This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare

Deliverable D7.3
Federated machine learning apps running in app store

Work Package
WP7 Integrated FeatureCloud health informatics platform and app store

Disclaimer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078. Any dissemination of results reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Copyright message

Document information

Grant Agreement Number: 826078			**Acronym: FeatureCloud**	
Full title	Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare			
Topic	Toolkit for assessing and reducing cyber risks in hospitals and care centres to protect privacy/data/infrastructures			
Funding scheme	RIA - Research and Innovation action			
Start Date	1 January 2019	**Duration**	60 months	
Project URL	https://featurecloud.eu/			
EU Project Officer	Christos MARAMIS, Health and Digital Executive Agency (HaDEA) - Established by the European Commission, Unit HaDEA.A.3 – Health Research			
Project Coordinator	Jan BAUMBACH, UNIVERSITY OF HAMBURG (UHAM)			
Deliverable	D7.3 Federated machine learning apps running in app store			
Work Package	WP7 Integrated FeatureCloud health informatics platform and app store			
Date of Delivery	**Contractual**	31/12/2021	**Actual**	17/12/2021
Nature	Demonstrator	**Dissemination Level**	Public	
Lead Beneficiary	01 UHAM			
Responsible Author(s)	Julian Matschinske, Niklas Probul, Mohammad Bakhtiari, Jan Baumbach, Nina Wenke & Christina Saak (UHAM)			
Keywords	Platform, Federated Apps, SMPC			

History of changes

Version	Date	Contributions	Contributors (name and institution)
V0.1	1/11/2021	Draft 1	Julian Matschinske, Mohammad Bakhtiari & Niklas Probul (UHAM)
V0.2	8/11/2021	Comments 1	Balazs Orban & Sandor Feyer (GND)
V0.3	24/11/2021	Draft 2	Julian Matschinske, Mohammad Bakhtiari & Niklas Probul (UHAM)
V0.4	6/12/2021	Comments 2	Dominik Heider (UMR)
V0.5	13/12/2021	Draft 3	Julian Matschinske & Niklas Probul (UHAM)
V1	15/12/2021	Final version	Jan Baumbach, Nina Wenke & Christina Saak (UHAM)
V1	16/12/2021	Submission	Miriam Simon (concentris)

Table of Content

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

A) Objectives of the Deliverable

Deliverable 7.3 “Federated machine learning apps running in app store” relates to task 1 “Programming interfaces and platform” and task 2 “App store and workflow management” of work package 7 as described in the Description of Action. In addition, it touches upon user interfaces, testing and evaluation (tasks 3 and 4). Like the previous deliverable 7.2, this deliverable contains all progress made in the past reporting period (months 24 - 36) related to the FeatureCloud platform and AI Store, including the overall system with additional and updated descriptions, figures and documentation.

B) Executive Summary

This deliverable has been split into five parts, looking at the advances from different angles: app usage (end-user perspective), application development and testing (developer perspective), evaluation (performance perspective), privacy enhancements (privacy perspective), and the system and implementation (technical perspective).

D7.2 already provided the basic functionality to develop apps, distribute them using the AI Store and execute them using the FeatureCloud controller. All of these parts have been extended to increase convenience for users and developers (see sections 1 and 2). In terms of evaluation, a flexible workflow has been created to assess the accuracy and performance of the platform (section 3). The FeatureCloud API has been extended to allow for implementation of more advanced privacy-enhancing techniques inside the apps themselves, as well as providing such techniques in the FeatureCloud system to gain more control over their execution (see section 4). In this context, the foundation has been laid to allow for integration of techniques to detect malicious behaviour developed by WP2 (see section 5).

C) Results

1 App Usage and Execution

Apps, as have been introduced in D7.2, are implementations of algorithms that are being executed locally in an isolated fashion (see section 1.3) and can be combined into a workflow (D7.2, section 3.3). An example workflow composed of apps available in the AI Store can be found in section 3.1. In this section, we provide information about how changes to the API (see section 5.2) and the overall FeatureCloud system impact the usage and execution of apps. The general project management process has been kept as described in D7.2. Figure 1 outlines this process again, similar to Figure 8 in D7.2, from an actor perspective.

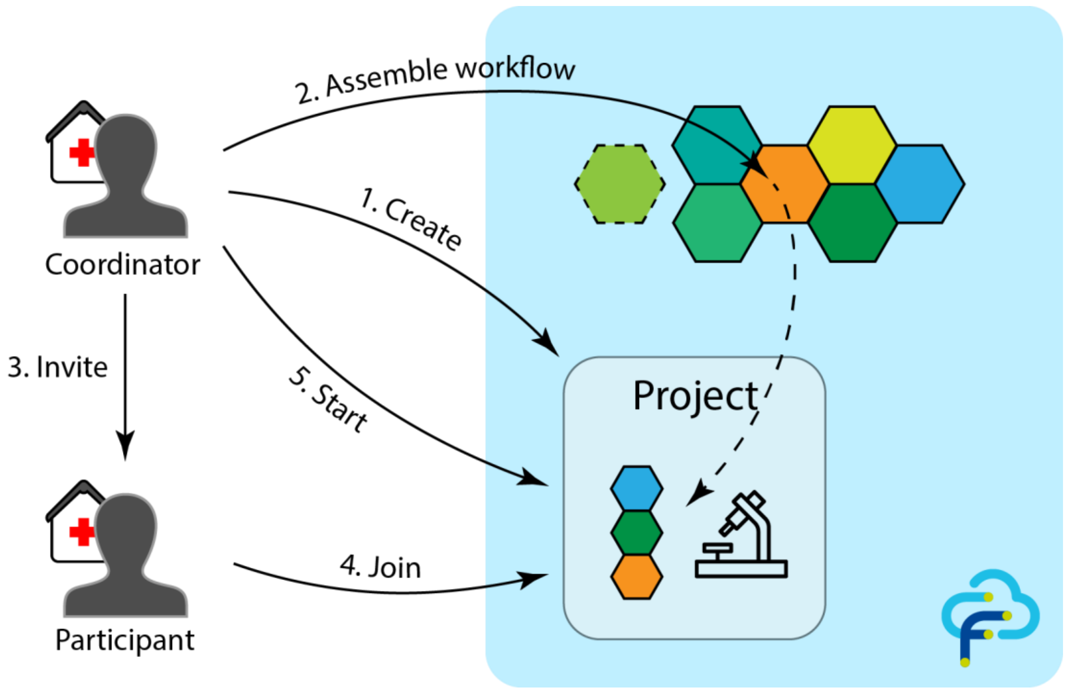
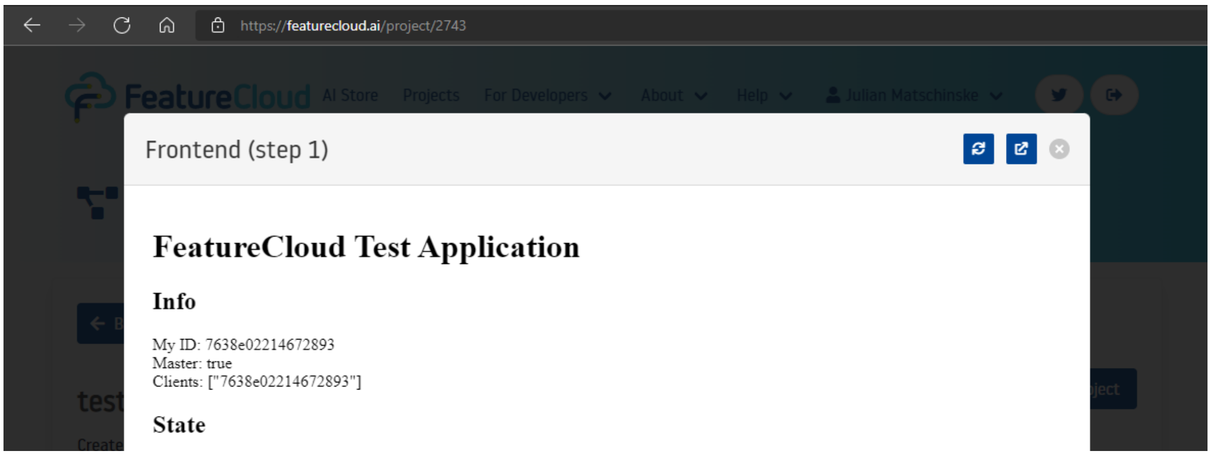


Figure 1. Creating and setting up a federated project. *One of the partners takes the role of the coordinator and creates a new project on the website (1). After that, the coordinator defines the workflow by adding the apps to the project's workflow (2). Then, the other partners are invited by sending a randomly generated token to each of them (3), which is unique and allows for joining the project (4). When all partners have joined, the coordinator triggers the execution on the FeatureCloud website and the workflow runs (5). During workflow execution, active interaction with the end-user can be required, depending on the apps.*

1.1 App Integration

Apps could already provide a graphical user interface (GUI) in the form of an app frontend. This frontend had to be opened in its own window and opened in a new browser tab. In order to integrate these apps better, hide technical details (e.g., frontend URL) and increase the user experience, their frontends are now embedded into the FeatureCloud frontend. They can be accessed directly on the project page (see Figure 2). This way, we further aim to remove the visible boundary between the FeatureCloud system and app implementations, which can be confusing to end-users.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.



FeatureCloud Test Application

Info

My ID: 7638e02214672893
Master: true
Clients: ["7638e02214672893"]

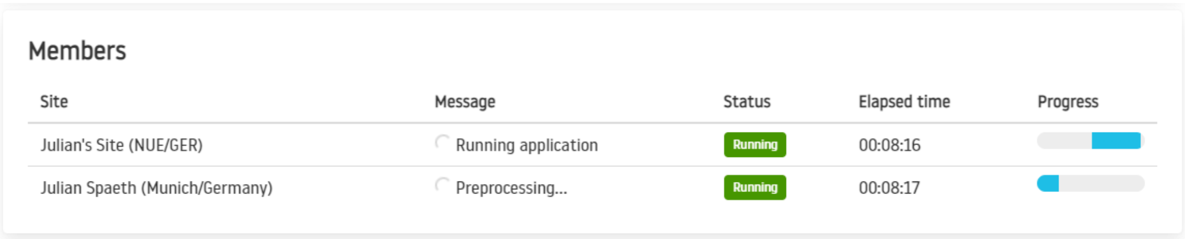
State

Figure 2. App frontend embedded in the FeatureCloud GUI using a modal dialog. *Like app logs, app frontends can be opened instantly inside the project page, hiding technical details such as the endpoint and providing a better user experience.*

Apps developers can now also provide the FeatureCloud system with additional information about the current execution status. They can now report

- Progress (a number between 0.0 and 1.0)
- Message (text of up to 40 characters)
- State (one of ‘running’, ‘error’, ‘action_required’)

during the execution in the app API (see section 5.2). This information is used to display a meaningful progress page and instantly inform the user about potential problems (see Figure 3).



Members

Site	Message	Status	Elapsed time	Progress
Julian's Site (NUE/GER)	Running application	Running	00:08:16	
Julian Spaeth (Munich/Germany)	Preprocessing...	Running	00:08:17	

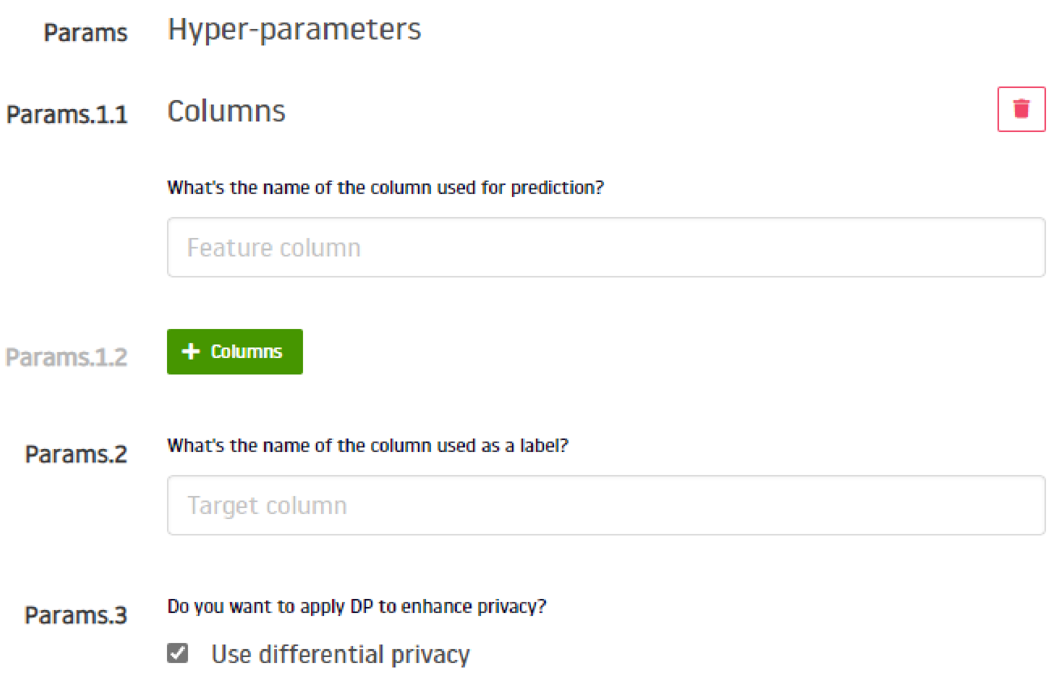
Figure 3. Project members overview providing feedback about the app executions. *The coordinator can see the progress for each member, its message and state.*

Apps usually require additional configuration parameters in order to run as expected (e.g., number of trees in a random forest, portion of test data, ...). This information is currently being put inside a configuration file that contains one section per app in the workflow. For less technical users, this could pose a barrier. In the AIMe[1] side-project, we implemented a specification language to define a form. This language will now be used to specify a parameter form that can be directly placed on the workflow page as well. Listing 1 shows a snippet of the specification language and Figure 4 shows the corresponding form that has been rendered from it.

[1] https://aime-registry.org/

```yaml
title: Hyper-parameters
id: HP
type: complex
children:
  - id: "1"
    title: Columns
    type: list
    default: []
    child:
      type: string
      default: ""
      title: Feature column
      question: What's the name of the column used for prediction?
  - id: "2"
    type: string
    default: ""
    title: Target column
    question: What's the name of the column used as a label?
  - id: "3"
    type: boolean
    default: true
    title: Use differential privacy
    question: Do you want to apply DP to enhance privacy?
```

Listing 1. YAML-based specification language. *This sample snippet shows how parameters can be specified for apps, including their type and default value.*


Params Hyper-parameters

Params.1.1 Columns

What's the name of the column used for prediction?

Feature column

Params.1.2 + Columns

Params.2 What's the name of the column used as a label?

Target column

Params.3 Do you want to apply DP to enhance privacy?

☑ Use differential privacy

Figure 4. HTML-form rendered from YAML specification. *This form has been generated from the specification shown in Listing 1.*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

1.2 App Types and Automated Testing

Apps can implement pre-processing, ML, post-processing, evaluation and other types of apps. In the AI Store, we currently distinguish between ‘Pre-processing’, ‘Analysis’, and ‘Evaluation’. An example workflow consisting of 2 pre-processing apps, 1 model training app and 1 evaluation app can be found in section 3.

While we encourage developers to implement their application such that they are compatible with existing apps, this is not being enforced currently. We are therefore exploring the possibility of automatically testing and embedding apps in a workflow context. For that, before apps are considered for certification or even shown in the AI Store, they could be built by the FeatureCloud system (e.g., as an extension to the backend). If the apps can be built without errors (first check), it is automatically executed and provided with input data formatted according to data formats we encourage to use. Conversely, the output should follow a certain structure as well. Both can be assessed by looking at the output of the app. We are currently in the conceptualization phase and are aiming to provide this as a feature similar to CICD pipelines known from GitHub or GitLab and are going to report on it in D7.6 and D7.7.

1.3 App Isolation

For security reasons, we isolate the apps as much as possible from the host system. We achieve this by running apps as Docker containers. In particular, direct access to the file system as well as to the internet is not allowed to any running app container, as described already in D7.2. Since apps can provide a frontend, which is run inside the browser, there is a potential security problem: Apps could request the user to open the frontend, funnel sensitive data via the internal frontend API to the browser and transfer the data to an external endpoint using the browser as man in the middle. This is now being prevented using Content Security Policy (CSP)[2]. The Controller, which sits between the frontend API and the browser, sets an HTTP header to instruct the browser to refuse any connections to URLs other than the local controller URL.

2 Development and Testing

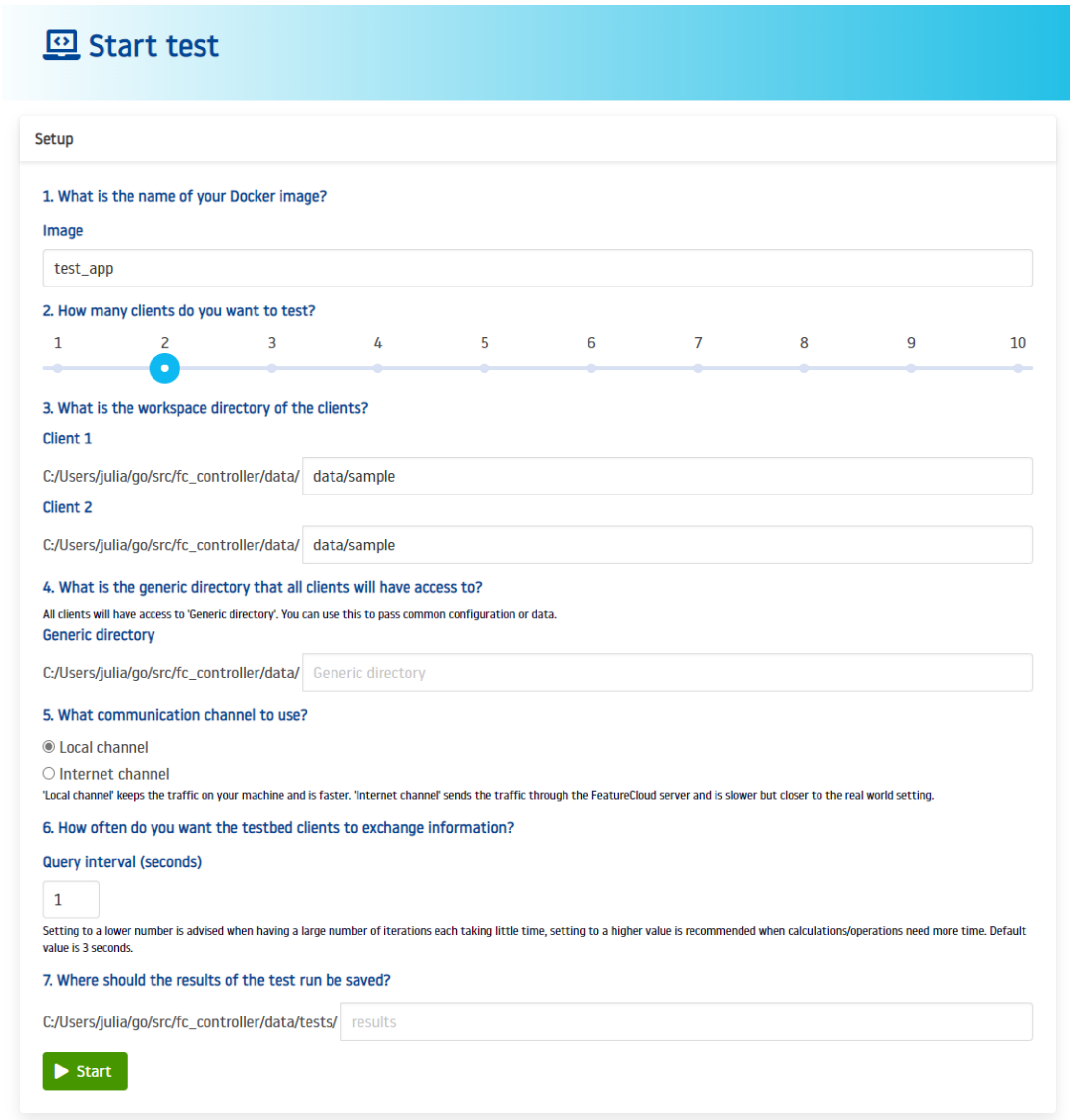
One of the core aspects of FeatureCloud remains the provision of developer tools that increase robustness, performance and speed of federated app development. Fast and convenient debugging cycles are crucial in this context. This involves the beginning of the implementation, for which we offer an app template, and the consecutive implementation, for which we provide a simulation tool.

2.1 App Simulation

In order to accelerate the development of apps, it needs to be possible to regularly test and debug its implementation. The app simulation tool has been extended for this purpose and now displays the new message, progress and state information. Also, the simulation tool now allows for specifying a common input directory, whose contents are put in all test input volumes, and modifying the output directory for the test results. This increases flexibility and testing capabilities, particularly in combination with the newly developed CLI (see section 2.3 and supplement).

[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.


Start test

Setup

1. What is the name of your Docker image?

Image

test_app

2. How many clients do you want to test?

1 [✓ 2] 3 4 5 6 7 8 9 10

3. What is the workspace directory of the clients?

Client 1

C:/Users/julia/go/src/fc_controller/data/ data/sample

Client 2

C:/Users/julia/go/src/fc_controller/data/ data/sample

4. What is the generic directory that all clients will have access to?

All clients will have access to 'Generic directory'. You can use this to pass common configuration or data.

Generic directory

C:/Users/julia/go/src/fc_controller/data/ Generic directory

5. What communication channel to use?

● Local channel

○ Internet channel

'Local channel' keeps the traffic on your machine and is faster. 'Internet channel' sends the traffic through the FeatureCloud server and is slower but closer to the real world setting.

6. How often do you want the testbed clients to exchange information?

Query interval (seconds)

1

Setting to a lower number is advised when having a large number of iterations each taking little time, setting to a higher value is recommended when calculations/operations need more time. Default value is 3 seconds.

7. Where should the results of the test run be saved?

C:/Users/julia/go/src/fc_controller/data/tests/ results

Figure 5. Testbed form has been extended. *Under section 4 you can now specify the general input directory, under section 7 the output directory can be specified. These options are also available via the new CLI.*

2.2 App Template

App developers should be able to quickly implement a new app starting from a scaffold and ideally only put the custom logic inside. Code dealing with serialization, interacting with the controller API, reading config information, etc. is very similar across all app implementations and can therefore be provided in a well-structured and documented app template.

2.2.1 State Machine Concept

We have investigated previous app implementations and came to the conclusion that almost all of them implement some kind of state machine: each app instance is in a certain state, usually synchronized across the workflow, while the coordinator can have its own states and manages the transitions. State machines are a well-investigated and commonly used concept for applications of

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

different levels of complexity, both in theoretical and applied computer science. They provide a logical structure and help across all phases of development: conceptualization, implementation, testing and debugging.

For this reason, we integrated the state machine concept into the template and added some assisting functionality, such as rendering state machine diagrams directly from the code (see Figure 6). For that, app developers need to extend an AppState class and implement its abstract methods register and run (see Listing 2). We use this information to infer a status message from the current state (e.g., ‘throw_die’) and display this information automatically on the FeatureCloud frontend. App developers can override this message by calling the `update` method.

```
@app_state('throw_die', Role.BOTH)
class DieState(AppState):
    def register(self):
        self.register_transition('aggregate', Role.COORDINATOR)
        self.register_transition('obtain', Role.PARTICIPANT)

    def run(self) -> str or None:
        self.update(progress=0.25)
        d = random.randint(1, 6)
        self.app.log(f'threw a {d}')
        self.configure_smpc(exponent=6, operation=SMPCOperation.ADD)
        self.send_data_to_coordinator(d, use_smpc=USE_SMPC)

        if self.app.coordinator:
            return 'aggregate'
        else:
            return 'obtain'
```

Listing 2. Sample app state implementation. *The* `register` *method requires developers to indicate which transitions are possible and for which role (coordinator/participant). The* `run` *method contains the actual app logic.*

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

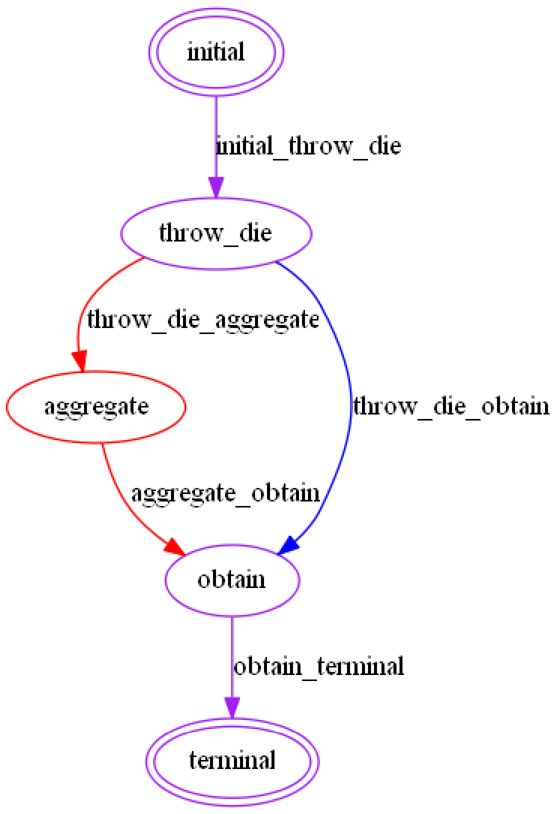


Figure 6. State diagram for the throw die example app. *Purple, red and blue colours indicate valid states and transitions for both roles, only the coordinator or only participants, respectively. Names for transitions and states are inferred from the code (see Listing 2) so that no additional effort is required from the developers.*

Every app should include a couple of files that more or less contain the same information or codes, regardless of their application. Among those, main.py is the most important one, which includes states. It always imports `api_server` and `web_server` from FeatureCloud API package to bind with the bottle app[3] (see Listing 3).

```
from bottle import Bottle
from api.http_ctrl import api_server
from api.http_web import web_server
import apps.examples.dice
from engine.app import app

server = Bottle()
```

Listing 3. The dice example app is imported from the FeatureCloud apps package. *Importing the app triggers the `@app_state` decorators to register the states and tie them to the app.*

[3] https://bottlepy.org/docs/dev/

Importing the states or defining them in the `main.py` file registers them into the app instance. Each app includes at least one and normally multiple states. All of the states should be registered for the same app instance. Accordingly, developers should always use app instances in the FeatureCloud engine package.

2.2.2 Roles, States, and Log Levels

Every client in the FeatureCloud platform should run an app instance, which can be either participant or coordinator. Generally, the coordinator can handle both local updates and global aggregations, which entails having access to the locally trained models. On the other hand, the participant role is restricted to local computations. These roles are not mutually exclusive, and developers can use three constants `COORDINATOR`, `PARTICIPANT`, and `BOTH` to assign a role to each state and state transition. Developers are expected to consider clients' roles when defining states and the possible transitions between them. FeatureCloud template includes a verification mechanism to ensure that clients' roles agree with states and transitions logic during execution.

Once states are executing, any exceptions or errors can happen, which the app will handle automatically. For reporting the situation for the front-end app to inform the end-users, developers can communicate `RUNNING`, `ERROR`, or `ACTION_REQUIRED` to the controller:

- `RUNNING`: the app is functioning normally
- `ERROR`: app execution is interrupted with an error and cannot recover from it. Consequently, app execution will stop
- `ACTION REQUIRED`: This expresses the demand for end-users' intervention. It is specifically provided for interactive federated learning or data analysis apps

For proper logging and reporting to the front-end, developers can employ `DEBUG`, `ERROR`, and `FATAL` logging levels to facilitate the debugging and reporting process. For debugging and possible error messages, developers may use `DEBUG` and `ERROR`, respectively. For `FATAL`, like `ERROR`, it can log erroneous events that the app may encounter during the execution but cannot recover from. The app execution stops in case of a fatal error.

2.2.3 App Class

In the FeatureCloud engine package, the `App` class is the central piece, responsible for state registration, transition, and execution. Despite acting as an interface between the app and controller and managing the app execution, App is a highly transparent class that demands minimum developer knowledge and interaction. In fact, developers are not obliged to be familiar with the `App` class; however, there is a verification mechanism in both `App` and `AppState` classes that developers should be aware of, which includes registering states and transitions by assigning role/s that are responsible/allowed to execute states or take transitions. The `App` class automatically checks the logic to ensure semantic errors in defining the workflow are minimized.

Each app should contain and start with an initial state. On the other hand, each app, by default, includes the terminal state that has no task or operation to accomplish other than explicitly marking the final state in the app. Once a state transitions to the terminal state, that state should be considered one of the app's possible exit states. The FeatureCloud app includes various methods that provide the ability to flexibly incorporate different states into the app and work as part of the verification mechanism. Once all the states are registered and ready to run, `app.register()` should be called to register all the transitions.

When the app instance is running, different errors may happen, or various results may be produced. Thereby, the app instance may need to communicate with the controller or front-end parts of FeatureCloud which can be easily done through helper functions in AppState class. Developers can use status attributes in the `App` class to send messages between the app container to the controller and/or indirectly with the front-end.

Availability of data to communicate. Once a client wants to communicate with other clients, regardless of role, and the data is ready, by setting `app.status_available` to `True`, the app instance sends the signal to the controller to execute the communication. Generally, this attribute will be used for communication methods and automatically handled by the FeatureCloud app.

Termination of app execution. The app instance can set the `app.status_finished` attribute as True to signal the controller that app execution is finished. Generally, this attribute will be set as True by the FeatureCloud app once the app enters the terminal state or some exceptions happen during the app run.

Messaging to the frontend. Once there is a specific message, e.g. the occurrence of some semantic errors, the app instance can use `app.status_message` to inform the end-user in the frontend. For sending messages to the frontend, developers can use app.update.

Overall progress of the app. During the run, app execution progress can be quantified based on different factors. Developers can quantify the app progress in the range of zero to one and share it with the end-user through the front-end using `app.update`.

Operational state of the app. During the app run, different operational states can be reported to the end-user using `app.update`.

Messaging to other clients. Once clients want to communicate with another client, they should provide the ID of the target client for the coordinator. Developers should use the destination argument in communication methods for this purpose and `status_destination` will be accordingly and automatically handled by the app instance.

Desired configuration of SMPC component. App developers can decide which parameters should be used for SMPC aggregation, and they can inform the controller about the configuration using `app.configure_smpc`.

Shared memory for states. Different states can be defined and registered to the app, and they may need to pass data to each other. The `App` class has an internal attribute, a dictionary that can be accessed through self.app.internal in each state to support a shared memory between different states.

2.2.4 Example Apps

In general, for developing apps in the FeatureCloud platform, apps should communicate with the FeatureCloud controller. For this purpose, app developers have multiple options that all include employing the `FeatureCloud.engine` package, which provides the basic means. Most simplistically, they can extend the `FeatureCloud.engine.app.State` class to define new custom states and use the `app_state` handler to register their state inside the app. FeatureCloud apps can

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

support different states and various communications; moreover, they can be used inside a workflow in conjunction with other apps. Inside each workflow, every app gets the input files from the output of the previous app in the workflow, except for the first one, which gets data from end-users. Accordingly, we use a convention that facilitates providing acceptable results for other accompanying apps in a workflow. Meanwhile, it increases readability and facilitates debugging. One of the simplifying ways for providing expected results for accompanying apps are using the `ConfigSate` from the FeatureCloud engine which also exemplifies how to extend `AppState` to define different levels of abstractions that can be used in multiple apps.

We provide 4 sample implementations or scaffolds[4] illustrating the capabilities of the FeatureCloud template, each of them being commented extensively to provide help to the developers:

- **Blank** - Blank scaffold for new apps
- **Throw die** - Simple state machine with different states for participants and controller
- **Library** - Implementation demonstrating AppState extensions
- **Round** - Sample app demonstrating peer-to-peer communication

2.3 Command-line Interface

While the FeatureCloud app simulation (see section 2.1) is easily accessible through a graphical web frontend, developers are often used to performing tasks reproducibly through a command-line interface (CLI). Therefore, we now provide a compact FeatureCloud CLI to allow for controlling app testing from the terminal. By this, developers can trigger different run scenarios for their apps inside bash scripts or from various programming languages, programmatically define different input data or parameters, and verify the test results using the CLI commands.

The FeatureCloud CLI will be further extended and made available as a pip package that can be installed globally. Its commands have the following shape:

```
<scope> <command> -param1 value1 -param2 value2 ...
```

To start a test, the command would be

```
test start --client-dirs ./test1,./test2 --app-image test_app
```

Here, the scope is `test`, the command is `start` and the parameters are `client-dirs` and `app-image`.

After the test run starts successfully, the test id will be returned. If something went wrong, the corresponding error message occurs, e.g. `{"detail":"Error: No such image: test_app"}`

A list of the currently available commands can be found in the supplement.

The CLI uses the same endpoints as the front end. Therefore, every action performed using the CLI will also be shown in the frontend and vice versa. The CLI is especially useful for developers of WP5

[4] https://github.com/FeatureCloud/app-template/tree/master/apps/examples

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

from SDU, who are developing unsupervised ML apps for FeatureCloud, which require regular and extensive testing.

In the future, we aim to merge the app template library and the CLI into a single pip package, such that all tools needed by developers are provided by the all-purpose pip package ‘featurecloud’.

3 Evaluation

We can distinguish two different types of evaluations needed in the context of machine learning apps in the FeatureCloud AI Store:

1. **Evaluation of an analysis performed by the end-user.** Here, similar to central ML pipelines, the performance of the trained model needs to be validated. The performance here is mainly dependent on the data that is used for training.
2. **Evaluation of the app algorithm itself, performed by the app developer.** Here, the app developer compares the performance and runtime of the federated algorithms with the central algorithm. This evaluation is data-independent. It should show that similar or identical results are achieved if the same dataset is used for the central and federated algorithms (if they were merged).

The following sections will describe how FeatureCloud enables both evaluation types to allow state-of-the-art machine learning workflows with apps from the AI store.

3.1 Evaluation Pipeline for End-Users

To evaluate the performance of a machine learning model and detect misbehaviours, such as underfitting or overfitting, a simple training of the machine learning model on training data is not enough in practice. The trained models need to be evaluated on an unseen set of data to make sure that models generalize well and do not only perform well on the already seen training data.

For this, FeatureCloud offers several apps in the AI store that make it easy to evaluate a machine learning model. As in medicine sample size and therefore data samples are the bottleneck, we support a cross-validation app that splits the local data into various splits. Every upcoming analysis app will perform the analyses on each of these splits in parallel. In the end, an evaluation app can calculate the corresponding scores like sensitivity, specificity, accuracy, or the Matthews correlation coefficient to determine the model’s performance through cross-validation.

Figure 7 shows an example of such a workflow. The cross-validation app (purple) splits the local data into three splits and creates the corresponding train and validation data. After that, the normalization app (green) performs a normalization (e.g., standardization) on each data split. After that, an analysis app (e.g., Linear Regression) performs a regression on the normalized data splits. Finally, the evaluation app evaluates the model by calculating different scores for each split and visualizes these scores to validate the model performance.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

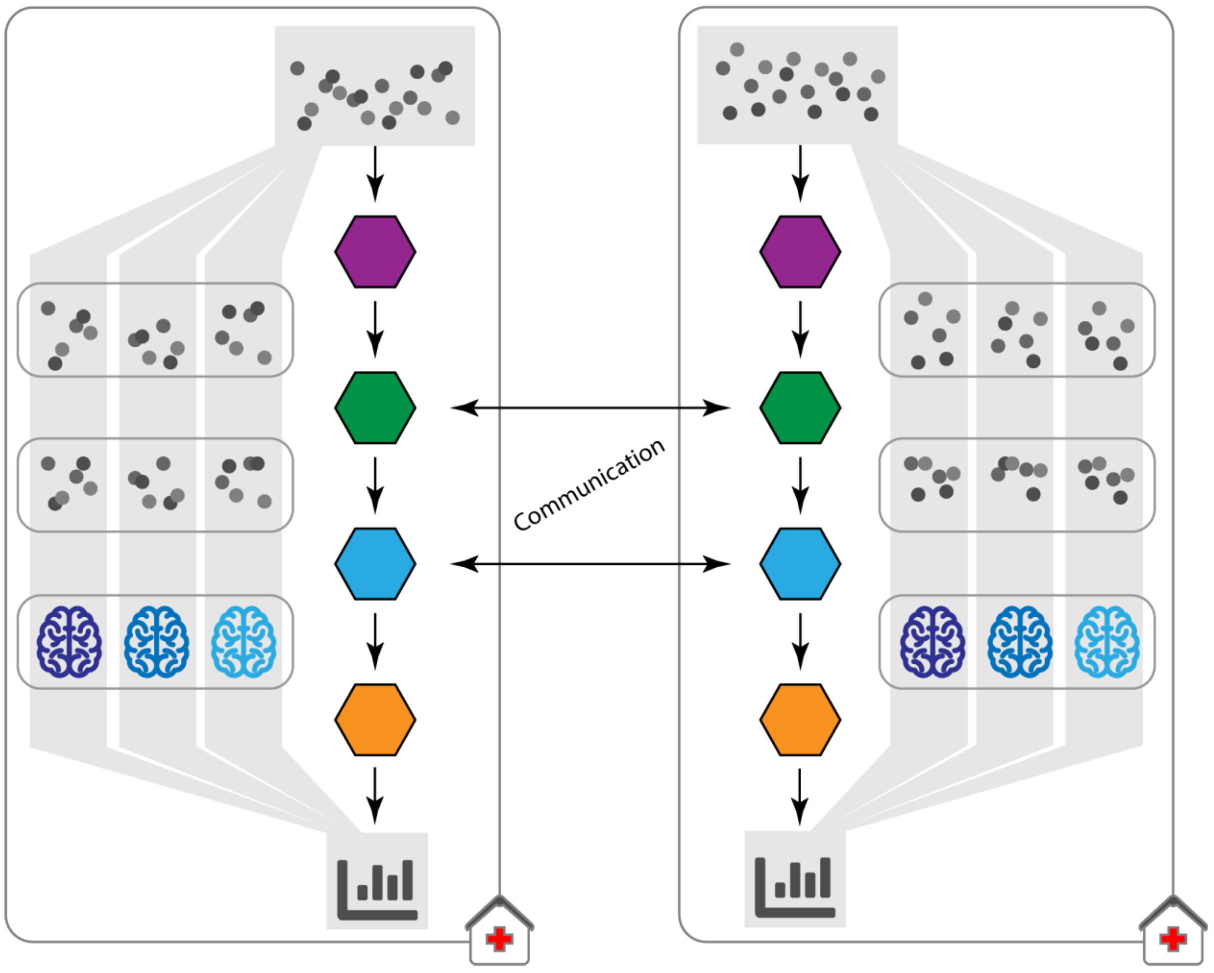


Figure 7. Workflow structure used for evaluation. *The first app (purple) creates splits for cross-valuation. All following apps perform their tasks on each split individually, in a federated fashion, only transmitting model parameters. The grey dots represent intermediate training/test data. The second app (green) performs normalization, and the third (blue) trains the models, generating a global model. The global model is evaluated in the evaluation app (orange). The evaluation results are finally aggregated to obtain an evaluation report based on the initial CV splits.*

The workflow approach in FeatureCloud and the cross-validation app and evaluation apps allow a state-of-the-art evaluation of machine learning algorithms in FeatureCloud for the federated analysis.

3.2 Evaluation of the AI Store apps

Representing the many apps in the AI store, we evaluated the performance and runtime of the FeatureCloud apps of four commonly used algorithms in machine learning: logistic regression (LR) and random forest (RF) for classification tasks and regression tasks. Each evaluation of the federated apps was run in the workflow that was previously described in section 3.1.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

3.2.1 Performance

Figure 8 shows the performance of the FeatureCloud apps on different datasets (subfigures) for the centralized algorithm (orange), the federated FeatureCloud app (blue), the individual training on each site on a central, common test dataset (dark grey) and the individual training on each site with local test data only.

As we can see in the Figure, the federated logistic regression app and the federated linear regression app perform identically to their centralized counterparts. The federated Random Forest does not perform identically but is comparable to their centralized counterparts. This is because the app does not compute each tree in a federated fashion. Each participant computes a forest on its own data that are finally merged and weighted into a global forest. Therefore, identical results are no longer possible, but the results have shown similar performance.

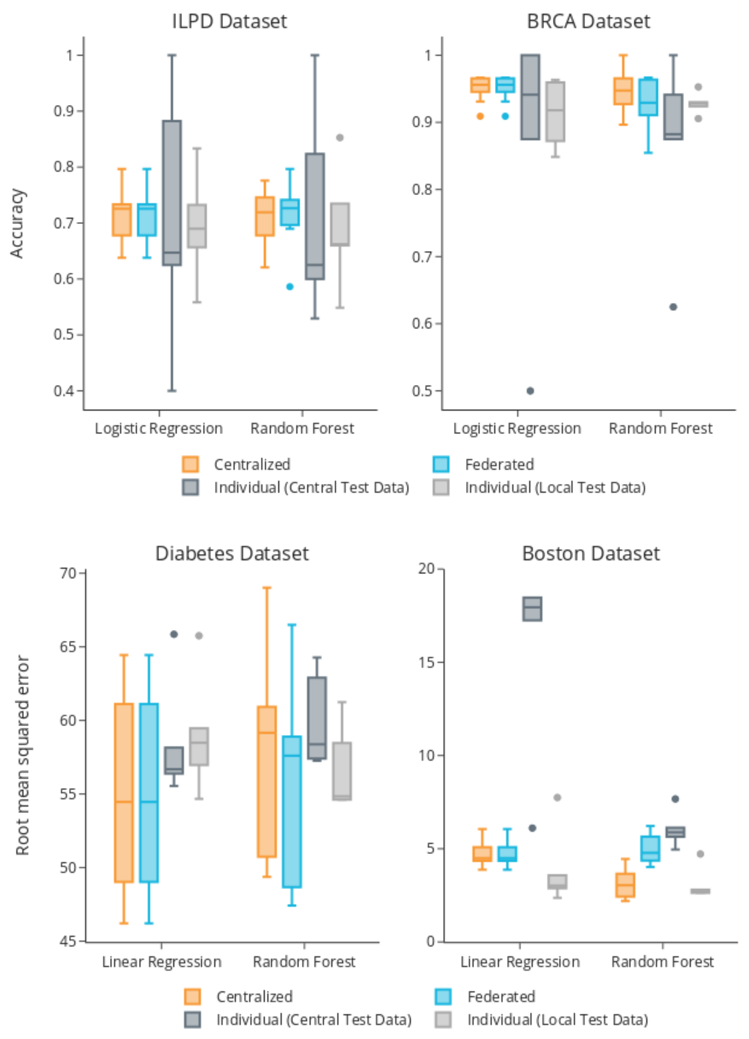


Figure 8. Performance evaluation of federated AI methods. *The boxplots show the results of a 10-fold CV for the different classification and regression models and datasets in multiple settings. The centralized results are shown in orange, the corresponding federated results in blue and the individual results obtained locally at each participant in grey. Each model was evaluated on the entire test set (dark grey) like the centralized and federated models, and on the individual (local)*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

parts of the test set (light grey). The federated logistic and linear regressions perform identically to their centralized versions and the federated random forest performs similar to its centralized version.

3.2.2 Runtime and Network Traffic

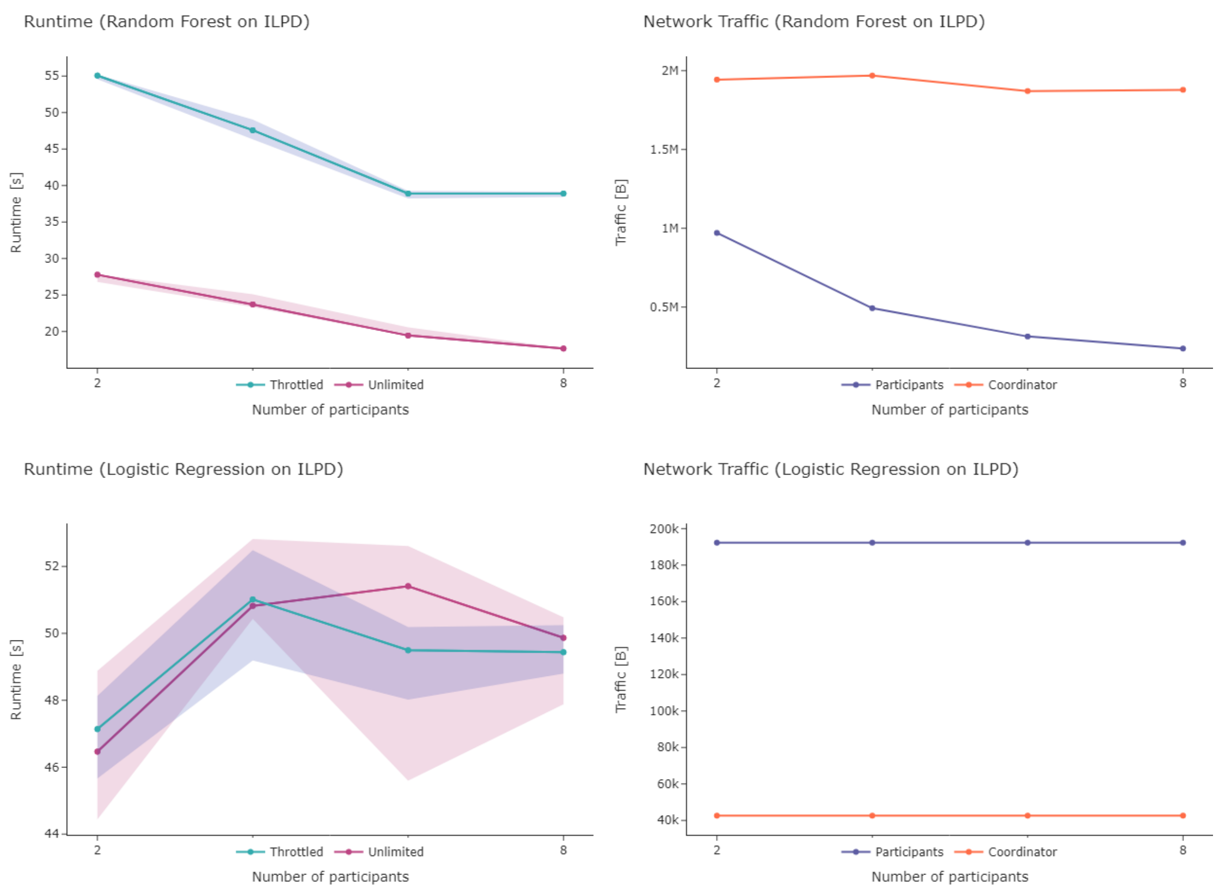


Figure 9. Runtime and network traffic. *The left plots show runtime for unlimited and throttled connections, the right plots show network traffic for coordinator and participants evaluated on the Indian Liver Patient Dataset. The lines represent the median values measured over 10 executions. The areas show the 25% and 75% quartiles to illustrate variance across the executions.*

4 Privacy

Privacy is one of the crucial aspects that FeatureCloud has to consider. While federated learning usually already provides a significantly higher level of privacy, it cannot generally be ruled out that the transmitted model parameters reveal information about the raw data. App developers can integrate additional privacy-enhancing techniques (PETs) such as differential privacy (DP), as suggested in D2.4. However, these implementations need to be verified manually and still leave the risk of flaws in the implementation, bugs, malicious intent etc. For this reason, the FeatureCloud consortium constantly evaluates possible extensions to the FeatureCloud system itself. Differential privacy of the app outputs (i.e., the learned model parameters) will thus be provided as an additional app that can be directly applied to the local workflow. In addition, modules for stochastic gradient descent will be available (cf. Section 5.3).

This section contains information about changes to the communication and other logic implemented in the FeatureCloud system and relates it to privacy considerations. Additive secret sharing is discussed as a first example for a privacy-enhancing nodule inside FeatureCloud.

4.1 Peer-to-peer Communication

The existing star-based communication logic (see D7.2, section 2.2) allowed for common use-cases in federated learning, where a global model is continuously being updated by aggregating local updates in one common model. Sending data to a single participant from the coordinator was not possible so far, let alone peer-to-peer (P2P) communication. This imposed restrictions in terms of the available range of privacy-enhancing techniques that can be integrated into app implementations. A possible workaround was emulating P2P communication inside this architecture which caused a significant increase in network traffic.

We therefore integrated P2P communication into the FeatureCloud system. While all traffic is still routed through the relay server, it logically provides a secure P2P channel. Please note that the relay server is a FeatureCloud system component (see 13) and has to be distinguished from the coordinator app instance, which is not involved here. In P2P communication, all app instances behave the same and their role (coordinator/participant) is of no significance.

To hide the traffic from the relay server, asymmetric encryption is being used, as described in section 4.2. Figure 10 shows the process of transmitting data from one participant to another via the relay server.

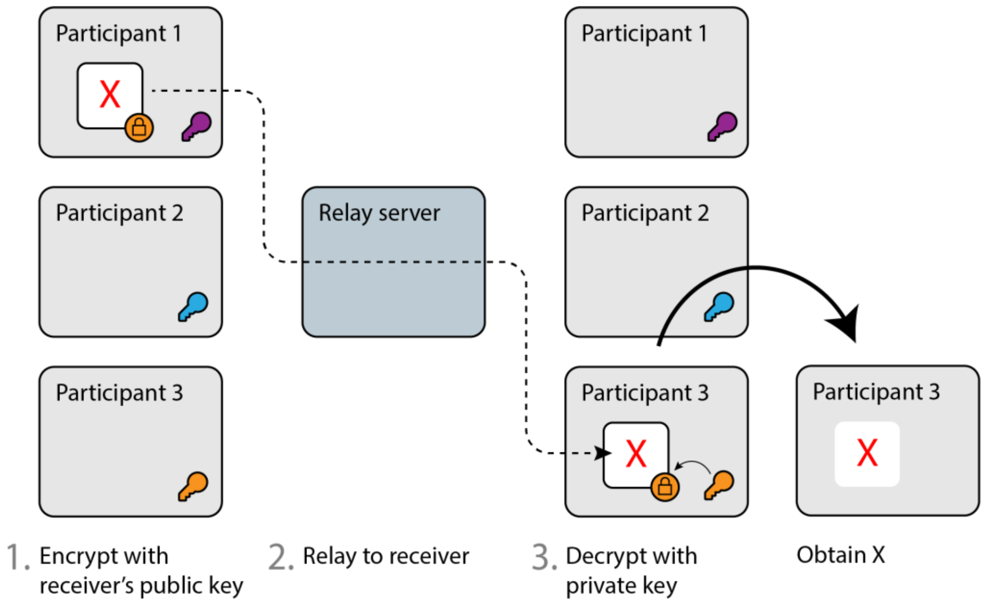


Figure 10. Peer-to-peer communication built on top of star architecture. *In step 1, participant 1 encrypts value X using the public key of participant 3. In step 2, the relay server sends the encrypted value to participant 3, which decrypts it in step 3.*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

In order to allow for P2P-communication, the API was extended. From an app developer's perspective, it is very simple to use this piece of functionality by specifying the destination client ID (see Listing 4).

```
{
      "available": true,
      "destination": "38b0da293e4ed6c1"
}
```

Listing 4. P2P API extension (JSON). *Specifying a destination ID causes the FeatureCloud system to deliver the data to the identified participant (here* `38b0da293e4ed6c1`*). The usual gather/broadcast behaviour is not applied in this case.*

4.2 Encryption

To hide the data from the relay server and other potential eves droppers, we apply a state-of-the art combination of asymmetric and symmetric encryption: First, a symmetric key is generated and used to encrypt the data. Before actually encrypting the data, a nonce (sequence of 12 random bytes) is prepended to make previously sent data indistinguishable from new data when comparing the ciphertexts. The symmetric key is then encrypted using the public key of the destination client.

Symmetric encryption. For the symmetric encryption, we apply the Advanced Encryption Standard (AES) with a 256-bit key.

Asymmetric encryption. For the asymmetric encryption part, we make use of elliptic hash curves, using the P-224 curve (see FIPS 186-3, section D.2.2). We decided to use this technique instead of the more popular RSA standard, because we can significantly reduce the key sizes of the involved participants that way.

4.3 Additive Secret Sharing

One of the crucial steps in FL is aggregating local models from multiple participants. This leads to an imbalance of required trust: while every participant will be able to see the aggregated model after an aggregation step, only the coordinator knows all individual models. To address this problem, an adapted additive secret sharing technique has been implemented. Each participant splits its local model into *n* pieces or *secrets*, a masked model ($M - r_1 - \dots - r_{n-1}$), and the masks $r_1, \dots, r_{n-1}$ which are equally distributed random values. Those secrets are then distributed to the other parties. They, in turn, sum up all received pieces individually and send their sum to the coordinator, which can calculate the global sum and redistribute it to the other parties again (see Figure 11).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

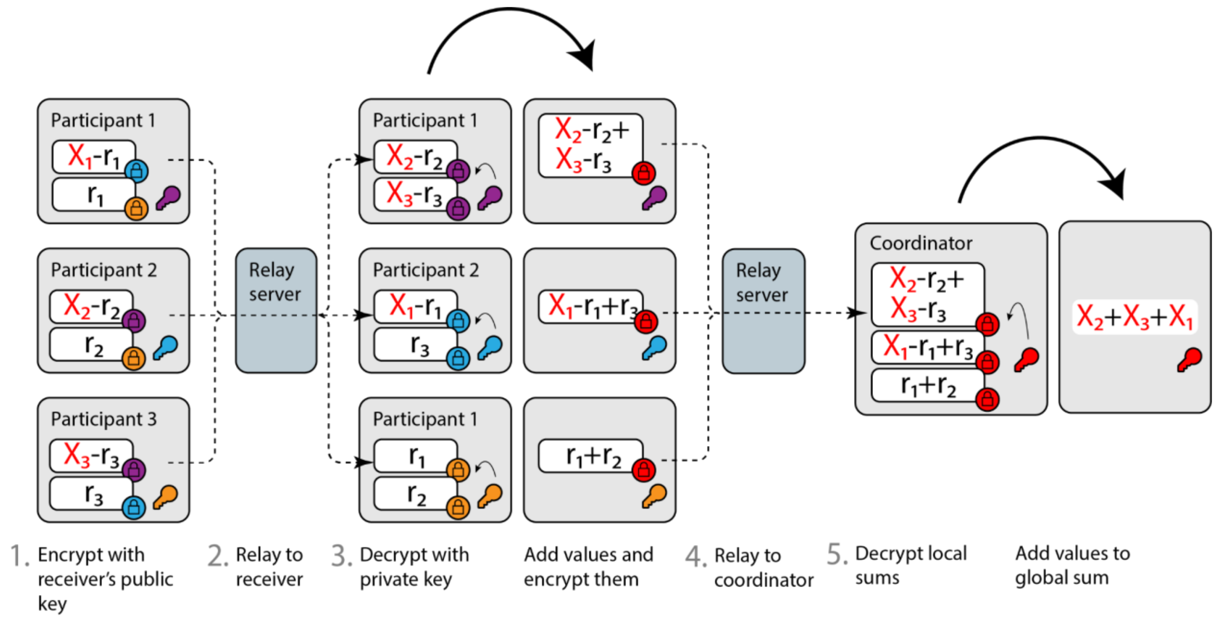


Figure 11. *Additive secret sharing implemented in FeatureCloud. Step 1 shows how two secrets are created by each of the three participants for their values X_1, X_2 and X_3. Step 2 distributes them according to the P2P protocol. In step 3, the received secrets are decrypted and summed up by each participant. Step 4 relays the local sums to the coordinator, which decrypts them in step 5 and calculates the global sum.*

When using this technique during training, at the beginning of each iteration, each participant first receives the global model (e.g., a randomly initialized neural network). Each participant then creates an updated model using its local data and masks the model with *n-1* different masks, one for each participant, and encrypts them with the respective participant's public key. The masked model, together with the encrypted masks, is then sent back to the coordinator. The coordinator relays the encrypted masks to the participants who can decrypt their share of the masks and calculate the sum, which is then sent back to the coordinator. The coordinator finally sums up the masked models and the sums received from the participants to obtain the sum of local models. While providing enhanced privacy for each participant, it leads to an increase in network traffic, growing quadratic with the number of participants.

4.4 Certification Process and App Evaluation

As an extension of the certification process and migration to a more structured approach, we are currently integrating the AIMe report tool into FeatureCloud. The current process entirely relies on a description by the developer, making manual checks of completeness of the reported pieces of information necessary. In order to solve this, an additional AIMe section entirely dedicated to privacy is being drafted, using the AIMe specification language shown in Listing 1. In order to be certified, each app then requires a privacy AIMe report providing details about potential privacy leaks for:

- A single execution
- Multiple executions on the same data
- Execution in a pipeline

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

This AIMe report will then be a mandatory part of the certification process and used as a basis for the verification of the risk assessment (see Figure 12).

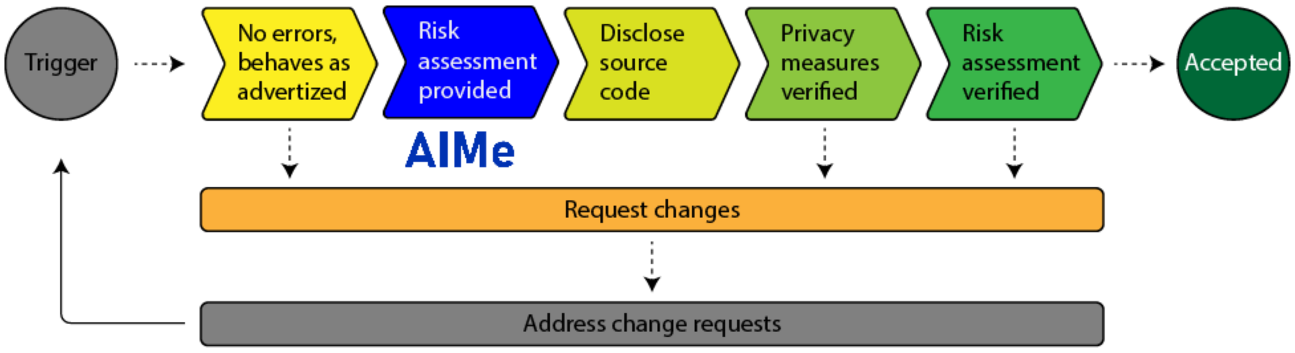


Figure 12. Outline of the certification process. *A new app and app updates need to undergo 5 stages during the certification process: Checking for errors, providing AIMe risk assessment, disclosing source code, verifying privacy measures, verifying risk assessment.*

5 System and Implementation

This section contains the updated description of the system and software architecture.

5.1 System Architecture

The FeatureCloud system consists of multiple components that are distributed across IT infrastructures of the workflow participants and servers hosted by FeatureCloud.

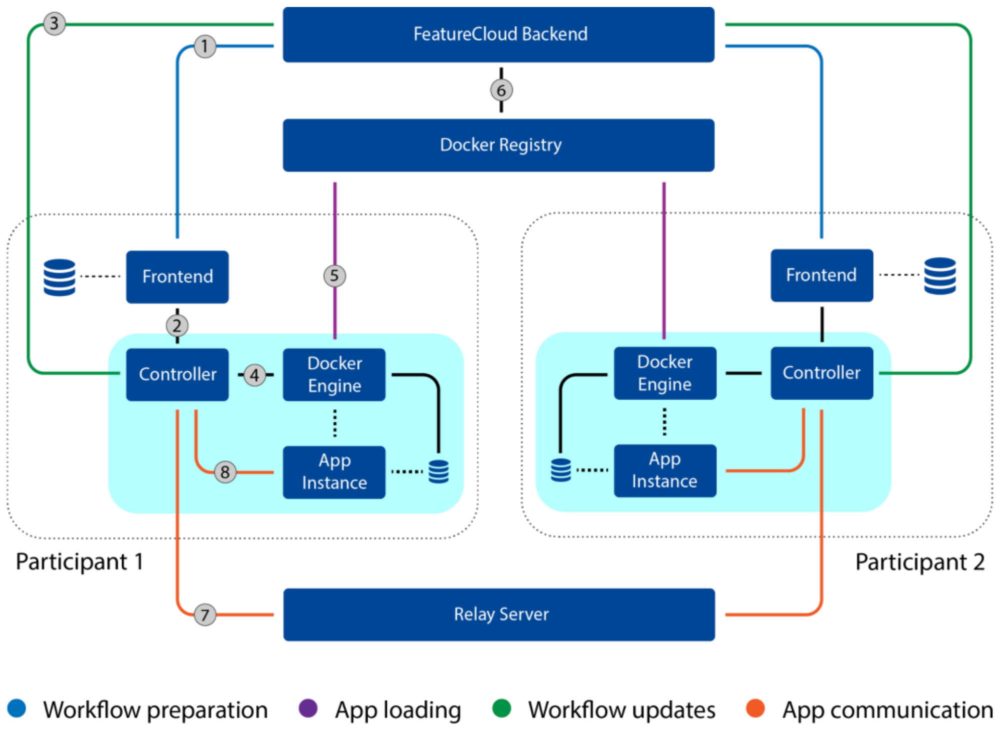


Figure 13. System architecture of FeatureCloud with two participants. *The Controller, Frontend, Docker Engine, and App Instances are running locally at the participants. The FeatureCloud Backend and Docker Registry are running on FeatureCloud servers. The Relay Server can be run on a separate server, or participants can use a provided instance from*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

FeatureCloud. The components are connected via TCP/IP connections (straight lines). All links are HTTP-based, except for link 7, which uses a raw socket connection. Links 1 - 4 use JSON for serialization, and links 4 - 6 use the Docker API.

Frontend. The frontend is implemented as a user-friendly web interface accessible through the website featurecloud.ai. It is implemented in Angular and allows users to create and manage federated and privacy-aware workflows without any programming skills. It is also used to create test runs for app implementations and to manage the local Controller.

FeatureCloud Backend. The FeatureCloud Backend is a central server hosted by FeatureCloud handling all general entities and information of the platform, such as user accounts or app meta-data (author, versions, user feedback) and project information (including onboarding of workflow participants). It is implemented in Python using the Django web framework.

FeatureCloud Registry. All app images are hosted on the FeatureCloud Registry as Docker images (see Technology section). When a WF is executed, all required apps are automatically pulled from there. App developers also use it to push new apps or new app versions. For that, we host a Docker registry with a custom proxy in front of it, implemented by us to control access and ensure only entitled users can push specific app images[5].

Controller. The Controller is a local server instance that is only running at the corresponding site. As it handles the sensitive local computations of the FL workflow, it is only reachable for the corresponding users of that site inside the local IT infrastructure. It is implemented in GoLang (see Figure 14) and takes care of executing and stopping the Docker containers of the FL app and the communication between the different participants. The Controller provides an HTTP API that implements and enables FL for the developed apps. Developers use it to develop their own apps. Developed apps that implement this API can be published in the AI Store and used in federated workflows by the FeatureCloud users. The complete API specification is available on the FeatureCloud website[6]. For an easy start, we provide an extensive template[7] with examples for different algorithms, making it easy to develop new federated algorithms for FeatureCloud.

[5] see https://featurecloud.ai/manual
[6] https://featurecloud.ai/assets/api/redoc-static.html
[7] https://github.com/FeatureCloud/app-template

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

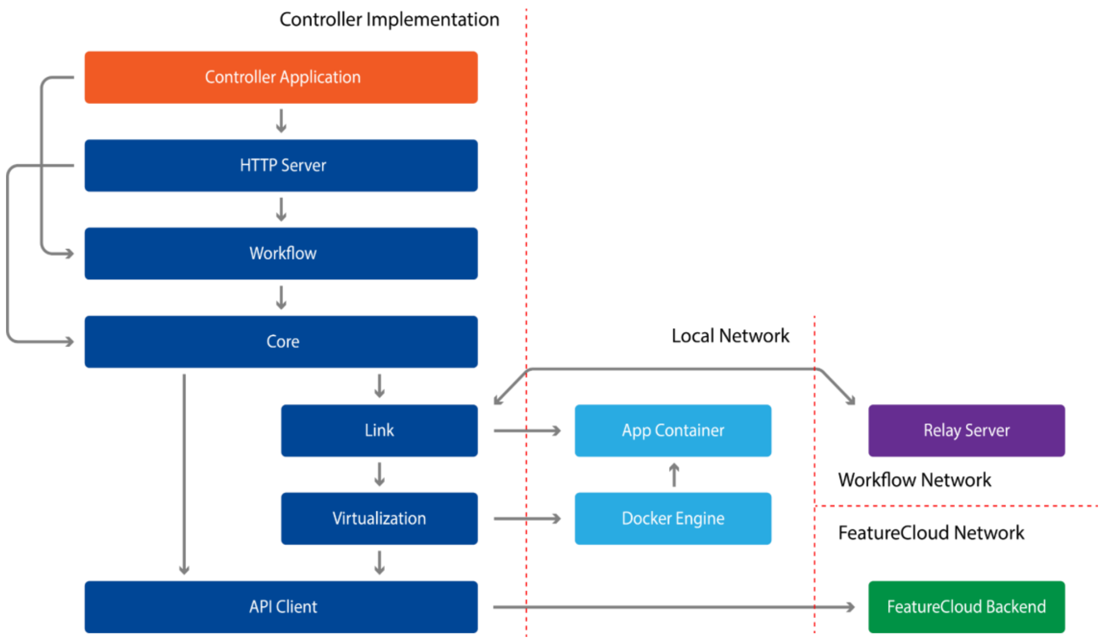


Figure 14. Software architecture of the FeatureCloud Controller and its context. *The Controller uses a layered architecture preventing arbitrary access across layers by enforcing a partially ordered access hierarchy (left side). It communicates with the App Containers and the Docker Engine to orchestrate the local part of the workflow (middle). It receives instructions on which container to execute next from the FeatureCloud Backend (right side). The shared parameters are sent through the Relay Server (right side), which can reside in its own network to which all participants need to have access to. Alternatively, FeatureCloud provinces a Relay Server instance for convenience.*

5.2 Application Programming Interface

The FeatureCloud API was updated to provide better feedback to the app users and more security to the exchanged parameters. Therefore, the *GET status* call was extended by a *status*, *progress*, *message, state, destination,* and *smpc* parameter.

```
{
  "available": false,
  "finished": false,
  "message": "Computing local parameters",
  "progress": 0.1,
  "state": "running",
  "destination": 1,
  "smpc": {
    "operation": "add",
    "serialization": "json",
    "shards": 2,
    "exponent": 8
  }
}
```

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

Listing 5. Sample API call. *The Controller API has been extended to allow for invoking further functionality of the Controller.*

Developers implementing the FeatureCloud API can now use the *message* parameter to inform end-users about the current info of the app (see Listing 5). The *progress* parameter can be used to estimate the progress and will be used to visualize a progress bar in the FeatureCloud workflow frontend. With the *state* parameter, developers can show users that the app is *running*, an *error* occurred, or f*eedback* is *needed* from the end-user.

The *destination parameter* gives developers more control over where to send their data. Before this, the aggregating coordinator could only broadcast the data to all clients. With the *destination* parameter, data could be sent to specific clients if not all clients receive the same data. From this feature, we also make use of internally when the *smpc* parameter is set. The *smpc* parameter can be set with an *operation*, *serialization*, the number of *shards*, and an *exponent* to perform additive secret sharing for the aggregation. It implements the approach explained in 4.3 and allows developers to use additive secret sharing and hide the shared local parameters from the aggregator without more profound technical knowledge.

A complete and up-to-date API description is available on our FeatureCloud website[8].

5.3 Compute Modules

The current API is still very generic and allows for implementing a large variety of applications. In order to make guarantees about privacy and security of algorithms, the FeatureCloud system needs to provide more pieces of functionality that are being used by apps. One first example has been implemented (SMPC), following recommendations from D2.4, but further such compute modules will be integrated into the platform to reduce the danger of malicious implementations. By providing such modules and restricting or entirely removing free communication between apps, the app implementations have considerably less opportunity to funnel private data outside of the local site.

From an architectural point of view, this is achieved by disabling parts of the API. For instance, if an app requires gradient descent, it would be part of its specification to not require direct communication, but the gradient descent compute module. The FeatureCloud Controller then disables all communication capabilities of the API and just provides the gradient descent module API, executed inside the Controller, which then takes care of federated communication internally.

[8] https://featurecloud.ai/assets/api/redoc-static.html

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

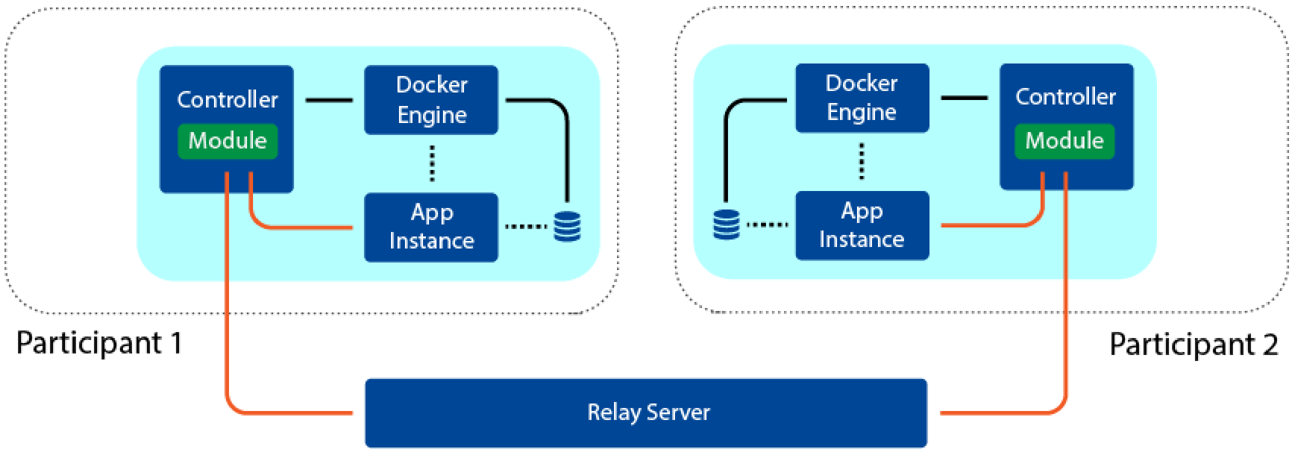


Figure 15. Compute module integration. *App instances cannot send data directly to one another. Instead, the compute module (green) takes care of data communication to achieve a piece of federated functionality.*

We are investigating the following compute modules for integration into the Controller:

- Gradient descent (applicable for neural networks and other continuous models)
- Extension of gradient descent module to make it differentially private, as proposed in D2.4
- Impurity optimization (applicable for decision trees / random forests)

6 Next steps

The current version of the platform provides all the basic functionality required for in-practice executions of federated ML studies.
However, apart from goals set out in the deliverables, the following aspects will be integrated in the process of refinement of the platform:

- Frontends for parameter specification rendered from structured parameter representation attached to each app
- Prepared workflows for specific application scenarios
- Storing of computed models (results of ML executions) on the FeatureCloud website for later use (e.g., to perform predictions on new samples)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

D) Table of Acronyms and Definitions

AI	Artificial intelligence
API	Application programming interface
CLI	Command-line interface
CI/CD	Continuous integration / continuous deployment
concentris	concentris research management GmbH
CSS	Cascading style sheets
CSV	Comma-separated values
DL	Deep learning
DP	Differential privacy
E/R	Entity/relationship
FL	Federated learning
GDPR	General Data Protection Regulation
GND	Gnome Design SRL
GUI	Graphical user interface
HE	Homomorphic encryption
HTML	Hypertext markup language
HTTP	Hypertext transfer protocol
HTTPS	Hypertext transfer protocol (secure)
IP	Internet protocol
JSON	JavaScript object notation
JWT	JSON web token
ML	Machine learning
MR	Merge request
MS	Milestone
MUG	Medizinische Universitaet Graz
Patients	In this deliverable, we use the term “patients” for all research subjects. In FeatureCloud, we will focus on patients, as this is already the most vulnerable case scenario and this is where most primary data is available to us. Admittedly, some research subjects participate in clinical trials but not as patients but as healthy individuals, usually on a voluntary basis and are therefore not dependent on the physicians who care for them. Thus, to increase readability, we simply refer to them as “patients”.
RF	Random forest
SDU	Syddansk Universitet
SMPC	Secure multiparty computation
SSH	Secure shell
SSL	Secure sockets layer
SVM	Support vector machine
TCP	Transmission control protocol
TUM	Technische Universitaet Muenchen
UHAM	Universität Hamburg
WP	Work package

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

E) Other Supporting Documents, Figures and Tables

Command-line Interface

The current CLI supports the following commands:

Start a test

```
test start --client-dirs ./test1,./test2 --app-image test_app
```

Starts a new test run with two clients (inferred from the number of directories) and the corresponding input data in folder test and test2 using the app image 'test_app'.

Obtain info

```
test info --test-id <id>
```

Get general information about the test run with id <id>.

Show logs

```
test logs --test-id <id>
```

Shows the logs of the test run with id <id>.

Stop test

```
test stop --test-id <id>
```

Stops the running test with test id <id>.

Delete test

```
test delete --test-id <id>
```

Delete the test run with id <id>.

Show traffic

```
test traffic --test-id <id>
```

Shows the traffic of the running test with test id <id>.

List all tests

```
test list
```

List all test runs that were executed or are still running.

Application API

The current API specification can be found on the FeatureCloud website on https://featurecloud.ai/assets/api/redoc-static.html

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.