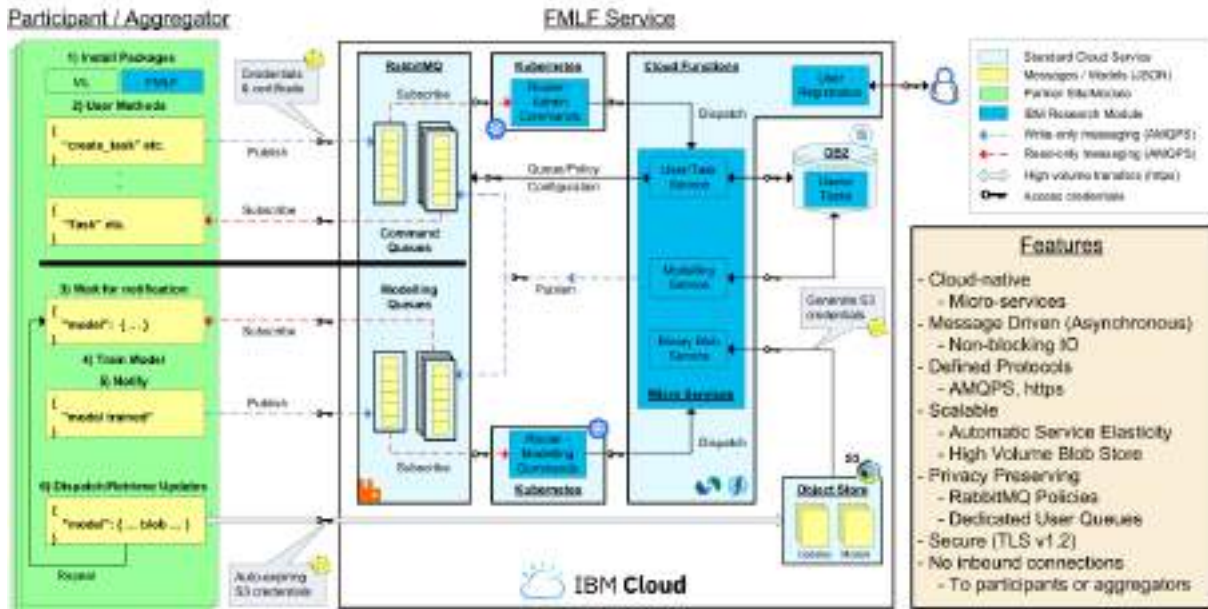
- The cloud platform uses message queues for asynchronous exchange of information required for federated learning, such as the latest version of the central model computed by the aggregator, or model updates computed by the participants on their local data. The platform itself is agnostic to the semantics of this information (generally it will not even be aware whether or not the information is encrypted); it is parsed and interpreted in the context of the federated learning algorithm processes running on the aggregator and participants' sides, respectively.
- Besides the exchange of information for the execution of the actual federated learning tasks, the platform also provides services to manage tasks throughout their lifecycle, such as: creating new tasks, browsing created tasks, joining tasks as a participant, or deleting tasks. The meta information that is required for task management is stored in a cloud database.

For the duration of the MUSKETEER project, four instances of the cloud platform have been instantiated, running in the IBM® Cloud™ [3] hosted in Frankfurt, Germany. One instance is used for continuous integration and testing of new platform features (within WP3), one instance for the development and testing of federated learning algorithms (within WP4, WP5, WP6), and two dedicated instances to support the real-world use cases in Smart Manufacturing and Healthcare, respectively (within WP7).

As we will see in the end-to-end walk-through in Section 3, credentials provided by IBM are required to avail of the cloud platform's services.

**Federated Machine Learning Framework - Architecture**

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



**Figure 2: MUSKETEER platform architecture (final version)**

## 2.2 Local platform

As an alternative to the cloud platform, we have also developed, in WP3, a local platform for enabling federated learning. The purpose of the local platform is to enable lightweight local development and experimentation with federated learning algorithms. The designated use case is when the aggregator and participants' compute processes are all running within the same local network, e.g. on the same laptop or within the same compute cluster. Thus, the local platform does not support the execution of federated learning algorithms in real-world scenarios where participants are geographically dispersed or hosted in separate compute environments.

The local platform provides advantages for the rapid development, testing and performance evaluation of federated learning algorithms. In particular, it doesn't incur the communication overhead of transmitting information between federated learning aggregators and participants via the central communication services hosted in the IBM® Cloud™. Another purpose of the local platform is to allow researchers to experiment with the federated learning algorithms developed in WP4 and WP5 without having to rely on the cloud platform or requiring credentials to access it.

The local platform is implemented using the Python Flask framework [4], where a lightweight webserver is deployed on the local compute host, and the aggregator and participants of the federated ML task exchange information via HTTP requests to that server. The implementation of the local platform is released open source in [1] under an Apache 2.0 license.

## 2.3 Platform APIs

The cloud platform exposes Application Programming Interfaces (APIs) allowing algorithm and web developers to create federated ML algorithms and user interfaces for end users leveraging the platform's functionality. At a low level, the APIs use the Advanced Message Queuing Protocol (AMQP) for communicating with the platform over the internet (see D3.2 for more details). To facilitate rapid development and to abstract from details of the message protocol, a higher-level Python API has been developed and open sourced under WP3 [2]. It provides Python function calls, e.g., to create new federated ML tasks in the platform, browse available tasks, join tasks, and execute actual training algorithms as aggregator or participant. The API is designed to work both with cloud and local platform deployments. This way, for example, algorithm developers can use a local platform deployment for development and testing, and then their algorithms can be used with the cloud platform for real-world deployments. A complete documentation of this API is included in D3.2.

## 2.4 Client package

As last component of the demonstrator, a light-weight client package has been developed in WP3 and – same as the platform Python API – released open source under an Apache 2.0 license [1]. We note that the final, full-scale client package for MUSKETEER is to be developed under WP7; the client package demonstrated in this deliverable only provides minimum functionality and serves only as a basic example of how the MUSKETEER platform enables end-to-end data sharing and federated learning workflows. The client package contains a simple federated learning algorithm for training a Convolutional Neural Network (CNN) classifier implemented in Keras [5]. The final WP7 client package will contain the full suite of algorithms developed in WP4; again, the package demonstrated in this deliverable only serves as a simple working example.

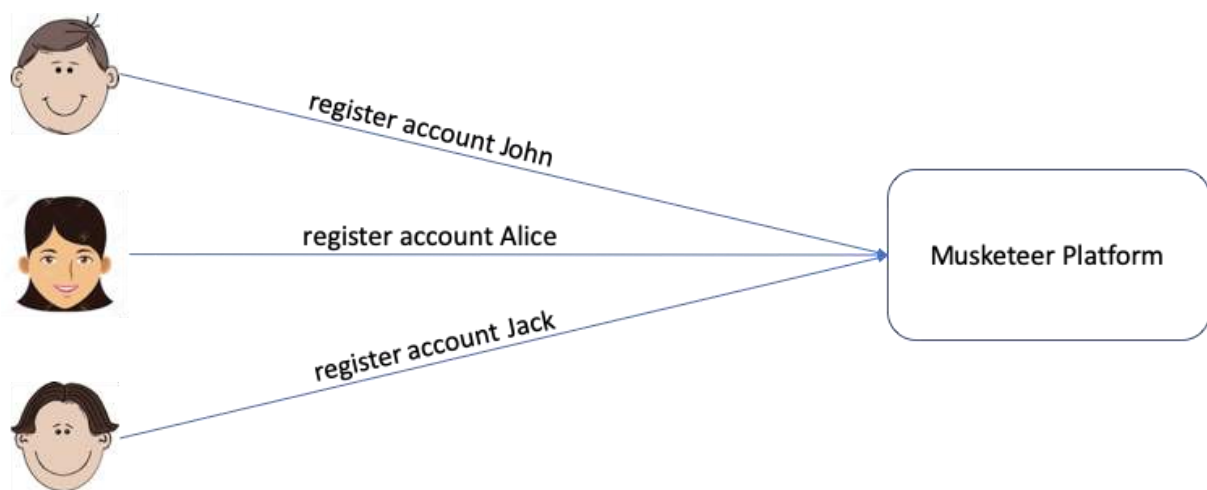
### 3 Examples of end-to-end executions

In this section we present actual examples demonstrating the end-to-end execution of data sharing and federated learning via the MUSKETEER platform. In Section 3.1, we first explain the workflow at a high-level. Section 3.2 provides instructions for setting up local compute environments to execute the demonstrator; finally, Section 3.3 and 3.4 provide detailed walk-throughs of the demonstrations on synthetic data and on real-world data from the Smart Manufacturing use case, respectively.

#### 3.1 Motivation

As a motivation, and to explain the basic interactions with the MUSKETEER platform at a high level, we provide a fictitious example. Consider Alice, a fictitious Data Scientist in a fictitious organization A, who has an ML task for which she would like to train a model, but no training data available within her organization. Therefore, she would like to harness the MUSKETEER platform to leverage training data owned by other parties. John and Jack, from organizations B and C, have access to local data that may be valuable for Alice’s task. Together, they can use the MUSKETEER platform to collaboratively train a ML model in a federated fashion, without having to share or centralize the actual data. Thereby, they will be able to unlock additional value of their data, and all will benefit from creating and training of the ML model.

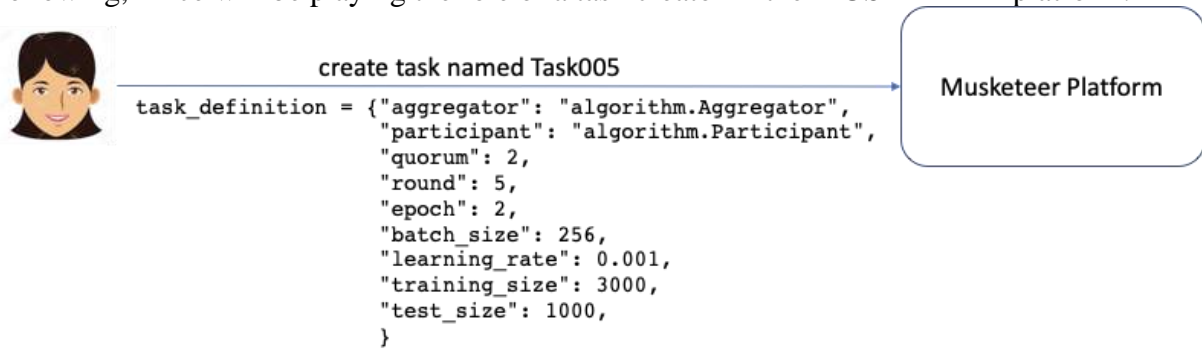
In order to use the MUSKETEER platform, Alice, John and Jack first must register their own respective user names and password (Figure 3). Once registered, all their subsequent interactions with the platform will utilize those credentials.



**Figure 3: User registration on the MUSKETEER platform**

After registering with the platform, Alice will create a federated ML task in the MUSKETEER platform. The task creation process requires 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, the number of training epochs and rounds, and other common parameters of ML model training such as batch sizes, learning rates, etc. Upon successful

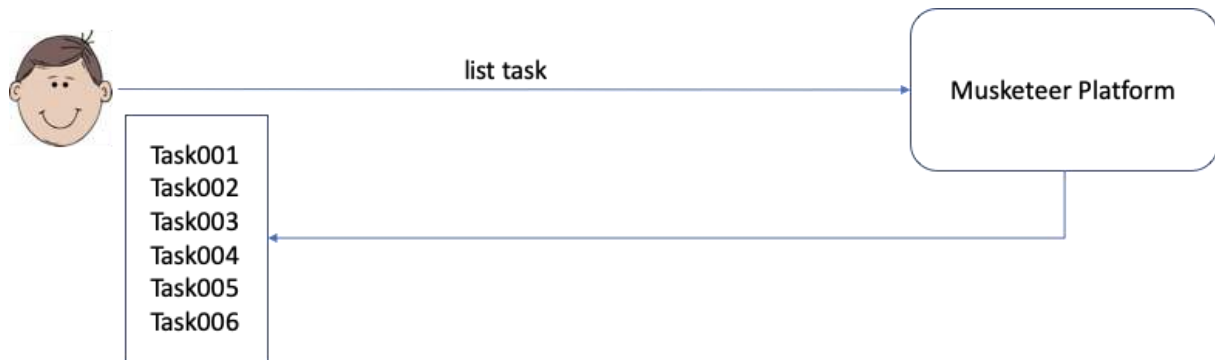
creation, the task will be assigned with a unique name (“Task005” in this example). In the following, Alice will be playing the role of a task creator in the MUSKETEER platform.



**Figure 4: Create Federated ML task on the MUSKETEER platform**

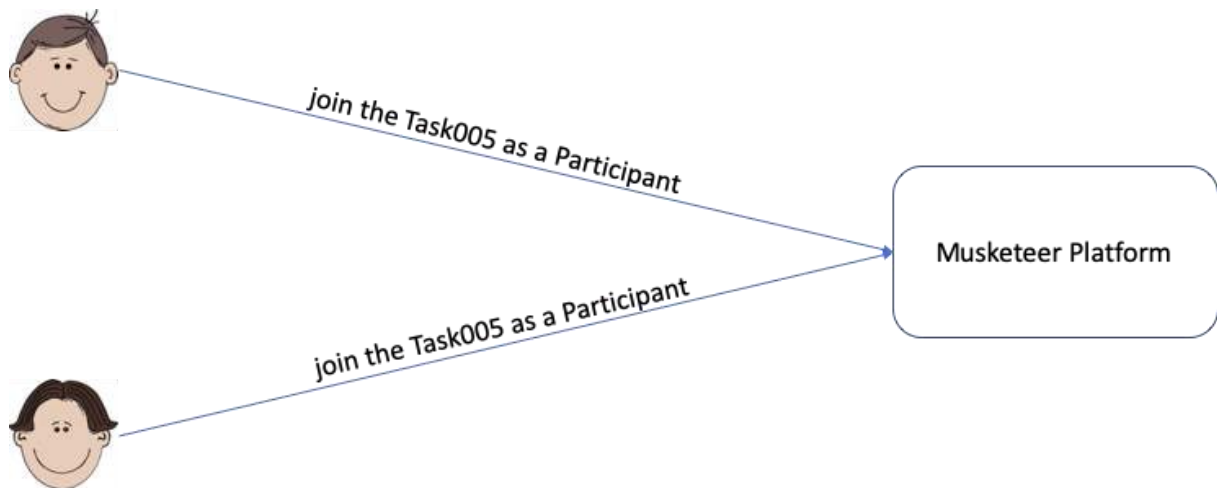
As task creator, Alice will also execute the aggregator side of the federated training process in her computational environment. In this example, the training quorum is 2 (see Figure 4), i.e. exactly two participants need to have joined the task before the training begins.

John and Jack can avail of the MUSKETEER platform services to explore federated ML tasks created by other users, including the task “Task005” that was created by Alice. By inspecting the definition of the task (which could also include meta information provided in full text), they can determine whether their available data fits the task’s requirements and ultimately decide whether they want to join that task as participant (Figure 5).



**Figure 5: List tasks on the MUSKETEER platform**

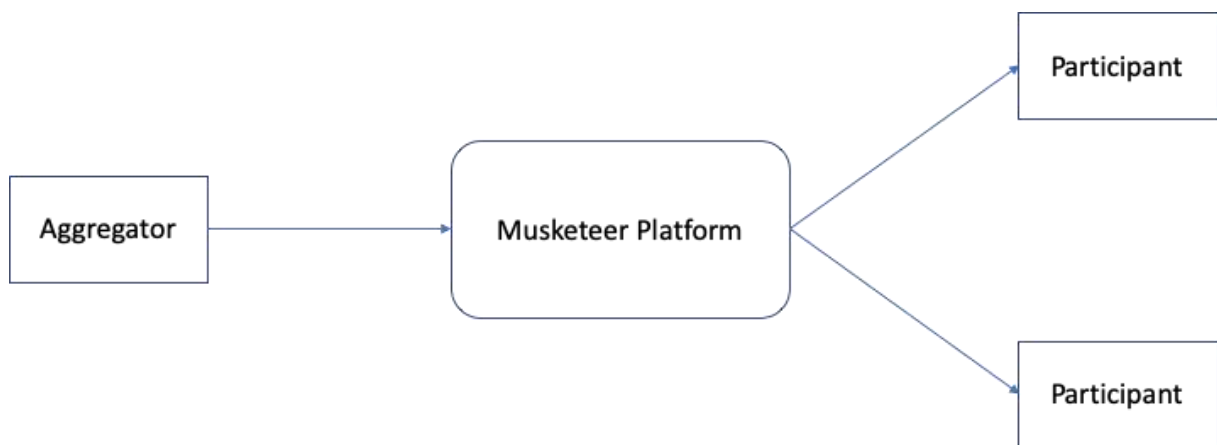
Once they make the decision to participate in a specific task (typically independently and unbeknownst of each other), they can avail of the platform services to join that task (Figure 6) and assuming the role of task participants in the following.



**Figure 6: Join task on the MUSKETEER platform**

With two participants having joined, the starting criterion 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 POM; in the following, we use standard federated ML training in accordance to POM1 (see D4.1) as a running example.

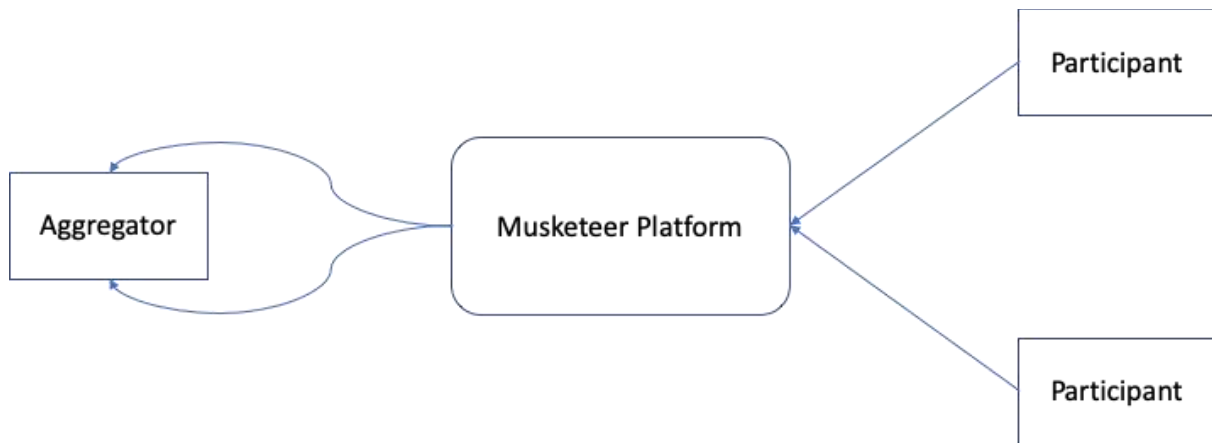
Firstly, the aggregator sends an initial version of the ML model to both participants. This can be done via a broadcast function in order to save communication costs, as illustrated in Figure 7. Without a broadcast function, the aggregator would have to send the same model repeatedly to the platform (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**

Upon completion, the final trained model could be stored by the aggregator in the MUSKETEER platform for later use, and/or sent to task participants who could then deploy the model in their respective local production environments.

### 3.2 Installation instructions

Before turning to the actual demonstrations in Section 3.3 and 3.4, we first provide instructions for setting up a local compute environment within which the interactions with the MUSKETEER platform can be performed. The instructions can also be found on the README page of the open source repository [1].

It is assumed that all development (e.g., of federated learning algorithms or of platform interfaces for end users) takes place in **Python**, using at least version 3.6. To speed up the creation of a development environment and generally for ease of use, an automated way for provisioning a **virtual machine (VM)** is provided. During the provisioning of the VM, all appropriate dependencies are installed. To take advantage of this automated build process, **Vagrant** [6] must be installed on the local compute system, and backed by a hypervisor such as **VirtualBox** [7].

For **Mac users**, these installations can be performed using HomeBrew [8]:

```
brew cask install virtualbox  
brew cask install vagrant  
vagrant plugin install vagrant-vbguest
```



Or, on **Windows**, using the following resources:

```
Install VirtualBox - https://www.virtualbox.org/wiki/Downloads
```

```
Install Vagrant - https://www.vagrantup.com/downloads.html
```

```
vagrant plugin install vagrant-vbguest
```

Users who choose not to bring up a VM should refer to the **Vagrantfile** in [1] for dependencies to manually install.

In the following it is assumed that the user has **cloned** or **downloaded** the **repository** [1] and has opened a **terminal** and navigated to the root directory of the local replica of the repository. At the time of writing, tag v0.1 in the repository is the latest release.

To **create the VM**, from the terminal, run:

```
vagrant up
```

This will take a few minutes. Upon completion, to **log into the VM**, run:

```
vagrant ssh
```

that the current directory will be shared between the host and the VM. To **stop the VM**:

```
vagrant halt
```

And to **delete the VM**:

```
vagrant destroy
```

Note: In order to avail of the MUSKETEER **cloud platform**'s services, **credentials** and the server **certificate** must be available. Those are available upon request from the IBM team.

There is a **test** provided which will **verify access** to the platform based on the available credentials. Logged into the VM, run:

```
python3 -m pytest tests/basic.py --credentials=<CREDENTIALS FILE> -srx -s
```

to perform the test, where <CREDENTIALS FILE> should be replaced by the name of the file containing the credentials and server certificate provided by IBM.

To facilitate research and rapid prototyping without dependency on cloud resources, we provide a **local version** of the MUSKETEER **platform** (see Section 2.2). This local version has the limitation that only one federated learning task can be running at a time. In order to **instantiate the local platform**, run:

```
python3 local_platform/musketeer.py
```

Finally, for this demonstrator we use **Jupyter notebooks** [9] as a simple **dashboard** and user interface for interacting with the MUSKETEER platform (the final client package developed in WP7 will provide more advanced graphical user interfaces). To **start the Jupyter notebook server**, run the following two commands in the terminal:

```
jupyter notebook password  
jupyter notebook --ip=0.0.0.0 &
```

The first command will result in a prompt to enter (and verify) a password for the Jupyter notebooks. After executing the second command, **open 127.0.0.1:8881 in your host browser** (tested with Chrome and Firefox), enter the password you chose and then you should see the navigation tree. The notebooks/ subfolder contains the actual notebooks that will be used to drive the demos in the following sections.

For users who do not use the VM and run the demo directly in their local environment, the Jupyter notebook server should be started as follows:

```
jupyter notebook --ip=127.0.0.1 &
```

### 3.3 Synthetic data

In this section, we present the demonstration on a synthetic example: training a CNN classifier on the MNIST dataset [10]. In this synthetic example, the aggregator and training participants will all have different random subsets of the MNIST dataset and use them for their local model updates and evaluations.

We first show how to run the demo via Python scripts in terminal windows, then walk through the demo driven by Jupyter notebooks, and finally show a mock-up of user interactions via a graphical user interface.

#### 3.3.1 Terminal windows

This demo is driven by the Python scripts contained in the demo/ subdirectory. Three different terminal windows need to be opened to run this demo. (If the local platform is used, then a fourth terminal needs to be opened to instantiate it; see above.) When using the VM, one needs to ssh into the VM in each of the three terminals. In the following, we will refer to the three terminals as **aggregator terminal**, **participant-1 terminal** and **participant-2 terminal**, respectively. In all three terminals, we first change to the demo/ subdirectory.

As first step in the aggregator terminal, we will register the user who is going to serve as task creator and aggregator:

```
python3 register.py --credentials <CREDENTIALS FILE> --user <AGGREGATOR USERNAME>  
--password <AGGREGATOR PASSWORD> --org <AGGREGATOR ORGANIZATION> --platform <CLOUD  
OR LOCAL>
```

The parameters here should be set as follows:

- <CREDENTIALS FILE> should be replaced by the name of the file containing the credentials and server certificate provided by IBM. If the local platform is used, set this to the `../local_credential_sample.json`
- <AGGREGATOR USERNAME> should be replaced by the username under which the aggregator registers. We recommend this to be a string of 8-16 characters without spaces or escape characters. Note: the same username can only be used once, i.e. if a user had registered with the same before, an error will be thrown.
- <AGGREGATOR PASSWORD> should be replaced by the password chosen by the aggregator user which will be used for authentication in further interactions with the platform. We recommend this to be a string of 8-16 characters without spaces or escape characters.
- <AGGREGATOR ORGANIZATION> should be replaced by the organization of the user. This parameter isn't strictly required for further platform interactions but rather for information purposes.
- <CLOUD OR LOCAL> should be replaced by either `cloud` or `local`, indicating whether the cloud or local platform is to be used. The credentials file above needs to be set accordingly.

An example of this step is shown in the screenshot in Figure 9.

```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 register.py --credentials ../cloud_credentials.json --user aggregator-demo3.3 --password password
--org MUSKETEER --platform cloud
2020-05-07 09:48:14.678 INFO    __main__ 139813315204928 :: User aggregator-demo3.3 created
vagrant@ubuntu-musketeer-client: /vagrant/demo
$
```

**Figure 9: Registering the aggregator user via terminal**

Next, we will register the two designated participants of this demonstration task, using the same command as for registering the aggregator except that the usernames, passwords and organization are those for the participants. Example of these registration steps are shown in Figure 10 and Figure 11.

We note that, while in this simple demonstration the two participants could register from the same terminal as the aggregator and use the same credentials file, in real-world scenarios the participants will register from different hosts, possibly belonging to different organizations and located in different geographies, and use their own dedicated cloud credentials.

```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 register.py --credentials ../cloud_credentials.json --user participant-1-demo3.3 --password password
--org MUSKETEER --platform cloud
2020-05-07 09:53:36.541 INFO    __main__ 140609247610688 :: User participant-1-demo3.3 created
vagrant@ubuntu-musketeer-client: /vagrant/demo
$
```

**Figure 10: Registering the participant-1 user via terminal**

```
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$ python3 register.py --credentials ../cloud_credentials.json --user participant-2-demo3.3 --password passwd
rd --org MUSKETEER --platform cloud
2020-05-07 09:54:29.851 INFO __main__ 148153918744896 :: User participant-2-demo3.3 created
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$
```

**Figure 11: Registering the participant-2 user via terminal**

Next, the aggregator user will create the federated learning task via the following command:

```
python3 creator.py --credentials <CREDENTIALS FILE> --user <AGGREGATOR USERNAME> -
-password <AGGREGATOR PASSWORD> --task_name <TASK NAME> --platform <CLOUD OR
LOCAL>
```

Here <CREDENTIALS FILE> and <CLOUD OR LOCAL> are the same as before, and <AGGREGATOR USERNAME> and <AGGREGATOR PASSWORD> are the username and password provided in the registration step above. <TASK NAME> is a name for the newly created task chosen by the aggregator, with the same recommendations regarding string length and escape characters as before; also, same as for usernames, task names need to be unique, hence, if a task of the same name had been created before, the platform will return an error message.

Figure 12 shows an example of the task creation step. We note that the message returned by the platform contains meta information that the platforms stores about the newly created task, such as its name, by whom and when it was created, and the actual task definition, which is in the form of a dictionary. Deciding on the specifics of the task definition is up to the task creator; in the above command for task creation the task definition does not explicitly occur; it is hard-coded in the creator.py script, lines 89-98.

```
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$ python3 creator.py --credentials ../cloud_credentials.json --user aggregator-demo3.3 --password passwdrd -
-task_name task-demo3.3 --platform cloud
2020-05-07 11:41:20.192 DEBUG __main__ 148442363057984 :: {'task_name': 'task-demo3.3', 'user_name': 'aggre
gator-demo3.3', 'status': 'CREATED', 'queue': '9808f645f12ce27ba54b363d7c7a793d9e990e96/aggregator/task-demo
3.3', 'topology': 'STAR', 'definition': {'aggregator': 'neural_network.Aggregator', 'batch_size': 256, 'epo
ch': 2, 'learning_rate': 0.001, 'participant': 'neural_network.Participant', 'quorum': 2, 'round': 5, 'test
size': 1000, 'training_size': 10000}, 'added': '2020-05-07T11:41:21.858311Z', 'updated': '2020-05-07T11:41
21.876820Z'}
2020-05-07 11:41:20.192 INFO __main__ 148442363057984 :: Task created.
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$
```

**Figure 12: Creating a federated learning task via terminal**

Here we look at the task definition more carefully:

```
{
  "aggregator": "neural_network.Aggregator",
  "participant": "neural_network.Participant",
  "quorum": 2,
  "round": 5,
  "epoch": 2,
  "batch_size": 256,
  "learning_rate": 0.001,
  "training_size": 10000,
  "test_size": 1000
}
```

The "aggregator" and "participant" fields reference the code that the aggregator and participants shall execute as part of the actual federated learning. It refers to the classes `Aggregator` and `Participant` in the Python module `neural_network.py`, which is located under the `fl_algorithm/` directory in the client package. In the final version of the MUSKETEER platform, algorithms from the federated ML library developed under WP4 will be developed at this place.

The "quorum" parameter specifies the number of participants that need to have joined the task before the training will start. "round" is the number of rounds during which participants locally compute updates on the latest model shared by the aggregator (compare with the high-level description of federated learning algorithms provided in Section 1.2), where the updates are obtained over "epoch" number of epochs, using batches of size "batch\_size" and the learning rate "learning\_rate". "training\_size" and "test\_size" specify the number of data points from the MNIST dataset that will be used locally by the participants for training/updating and testing the ML model.

Upon creation of the task, the aggregator can now initiate the training process by locally starting the aggregator process:

```
python3 aggregator.py --credentials <CREDENTIALS FILE> --user <AGGREGATOR  
USERNAME> --password <AGGREGATOR PASSWORD> --task_name <TASK NAME> --platform  
<CLOUD OR LOCAL>
```

Here <TASK NAME> refers to the task created above by its unique name. Effectively, in line 91, the `aggregator.py` script will retrieve the corresponding task definition from the MUSKETEER platform, and in line 102 execute the `.start()` method implemented by the `neural_network.Aggregator` class. We note that in that method (line 176 in the `neural_network.Aggregator` class), the aggregator training process will wait until the number of participants specified in the "quorum" field of the task definition has joined the task; this can also be seen in the log message after starting the aggregator process (see Figure 13). Inspecting the log messages, we can also see that the aggregator process has already started the Keras TensorFlow backend, which will be used later on for performing machine learning operations, and downloaded the MNIST dataset from <https://s3.amazonaws.com/img-datasets/mnist.npz>.



```
vagrant@ubuntu-musketeer-client: /vagrant/demo  
$ python3 aggregator.py --credentials ../cloud_credentials.json --user aggregator-demo3:3 --password passwd@  
d --task_name task-demo3:3 --platform cloud  
Using TensorFlow backend.  
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz  
11493376/11498434 [=====] - 9s 1us/step  
2020-05-07 12:22:48.792 INFO neural_network 140319238518592 :: Waiting on quorum  
2020-05-07 12:22:50.249 INFO neural_network 140319238518592 :: Waiting on workers to join (0 of 2 present)
```

Figure 13: Starting the aggregator process

In parallel, from their respective terminal windows, participants 1 and 2 can list the available tasks in the platform using the following command:

```
python3 listing.py --credentials <CREDENTIALS FILE> --user <PARTICIPANT USERNAME>  
--password <PARTICIPANT PASSWORD> --platform <CLOUD OR LOCAL>
```

For participant 1, this operation is shown in Figure 14. We note that, in general, many more tasks besides the one created above may be listed via this command, e.g. tasks created by other users in the past, including tasks for which the training may already have been completed.

```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 listing.py --credentials ../cloud_credentials.json --user participant-1-demo3.3 --password password --platform cloud
2020-05-07 12:32:59.780 INFO      __main__ 148476065929024 :: task-demo3.3 - CREATED
vagrant@ubuntu-musketeer-client: /vagrant/demo
$
```

**Figure 14: Listing federated learning tasks**

The next step is for participants 1 and 2, in their respective terminal windows to join the task task-demo3.3 that was created by the aggregator above. This is done using the following command:

```
python3 join.py --credentials <CREDENTIALS FILE> --user <PARTICIPANT USERNAME> --password <PARTICIPANT PASSWORD> --task_name <TASK NAME> --platform <CLOUD OR LOCAL>
```

Figure 15 shows this step for participant 1.

```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 join.py --credentials ../cloud_credentials.json --user participant-1-demo3.3 --password password --task_name task-demo3.3 --platform cloud
2020-05-07 12:48:45.237 DEBUG    __main__ 139748996749128 :: Joined task
vagrant@ubuntu-musketeer-client: /vagrant/demo
$
```

**Figure 15: Joining a task as participant**

After both participant 1 and 2 have joined, we can see in the log messages from the aggregator process that the quorum of two participants is found to be present (see Figure 16). At this point, the aggregator process will start the actual training and – as can be seen from the log messages – send the initial version of the neural network to the participants for them to update it on their local data.

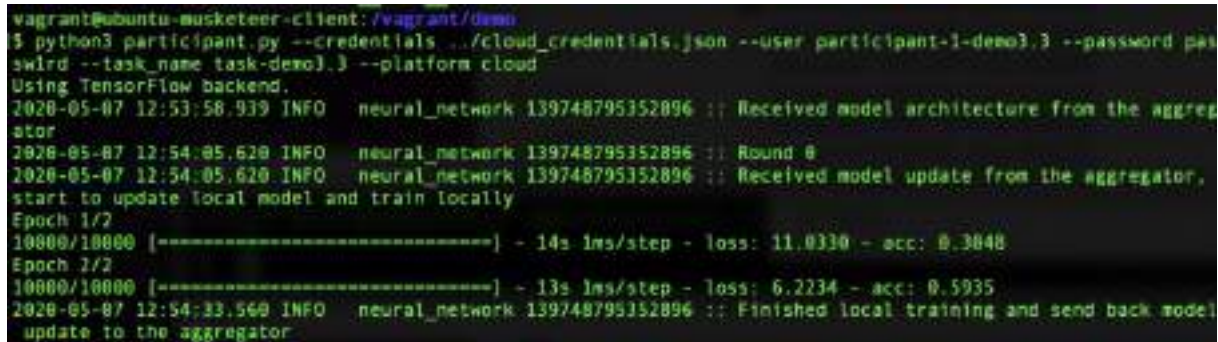
```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 aggregator.py --credentials ../cloud_credentials.json --user aggregator-demo3.3 --password password --task_name task-demo3.3 --platform cloud
Using TensorFlow backend.
2020-05-07 12:51:27.395 INFO      neural_network 148244833609536 :: Waiting on quorum
2020-05-07 12:51:29.807 INFO      neural_network 148244833609536 :: Waiting on workers to join (1 of 2 present)
2020-05-07 12:52:22.745 INFO      neural_network 148244833609536 :: Quorum of workers found
2020-05-07 12:52:22.745 INFO      neural_network 148244833609536 :: Starting training
2020-05-07 12:52:22.886 INFO      neural_network 148244833609536 :: Distributing neural network architecture to participants
2020-05-07 12:52:24.453 INFO      neural_network 148244833609536 :: Round 0
2020-05-07 12:52:24.453 INFO      neural_network 148244833609536 :: Asking participants to update model weights
- do local training and send back model update
```

**Figure 16: Performing the federated learning on the aggregator side (start)**

In order to close the loop and initiate the corresponding actions on the participants' side, participant 1 and 2 need to start their local training processes via the following command:

```
python3 participant.py --credentials <CREDENTIALS FILE> --user <PARTICIPANT USERNAME> --password <PARTICIPANT PASSWORD> --task_name <TASK NAME> --platform <CLOUD OR LOCAL>
```

Figure 17 shows this step for participant 1. As can be seen, upon starting the participant’s training process it will immediately update the model received from the aggregator on the local data and then send the model update back to the aggregator.

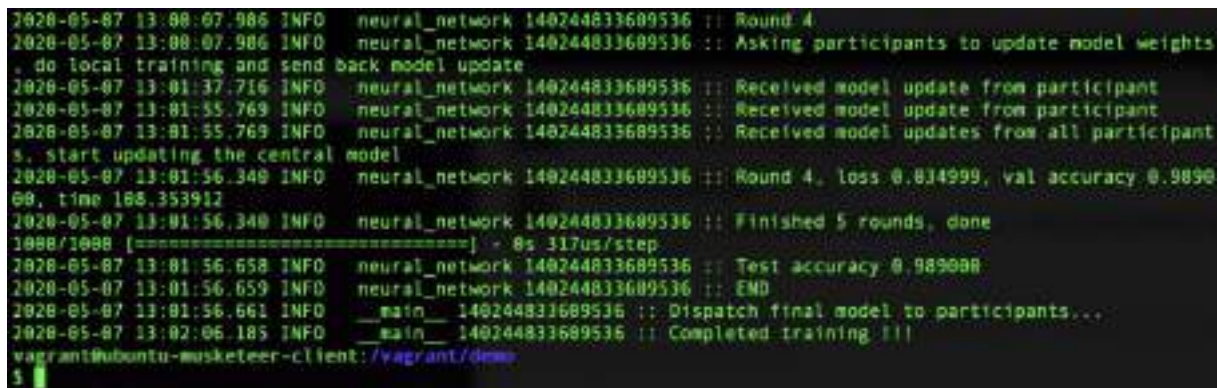


```
vagrant@ubuntu-musketeer-client: /vagrant/demo
$ python3 participant.py --credentials ../cloud_credentials.json --user participant-1-demo3.3 --password password --task_name task-demo3.3 --platform cloud
Using TensorFlow backend.
2020-05-07 12:53:58.939 INFO    neural_network 139748795352896 :: Received model architecture from the aggregator
2020-05-07 12:54:05.620 INFO    neural_network 139748795352896 :: Round 0
2020-05-07 12:54:05.620 INFO    neural_network 139748795352896 :: Received model update from the aggregator, start to update local model and train locally
Epoch 1/2
10000/10000 [-----] - 14s 1ms/step - loss: 11.0330 - acc: 0.3048
Epoch 2/2
10000/10000 [-----] - 13s 1ms/step - loss: 6.2234 - acc: 0.5935
2020-05-07 12:54:33.560 INFO    neural_network 139748795352896 :: Finished local training and send back model update to the aggregator
```

Figure 17: Performing the federated learning on the participant side (start)

From this point onwards the training process between the aggregator and the two participants is performed iteratively as described in Section 1.2 throughout the total 5 rounds (numbered 0, 1, 2, 3, 4) prescribed in the task definition.

Figure 18, Figure 19 and Figure 20 show the log messages from the last round of updates and the termination messages for the aggregator, participant 1 and participant 2. As can be seen, the final testing accuracy (evaluated on the aggregator side) is around 99%. Before the termination, the aggregator dispatches the final model to the participants. In this simple demonstration, no further action is taken on their side, however, in future versions of the platform with the fully developed client connectors there may be end-to-end support for putting the trained models into production on the participants’ sides.



```
2020-05-07 13:00:07.986 INFO    neural_network 140244833689536 :: Round 4
2020-05-07 13:00:07.986 INFO    neural_network 140244833689536 :: Asking participants to update model weights, do local training and send back model update
2020-05-07 13:01:37.716 INFO    neural_network 140244833689536 :: Received model update from participant
2020-05-07 13:01:55.769 INFO    neural_network 140244833689536 :: Received model update from participant
2020-05-07 13:01:55.769 INFO    neural_network 140244833689536 :: Received model updates from all participants, start updating the central model
2020-05-07 13:01:56.340 INFO    neural_network 140244833689536 :: Round 4, loss 0.834999, val accuracy 0.989088, time 106.353912
2020-05-07 13:01:56.340 INFO    neural_network 140244833689536 :: Finished 5 rounds, done
10000/10000 [-----] - 0s 317us/step
2020-05-07 13:01:56.658 INFO    neural_network 140244833689536 :: Test accuracy 0.989088
2020-05-07 13:01:56.659 INFO    neural_network 140244833689536 :: EMD
2020-05-07 13:01:56.661 INFO    __main__ 140244833689536 :: Dispatch final model to participants...
2020-05-07 13:02:06.185 INFO    __main__ 140244833689536 :: Completed training!!!
vagrant@ubuntu-musketeer-client: /vagrant/demo
$
```

Figure 18: Termination of the federated learning on the aggregator side

```
2020-05-07 13:08:28.446 INFO neural_network 139748795352896 :: Round 4
2020-05-07 13:08:28.447 INFO neural_network 139748795352896 :: Received model update from the aggregator,
start to update local model and train locally
Epoch 1/2
10000/10000 [=====] - 30s 3ms/step - loss: 0.8835 - acc: 0.9735
Epoch 2/2
10000/10000 [=====] - 27s 3ms/step - loss: 0.8731 - acc: 0.9768
2020-05-07 13:01:25.582 INFO neural_network 139748795352896 :: Finished local training and send back model
update to the aggregator
2020-05-07 13:01:31.398 INFO neural_network 139748795352896 :: Finished 5 rounds, done.
2020-05-07 13:02:11.245 INFO neural_network 139748795352896 :: Received the final model from aggregator
2020-05-07 13:02:11.253 INFO __main__ 139748795352896 :: Completed training !!!
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$
```

Figure 19: Termination of the federated learning on the participant 1 side

```
2020-05-07 13:08:27.186 INFO neural_network 140692055484224 :: Round 4
2020-05-07 13:08:27.186 INFO neural_network 140692055484224 :: Received model update from the aggregator,
start to update local model and train locally
Epoch 1/2
10000/10000 [=====] - 28s 3ms/step - loss: 0.8812 - acc: 0.9758
Epoch 2/2
10000/10000 [=====] - 29s 3ms/step - loss: 0.8654 - acc: 0.9888
2020-05-07 13:01:24.141 INFO neural_network 140692055484224 :: Finished local training and send back model
update to the aggregator
2020-05-07 13:01:33.189 INFO neural_network 140692055484224 :: Finished 5 rounds, done.
2020-05-07 13:02:11.681 INFO neural_network 140692055484224 :: Received the final model from aggregator
2020-05-07 13:02:11.684 INFO __main__ 140692055484224 :: Completed training !!!
vagrant@ubuntu-musketeer-client:~/vagrant/demo
$
```

Figure 20: Termination of the federated learning on the participant 2 side

### 3.3.2 Jupyter notebooks

Next, we show how to drive the interactions with the MUSKETEER platform through Jupyter notebooks. Here, we assume the Jupyter notebook server has been started and the tree view been opened in a local web browser (see Section 3.2), thus the user should see a screen similar to what is shown in Figure 21.





Figure 21: Jupyter notebook tree view

Opening the notebooks/ folder, the user should then see the two notebooks that will drive the demonstrator: task\_creator.ipynb and task\_participant.ipynb (see Figure 22).



Figure 22: Demonstrator notebooks

First, we look at the notebook task\_creator.ipynb which drives the interactions of a task creator / aggregator with the MUSKETEER platform. The first cell (Figure 23) loads the required dependencies into the Python notebook environment.



Figure 23: Task creator notebook - loading prerequisites

The next two cells (Figure 24) allow the task creator to register a username and a password with the MUSKETEER platform. The same comments as above regarding the length, escape characters and the uniqueness of the username apply.

```

Create a user name and password

In [2]: ccredentials = './cloud_credentials.json' # e.g. './local_credentials.json'
        user = 'aggregator-demo3.3-notebook' # note [local] mode requires that user names must be different for aggregator and pu
        password = 'password'
        org = 'MUSKETEER'

In [3]: platform = 'cloud'
        context = utils.platform(platform, credentials)

        # Only run this if you had not created this user before:
        try:
            register.create_user(context, user, password, org)
        except Exception as err:
            print("Error:", err)
    
```

**Figure 24: Task creator notebook - registering user**

The next three cells (Figure 25) let the task creator retrieve a list of existing federated learning tasks in the MUSKETEER platform, display the total number of existing tasks, and list details (name, creation time and status) of tasks that were created within the last 24 hours. Note: depending on the recent usage of the platform, this may be an extensive list.

```

Listing existing Federated ML tasks

In [4]: #Now create the platform context for the new user
        context = utils.platform(platform, credentials, user, password)
        tasks = listing.get_tasks(context)

In [5]: print("Number of tasks:", len(tasks))
        Number of tasks: 147

In [6]: # List all tasks that have been added in the last 24 hours
        for task in tasks:
            if task['added'] >= strftime('%Y-%m-%dT%H:%M:%S', gmtime(time() - 3600*24)):
                print(task['task_name'], '\t', task['added'], '\t', task['status'])

task-demo3.3    2020-05-27T11:41:21.058311Z    COMPLETE
    
```

**Figure 25: Task creator notebook - listing existing tasks**

The next five cells (Figure 26) drive the creation of a new federated learning task. The task definition is the same as in the previous section, and the same comment regarding task naming applies. As shown in the following cells, the task creator is able to confirm that the newly created task is indeed available in the MUSKETEER platform, along with its meta information and the task definition details.

```

Creating a new Federated ML task

In [7]: # Note: task names used in the original, so you need to use a new name for each task that you create
task_name = 'task-demo.3-notebook'

task_definition = {
    "aggregator": "neural_network.Aggregator",
    "participant": "neural_network.Participant",
    "quorum": 2,
    "round": 5,
    "epoch": 2,
    "batch_size": 256,
    "learning_rate": 0.001,
    "training_size": 10000,
    "test_size": 1000,
}

In [8]: try:
        result = creator.create_task(context, task_name, task_definition)
    except Exception as err:
        print("Error:", err)

In [9]: tasks = listing.get_tasks(context)

In [10]: print('New task available in Musketeer: ', task_name in [t["task_name"] for t in tasks])

New task available in Musketeer: True

In [11]: print([t for t in tasks if t["task_name"] == task_name][0])

{'task_name': 'task-demo.3-notebook', 'status': 'CREATED', 'added': '2020-05-07T20:28:47.804393Z', 'topology': 'STAN', 'definition': {'aggregator': 'neural_network.Aggregator', 'batch_size': 256, 'epoch': 2, 'learning_rate': 0.001, 'participant': 'neural_network.Participant', 'quorum': 2, 'round': 5, 'test_size': 1000, 'training_size': 10000}}
    
```

**Figure 26: Task creator notebook - creating a new task**

The final cell in this notebook (Figure 27) starts the aggregator part of the federated learning process. Same as in the previous sections, the first few log messages show that the aggregator is waiting for the quorum of two participants to be met.

```

Running that task as aggregator

In [*]: try:
        aggregator.run(context, task_name)
    except Exception as err:
        print("Error:", err)

Using TensorFlow backend.
2020-05-07 20:29:33.687 INFO neural_network 146712935860832 :: Waiting on quorum
2020-05-07 20:29:34.598 INFO neural_network 146712935860832 :: Waiting on workers to join (1 of 2 present)
    
```

**Figure 27: Task creator notebook - aggregator process (start)**

The second notebook, `task_participant.ipynb`, drives the interactions of a task participant with the MUSKETEER platform. Since for this demonstration we need a quorum of two participants, we need to make a copy of this notebook and drive the interactions of the two participants through two different notebooks. (Note: a copy can simply be created, after opening the `task_participant.ipynb` notebook, by selecting “File -> Make a Copy...” from the top menu in the notebook.) In the following, we show the interactions for participant 1.

In the first cell (Figure 28), prerequisites are loaded, same as in the task creator notebook.

### Load prerequisites

```
In [1]: import sys
sys.path.append('.')
sys.path.append('../demo')
from demo import register, listing, join, creator, participant
from demo import platform_utils as utils

from time import time, gmtime, strftime

import logging
logger = logging.getLogger()
handler = logger.handlers[0]
handler.setLevel(logging.INFO)
```

Figure 28: Task participant notebook - loading prerequisites

The next two cells (Figure 29) drive the registration of the participant’s username and password, same as in the aggregator notebook.

### Create a user name and password

```
In [2]: credentials = '../cloud_credentials.json' # e.g. '../local_credentials_sample.json'
user = 'participant-1-demo1-1-notebook' # note local mode requires that user names must be different for aggregator and
password = 'password'
org = 'MUSKETEER'

In [3]: platform = 'cloud'
context = utils.platform(platform, credentials)

# Only run this if you had not created this user before:
try:
    register.create_user(context, user, password, org)
except Exception as err:
    print("Error:", err)
```

Figure 29: Task participant notebook – registering user

The next three cells (Figure 30) allow the task participant to retrieve a list of all federated learning tasks available in the MUSKETEER platform, show their total number, and display details of tasks that were created within the last 24 hours. Among those, the task participant should be able to see the task that was just created by the task creator.

### Listing existing Federated ML tasks

```
In [4]: #Now create the platform context for the new user:
context = utils.platform(platform, credentials, user, password)
tasks = listing.get_tasks(context)

In [5]: print("Number of tasks:", len(tasks))
Number of tasks: 348

In [6]: # List all tasks that have been added in the last 24 hours:
for task in tasks:
    if task['added'] == strftime('%Y-%m-%dT%H:%M:%S', gmtime(time() - 3600*24)):
        print(task['task_name'], '\t', task['added'], '\t', task['status'])

task-demo1.1      2020-05-07T11:41:21.058311Z    COMPLETE
task-demo1.1-notebook  2020-05-07T20:26:47.804391Z    CREATED
```

Figure 30: Task participant notebook - listing tasks

In the next five cells (Figure 31), the task participant can first inspect details of a task that she is interested in, join the task, and finally confirm that the MUSKETEER platform has correctly registered her participation.

```

Join a Federated ML task

In [7]: # Name of the task to be joined:
task_name = 'task-demo1.1-notebook'

# Note: this should be an existing task with status "CREATED"

In [8]: tasks = listing.get_tasks(context)
print([t for t in tasks if t['task_name'] == task_name][0])

{'task_name': 'task-demo1.1-notebook', 'status': 'CREATED', 'added': '2020-05-07T20:26:47.894393Z', 'topology': 'STAN', 'definition': '{"aggregator": "neural_network.Aggregator", "batch_size": 256, "epoch": 2, "learning_rate": 0.001, "participant": "neural_network.Participant", "quorum": 2, "round": 5, "test_size": 1000, "training_size": 10000}'}

In [9]: try:
    join.join_task(context, task_name)
except Exception as err:
    print("Error:", err)

In [10]: joined_tasks = join.get_user_assignments(context)

In [11]: for task in joined_tasks:
    print(task['task_name'], '\t', task['added'], '\t', task['status'])

task-demo1.1-notebook    2020-05-07T20:54:13.078994Z    CREATED
    
```

Figure 31: Task participant notebook - joining a task

The final cell in the notebook (Figure 32) will launch the participant’s part of the federated learning process.

```

Running that task as participant

In [*]: try:
    participant.run(context, task_name)
except Exception as err:
    print("Error:", err)

Using TensorFlow backend.
2020-05-07 20:38:41.126 INFO neural_network 14073823098400 :: Received model architecture from the aggregator
2020-05-07 20:38:50.339 INFO neural_network 14073823098400 :: Round 0
2020-05-07 20:38:50.342 INFO neural_network 14073823098400 :: Received model update from the aggregator, start to update local model and train locally

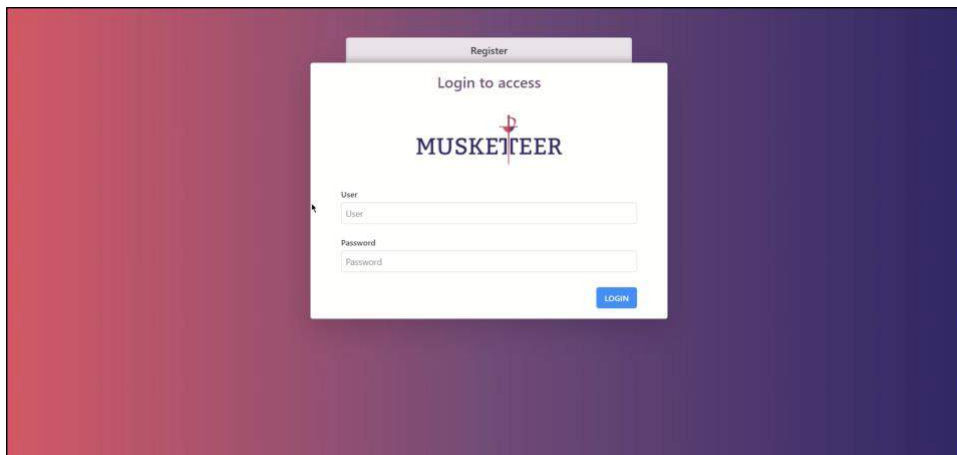
Epoch 1/2
8448/10000 [#####.....] - ETA: 3s - loss: 8.2091 - acc: 1.4570
    
```

Figure 32: Task participant notebook - participant process (start)

The log messages that are displayed in the aggregator and the participants’ notebooks follow the same format as in the terminal-driven demo described in the previous section (they result from the execution of the same code implemented in the `neural_network.Aggregator` and `neural_network.Participant` classes; the exact numerical results may slightly differ though due to different random initializations of the neural network, different random sampling of local training / test data, and different random shuffling of local training data during the local model updates).

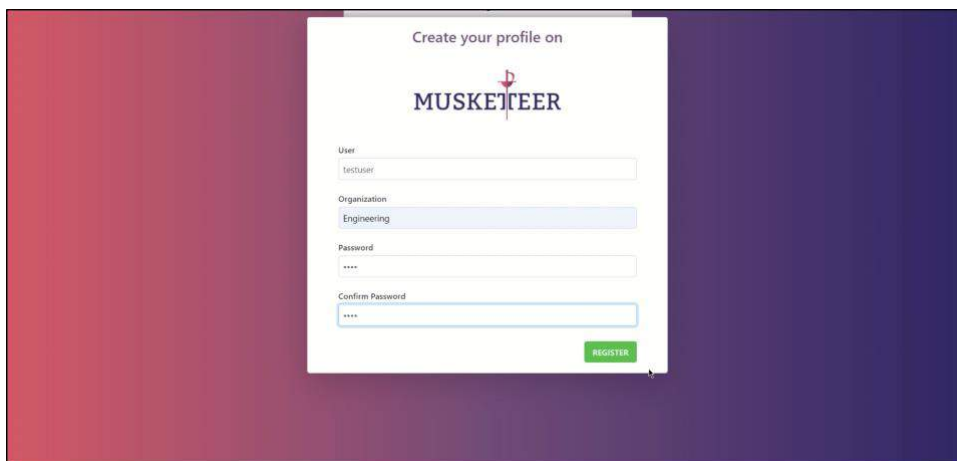
### 3.3.3 Graphical user interface mock-up

Finally, we show how the user interactions with the platform could be performed via a graphical user interface, using the platform Python APIs in the backend. Figure 33 shows the login phase where an already registered user can login to the platform by entering her credentials and will receive an error message in case the user name or password are wrong.



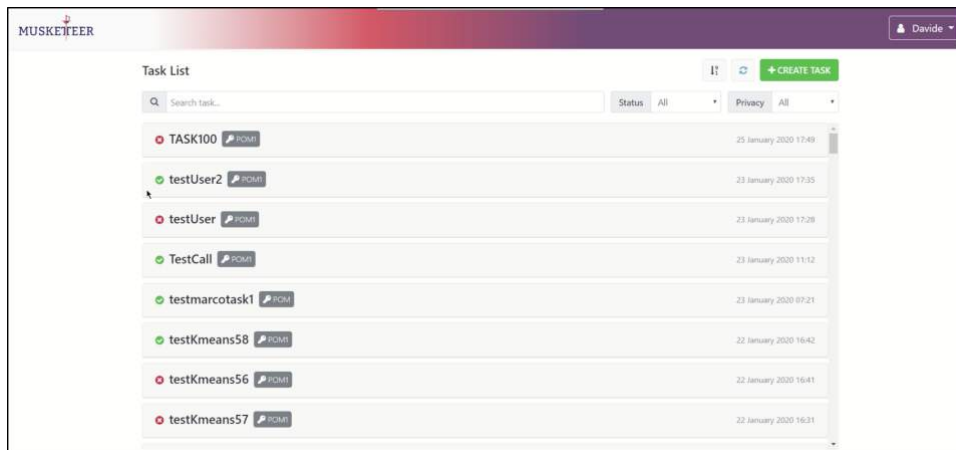
**Figure 33: Graphical user interface - login page**

Figure 34 shows a screen which allows a new user to register by providing a user name, password (which has to be confirmed in an extra field), and an organization.



**Figure 34: Graphical user interface - user registration**

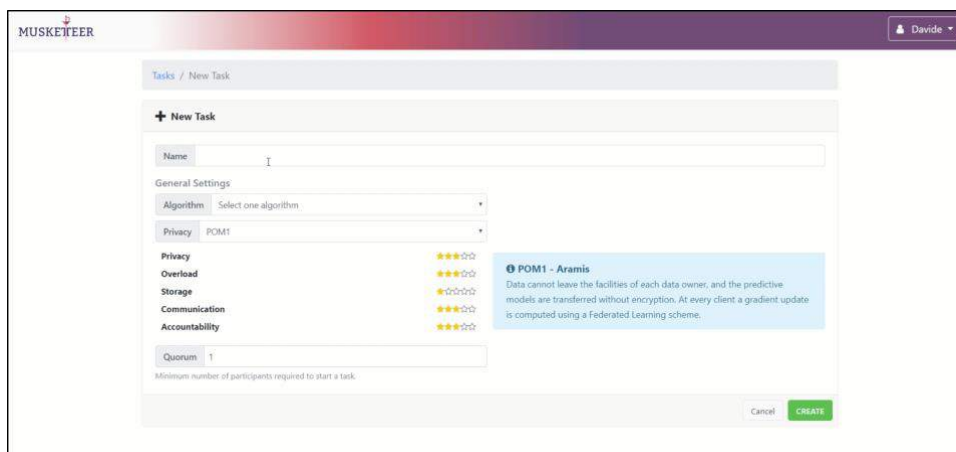
Figure 35 shows the tasks listing page that is available to the user after logging in. The page displays all the created tasks; they are marked by their name, an icon to show their current status, the privacy operation mode chosen for that task (if available as part of the task definition), further details of the task and the task creation date. By clicking on the task, it is possible to obtain additional information about the task. The page allows for task browsing and searching for specific tasks via a search bar or through different filters. Finally, using the button on the right side allows the user to create a new task.



**Figure 35: Graphical user interface - task listing**

Figure 36 shows the task creation page. In this mock-up example, the following task definition information can be provided:

- Task name: the name under which the task should be displayed;
- Federated Machine Learning algorithm: once the algorithm is selected it is possible to set the parameters of that algorithm;
- Privacy level: indicated by a privacy operation modes number (where for each mode a brief description and summary of its specifications is available);
- Quorum: the minimum number of participants required for the task.



**Figure 36: Graphical user interface - task creation**

The screen in Figure 37 displays a task summary, including its status and the operations that the user can perform on this task. Next to the summary on the right hand side are four buttons which allow the users to inspect details of the task definition, run the task as a participant, run the task as an aggregator (this option is only available if the user is the creator of that task), delete the task (this feature is not supported by the first prototype of the platform, but it will be in the final version).

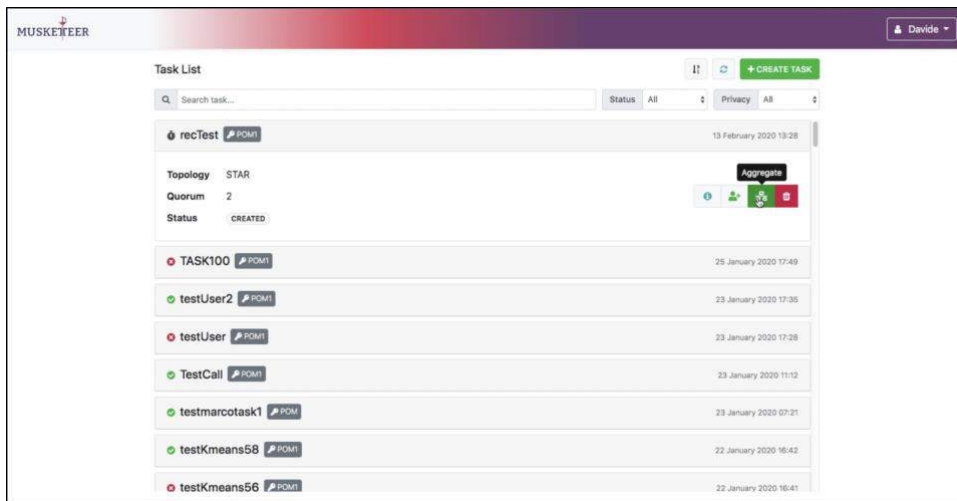


Figure 37: Graphical user interface - task operations

Upon clicking on the join/participate button, the dialogue box shown in Figure 38 will appear. Here, on the left side, the user can choose a dataset to use for that task, while on the right side the user can add new datasets from the local file system.

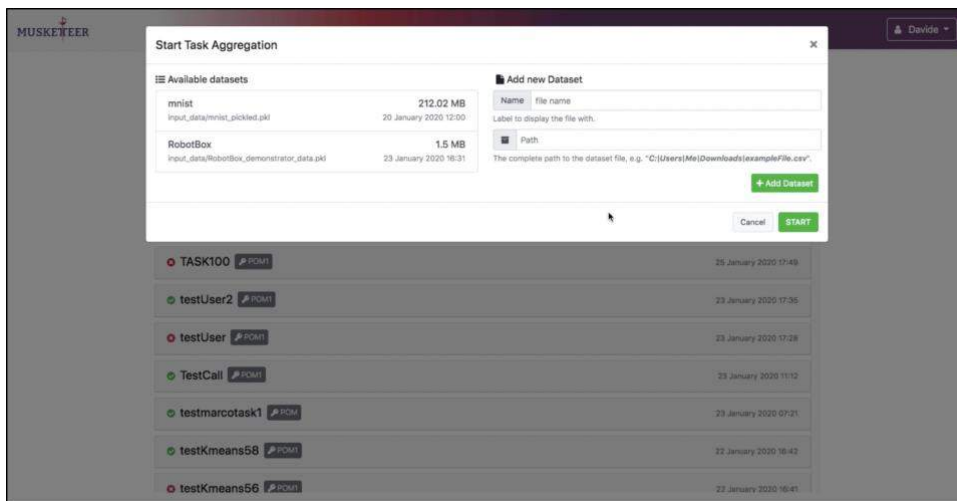


Figure 38: Graphical user interface - task execution settings

The screen in Figure 39 shows a chart with the result of a task execution. In this example the result is the clustering of the data obtained using the K-Means algorithm, shown in a scatter-plot with the two resulting clusters displayed in different colors.



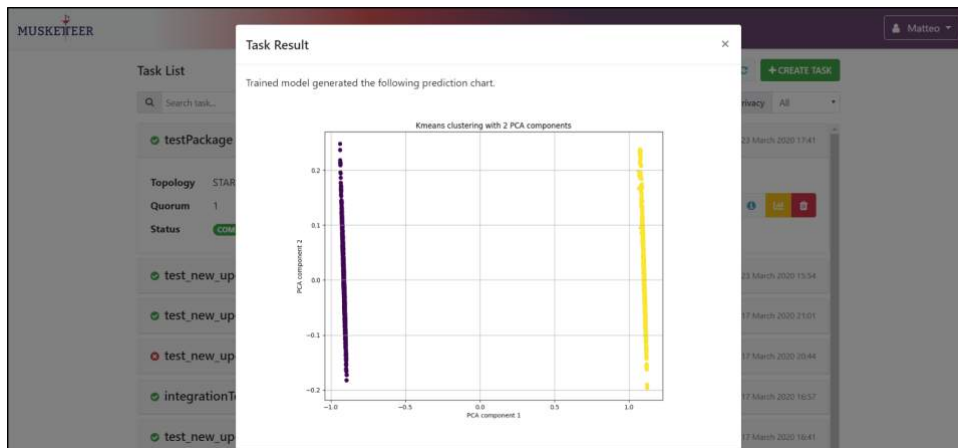


Figure 39: Graphical user interface - task result

### 3.4 Use case: Smart manufacturing

In this section, we explain how to use the prototype of the MUSKETEER platform to support a real-world use case: training predictive models with data from Smart manufacturing devices. For detailed background on this use case we refer to D2.1. We would also like to note that the final development of the MUSKETEER use cases is outside the scope of this deliverable but will be performed under WP7 (see Figure 1).

Here we briefly recapitulate the motivation at a high level: the context of this use case is the deployment of manufacturing robots (supplied by COMAU) in car manufacturing plants (owned and operated by FCA). More specifically, the robots considered in this use case are applied in the car welding process. Whilst operating, the welding guns collect meta information and sensor data for each welding point. From a business perspective, the goal of this use case is to harness those data to automatically identify potential quality issues during the welding process as soon as possible, and to optimize robot maintenance schedules. If the operational data from all robots was centrally available, COMAU could create predictive or prescriptive analytical models for this purpose following conventional Data Science methodologies. However, since the data is proprietary to FCA, this is not possible. Federated learning therefore offers a compromise where COMAU can leverage operational data from the robots to create ML models for the aforementioned purpose, while the data resides within FCA and no proprietary or confidential information is revealed.

In this demonstration we will utilize data from two robots applying welding points to the car: **FPS2**, operating on the left side, and **FPD2**, operating on the right side. The two robots symmetrically apply 5 welding points to each car model passing through the assembly line, denoted by P081, P082, P083, P085, P086. For the purpose of this demo, we will focus on the welding point **P083**.

#### 3.4.1 Data Science workflow

Before demonstrating how the MUSKETEER platform can support this use case, we will first walk through the Data Science workflow. For complete documentation, we refer to the

Jupyter notebook in [11], however, note that because of their proprietary nature the actual data and the code for parsing the raw data are *not* included in this deliverable, therefore the reader will not be able to run the notebook.

The raw meta-information and sensor data from the robots is stored in .xml files, where each file contains the information for one car door. For this analysis, **49,627 files** from the left door (**FPS2**) and **30,983 files** from the right door (**FPD2**) were used. The first step in the workflow is to extract from the raw data the fields that are relevant for creating the ML models:

- **WELCNT**: This is a counter which tracks how many welding points the robot has been applied since the last dressing of the welding electrode.
- **WDRSCNT**: This is a counter which tracks the number of dressings of the welding electrode before its complete replacement.
- **WELR**: Electric resistance when applying the electrode to this welding point.
- **WELCUR1\_S**: Electric current when applying the electrode to this welding point.
- **WCLSRES**: An automatically generated quality index, with integer values between 0 and 10. Here, a quality index of 6 or 7 means that the performance of the welding electrode was within the permissible operating range; other values indicate that the welding electrode has three possible problems: “squeezed”, “pasted” or “short circuit”.

After parsing the raw data, we obtain a total of **19,288** complete data records from **FPS2**, and **11,143** from **FPD2**. Applying an 80:20 split, we divide those into **13,831 / 3,457 training / test** samples for **FPS2**, and **8,915 / 2,228 training / test** samples for **FPD2**.

The ML problem that we would like to solve is predicting **WCLSRES** as a function of **WELCNT**, **WDRSCNT**, **WELR** and **WELCUR1\_S**. From the business perspective, solving this ML problem would be valuable because it would allow us to predict quality indices for robots which do not automatically generate them, based on the available meta information and sensor data.

In order to cast this ML problem as a 4-class classification task, we create a binned variable **WCLSRES\_BIN** from **WCLSRES** as follows:

- **WCLSRES\_BIN** = 0 if **WCLSRES** is less than or equal to 3;
- **WCLSRES\_BIN** = 1 if **WCLSRES** is 4 or 5;
- **WCLSRES\_BIN** = 2 if **WCLSRES** quality is 6 or 7;
- **WCLSRES\_BIN** = 3 if **WCLSRES** is greater than 7.

We note that the classes are highly imbalanced: class 0 does neither occur in FPS2 nor in FPD2, and class 1 only occurs in FPD2. The dominating class is 2, and class 3 has the second-most occurrences. Despite the lack of observations for class 0, we still cast this as a 4-class classification task to make the model applicable in settings where all four classes occur. We do not undertake any measures (e.g. class weighting) to mitigate the imbalance during the ML model training, however, this could be an approach to be explored in future work to improve the model performance.

As ML model, we choose a fully-connected neural network with one hidden layer of 1,000 units and ReLU activations, implemented in Keras [5]. For training, we use the Keras Adam optimizer on cross-entropy loss. We train for 64 epochs using a batch size of 128.

We standardize each of the four feature columns **WELCNT**, **WDRSCNT**, **WELR**, **WELCUR1\_S** to have zero mean and unit variance by subtracting the sample mean and dividing by the sample standard deviation computed on the training set.

We run three different baseline experiments which we will use later as a comparison to the performance when performing the training in a federated setting on the MUSKETEER platform:

1. We **train** the model on the training data from **FPS2** and **test** it on the test data from **FPS2** and **FPD2**. We obtain **92.68%** test accuracy on **FPS2**, **79.35%** on **FPD2**, and **87.46%** on the combined testing data from **FPS2** and **FPD2**. See Table 1, Table 2 and Table 3 for the respective confusion matrices.
2. We **train** the model on the training data from **FPD2** and test it on the test data from **FPS2** and **FPD2**. We obtain **89.35%** test accuracy on **FPS2**, **88.46%** on **FPD2**, and **89.01%** on the combined testing data from **FPS2** and **FPD2**. See Table 4, Table 5 and Table 6 for the respective confusion matrices.
3. We **train** the model on the combined training data from **FPS2** and **FPD2** and test it on the test data from **FPS2** and **FPD2**. We obtain **91.64%** test accuracy on **FPS2**, **87.03%** on **FPD2**, and **89.83%** on the combined testing data from **FPS2** and **FPD2**. See Table 7, Table 8 and Table 9 for the respective confusion matrices.

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	2,964	52
3	0	0	201	240

**Table 1: Confusion matrix when training on FPS2 and testing on FPS2**

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	29	0
2	0	0	1,479	324
3	0	0	107	289

**Table 2: Confusion matrix when training on FPS2 and testing on FPD2**

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	29	0
2	0	0	4,443	376
3	0	0	308	529

Table 3: Confusion matrix when training on FPS2 and testing on FPS2 + FPD2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	37	2,956	23
3	0	0	308	133

Table 4: Confusion matrix when training on FPD2 and testing on FPS2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	25	4	0
2	0	3	1,714	86
3	0	0	164	232

Table 5: Confusion matrix when training on FPD2 and testing on FPD2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	25	4	0
2	0	40	4,670	109
3	0	0	472	365

Table 6: Confusion matrix when training on FPD2 and testing on FPS2 + FPD2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	2	2,988	26
3	0	0	261	180

Table 7: Confusion matrix when training on FPS2 + FPD2 and testing on FPS2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	11	18	0
2	0	0	1,684	119
3	0	0	152	244

Table 8: Confusion matrix when training on FPD2 + FPS2 and testing on FPD2

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	11	18	0
2	0	2	4,672	145
3	0	0	413	424

**Table 9: Confusion matrix when training on FPS2 + FPD2 and testing on FPS2 + FPD2**

The experiments suggest that training a model on the combined training data from FPS2 and FPD2 yields better performance on average than using a model that is trained on data only from FPS2 or FPD2.

### 3.4.2 Integration with the MUSKETEER platform

In this section we explain how to use the prototype of the MUSKETEER platform to set up and execute a federated learning task that trains a ML model on the combined training data from FPS2 and FPD2 without having to centralize those data.

The attachments [12], [13], [14], [15] contain all the code that is required for executing this use case, however, because of their proprietary nature the actual data are *not* included in this deliverable, therefore the reader will not be able to run the code.

Two main customizations of the demonstrator from Section 3.3 are required to port this use case to the MUSKETEER platform:

- Modifications of the classes `neural_network.Aggregator` and `neural_network.Participant` to support the specific data preprocessing requirements and model definition for this use case; those are implemented in `neural_network_usecase.py` [12].
- Modifications of the user inputs in the notebooks for the task creator and participants; those are provided in `task_creator_usecase.ipynb` [13], `task_participant_usecase_1.ipynb` [14] and `task_participant_usecase_2.ipynb` [15].

We would like to emphasize, however, that those modifications are minimal; in future versions of the end-to-end MUSKETEER platform – with the fully developed algorithm library from WP4 and client connectors from WP7 – we expect that those modifications will only require changes to task definitions, connector configurations etc. and no changes to the actual code.

First, we explain the customization that we made in the implementations of the `Aggregator` and `Participant` classes in the `neural_network_usecase.py` module [12] to execute this use case:

- The first change is the implementation of the `load_data` function of the `Participant` class (lines 219-221), which loads the local training and testing data from the `.pkl` files prepared in the notebook [11].

- The second change is the instantiation of the ML model described in the previous section as part of the `start` function of the `Aggregator` class (lines 142-145).
- The final change is logic that we added in the `Aggregator` and `Participant` classes (lines 154-168 and 238-256, respectively) for appropriate standardization of the local training and test data. This is done via the following steps: (1) Each participant sends – for each feature – the sum of all values,  $m_i$ , the sum of squares of all values,  $s_i$ , and the total number of training samples,  $n_i$ , to the aggregator. (2) The aggregator uses this information to compute the total number of training samples  $n = (n_1 + n_2 + \dots)$ , the global feature mean  $m = (m_1 + m_2 + \dots) / n$ , and the global standard deviation  $s = \sqrt{(s_1 + s_2 + \dots - n \cdot m^2) / (n - 1)}$ , and then broadcasts  $m$  and  $s$  to all participants. (3) The participants standardize their local training data using  $m$  and  $s$ .

Besides those customizations, the logic is identical to the `neural_network.py` module used in the demonstrator on synthetic data in the previous section. We note that, through the contributions from WP4 and WP7, such hard-coded changes to algorithm implementation ultimately may not be necessary; in particular, the exact model definition and data standardization logic may be configurable through the task definition, and the logic for data loading and parsing may be configurable through data connector interfaces in the client package.

Finally, let us look at the changes in the notebooks for the task creator and participants. In the task creator notebook [13], the only noteworthy customization is the change in the task definition (see Figure 40). In particular, the "aggregator" and "participant" fields here reference the `Aggregator` and `Participant` classes in the modified Python module `neural_network_usecase.py` [12]; moreover, the task definition contains a field "point" which allows the task creator to prescribe for which welding point the ML model shall be trained. All other fields have the same meaning as in the task definition for the synthetic use case.

**Creating a new Federated ML task for this use case**

```
In [3]: # Note: task names need to be unique, so you need to use a new name for each task that you create.
task_name = 'task-000011-000000'

task_definition = {
    'aggregator': 'neural_network_usecase.Aggregator',
    'participant': 'neural_network_usecase.Participant',
    'quorum': 2,
    'rounds': 5,
    'epoch': 5,
    'batch_size': 128,
    'point': '9993'
}
```

**Figure 40: Use case task creator notebook – task definition**

In the participants' notebooks ([14], [15]), it is worth noting that the `.run` function of the `Participant` class here takes an extra argument `directory` (see Figure 41). This argument allows the task participants to specify where in their local compute environment the `.pkl` file containing the processed training and testing data is located. This file is then loaded by the `load_data` function of the `Participant` class in preparation of the federated learning process.

As can be seen by the respective values of the directory argument, participant 1 loads the data from FPS2, and participant 2 the data from FPD2.

**Running that task as participant**

```
In [13]: try:
        participant.run(context, task_name, directory='../cmas/030803_fnc_musketeer/FPS2/030801')
    except Exception as err:
        print("Error: ", err)

Using TensorFlow backend.
2025-10-08 10:49:39.774 INFO neural_network_utils 139625454872384 :: Received model architecture from the aggregator
2025-10-08 10:49:49.151 INFO neural_network_utils 139625454872384 :: Received global mean and standard deviation from the aggregator
2025-10-08 10:49:49.382 INFO neural_network_utils 139625454872384 :: Standardized local data
2025-10-08 10:49:50.681 INFO neural_network_utils 139625454872384 :: Found 3
2025-10-08 10:49:50.683 INFO neural_network_utils 139625454872384 :: Received model update from the aggregator, start to update local model and train locally.

Epoch 1/5
13831/13831 |-----| - 1s 41ms/step - loss: 0.3804 - acc: 0.8994
Epoch 2/5
13831/13831 |-----| - 0s 31ms/step - loss: 0.2352 - acc: 0.9214
Epoch 3/5
13831/13831 |-----| - 0s 33ms/step - loss: 0.2275 - acc: 0.9259
Epoch 4/5
13831/13831 |-----| - 0s 22ms/step - loss: 0.2254 - acc: 0.9257
Epoch 5/5
13831/13831 |-----| - 0s 15ms/step - loss: 0.2238 - acc: 0.9269
```

**Figure 41: Use case task participant notebook – training execution (start)**

The log messages shown in Figure 41 also report on the standardization of the local data by the global means and standard deviations computed via the federated approach described above.

From the last log messages in the participants’ notebooks, we obtain that the final ML model – dispatched to the participants after the training process has ended – achieves **92.13%** test accuracy on the data from **FPS2**, and **83.98%** on the data from **FPD2**. On the combined test data from **FPS2** and **FPD2** (taking into account the different sample sizes of 3,457 and 2,228, respectively), the accuracy is **88.94%**. The corresponding confusion matrices are shown in Table 10, Table 11 and Table 12, respectively. Compared to the baseline above, where the model was trained in a non-federated fashion on the combined training data from FPS2 and FPD2, the average accuracy is just about 1% lower. We would like to emphasize, however, that in this demonstration we did not attempt to fine-tune the hyperparameters of the ML model and the federated learning algorithm. We expect that, with more advanced algorithms developed in WP4, the accuracy of a model trained with federated learning may more closely match the accuracy of a conventionally trained model.

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	2,982	34
3	0	0	238	203

**Table 10: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPS2**

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	29	0
2	0	0	1,577	226

3	0	0	102	294
---	---	---	-----	-----

**Table 11: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPD2**

True/Predicted	0	1	2	3
0	0	0	0	0
1	0	0	29	0
2	0	2	4,559	260
3	0	0	340	497

**Table 12: Confusion matrix for federated training on FPS2 + FPD2 and testing on FPS2 + FPD2**

## 4 Conclusions

In this deliverable we presented demonstrations of data sharing and federated learning via the MUSKETEER platform. We explained key components – the cloud platform, the local platform, the Python API and the sample client package – and demonstrated interactions with the platform via Python scripts and Jupyter notebooks. We used a synthetic example and a real-world Smart manufacturing use cases to demonstrate the workflow end-to-end. The sample client package and the Python API have been released open source on GitHub under an Apache 2.0 license, thus enabling the reader to experiment with the platform functionality.

While the architecture of the MUSKETEER platform is finalized (and described in detail in D3.2), both functional and non-functional extensions and improvements are envisioned towards the final version of the platform. This may include, e.g., more advanced features for managing models created via federated learning tasks or for storing information about the estimated value of the data contributed by different task participants, as well as refinements of the platform APIs to improve the usability for algorithm and application developers.

## 5 References

- [1] <https://github.com/IBM/Musketeer-Client>
- [2] <https://github.com/IBM/pycloudmessenger>
- [3] <https://cloud.ibm.com/>
- [4] <https://flask.palletsprojects.com/en/1.1.x/>
- [5] <https://keras.io/>
- [6] <https://www.vagrantup.com/>
- [7] <https://www.virtualbox.org/>
- [8] <https://brew.sh/>
- [9] <https://jupyter.org/>
- [10] <http://yann.lecun.com/exdb/mnist/>
- [11] D3.3 attachment: welding\_data\_use\_case\_D3.3.ipynb
- [12] D3.3 attachment: neural\_network\_usecase.py
- [13] D3.3 attachment: task\_creator\_usecase.ipynb
- [14] D3.3 attachment: task\_participant\_usecase\_1.ipynb
- [15] D3.3 attachment: task\_participant\_usecase\_2.ipynb
- [16] <https://www.internationaldataspaces.org/wp-content/uploads/2020/01/IDSA-Strategy-paper-certification-scheme-V.2.pdf> p.13 et seq.