Federated decentralized trusted dAta Marketplace for Embedded finance

D5.4 - Energy Efficient Analytics Toolbox. II

Title	D5.4 - Energy Efficient Analytics Toolbox. II
Revision Number	3.0
Task reference	T5.3 T5.4 T5.5
Lead Beneficiary	LXS
Responsible	
Partners	ATOS, UPRC
Deliverable Type	DEM
Dissemination Level	PU
Due Date	2025-03-31 [Month 33]
Delivered Date	2025-09-29
Internal Reviewers	INNOV TRB
Quality Assurance	UPRC
Acceptance	Coordinator Accepted
Project Title	FAME - Federated decentralized trusted dAta Marketplace for Embedded finance
Grant Agreement No.	101092639
EC Project Officer	Stefano Bertolo
Programme	HORIZON-CL4-2022-DATA-01-04

This project has received funding from the European Union's Horizon research and innovation programme under Grant Agreement no 101092639

Revision History

Version	Date	Partners	Description
0.1	2025-07-03	LXS	TOC
0.2	2025-08-29	ATOS	Input in Sections 3.3, 3.4 and 4.3, 4.4
0.3	2025-08-29	JSI	Input in Sections 3.2, and 4.2
0.4	2025-09-03	LXS	Input in Sections 3.1, and 4.1
0.5	2025-09-03	LXS	Input in Executive Summary, Intro and Conclusions
0.6	2025-09-08	LXS	Input in Section 2
0.7	2025-09-09	ATOS	Input in Section 2
0.8	2025-09-10	JSI	Input in Section 2
1.0	2025-09-11	LXS	Version for Internal Review
1.1	2025-09-20	TRB	Review from TRB
1.2	2025-09-26	INNOV	Review from INNOV
2.0	2025-09-28	LXS	Version for QA
3.0	2025-09-29	LXS	Version for submission

Definitions

Acronym	Definition
ACM	Association for Computing Machinery
AI	Artificial Intelligence
API	Application Programming Interface
APP	Application, usually referred to the Project WEB application
ATOS	Atos It Solutions And Services Iberia Sl
CI	Continuous Integration
CPU	Central Processing Unit
CSV	Comma Separated Value files
DB	Data Base
DL	Deep Learning
ES	Expected Shortfall
FAME	Federated decentralized trusted dAta Marketplace for Embedded finance
FDAC	Federated Data Assets Catalogue
FML	Federated Machine Learning
GA	Grant Agreement|General Assemply
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ID	Identity
IEEE	Institute (of) Electrical (and) Electronic Engineers
IP	Internet Protocol
ISO	International Organization for Standardization
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
ML	Machine Learning
MOH	Motor Oil (Hellas) Diilistiria Korinthou A.E.
ODBC	Open Database Connectivity
ORM	Object-Relational Mapping
RBAC	Role-Based Access Control
REST	Representational State Transfer
RFM	Recency, Frequency, and Monetary
SA	Solution Architecture
SAX	Situation Aware eXplainability

SQL	Structured Query Language
TLS	Transport Layer Security

Other acronyms and abbreviations not present in the table, are introduced in the text along with their definitions.

Executive Summary

This deliverable presents the work that has been carried out during the second phase of the project regarding the *Energy Efficient Analytics Toolbox.* This work has been performed under the scope of three tasks of WP5 ("*Trusted and Energy Efficient Analytics"*) and more precisely T5.3 ("*Incremental Energy Efficient Analytics"),* T5.4 ("*Edge Data Management and EdgeAI Optimization")* and finally T5.5 ("*FML for Privacy Friendly and Energy Efficient Data Markets*").

The outcomes of this work are several building blocks that the *Energy Efficient Analytics Toolbox* consists of. Firstly, the *Incremental Analytics* component, which is responsible for the energy efficient query execution, exploiting the novel incremental query operators of the underlying decentralized data management system. Secondly, the *Analytics CO2 Monitoring* whose main focus is to monitor and predict CO2 emissions for artificial intelligence or machine learning algorithms and SAX techniques developed within the FAME project. Then, the *Smart Deployment* implements a powerful way to deploy models and services into the target infrastructure, automating the role of the MLops engineer, being capable to deploy those models from a centralized cloud infrastructure to edge devices. Last but not least, the *FML Orchestrator* enables federated machine learning techniques.

The technology outcomes presented in this deliverable are tightly connected with the *Objective 5* ("*Trusted and Energy Efficient Analytics Toolbox")* as described in the DoA and more precisely to the **incremental query execution** that compute their results leveraging past executions of analytical functions and computing it only for data that have changed. This is key factor for the overall reduction of CO2 emissions of analytical functions, which is measured by the **analytics CO2 monitoring.** Moreover, this project scope objective is also related with the ability to move AI/ML operations to the edge, using the **smart deployment** component presented in this deliverable, while enabling federated machine learning techniques via the **FML framework**. The target objective of the work presented in this deliverable is to minimize the CO2 emissions of analytical functions based on a drastic reduction of data transfers and I/O operations, exploiting the outcomes of the aforementioned technologies in order for FAME to integrate energy efficient techniques that automatically select and deploy the most energy efficient analytics functions for a given application.

This is the second and final version of the deliverable, accomplishing the target objectives of milestone MS09. With respect to its first version, this document provides more details about their internal architecture, along with an extensive technical specifications and interface documentation, when applicable. These technical specifications are accompanied by a respective demonstration of their use. It includes their documentation and how to properly install, deploy and use, along with an extensive evaluation per component. As this is the final version of this deliverable, we present the final specifications and demonstrators, along with the evaluation of the technology outcomes with respect to the relevant KPIs, taken from the DoA's *Objective 5.*

It should be noted that, as this is a second version of a previous deliverable (D5.2), there are some text that has been maintained from that version to ease the readability of the document.

Table of Contents

List of Figures

List of Tables

1 Introduction

This deliverable presents the *Energy Efficient Analytics Toolbox* architectural layer of the overall FAME integrated solution. It summarizes the work that has been performed during the second phase of the project, which accomplishes the target outcomes of the milestone MS09. This work has been performed under the scope of several tasks: T5.3 ("*Incremental Energy Efficient Analytics")* whose main technological outcome is the *Incremental Analytics* component, T5.4 ("*Edge Data Management and EdgeAI Optimization")*, which is responsible for both the *Analytics CO2 Monitoring* and *Smart Deployment* components and finally T5.5 ("*FML for Privacy Friendly and Energy Efficient Data Markets*") under whose responsibility is the *FML Framework*.

In this document, we firstly introduce its content, presenting the main technological pillars that it concerns, along with the objectives of the deliverable, giving insights and dependencies from other tasks. Then, we analyse how the components of this report are positioned with respect to the overall FAME SA, which of the involved assets are indexed into the FAME Marketplace, along with the coverage of their respective KPIs. After positioning into the FAME SA, we provide further deep details per component, starting with a brief description of its purpose. Whereas the first version of this deliverable focused on the related work behind each prototype and presented their initial design at the first phase of the project, we now report their final design, accompanying with a wider technical specification, presenting the 3rd level of the C4 architecture per each component, which depicts the internal sub elements of them and how they interact both among them and with the external technological entities that the overall FAME integrated solution consists of. For the components' interaction, we have included a specification of the corresponding interfaces, which may include different level of detail according to the level of maturity of each of the components presented in this document, at this phase of the project. A description of the baseline technology used is also provided. At this second version of the deliverable, we provide an extensive demonstrator of each component, which includes the documentation of its use. Finally, we give an extensive evaluation per component.

1.1 Objective of the Deliverable

The main objective of this deliverable is to present the work that has been carried out under the scope of the tasks related with the *Energy Efficient Analytics Toolbox* architectural layer of FAME SA. This implies the technology specification of the components that this layer consists of, the description of the work related with them, and a technical documentation regarding the internal 3rd level of the C4 architecture and the description of the relevant interfaces. Another objective is to include a demonstrator of the use of each component that takes part in the overall *Energy Efficient Analytics Toolbox.* Last objective of this document but not least, is to provide a validation of the implementations with respect to the project level objectives, and more precisely with the *Objective 5* ("*Trusted and Energy Efficient Analytics Toolbox")* as described in the DoA. Towards this direction, we evaluated our prototypes against the corresponding KPIs of this objective

1.2 Insights from other Tasks and Deliverables

The work presented in this deliverable has been carried out under the scope of tasks T5.3 ("*Incremental Energy Efficient Analytics"),* T5.4 ("*Edge Data Management and EdgeAI Optimization")* and T5.5 ("*FML for Privacy Friendly and Energy Efficient Data Markets*"). As a result, it holds strong interactions with the remaining of the tasks of WP5 ("*Trusted and Energy Efficient Analytics")* and as a fact, the work described here consists of the one the two main technology pillars of that particular architecture layer. Moreover, it is well connected with T2.1 ("*Requirements, Specification and Co-Creation*"), as this task provides the requirements for the

work to be performed here, along with T2.2 ("*Platform Architecture and Technical Specifications*") which provides the design principles and guidelines of the overall architecture, and how all the internal components should interact. T2.3 ("*Data Marketplace and Platform Integration"*) provides the guidelines and operational tools for the deployment and runtime execution of the software presented in this deliverable, while this report gives input to T2.4("*FAME Dashboard and Open APIs").* Moreover, this deliverable describes the technology that can be used to provide the energy efficient incremental analytics, not the *ML/AI analytics* or *SAX Techniques* themselves, and therefore they are not directly connected with the *Data Assets Catalogue* or other components resigned into the *Federation Manager,* whose responsibility are WP3 and WP4. However, they are indirectly connected through the *incremental analytics* components and the *Analytics CO2 Monitoring* that both can be discoverable and give carbon emission insights to be later taken into consideration by the components responsible for the FAME marketplace. Moreover, the work presented in this deliverable will be finally exploited and validated by the use cases defined in WP6. As a result, we can see how tightly interconnected this deliverable is with the majority of the tasks of the FAME project.

1.3 Structure

This deliverable is structured as follows:

- *Section 1* introduces the document.
- *Section 2* positions the work presented here into the FAME SA, while provides information about KPI coverage and indexed assets in the FAME Marketplace
- *Section 3* provides the technical specification per component.
- *Section 4* demonstrates the use of each component and provides an extensive evaluation
- *Section 5* concludes the document.

1.4 Summary of changes from D5.2

In this second version of this deliverable, we have included additional subsections under section 2 to provide details about which of the assets are indexed and now available in the FAME marketplace, along with the description of their respective KPI coverage. Sections 3 and 4 have been re-written to focus on the final design of the prototypes and provide an extensive technical specification. Finally, we have included a component evaluation subsection for all our reported prototypes.

2 Positioning into FAME SA

This section describes where the *Energy Efficient Analytics Toolbox* is positioned with regards to the overall FAME SA. **Errore. L'origine riferimento non è stata trovata.** depicts the overall architecture at its current version described with more details in the D2.2 ("Technical Specifications and Platform Architecture"). In the Figure 1, we can identify the bottom layer, called *Energy Efficient Analytics Services,* which can be divided in two major blocks of components. From one hand, we have the components responsible for the provision of the *Trusted and Explainable AI Techniques,* and on the other hand, depicted inside the red outline, there are the building blocks responsible to deliver the *Energy Efficient Analytics Toolbox*. The latter is the concern of this document.

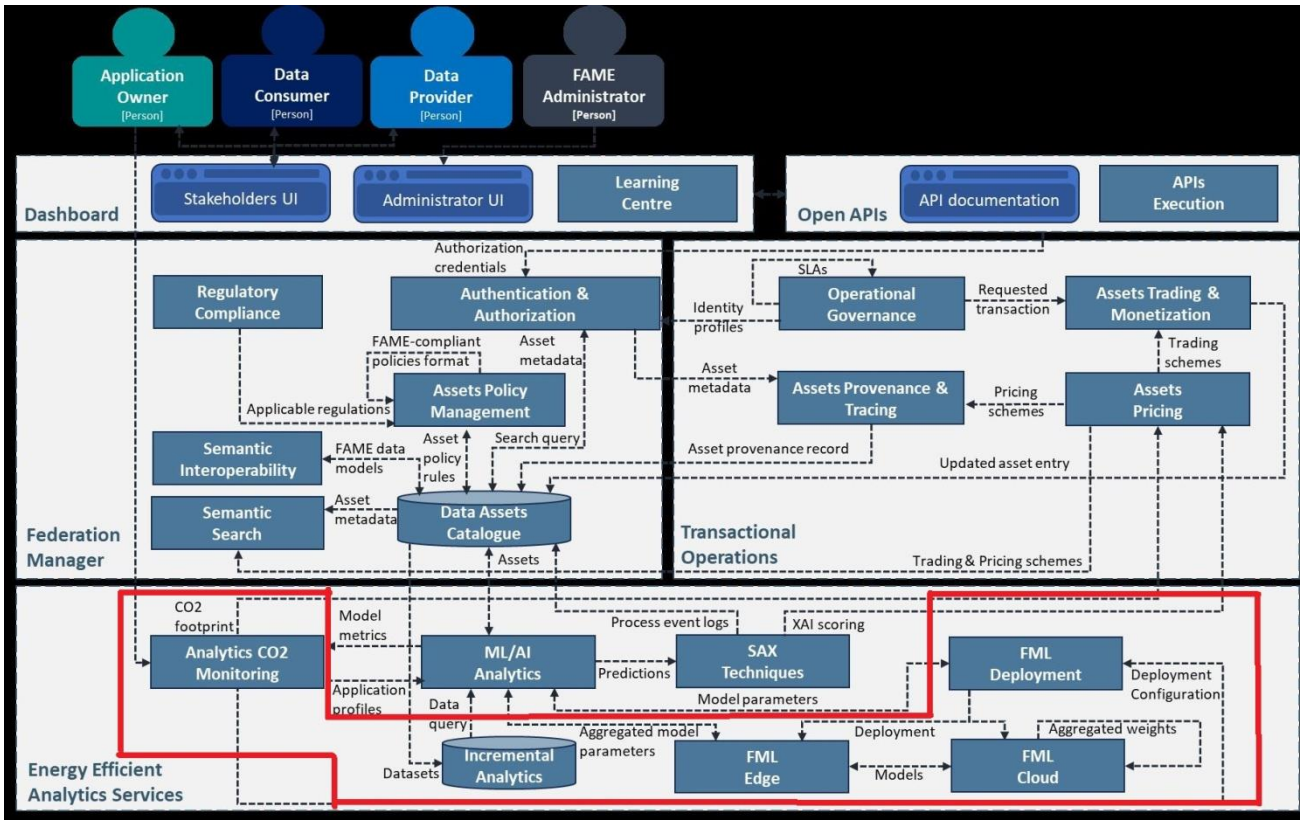


Figure 1: FAME SA C4 container diagram

The *Energy Efficient Analytics Toolbox* consists of several components. Firstly, we have the *Incremental Analytics,* which provides the decentralized data management system of the overall FAME solution, enhanced with novel capabilities with regards the energy efficient query processing and the implementation of novel incremental analytics database operators. This is used mainly by the *ML/AI Analytics* available through the overall *Energy Efficient Analytics Services* architecture layer of the FAME SA. Secondly, the *Analytics CO2 Monitoring* component focuses on the monitoring and prediction of such analytics. The *FML Deployment,* currently known as *Smart Deployment* in this document, aims to provide the *ML/AI Analytics* and *SAX Techniques* with an easy and powerful way to deploy models and services, automating the role of a MLOps engineer. Finally, the *FML Edge* and *FML Cloud* are encapsulated in this deliverable under the *FML Orchestrator.* Its purpose is to provide to the FAME integrated solution the means to enable federated machine learning in order for customers to train purchased models in a federated manner using data hosted on different servers.

The following subsection provide details regarding which of the assets developed or used by the aforementioned components are indexed into the FAME marketplace, along with a description related with the KPI coverage that the presented prototypes are linked to.

2.1 Indexed Assets

Table 1 presents the assets indexed in the FAME platform in the scope of T5.3, T5.4 and T5.5, reported in this deliverable and categorized by asset name, relation to pilot, type, and license. The detailed descriptions of the assets are provided in the respective subsections of Section 3, while their validation is presented in Section 4.

Table 1: Indexed Data Assets

#	Data asset	Pilot	Type	License
1	Incremental Analytics	Pilot Agnostic	Software	Proprietary
2	Analytics CO2 Monitoring	Pilot Agnostic	Software	Open Source
3	Smart Deployment	Pilot Agnostic	Software	Open Source
4	FML Framework	Pilot Agnostic	Software	Open Source
5	Fast Data Importer	Pilot#7	Software	Open Source
6	Anomaly Detection FL Root Server	Pilot#7	Software	Proprietary
7	Anomaly Detection FL Center Server	Pilot#7	Software	Proprietary
8	Anomaly Detection FL Client	Pilot#7	Software	Proprietary
9	Federated KMeans Server	Pilot#1	Software	Proprietary
10	Federated KMeans Client	Pilot#1	Software	Proprietary
11	SD API-Server	Pilot Agnostic	Software	Proprietary
12	SD Node-Resolver	Pilot Agnostic	Software	Proprietary
13	SD Model-Server	Pilot Agnostic	Software	Proprietary
14	SD WebAPP	Pilot Agnostic	Software	Proprietary

2.2 Key Performance Indicators

As written in the DoA of the project, the success criteria for the activities performed under the scope of the tasks that this document reports are evaluated by the following KPIs:

KPI5.2: Energy Efficient AI techniques at the edge>=6 (ATOS)

This document reports on 6 different energy efficient AI techniques at the edge, which can be summarized as follows:

- Incremental Analytics (work performed under the scope of T5.3)
- Smart Deployment (work performed under the scope of T5.4)
- Data drift monitoring for continual learning (work performed under the scope of T5.4/T5.5)
- Hierarchical federated learning (work performed under the scope of T5.5)
- Client-server federated learning (work performed under the scope of T5.5)
- Federated anomaly detection model (work performed under the scope of T5.5)

All these techniques are further presented in more detailed in section 3 and their documentation and evaluation is presented in section 4 of this document.

KPI5.4: Average CO2 reduction based on incremental analytics and EdgeAI>=200%

This KPI targets an average CO_2 emissions improvement of at least 200% (corresponding to a 66.6% reduction compared to the baseline) within the smart-migration framework. It has been successfully achieved as demonstrated in Section 4.2.2 of the report. The component evaluation employed a benchmark using the lightweight Busybox container to simulate workload migration across multiple European countries, comparing mean CO_2 emissions with optimally achievable emissions under different scenarios of country availability. Results showed substantial emission reductions, with most scenarios exceeding the KPI threshold and even achieving reductions above 80% in several cases.

KPI5.5: Increase in Power Usage Efficiency (PUE) for analytics >=80%

This KPI is addressed by the *Incremental Analytics* component. Section 4.1.2 provides an extensive evaluation of our implementation. There, we present the overall performance acceleration of the prototype, along with a benchmark evaluation regarding the energy consumption. We present that for the given dataset that was taken into account, we managed to achieve 67% increase of power usage efficiency, when using the *online aggregates* operators that have been designed especially for AI analytics. However, this improvement in power usage efficiency could be increased even more, if the volume of the reference dataset is even bigger. In a nutshell, the higher the target data volume that our implementation needs to process, the more efficient in terms of power usage can be our prototype, compared to the vanilla background technology.

3 Components Specification

3.1 Incremental Analytics

3.1.1 Description

The purpose of the component named as *Incremental Analytics* is twofold: firstly, as its name implies, to provide analytic operations in an incremental fashion. That means that the result of an analytical query (i.e. aggregation, summarization, counting, etc.) is not being calculated each time a request arrives to this component, rather than it is pre-calculated on the-fly, as new data is being ingested at the same time. This allows to provide the result of the request incrementally. The second important purpose of this component is to provide query processing in an energy efficient manner. Towards this direction, new innovations have been developed under the purpose of the task T5.3 ("*Energy Efficient Incremental Analytics"*) that have been currently merged into this component and can be exploited for any data user or ML/AI analytic processing. The major impact of these new innovations is the significant reduce of the carbon footprint associated with a particular request for query processing, as a result of the new developments that target to reduce the overall energy consumption.

Towards the second phase of the project, the *Incremental Analytics* component was further developed in order to be used in use cases related with the federated machine learning. The activities related with the federated machine learning span across the scope of both T5.3 and T5.5. T5.5 is mostly focusing on the AI analytics developed for federated machine learning, while T5.3 primarily focuses on the data management aspects, leveraging the innovations provided by the *Incremental Analytics* component. Towards this direction, the *Incremental Analytics* can be now used in federated machine learning scenarios, where the edge nodes store the respective local data and the master node is periodically updated with the latest aggregated snapshots of the edge nodes. With our approach, we let the AI analytics for federated machine learning to focus on the business aspects (i.e. implement the corresponding ML/DL algorithm to provide the corresponding predictions, forecasting) and connect to the local data store, while the *Incremental Analytics* is focusing on the data management aspects, and how to synchronize the master nodes with the latest updates of the local datasets, incrementally and in an energy efficient fashion. For that, during the second phase of the project we have designed and implement the *federation-agent.* The agent is an additional component that supplements the *Incremental Analytics*. It can be used as either a client, deployed in the edge nodes of the federated machine learning, or as a server, deployed on the master node of the federated machine learning. Both server and client agents communicate directly with the *Incremental Analytics* and interact with each other to send updated data slices to the master node, in order to keep it always updated. In the next subsection we will provide the technical specification of the novel *federation-agent.*

3.1.2 Technical Specification

3.1.2.1 Component-level C4 Architecture

In this subsection, we give more details about the internal architecture of the *Incremental Analytics* component, now supplemented with the novel *federation-agent* develop during the second phase of the project. The FAME project follows the C4 Architecture Model, already presented in the corresponding D2.2 ("*Technical Specifications and Platform Architecture"*) deliverable. The 3rd level of the C4 Architecture with regards to the vanilla *Incremental Analytics,* described in the previous version of this deliverable is depicted in Figure 2. In its latest version finally delivered at the end of this task, the internal architecture remained unchanged, along with the core

functionalities provided by its internal sub-components and their interactions. As a result, the main focus of the second phase of the project with regards to the vanilla *Incremental Analytics* was to fix bugs, improve the internal query processing and finally to provide benchmark evaluation metrics which are presented in detailed in Section 4.

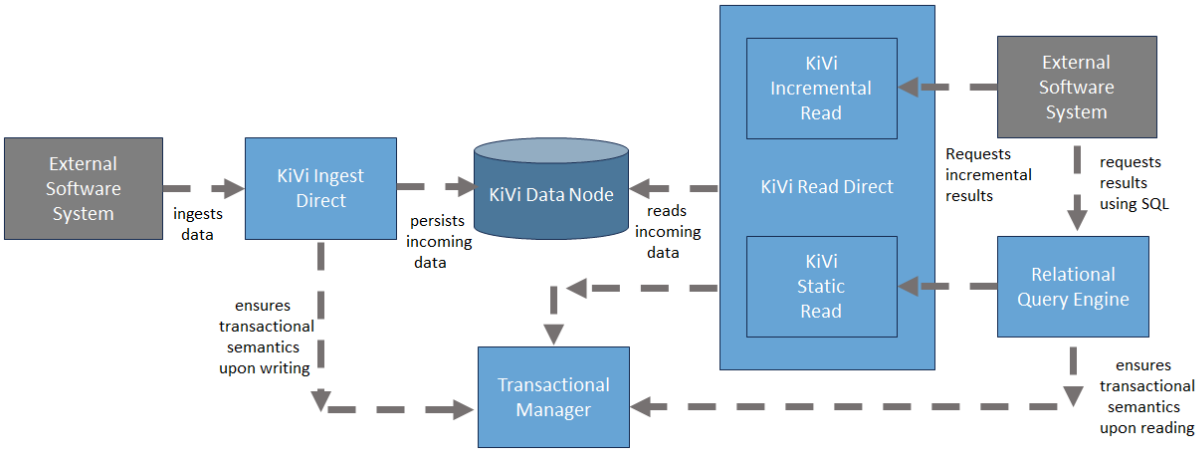


Figure 2: Incremental Analytics 3rd level of C4 architecture

Moreover, in this second phase of the project, we have extended the *Incremental Analytics* component to be used in federated machine learning scenarios. In such cases, we let the ML/DL algorithms to focus on reading the data from their respective datastore, while we have developed the *federation-agent* that is responsible to synchronize the data from the edge to the master data nodes. Figure 3 illustrates the 3rd level of the C4 Architecture with regards to the federated Incremental Analytics.

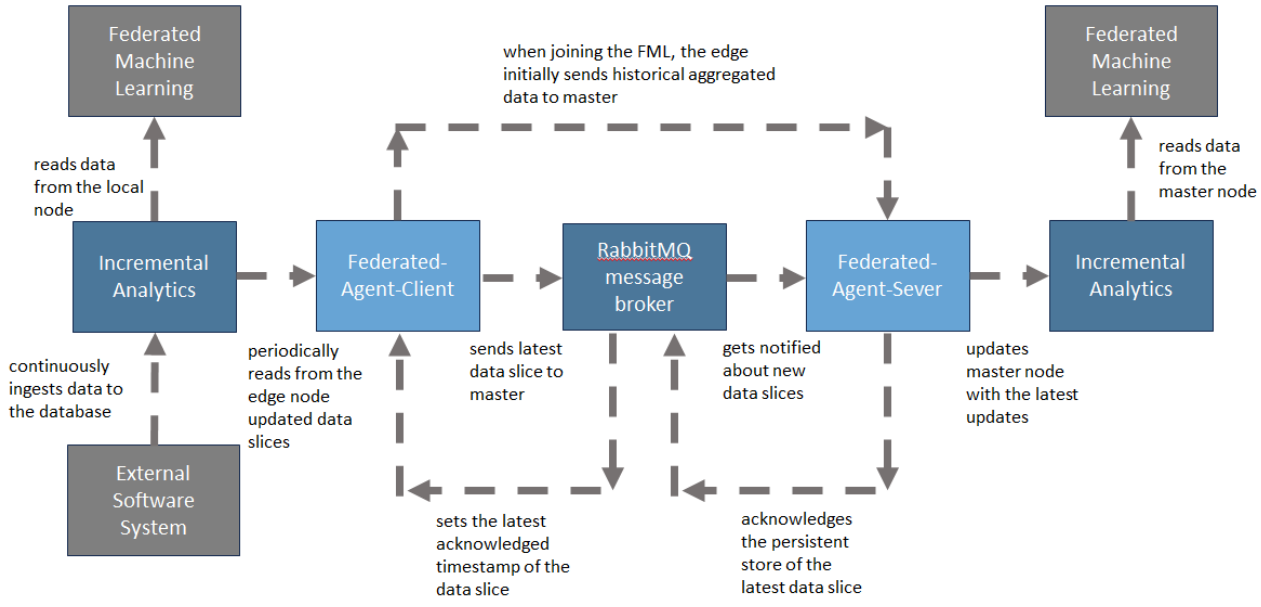


Figure 3: Federated Incremental Analytics 3rd level of C4 architecture

In this figure we can see the various internal subcomponents of our solution. The vanilla *Incremental Analytics* is now depicted as a black box, as its internal subcomponents were depicted there and described in more detailed in the previous version of this deliverable. The *federation-agent* that extends the *Incremental Analytics* for the federated machine learning scenarios are depicted as separate components: one (the Server) dedicated to the master node, and the ones (the Clients) dedicated to the edge nodes. We can have as many clients as the edge nodes in the

federated machine learning. In the figure, a RabbitMQ message broker is also depicted as a black box, which allows the asynchronous interaction between the client agents and the server. Last but not least, the figure includes external software system in the edge nodes that continuously generates new data that needs to be stored in the edge nodes, while the AI analytics processes are directly interacting with the datastores, using their standard interfaces, as described in the previous version of this deliverable. In this version, we will focus on the functionality of the *federation-agent* along with its interactions.

Firstly, when the *federation-agent* microservices are initialized, both in the client and the server nodes, they create a new table in their respective data nodes to hold meta-information regarding the information flow. The schema of this table is the same for both the clients and the master and it includes client's hostname that is subscribed to periodically send updates to the master node, the target data table that the user wants to synchronize, the time period that the user wants the synchronization to occur, the RabbitMQ topics to send the updates and to receive the acknowledgements, along with the last timestamp that the synchronization took place. This is depicted in Figure 4.

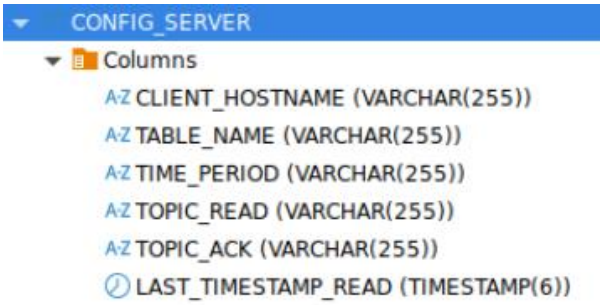


Figure 4: Federated Agent Meta-Information schema

The data administrators can choose one of the tables in the client side that they want to synchronize the data, along with the periodicity via an external exposed OpenAPI. Once this REST endpoint is invoked, the client interacts with the server via an internal OpenAPI to subscribe itself. It sends the aforementioned information like the table name and the periodicity, along with some meta-information like the client's host and port. When the server receives this request, it adds the specific table for that particular client into a list of *listeners* and it creates two specific topics in the RabbitMQ message broker: one for receiving new data slices for that particular client and one for sending acknowledgements, as we will see later on. Once the new request is added as a *listener* and the topics are created, the server returns back the HTTP response that contains the original message of the request, along with some meta-information regarding the topics that both the client and the server would need to use in order to exchange messages. Both clients and server persistently store this information in the corresponding data table that was depicted in the previous figure.

Immediately after client and server establish the way to communicate via the RabbitMQ topics, the server sends a request to an internal OpenAPI of the client to receive all historical data for the data table of interest until the current point in time. Instead that the client should send all historical data of that particular table, it makes use of the *Incremental Analytics* feature of the database and it will only send the aggregated information of that particular table, grouped by the time unit of interest. In fact, in the federated machine learning scenarios, there is no interest of the master nodes to receive all (possible private) information from the clients, but instead, they are interested in aggregated information from the original data tables. And this is where the *Incremental Analytics* play a significant role as they can calculate the aggregations very efficiently, reducing the carbon footprint of the operation by implementing it in an energy efficient manner.

It is important to highlight that receiving all historical data from the client, even aggregated, might require the data transmission of a significant amount of data via an HTTP request. A typical HTTP request-response communication would require the master node to send the request, then the client node, and then the latter to prepare the response before sending it back to the master node. However, when it comes to Big Data, this response might contain several GBs of data. This would firstly take too long to prepare the response with the risk for a time out exception, or even worse, the client node could run out of memory while preparing the response. Due to that, in our implementation we made use of a *StreamingResponseBody.* With that, when the client node receives the request to send all historical data, it immediately returns back the response to the caller, but leaves the communication channel open, In another thread, it starts reading all historical data from the database and writes the output to that channel that has been remained open. Internally, it makes use of a buffer and writes to the channel when the buffer is full of records. Once the whole historical data has been sent, the communication channel closes. By doing this, there is never a risk for a time out, as the caller receives the response immediately. Moreover, the client node does not risk to run out of memory, as it holds internally only a small buffer of the overall dataset. Lastly, the whole communication takes place in parallel, so the master node (the caller) is storing the historical data in parallel, as the client node is reading them from the local database.

Once the client node has received a confirmation for the subscription to the master node, it *schedules* a thread to periodically send new data slices to the master via the RabbitMP topic that it received. Internally, it checks the meta-info data table about the latest confirmed timestamp, and send only the data slice that corresponds to the most recent aggregated data from that particular timestamp. Given that typically the frequency of the scheduler equals the time unit of the aggregation, the data slice contains 1 or 2 data records. The client node sends the message to RabbitMQ without having to care if the message was received and the data slice was eventually stored. It periodically sends messages according to the latest timestamp.

The master node is listening to that particular topic and eventually receives messages asynchronously. Once a message is received, the master persistently stores the new data slice to its respective master node, and then acknowledges this action by sending an appropriate message to the other topic, that includes the latest timestamp of the data slice that was stored. Once the client node receives the acknowledgement, it updates its internal meta-info data table with this new timestamp, so that the scheduled thread can proceed and send new updates afterwards.

The overall communication of the client and the master nodes is based on the micro service architecture, where two independent microservices communicate asynchronously via a message broker. In fact, they only synchronous communication takes place when a client joins the federated machine learning deployment, when it initially sends a request to subscribe itself, receives a confirmation and sends the historical data up to that point. The information flow afterwards is based on a microservices choreography SAGA, where the involved microservices do not need a central orchestrator, rather than they interact upon messages they receive from the message broker. As it has been described, there are also no compensation methods, neither in the master nor in the client nodes, to react upon failures and probably rollback some actions. The reason for not having compensation methods is that all actions are upsert-only, which allows the re-do of specific actions. In fact, if a new message containing a data slice is lost, or the corresponding acknowledgement is lost, the client would have not updated its internal *last updated timestamp.* This will cause the scheduler to re-send the next time the previous data slice, along with the current. In case that the master failed to store a data slice, it will receive it the next time until it acknowledges a specific message. If the master receives data slices that had already stored beforehand, it will re-do the action and the data slice will override the already received one, having the same data. Eventually,

the client node will receive an acknowledgement and will forward the internal last updated timestamp.

3.1.2.2 Baseline Technologies and Tools

As it has been mentioned in the previous subsection, the *Federated Incremental Analytics* component of the FAME integrated solution relies on the background technology of LeanXcale database, that is being enhanced with the foreground technologies developed under the scope of T5.3 ("*Energy Efficient Incremental Analytics"*), along with the novel *federation-agent*. The following Table 2 presents the overall baseline technologies and tools used for implementation.

Table 2: Baseline Technologies and Tools for Incremental Analytics

Baseline Technology	Description	Added value to FAME
LeanXcale DB	An innovative NewSQL database that combines the benefits of both SQL and NoQL worlds.	It provides the database management system of the FAME integrated solution, with its advanced capabilities for data ingestion at very high rates and real time analytics over data being ingested simultaneously.
RabbitMQ	The message broker for the communication of the *federation-agents*	It provides a persistent and fault-tolerant message broker for the various agents to exchange messages asynchronously
SpringBoot	The baseline framework for developing the microservices for the *federation-agent*	Facilitates development, deployment and monitoring of microservices.

3.1.3 Interfaces

The Incremental Analytics component relies on one hand on the LeanXcale DB, which can be considered as a relational, full SQL compliant database. They interact with the database with the standard JDBC and ODBC connectivity mechanisms, being already integrated with popular analytical frameworks (i.e. Apache Spark, Apache Flink, etc.) and popular ORM frameworks (i.e. Apache OpenJPA, Hibernate, etc.).

With regards to the *federation-agent,* the latter firstly exposes a set of external OpenAPIs for the interaction with the end-users and data administrators that want to configure the agents to periodically update the master nodes with the latest updated data slices. Moreover, it exposes a set of internal OpenAPIs for the communication of the server and the client nodes. These interfaces are internal to the component and are not meant to be used by external systems or users. Last but not least, the *federation-agent* interacts asynchronously via the RabbitMQ message broker through additional interfaces. All these interfaces are described in this subsection. Firstly, the following code snippet depicts the *subscription* DTO that is widely used in the interfaces.

```
{
  "tableName": "K2201",
  "timeGranularity": "HOUR",
  "lastDateRead": "2025-07-15",
  "lastTimeRead": "08:52:55",
  "topic_read": "127.0.0.1:K2201_HOUR_data",
  "topic_ack": "127.0.0.1:K2201_HOUR_ack",
  "client_host": "127.0.0.1",
  "client_port": 8081
}
```

External Interfaces (implemented as OpenAPIs)

GET: /api/client/configure?machineID={machineID}&period={period}. It is invoked by the data administrator to join a client node to the federated machine learning deployment. Receives the Subscription DTO

Internal Interfaces (implemented as OpenAPIs)

POST: /api/server/subscribe It is invoked by the client node to subscribe itself to the master node, in the federated machine learning scenario. It sends a Subscription DTO as a body message, and receives the same Subscription DTO in the response, additionally filled with some meta-information like the topic to read and acknowledge messages.

POST: /api/client/get-all-data/stream It is invoked by the master node to get all historical data from the client. It returns a *StreamingResponseBody* over which the historical data are being sent as a stream byte array.

Asynchronous Interfaces (implemented as RabbitMQ consumers or producers)

RabbitMQClientProducer.send(String queue name, ClientRequestDataDTO): It sends the DTO to the specific queue of RabbitMQ. The DTO extends the Subscription DTO by additionally having a String that contains the data slice.

RabbitMQClientConsumer.consumeMessage(String queue_name): It listens for acknowledgements from the master node in the particular queue. It receives a byte array that contains the message. The message has three attributes: the table name, the time granularity nad the latest updated timestamp.

RabbitMQServerConsumer.consumerMessage(String queue_name): It listens for new data slices sent by the client. . It receives a byte array that contains the message. The message can be deserialized to ClientRequestDataDTO, which contains the data slice.

RabbitMQServerProducer.sendMessage(String queue_name, String ServerACKsWriteDataDTO dto): The master node sends the acknowledgement in the particular queue.

3.2 Analytics CO2 Monitoring

3.2.1 Description

The Analytics CO2 Monitoring component is designed to provide comprehensive monitoring of CO2 emissions associated with any containerized component within a Kubernetes cluster, specifically tailored for use in the smart deployment component designed by ATOS. Its core functionality involves collecting real-time power consumption data from services, such as Kepler, and combining it with dynamic CO2 intensity data from electricity grids in different countries. This allows for a precise calculation of the carbon footprint of workloads. The component aims to identify and recommend possible CO2-aware deployment strategies, thereby contributing to the overall reduction of CO2 emissions within the FAME ecosystem. It supports any containerized deployed workflow, providing granular insights into their environmental impact in the form of timeseries data.

3.2.2 Technical Specification

3.2.2.1 Component-level C4 Architecture

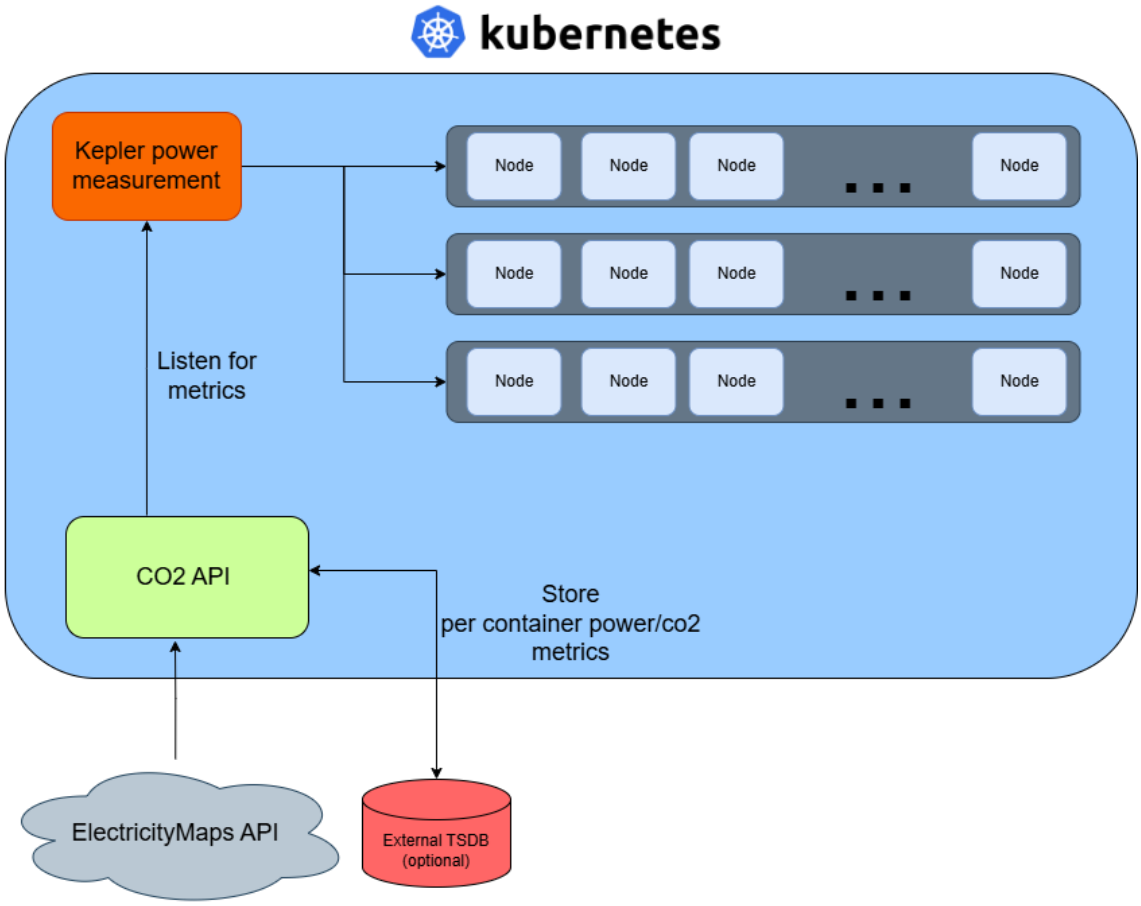


Figure 5: Visual C4 diagram of Integrated Component on distributed Cloud system

The architecture of the Analytics CO2 Monitoring component, as depicted in Figure 5 above, involves several key processes:

- **Collection of Relevant Data:** This includes internal and external data sources, such as power consumption metrics from containerized environments (e.g., via the Kepler exporter) and real-time CO2 intensity data from national electricity grids.
- **Generation of CO2 Emission Data:** This involves a continuous process of calculating CO2 emissions. The system scrapes power consumption metrics (e.g., kepler_container_joules_total) from monitoring endpoints like http://kepler.kepler.svc.cluster.local:9102/metrics. Concurrently, it fetches real-time CO2 intensity values (grams of CO2 per kWh) for various countries from sources like https://app.electricitymaps.com. These two data streams are then combined to compute the average pod-level grams of CO2 per second in the form of timeseries with 10 second increments.
- **CO2 Analytics:** The collected and calculated CO2 emission data is used to analyze the environmental impact of different workflows. This analysis can inform decisions on later CO2-aware migrations.

- **Data Storage:** Power consumption samples and CO2 data are then temporarily stored in in-memory deques for recent history. For persistent storage and historical analysis, the system supports integration with a TimescaleDB database if configured on deployment.

The component exposes a Flask-based REST API for interaction, enabling other FAME components and external systems to query and retrieve CO2 emission data.

3.2.2.2 Baseline Technologies and Tools

The component relies upon Kepler and ElectricityMaps, for its datastreams. Both components and their roles are described below:

- **Kepler:** Provides cumulative pod-level Energy consumption metrics for a Kubernetes cluster. Chosen due to its inference capabilities, to make up for lack of RAPL support on AMD-CPU host machines. [1]
- **ElectricityMaps:** Provides hourly updates on regional CO2 Intensity (grams of CO2 equivalent per kWh) these are then used for CO2 metrics calculation and taken into account for a distributed system's possible deployment regions. [2]

3.2.3 Interfaces

The Analytics CO2 Monitoring component interacts through the following interfaces:

- **Internal Interfaces:**
  - **Kepler Exporter:** HTTP GET requests to http://kepler.kepler.svc.cluster.local:9102/metrics to retrieve kepler_container_joules_total metrics for power consumption.
  - **Electricity Maps (Web Scraping):** requests to electricitymaps.com to obtain CO2 intensity data for various countries.
  - **TimescaleDB Database (Optional):** When configured in "db" mode, the component interacts with a TimescaleDB database for persistent storage of container metrics, including time, pod ID, namespace, CO2 emissions, energy consumption, country ISO2, and container name.
- **External REST API Interfaces:** The component exposes a Flask-based REST API with the following endpoints:
  - GET /api/containers: Returns a JSON list of active (pod, container) pairs being monitored.
  - POST /api/power-consumption: Accepts a JSON payload (e.g., {"n": 5}) to retrieve the last n power consumption samples for all monitored containers.
  - POST /api/co2-per-container: Accepts a JSON payload (e.g., {"pod": "my-pod", "container": "my-container", "n": 5}) to calculate and return the CO2 emissions (grams_co2_per_sec) for a specified container based on its power consumption and the CO2 intensity of the associated country.

3.3 Smart Deployment

3.3.1 Description

The Smart Deployment (SD) component aims to provide the rest of the WP5 components with an easy and powerful way to deploy models.

Nowadays, almost every human action is somehow recorded, indicating that there is a vast quantity of data available for exploitation. Machine Learning (ML), a branch of Artificial Intelligence, focuses on this task. It offers a set of tools and statistical techniques aimed at using this data to teach machines capabilities similar to human intelligence. We identify two main and distinct roles that collaborate in the implementation of a successful ML project: The Data Scientist and the MLOps Engineer. While the former develops the model, the latter focuses on its deployment.

The SD component is developed as a bridge between the Data Scientists and the MLOps engineers. The Data Scientist task is to create new models, while the MLOps engineers oversee the proper deployment of those ML models. The SD is the key in this scenario: it enables model deployment with a focus on minimizing the CO_2 emissions, helping the MLOps engineer with a centralized way in an easy-to-use interface, handling the management of the necessary Kubernetes resources to achieve this task.

It is important to note that the SD does not cover the training phase: this task is assigned to the Data Scientist. Given an ML model, the component can track it from the model repository and deploy it through a Docker image on a Kubernetes cluster.

The principal objective of the SD component is to establish a system capable of deploying a model given an infrastructure. To achieve this, several aspects have been addressed:

- Models must be versioned, registered and made available to the SD component.
- Orchestration capabilities are required to perform all the different steps involved in model deployment, from identifying the model and giving the correct access to the model, making it available to users/clients.
- A Docker image is required to serve the model.

3.3.2 Technical Specification

3.3.2.1 Component-level C4 Architecture

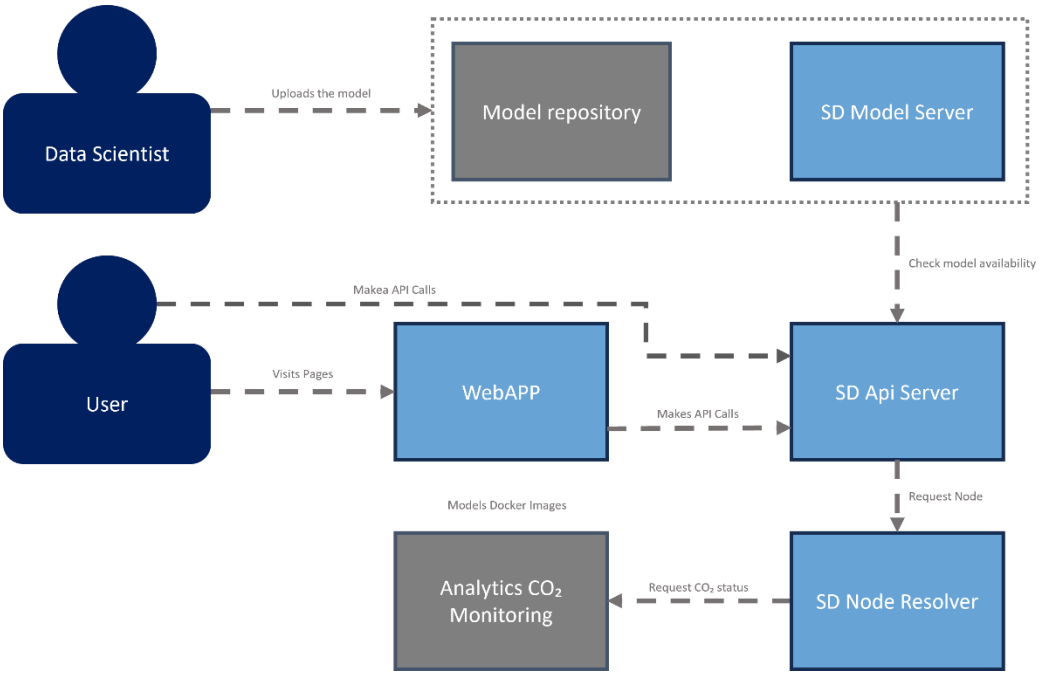


Figure 6: Smart Deployment component C4 architecture

This component is integrated within the FAME marketplace to provide customers with a user-friendly way to deploy the models developed in WP5. The customer will be able to decide which components of the SD want to purchase.

A high-level schema is shown inFigure 7. The SD component is made to be deployed on Kubernetes, and is divided in four parts:

- **API-Server**. Builds everything needed by the model to be deployed.
- **Node-Resolver**. Decides in which node the model will be deployed on.
- **Model Server**. Serves the model itself. MLFlow is needed for this module to work correctly.
- **WebAPP**. Gives a GUI over the SD API-Server.

Additionally, the models can be stored in an MLFlow storage, where they can be tracked and versioned.

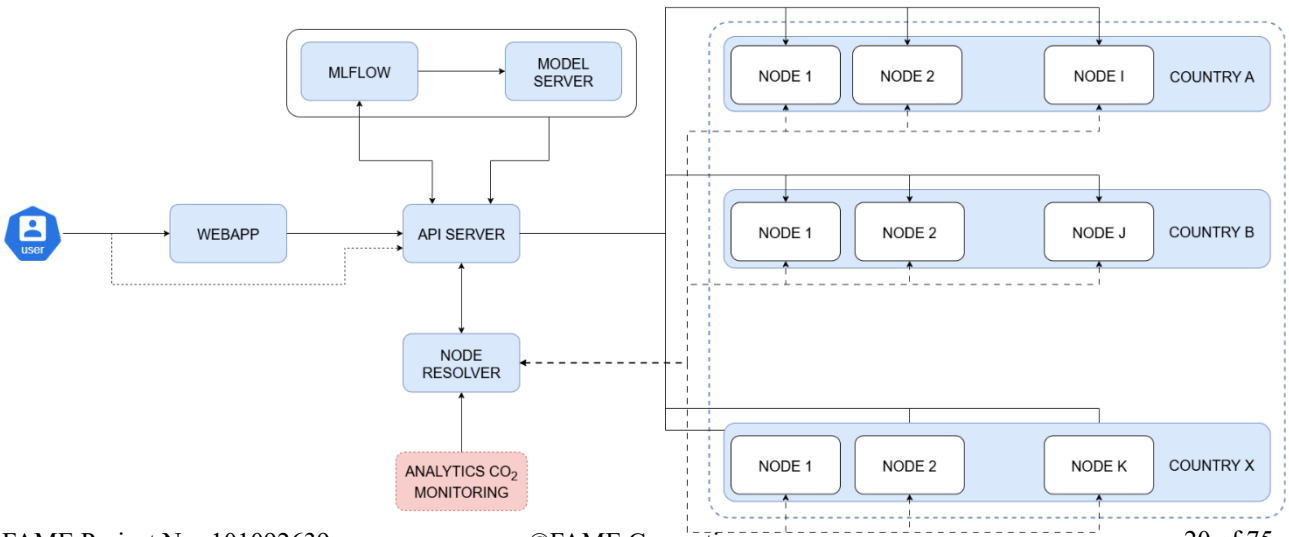


Figure 7 - Smart Deployment schema

3.3.2.1.1 API-Server

The API-Server (SD AS) is the core of the SD component. It is a pod inside the Kubernetes cluster with which the user connects and gives the necessary information about the model to be deployed through a REST API. This module manages all aspects related to the deployment of the models, including Kubernetes secrets, services, config-maps and deployments.

The user makes a request to the SD AS with all the information needed to build the Kubernetes resources: the name of the model, the base Docker image to deploy, the exposed port by the pod, etc. All the input needed for the model deployment, is specified in annex 0.When this is information is received, the SD AS takes the following steps:

1. It confirms the existence of the model in the MLFlow instance, passing the necessary information via secrets already existing in the cluster.
2. It confirms that there is a node with resources to accept new models. This confirmation is made via the connection of the node-resolver, which connects with the Analytics CO_2 Monitoring and selects the node with available resources and the lowest CO_2 footprint.
3. It builds the Kubernetes resources specifications and checks the non-existence of models with the same name and avoid possible incoherences of the specifications.
4. Execute the resources and returns the necessary information for the user to access the model.

The SD AS brings other functionalities to the user, such as:

- List the available models,
- Model redeployment,
- Deployed model deletion,
- Creation/modification of secrets with credentials.

3.3.2.1.2 Node-Resolver

The Node-Resolver (SD NR) is a necessary complement of the SD AS. This module has been built as a bridge between the SD AS and the Analytics CO_2 Monitoring, with the ability of decision to decide the node which is the best option to deploy the model.

Firstly, the SD NR receives the petition from the SD AS to receive the node. Then connects with the Analytics CO_2 Monitoring, which returns a list of countries with their CO_2 emissions. Once this information is given to the SD NR, it sorts the list by increasing CO_2 and starts searching for an available node in the country with less emissions, checking the node availability. Then, the SD NR returns the node where the model must be deployed to the SD AS.

If the Analytics CO_2 Monitoring is not reachable (or is not deployed) by the SD NR, it selects a random available node, checking if the node is available to accept new deployments.

3.3.2.1.3 Model-Server

The Model Server (SD MS) is a Docker image that contains everything needed for the model to perform inferences. This image must be available to the Kubernetes cluster, either through a public or private repository, giving the credentials if necessary.

There is a base Model-Server that is built as a REST API that downloads the model from a MLFlow instance and then serves it through an endpoint, but the SD system can deploy any other server provided by the user.

3.3.2.1.4 WebAPP

This module has been made to bring to the users a more comfortable experience than the SD API-Server to deploy the models.

This is a WebAPP where the deployment is made by a user-friendly manner, as it provides a GUI to the user that replicates all the operations made by the SD API-Server, i.e., the WebAPP module uses the SD API-Server as a base for all the operations.

3.3.2.2 Baseline Technologies and Tools

All the modules have been developed using Open Containers Initiative (OCI) according to the standard for Machine Learning Operations (MLOps), where the workloads and services are containerized, facilitating both declarative configuration and automation.

The baseline technologies used in the development and operation of the Smart Deployment are:

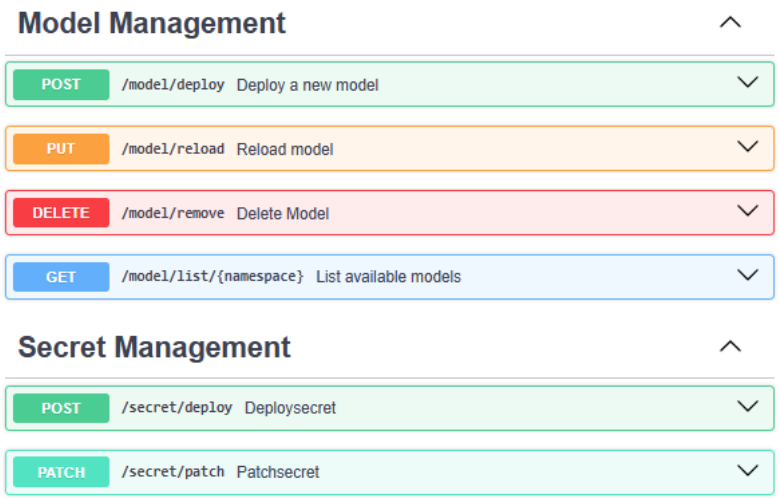
- **Kubernetes[1].** Open-source system for automating deployment, scaling, and management of containerized applications.
- **Podman[2].** Open-source container image management engine.
- **FastAPI[3].** Web framework for building Python-based APIs on standard Python type hints.
- **MLFlow[4].** A machine learning platform designed for end-to-end lifecycle model management, offering advanced experiment tracking, a model registry, and a model serving feature.
- **Streamlit[5].** A Python framework for building WebAPP.

3.3.3 Interfaces

The interfaces from all the modules, except the SD WebAPP, are REST APIs.

3.3.3.1 SD API-Server

In the image bellow are the endpoints available to manage the SD system. A detailed view for each endpoint with its own input is shown in the annex SD References.



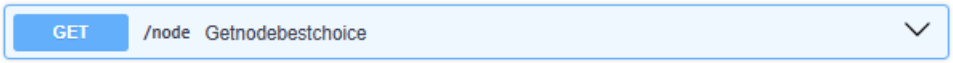
[1] https://kubernetes.io/
[2] https://podman.io/
[3] https://fastapi.tiangolo.com/
[4] https://mlflow.org/
[5] https://streamlit.io/

The endpoints are:

- Model management
  - */model/deploy*. Deploy a new model.
  - */model/reload*. Reload an already deployed model.
  - */model/remove*. Remove an existent model.
  - */model/list/{namespace}*. List all the deployed models. Return the given name, the country and the status.
- Secret management
  - */secret/deploy*. Create a secret.
  - */secret/patch*. Modify an existent secret.
  -

3.3.3.2 SD Node-Resolver

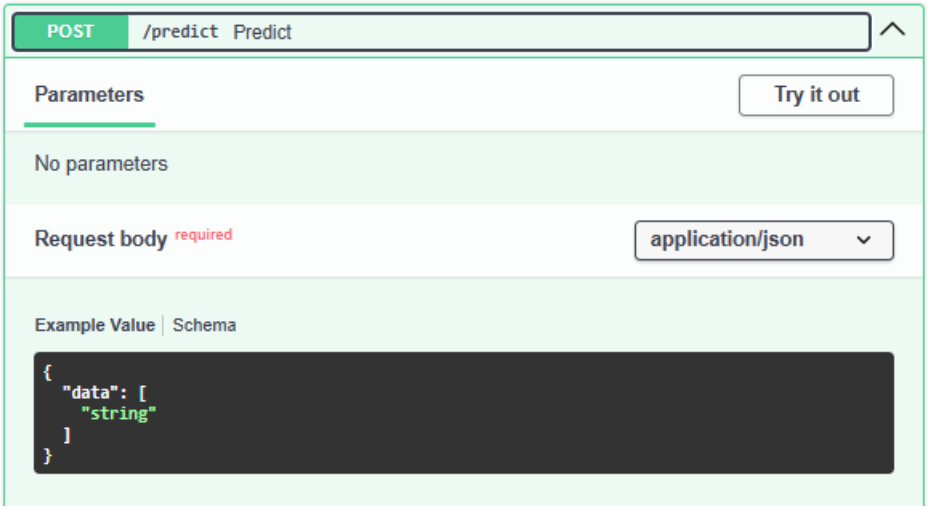
The SD Node-Resolver only has one available endpoint, and this is only used by the SD API-



Server:

This endpoint returns the node where the model must be deployed.

3.3.3.3 SD Model-Server



The SD Model-Sever only has one available endpoint to make predictions

The user must provide a list with the information to make a prediction.

3.3.3.4 SD WebAPP

This module is an extension of the SD API-Server. With a more user-friendly view, the user can perform the same actions than the SD API-Server, for instance:

- Deploy a new model: Here, the user can enter manually the model specification, and then, with a simple click deploy the model, as shown in Figure 8.

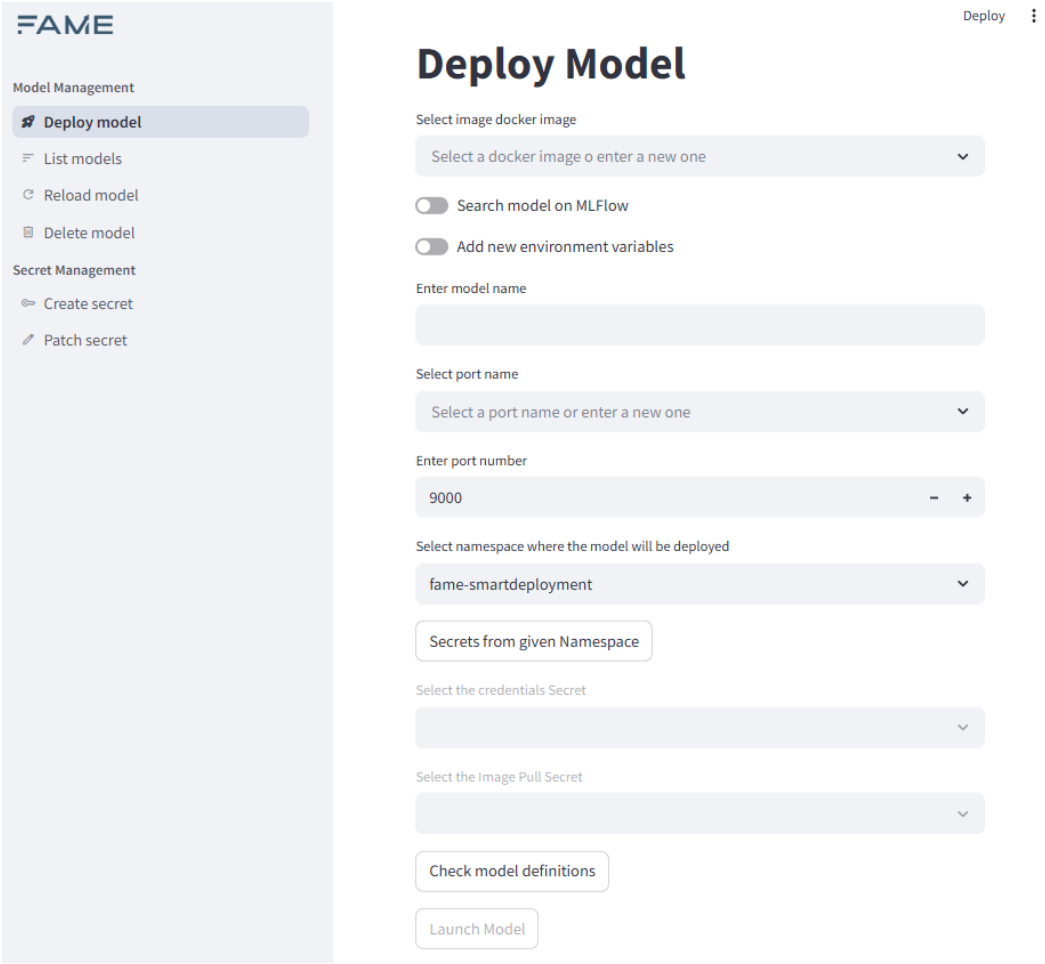


Figure 8 - SD WebAPP Deploy model page

- List the existent models: In this case, in addition returns an interactive map where the user can see where the models are deployed, as shown in Figure 9.

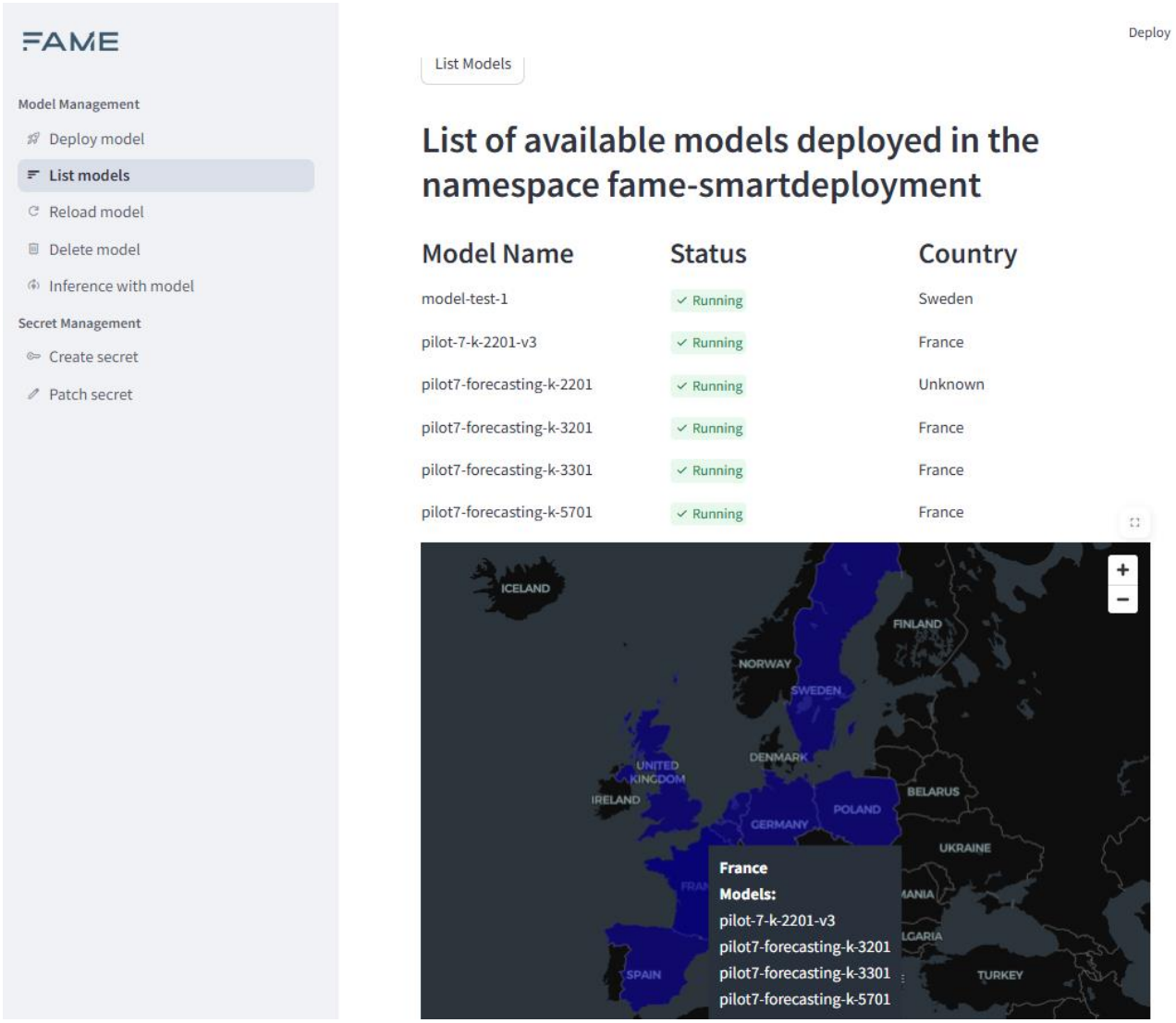


Figure 9 - SD WebAPP List models page

3.4 FML Framework

This section presents the FML framework indexed in the FAME marketplace to provide its customers with Federated Learning capabilities.

3.4.1 Description

Federated Learning (FL) is a distributed learning approach that enables the training of a common model across several clients while maintaining data privacy. As reported in D5.2, this learning paradigm provides opportunities to improve the industry by leveraging distributed datasets that otherwise could not be used due to privacy and data protection issues.

In its most basic deployments, FL consists of multiple clients that train a local copy of the model with their local data, then transmit this model to a central server, which in turn aggregates the models arriving from all clients. This process is repeated until the aggregated model converges. Since data remains local to the client that collected it and only the locally trained models are shared, data privacy is maintained throughput the whole process. In order to prevent data leakage from the trained models, secure sum (Bonawitz, et al., 2017) and differential privacy (McSherry & Talwar, 2007) can be applied.

In addition to the traditional client-server architecture, hierarchical and swarm deployments are also common in FL. As depicted in Figure 10, hierarchical deployments consist of multiple layers of federations. A first layer of federated orchestration occurs at the silo level, where all components typically are deployed by the same entity. In the next level, the server aggregates each silo's model, thereby obtaining the global model. On the contrary, in swarm deployments there is no centralized server, but the server role is distributed among all the nodes participating in the federation.

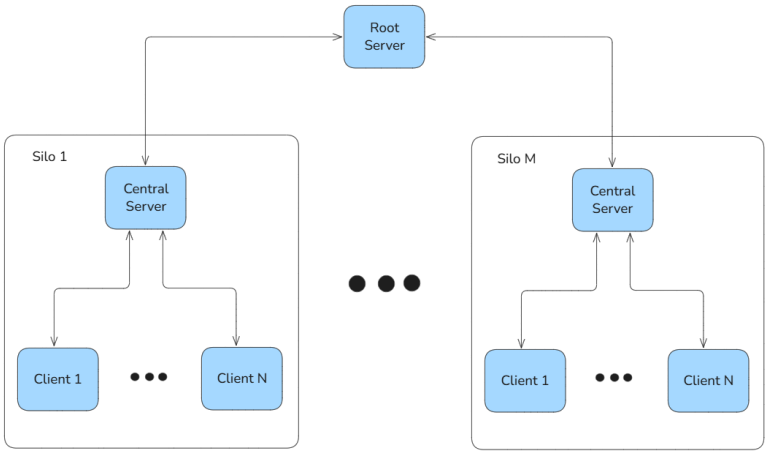


Figure 10: Hierarchical Deployment

As described in Section 3.4.2.2, the FML framework is designed in a modular manner, thereby enabling all three deployments described above. However, in the remainder of this document we focus on the client-server and hierarchical deployments, as they align better with the FAME marketplace and FAME pilot needs.

3.4.2 Technical Specification

3.4.2.1 Component-level C4 Architecture

This subsection provides a more detailed description of the different components that made up the FML framework indexed in the FAME marketplace. Figure 11 depicts the C4 Architecture diagram with all the different components in the framework, the interactions between them and other components of the FAME marketplace, as well as the user and service providers. The main

differences with the C4 component diagram reported in D5.2 are i) the FML Central Server component to enable hierarchical deployments, ii) the Data Drift Monitoring component to perform continual learning, and iii) the direct connection between the Model Registry and the FDAC, such that the FML Root Server does not directly index the model in the FDAC, but the Model Registry entry is used instead. Moreover, the FML Orchestrator is no longer present, since it is not considered a core component of the FML framework itself, but the orchestration is performed by Docker or Kubernetes (see Section 4.4.1).

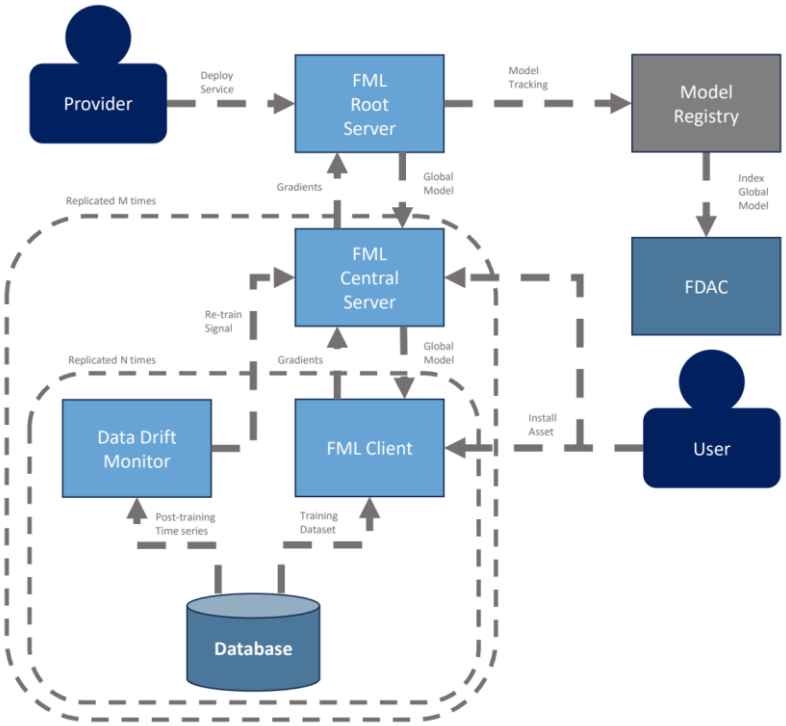


Figure 11 - FML Framework C4 Architecture Diagram

The *FML Root Server* implements the necessary functionalities for client subscription and selection, model aggregation and optimization, and model broadcasting. The implemented client selection mechanism randomly samples the set of all subscribed clients, and it is a key feature of the framework, as it has a direct impact on the resource usage (i.e., training, transmission, and aggregation) in each round, while still allowing the model to converge to a good performance by exploiting clients with high quality data (Abdelmoniem, Sahu, Canini, & Fahmy, 2023). The percentage of clients participating in each round is a configurable parameter by the *Provider* of the service. After every model update, this component tracks the new global model in a *Model Registry*. The final model is indexed and published in the FDAC.

The FML Root Server exchanges gradients and models with its clients. In a traditional client-server deployment, the *FML Client* component trains the model with its local dataset and forwards it to its server, while the *FML Central Server* acts as the combination of a client and a server in hierarchical deployments, i.e., it forwards the model to its server after aggregation. This is possible because both components share the same endpoints for gradient transmission and global model reception, as well as a fully configurable client subscription mechanism.

The FML Central Server has a double function. On the one hand, it acts as the silo server in the intra-silo rounds by aggregating the silo's model. On the other hand, in cross-silo rounds it acts as a client for the root server. This component is only required in hierarchical deployments and all the other components in the diagram above can operate without it.

The FML Client component begins its execution by subscribing to its configured server (i.e., FML Root/Central Server, depending on the deployment). Once the training starts, this component trains the model with local data upon request and sends the locally updated model back to the server. Each client has access to a local *Database*, which can be queried to obtain the training dataset.

The final component is the *Data Drift Monitor*, which has been jointly developed with T5.4 to improve the performance of the system in production environments. All the components described above take part on the model training phase. However, the Data Drift Monitor component operates once the model has already been trained and deployed, namely the production phase. As new data is produced (e.g., sensor timeseries), model retraining will only be required if the statistical deviation of the new data with the training dataset is large enough that the trained model does not generalize well for it. The Data Drift Monitor perform such a statistical analysis and triggers a new training phase whenever is required.

3.4.2.2 Baseline Technologies and Tools

The implementation of the components described in Section 4.4.2 is based on Eviden's proprietary framework for FL (the FL framework in the following). The FL framework is based on the pipes (or *wires*) and filters (or *pods*) paradigm:

- Pods operate on input objects to generate output objects.
- Wires transport objects from one pod to the next one.

The FL framework is implemented in Python and is shared with the consortium as an obfuscated wheel file.

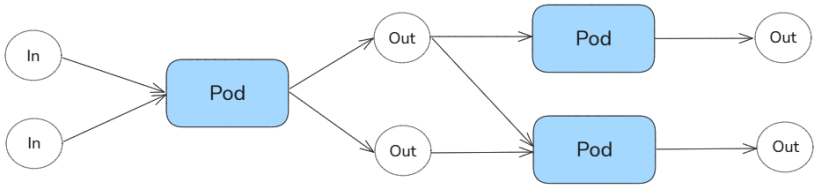


Figure 12 - Pipes and Filters

Figure 12 depicts an illustrative pipeline showing how multiple pods can be interconnected. Following this design paradigm allows the FL framework to implement complex behavior via the interconnection of simple pods. As a result, the FL framework is highly customizable, as more pods can be added with relative ease and integrated into existing pipelines to enhance or increase their functionalities. Figure 13 and Figure 14 show how the FML Root Server and FML Client described in Section 3.4.1 can be implemented as a pipe of pods in the FL framework, where the blue boxes represent pods and the labels at both end of the arrows the input and output wires.

The Server begins the execution with the *Starter* pod by triggering a *Keras* pod to load the base model, which is in turn broadcasted to the clients via an *HTTP* pod. A *Server* pod manages the federated rounds by gathering client updates. As enough updates are gathered in a *Collector* pod, all the updates are aggregated into a single model in an *Aggregator* pod. Finally, the *Keras* pod evaluates the performance of the aggregated model, for which the test dataset is loaded with a *CSV* pod.

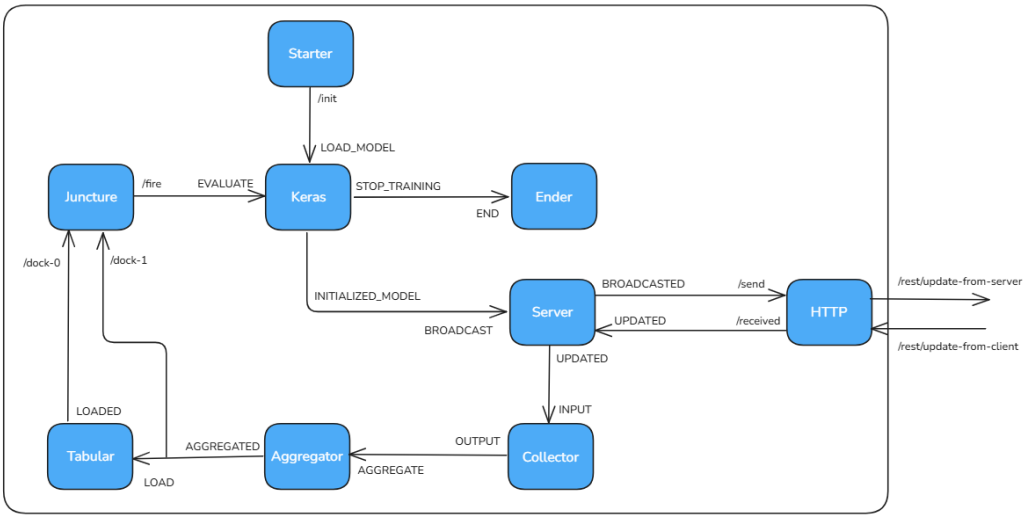


Figure 13 - Server pods

In the other communication end, the Client awaits updates from the server with its own HTTP pod. The *Client* pod performs the round management at the client side. As a new update arrives, the CSV pod loads the local dataset. Both the model update from the server and the loaded dataset are merged into a single request by a *Juncture* pod before model training in the *Keras* pod. The trained model is then transmitted to the server.

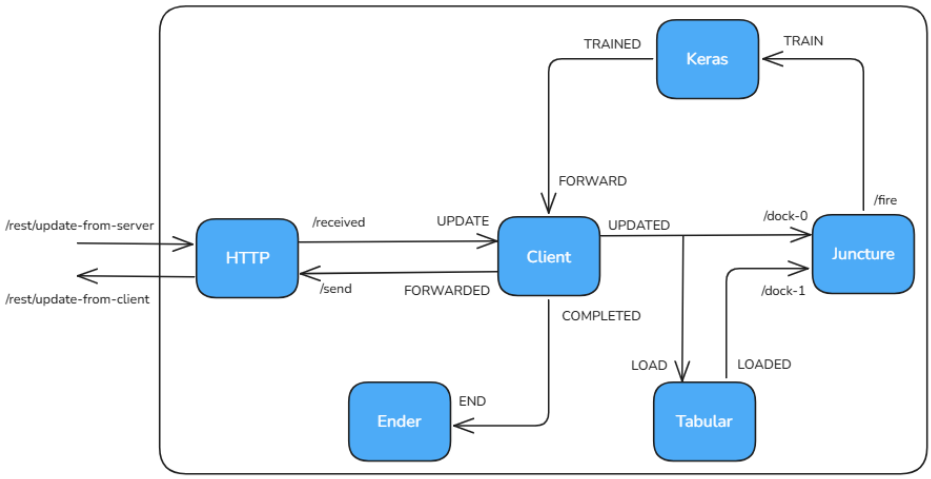


Figure 14 - Client pods

The server.py and client.py above represent an example of the minimum set of pods required to train in a federated manner using the FML framework. However, the server and client functionalities can be extended using other pods available in the framework. The following table collects the most important pods implemented in the FML framework. Although the framework supports the most common ML frameworks (i.e., Keras/TensorFlow, PyTorch, and scikit-learn) by default, it also provides a high customizability with a *Custom* trainer pod, which allows the user to define the behavior of this pod tailored to their needs. Similar custom pods are also available for data loading and model aggregation. These have been key components in the developments of federated models in the FAME project (see Section 4.4).

Table 3: Most important pods implemented in the FML Framework

Pod Type	Support
Data Loaders	Tabular, Image, Custom
Trainers	Keras, PyToch, scikit-learn (DNNs, linear regression, decision trees, and KMeans), Custom
Aggregators	FedAvg, FedOpt (Adam, Adagrad, Yogi), Krum, Median, Trimmed-mean, GeoMedian, Custom, KMeans
Communication	HTTP, Kafka
Privacy	Secure-sum, differential privacy, TLS
Third-party apps	MLFlow, MinIO, Prometheus, Grafana, Evidently

Table 4 - FML framework pod types

The following sections introduce in more detail the pods that have been implemented for the FML Framework in the context of FAME.

Client Subscription and Selection

Originally, the client configuration mechanism implemented in Eviden's FML framework was manual, i.e., the clients were configured at the server by the user via environment variables. This mechanism was designed with simplicity in mind. However, using this mechanism becomes unfeasible as the number of clients grows (see Section 4.4.2.2). To provide support for FL to Pilot 7, an automatic client subscription has been implemented with two new pods.

The *Subscriber* pod generations the subscription and unsubscription request at the client side. Its constructor takes as input attributes the *client_id*, and subscription *metadata*. Currently, the IP/domain name and port must be transmitted to the server as metadata. An example of client subscription request looks as follows:

{'id': 'client_one', 'client_one': {'ip': '127.0.0.0', 'port': 80}}

The *SubscriptionManager* pod maintains a list of clients at the server side. Thus, it provides the mechanisms for subscription and unsubscriptions.

As the number of clients grows, using all of them in the same federated round becomes i) unfeasible, since the load at the server increases linearly with the number of clients per round, ii) improbable, as clients may crush and their updates never be received, and iii) unnecessary, as training with a subset of all available clients has been proved to approach the performance obtained when training with the complete set (Abdelmoniem, Sahu, Canini, & Fahmy, 2023). Two pods have been implemented to provide support for dynamic client selection. The *RandomSelector* pod takes the percentage of elements to select from a list, and outputs the selected elements, randomly drawn following a uniform distribution. The *RoundRobinSelector* pod generates a subset from a list in a round-robin fashion.

Fault-Tolerant Round Management

As the number of clients grows, so does the probability that a client crushes in the middle of a federated round or that stragglers appear that delay the round completion. Therefore, a timeout round management system is required in order to improve the FML framework's scalability. Otherwise, the server would keep waiting for the update from all clients forever. A *Timer* pod has been implemented that, once the START wire has been triggered, it signals TIMEOUT after a configurable period.

Server and Client pods

The *Server* and *Client* pods are central to the correct operation of the FL training with the framework presented here. On either communication side, these pods manage the federated rounds, client subscription, and model transmission. In other words, they orchestrate the complete FL process. In view of the newly added pods mentioned above, these pods have been reimplemented: from a monolithic implementation, they have been turned into a collection of smaller, simpler pods. This refactoring results in a more scalable FML framework thanks to its support for a larger client cohort and its tolerance to client failures.

KMeans

Eviden's framework has been extended to provide support for Federated KMeans (Garst & Reinders, 2024). A *ScikitLearnKMeans* pod has been implemented, which uses scikit-learn's *KMeans* library to perform the clustering. This pod implements the traditional operations with KMeans models, namely loading, saving, training, and evaluating. Following (Garst & Reinders, 2024), the *KMeansAggregator* pod aggregates the cluster centroids received from all clients by in turn applying KMeans on the centroids themselves.

Evidently

Model training is completed once the model's performance on the test set is high enough. Only then is the model deployed in production and interacts with the real environment it was designed to operate on. The evaluation on the test set ensures the model generalizes well to unseen data in the training phase. However, it can still occur that, once in production, the model faces new data whose statistical distribution significantly differs from the train and test sets, thereby resulting in poor performance.

Continual learning (Hadsell, Rao, Rusu, & Pascanu, 2020) addresses this issue by detecting these statistical differences in real-time once the model is put in production. As a result, re-training is only triggered when strictly necessary and saving precious resources while the model performs well.

Evidently is an open-source Python library providing data drift evaluation metrics off-the-shelf. This continual learning library has been integrated into the FML framework with the implementation of the *EvidentlyMonitor* pod. This pod provides the necessary wires to trigger the evaluation of data drift monitoring, signalling when it has been detected. It is up to the user how to react to a data drift detection, e.g., triggering a complete model re-train adding the new data in the training dataset, forcing the inclusion of the client detecting the drift in the next federated round, etc.

3.4.3 Interfaces

This section describes the interfaces for the root server, center server, and client components described above. These interfaces are implemented as REST APIs, on which the components expect a Python dictionary with certain entries. The REST APIs are exposed via the HTTP pod and are in turn connected to the destination pod that triggers the desired behavior.

The root server interfaces are:

- */rest/subs*: it triggers a client subscription. The root server expects a dictionary with two entries, i) *"id":* <*client-id*>, and ii) <*client-id*>, which in turn contains a dictionary with the *"ip"* and *"port"* of the client.
- */rest/upstream*: it triggers the collection of *"weights"* from the clients. Once enough weights are collected (or a timeout is triggered), the server proceeds to aggregate them.

The center server interfaces are:

- */rest/subs*: it triggers client subscriptions in the center server. It expects the same dictionary as the */rest/subs* interface for the root server above.
- */rest/downstream*: it forwards the *"weights"* from its server to its clients.
- */rest/upstream*: it triggers the same behavior as the */rest/upstream* interface in the root server above.
- */rest/terminate*: it triggers execution termination.

The client interfaces are:

- */rest/downstream*: it triggers model training with received *"weights"*.
- */rest/terminate*: it triggers execution termination.

4 Components Demonstration

4.1 Incremental Analytics

The *Incremental Analytics* component serves as the data management system of the entire FAME solution, offering some advanced capabilities developed within the project, which enable both energy efficient and incremental analytics to the data users. As depicted in **Errore. L'origine riferimento non è stata trovata.**, the novel *federated incremental analytics* component consists of several sub-components that interact with each other internally in order to provide the overall functionality. These consists of the datastore offering the vanilla *Incremental Analytics* functionality, the newly developed *federation-agent* that enables the use of the later under federated machine learning scenarios, and a RabbitMQ message broker. With regards to the datastore offering the vanilla incremental analytics, details about the prerequisites, installation guidelines and documentation has been provided in the previous version of this deliverable. These are still relevant and up-to-date, so this section will provide information about the rest of the components: the *federation-agent* and the RabbitMQ message broker.

4.1.1 Prerequisites, Installation Environment and Documentation

There are several ways that someone could install the overall environment to run the overall *Incremental Analytics* component. It could be either installed locally on the client's premises, or to be offered in an as-a-Service manner. Moreover, in cases of local deployments, it could be installed on a bare metal, or the client has the possibility to make use of predefined docker images and create a running container. As this *Incremental Analytics* is built upon the background proprietary technology of LeanXcale DB, it is out of the scope of the project to provide the binaries of this component directly to third parties to enable remote deployments on their premises. It can be only deployed using docker images. The overall documentation has been provided in the previous version of this deliverable.

Regarding the *federated-agent,* this is released as open source and can be manually compiled and executed using the source code. The source code is available under the FAME's private Gitlab[6] and can be downloaded in the local machines of the users. The prerequisites for the compilation are the existence of Java 17 along with Apache Maven 3.5. However, our solution is released with a Dockerfile to let the users build the corresponding images locally and experiment,

The first step towards the installation of the *federated-agent* component is to create a docker image for our solution. Given that a user can have access to the source code of the component, we have provided a *dockerfile* that can be used to create that image. This can be depicted in the following code snippet:

```
FROM ubuntu:20.04

MAINTAINER  Pavlos Kranas <pavlos@leanxale.com>

RUN apt-get update && apt-get -y install software-properties-common \
 && apt-get update \
 && apt-get -y install screen iputils-ping netcat vim vsftpd net-tools \
 openssh-server virtualenv telnet apt-transport-https openjdk-17-jdk curl maven

#install leanxcale driver locally
```

[6] gitlab.gftinnovation.eu/fame/lf-leanxcale-agent

```
COPY jars/qe-driver-1.9.13_latest-jdbc-client.jar /tmp
RUN cd /tmp \
 && mvn install:install-file -Dfile=/tmp/qe-driver-1.9.13_latest-jdbc-client.jar -
DgroupId=com.leanxcale -DartifactId=qe-driver -Dversion=1.9.13 -Dpackaging=jar



RUN mkdir /tmp/src
COPY pom.xml /tmp
COPY src/ /tmp/src
RUN cd /tmp \
 && mvn clean install -DskipTests


EXPOSE 22 23 8081

WORKDIR /tmp

CMD cd /tmp \
 && java -jar target/federation-leanxcale-agent-0.0.1-SNAPSHOT.jar
```

The docker image makes use of the base image of Ubuntu 20.04 and after installing the prerequisites for the technology that is required (i.e. Java 17, maven, etc.), the source is being copied from the local filesystem into the image. Before compiling the source, we install in the docker image the dependency for the driver to connect to the datastore. A jar file that contains the driver has been previously copied, and then we rely on maven to install this dependency. Once installed, we make use of maven to compile the source core. The final command starts the microservice upon start up.

It is important to highlight that the creation of this docker image is the responsibility of the CI pipelines defined for the project, where more details can be found at the relevant D2.3 (“Integrated FAME Data Marketplace”). That means that whenever we finalize (or we solve a potential issue or bug) a development cycle and we merge new updates into the main distribution, the latter is uploaded to the GitLan and the CI pipeline is triggered, using the aforementioned *dockerfile* to create the image that will be now available for the FAME integrated solution. It is also important to mention that the same image can be used for acting as either the client or the master in the federated machine learning scenario.

Having the image already created, we rely on the Kubernetes orchestration to actually deploy our implementation to the FAME’s cluster, or to possible external resources in case we foresee the installation of the overall FAME solution remotely. In order to do that, we need to create a number of different *resources* in the Kubernetes environment. For what concerns the vanilla *Incremental Analytics,* the aforementioned resources have been described in the previous version of this deliverable.

For what concerns the *federated-agent*, first we need to provide the resources for the RabbitMQ message broker. For that, we would need to define the network resource to enable access from both the clients and master nodes. The following code snippet depicts this:

```
apiVersion: v1
kind: Service
metadata:
  name: fame-fl-federation-lx-agent-rabbitmq-service
  namespace: fame
  labels:
    app: fame-fl-federation-lx-agent-rabbitmq
spec:
  ports:
    - name: "5672"
```

```
      port: 5672
      targetPort: 5672
    - name: "15672"
      port: 15672
      targetPort: 15672
  selector:
    app: fame-fl-federation-lx-agent-rabbitmq
  type: ClusterIP
```

We enable access to both 5672 and 15672 ports for the communication of the microservices and the administration web console.

Having set up the network, the following code snippet deploys the RabbitMQ:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fame-fl-federation-lx-agent-rabbitmq
  namespace: fame
  labels:
    app: fame-fl-federation-lx-agent-rabbitmq
spec:
  serviceName: fame-fl-federation-lx-agent-rabbitmq-service
  replicas: 1
  selector:
    matchLabels:
      app: fame-fl-federation-lx-agent-rabbitmq
  template:
    metadata:
      labels:
        app: fame-fl-federation-lx-agent-rabbitmq
    spec:
      containers:
        - name: fame-fl-federation-lx-agent-rabbitmq
          image: rabbitmq:3-management
          imagePullPolicy: Always
          ports:
            - containerPort: 5672
            - containerPort: 15672
          env:
          - name: RABBITMQ_DEFAULT_USER
            value: "user"
          - name: RABBITMQ_DEFAULT_PASS
            value: "password"
      restartPolicy: Always
      imagePullSecrets:
      - name: regcred
```

We provide the credential to the RabbitMQ as environment variables inside this resource, and we rely on the *rabbitmq3-management* official image. We additionally assign the already defined network service to this instance.

Having the RabbitMQ deployed, we would now need to deploy the master node of the *federated-agent*. As before, we need to create the network service first:

```
apiVersion: v1
kind: Service
metadata:
  name: fame-fl-federation-lx-agent-server-service
  namespace: fame
  labels:
    app: fame-fl-federation-lx-agent-server
```

```
spec:
  ports:
    - name: "8081"
      port: 8081
      targetPort: 8081
  selector:
    app: fame-fl-federation-lx-agent-server
  type: ClusterIP
```

Then, we will define the environment variables of this component in a configuration mapping file, as the following code snippet depicts:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fame-fl-federation-lx-agent-server-configmap
  namespace: fame
data:
  app.rabbitmq.hostname: "" #rabbitmq hostname, i.e fame-fl-federation-lx-agent-rabbitmq-
service.fame.svc.cluster.local
  app.rabbitmq.port: "5672" #rabbitmq port
  app.rabbitmq.username: "user" #rabbitmq username
  app.rabbitmq.password: "password" #rabbitmq password
  app.server.hostname: "" #hostname of the server agent, i.e fame-fl-federation-lx-agent-client-
service.fame.svc.cluster.local
  app.server.port: "8081" #port of the server agent, i.e fame-fl-federation-lx-agent-server-
service.fame.svc.cluster.local
  app.database.domain: "" #hostname of the server database, i.e leanxcaledb-
service.fame.svc.cluster.local
  app.database.port: "1529" #port of the server database
  app.database.name: "MOH" #logical database name to write data
  app.database.username: "APP" #logical database schema/user to write data
```

In the code snippet, there are self-explanatory comments for the purposes of each variable. To summarize, the user needs to define the RabbitMQ hostname, port and credentials, the hostname, port and credentials to the respective master data node, along with some details of each its own that will be published, like the hostname and port to be deployed.

Once the configuration mapping is defined, the following code snippet can be used to deploy the master node of the *federated-agent*:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fame-fl-federation-lx-server-client
  namespace: fame
  labels:
    app: fame-fl-federation-lx-server-client
spec:
  serviceName: fame-fl-federation-lx-agent-server-service
  replicas: 1
  selector:
    matchLabels:
      app: fame-fl-federation-lx-agent-server
  template:
    metadata:
      labels:
        app: fame-fl-federation-lx-agent-server
    spec:
      containers:
        - name: fame-fl-federation-lx-agent-server
          image: harbor.gftinnovation.eu/fame/federation-leanxcale-agent:0.0.1
```

```
        imagePullPolicy: Always
        ports:
          - containerPort: 8081
        env:
        - name: RABBITMQ_HOSTNAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.rabbitmq.hostname
        - name: RABBITMQ_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.rabbitmq.port
        - name: RABBITMQ_USER
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.rabbitmq.username
        - name: RABBITMQ_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.rabbitmq.password
        - name: SERVER_HOST
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.server.hostname
        - name: SERVER_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.server.port
        - name: LX_DOMAIN
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.database.domain
        - name: LX_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.database.port
        - name: DATABASE_NAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.database.name
        - name: DATABASE_USERNAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-server-configmap
              key: app.database.username
      restartPolicy: Always
      imagePullSecrets:
      - name: regcred
```

As the code depicts, for the master node we make use of the *harbor.gftinnovation.eu/fame/federation-leanxcale-agent:0.0.1* image that was generated automatically by the FAME's CI pipelines using the aforementioned dockerfile. The image had been pushed to the project's private docker registry. Then, we assigned the already defined fame-fl-

federation-lx-agent-server-service network, while we rely on the previously defined configuration mapping to assign the values in the environment variables of the component.

Regarding the client nodes of the *federated-agent,* the following code snippet depicts how we can create the corresponding *service* resource to be used for networking. It is similar to what we used for the master node:

```
apiVersion: v1
kind: Service
metadata:
  name: fame-fl-federation-lx-agent-client-service
  namespace: fame
  labels:
    app: fame-fl-federation-lx-agent-client
spec:
  ports:
    - name: "8081"
      port: 8081
      targetPort: 8081
  selector:
    app: fame-fl-federation-lx-agent-client
  type: ClusterIP
```

Again, we need to define the corresponding configuration mapping for the client nodes, as depicted in the following code snippet:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fame-fl-federation-lx-agent-client-configmap
  namespace: fame
data:
  app.rabbitmq.hostname: "" #rabbitmq hostname, i.e fame-fl-federation-lx-agent-rabbitmq-
service.fame.svc.cluster.local
  app.rabbitmq.port: "5672" #rabbitmq port
  app.rabbitmq.username: "user" #rabbitmq username
  app.rabbitmq.password: "password" #rabbitmq password
  app.client.hostname: "" #hostname of the client agent, i.e fame-fl-federation-lx-agent-client-
service.fame.svc.cluster.local
  app.client.port: "8081" #port of the client agent
  app.server.hostname: "" #hostname of the server agent, i.e fame-fl-federation-lx-agent-server-
service.fame.svc.cluster.local
  app.server.port: "8081" #port of the server agent, i.e fame-fl-federation-lx-agent-server-
service.fame.svc.cluster.local
  app.database.domain: "" #hostname of the server database, i.e leanxcaledb-
service.fame.svc.cluster.local
  app.database.port: "1529" #port of the server database
  app.database.name: "MOH" #logical database name to write data
  app.database.username: "APP" #logical database schema/user to write data
```

The configuration mapping contains similar environment variables, as the one used for the master node. Finally, the following code snippet eventually deploys the client nodes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fame-fl-federation-lx-agent-client
  namespace: fame
  labels:
    app: fame-fl-federation-lx-agent-client
spec:
```

```
serviceName: fame-fl-federation-lx-agent-client-service
replicas: 1
selector:
  matchLabels:
    app: fame-fl-federation-lx-agent-client
template:
  metadata:
    labels:
      app: fame-fl-federation-lx-agent-client
  spec:
    containers:
      - name: fame-fl-federation-lx-agent-client
        image: harbor.gftinnovation.eu/fame/federation-leanxcale-agent:0.0.1
        imagePullPolicy: Always
        ports:
          - containerPort: 8081
        env:
        - name: RABBITMQ_HOSTNAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.rabbitmq.hostname
        - name: RABBITMQ_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.rabbitmq.port
        - name: RABBITMQ_USER
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.rabbitmq.username
        - name: RABBITMQ_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.rabbitmq.password
        - name: CLIENT_HOST
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.client.hostname
        - name: CLIENT_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.client.port
        - name: SERVER_HOST
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.server.hostname
        - name: SERVER_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.server.port
        - name: LX_DOMAIN
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.database.domain
        - name: LX_PORT
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
```

```
              key: app.database.port
        - name: DATABASE_NAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.database.name
        - name: DATABASE_USERNAME
          valueFrom:
            configMapKeyRef:
              name: fame-fl-federation-lx-agent-client-configmap
              key: app.database.username
      restartPolicy: Always
      imagePullSecrets:
      - name: regcred
```

It is important to highlight that we need to repeat the deployment of the client nodes to as much as data nodes client instances are available to the federated machine learning scenario, paying attention to properly configure the environment variables for each client.

Having everything up and running, let's assume that the client data nodes are being continuously ingested with IoT data from the local sensors. The following figure shows for instance a data table that has been loaded with IoT data.

K2201 1

SELECT "TIMESTAMP", COL22SI101, COL22SI101_I Enter a SQL expression to filter results (use Ctrl+Space)

	TIMESTAMP	COL22SI101	COL22SI101 ISVALID	COL22TI120	COL22TI120 ISVALID	COL22TI121	COL22TI121 ISVALID	COL22TI123
1	2017-01-01 00:00:00.000	9,799.456771	0	57.68009949	0	43.1905572	0	43.73602174
2	2017-01-01 00:05:00.000	9,799.782656	0	57.59870021	0	43.10280834	0	43.71652617
3	2017-01-01 00:10:00.000	9,799.466908	0	57.58242035	0	43.07692337	0	43.66320934
4	2017-01-01 00:15:00.000	9,799.571429	0	57.58242035	0	43.04420086	0	43.52901999
5	2017-01-01 00:20:00.000	9,799.26	0	57.58242035	0	43.0510384	0	43.46764374
6	2017-01-01 00:25:00.000	9,800.123333	0	57.5205543	0	43.07692337	0	43.53927511
7	2017-01-01 00:30:00.000	9,799.942422	0	57.4847374	0	43.02466503	0	43.41196663
8	2017-01-01 00:35:00.000	9,799.440885	0	57.55669698	0	42.97573566	0	43.3699646
9	2017-01-01 00:40:00.000	9,799.086667	0	57.57875193	0	42.87798617	0	43.3699646
10	2017-01-01 00:45:00.000	9,799.166667	0	57.48482422	0	42.78396896	0	43.3699646
11	2017-01-01 00:50:00.000	9,799.416667	0	57.44917784	0	42.78388214	0	43.33724209
12	2017-01-01 00:55:00.000	9,799.933333	0	57.57865891	0	42.76418351	0	43.27228546
13	2017-01-01 01:00:00.000	9,800.005547	0	57.43256588	0	42.7415817	0	43.27228546
14	2017-01-01 01:05:00.000	9,799.021107	0	57.45892871	0	42.6729892	0	43.27228546
15	2017-01-01 01:10:00.000	9,798.18	0	57.39333143	0	42.58868666	0	43.08929649
16	2017-01-01 01:15:00.000	9,799.287337	0	57.27619244	0	42.57875595	0	43.06715546
17	2017-01-01 01:20:00.000	9,800.027669	0	57.25665661	0	42.59471021	0	42.99568689
18	2017-01-01 01:25:00.000	9,799.675	0	57.19169998	0	42.49084473	0	42.96931352
19	2017-01-01 01:30:00.000	9,799.667331	0	57.19169998	0	42.47130775	0	42.97924423
20	2017-01-01 01:35:00.000	9,799.282656	0	57.19169998	0	42.45812074	0	42.96963912
21	2017-01-01 01:40:00.000	9,799.385	0	57.24395832	0	42.37020718	0	42.97924423

Figure 15: Raw data in the client data nodes

Then, the *Incremental Analytics* data store is calculated incrementally the aggregated information, so that it can respond with the corresponding result sets in an energy and time efficient manner. As described in the previous version of this deliverable, internally it creates specific data structures that are holding this information, preserving the data consistency on the same time while data is being continuously ingested. Figure 16 shows the aggregated information derived from the previous data table.

	TIMESTAMP	COUNT COL22SI101 ISVALID	COUNT COL22TI120 ISVALID	COUNT COL22TI121 ISVALID	COUNT COL22TI123 ISVALID	COUNT COL22TI111 ISVALID
1	2017-01-01 00:00:00.000	0	0	0	0	
2	2017-01-01 01:00:00.000	0	0	0	0	
3	2017-01-01 02:00:00.000	0	0	0	0	
4	2017-01-01 03:00:00.000	0	0	0	0	
5	2017-01-01 04:00:00.000	0	0	0	0	
6	2017-01-01 05:00:00.000	0	0	0	0	
7	2017-01-01 06:00:00.000	0	0	0	0	
8	2017-01-01 07:00:00.000	0	0	0	0	
9	2017-01-01 08:00:00.000	0	0	0	0	
10	2017-01-01 09:00:00.000	0	0	0	0	
11	2017-01-01 10:00:00.000	0	0	0	0	
12	2017-01-01 11:00:00.000	0	0	0	0	
13	2017-01-01 12:00:00.000	0	0	0	0	
14	2017-01-01 13:00:00.000	0	0	0	0	
15	2017-01-01 14:00:00.000	0	0	0	0	
16	2017-01-01 15:00:00.000	0	0	0	0	
17	2017-01-01 16:00:00.000	0	0	0	0	

Figure 16: Aggregated data in the client data nodes

Now that the *federated-agent* is up and running for the federated machine learning scenario, the data user can configure the various client nodes to subscribe to the master node, in order for the clients to periodically send this aggregated information. Firstly, the data administrator can visit the swagger page, where the OpenAPIs are documented. In Figure 17, all OpenAPIs available are illustrated, both internal and external. This is for demonstrating purposes and in real scenarios, only the external API should be visible.

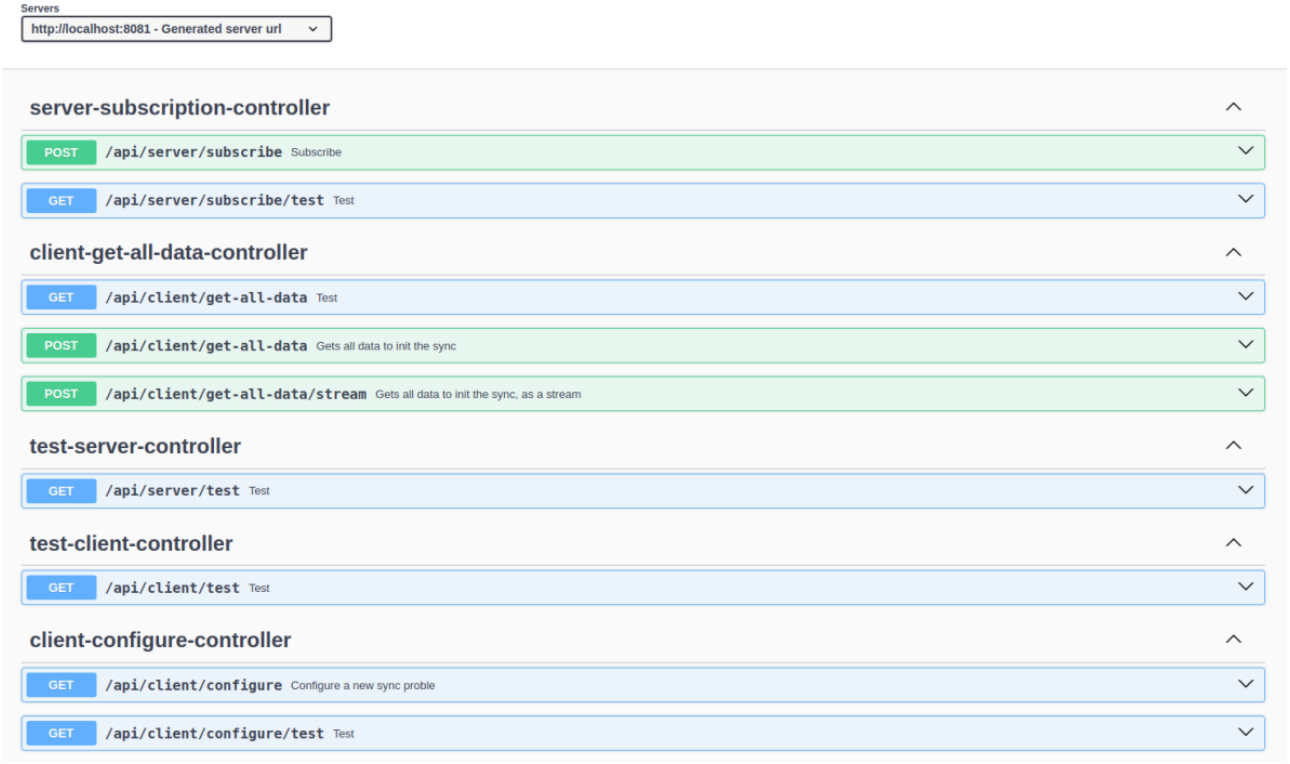


Figure 17: Federated Agent OpenAPIs

Given the interface specification provided in the previous section, the data user can invoke the /api/client/configure REST endpoint in order to configure each client node. This will trigger the client to subscribe itself to the master node, using its /api/server/subscribe internal endpoint.
Having established the communication flow, we can see that the internal meta-info data table created by the agents has been populated with information, as Figure 18 depicts:

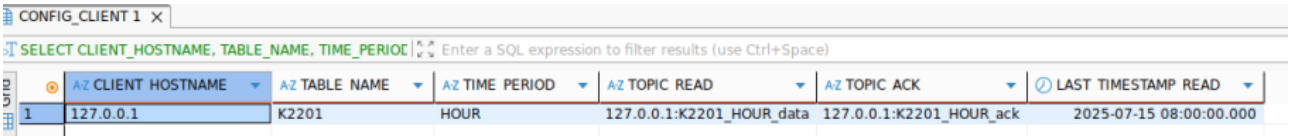


Figure 18: Federated Agent meta-info data table

As we can notice, the 127.0.0.1 client has been subscribed to the master node to send updates that concern the data table K2201 (whose data were illustrated before) every hour. The meta-info data table also holds information about the topic to read and acknowledge the messages, along with the latest updated timestamp.

Finally, we can visit the administrative web console (Figure 19) of the RabbitMQ to inspect the messages that are being sent between the clients and the master nodes.

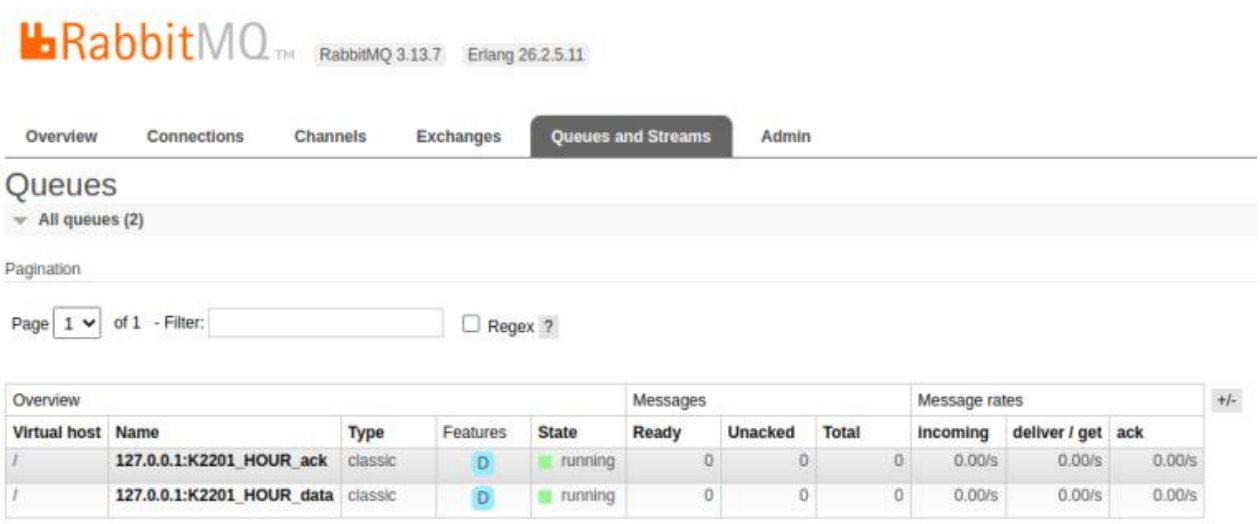


Figure 19: RabbitMQ Web Console for the Federated Agent

We can see that the two topics mentioned before has been created. We can further inspect them to debug the messages that have been exchanged among the various agents that participate in the federated machine learning scenario.

4.1.2 Component Evaluation

For the evaluation of the *energy efficient incremental analytics* component, we relied on the Pilot#7 ("Assessing the Quality and Monetary Value of Data Assets"). Pilot#7 has made available a dataset to the project, while they built a specific application to solve their internal needs, based on the technologies developed within the FAME project. As a result, they benefit from the use of the *energy efficient incremental analytics* component, while the latter took advantage of this pilot PoC to evaluate the foreground technology developed within the project.

The dataset that we made use of is related with metric information that is monitoring the status of various machines of the MOH industry. As a result, it can be considered as timestamped data IoT data, which makes our technology a perfect candidate to boost the performance of the pilot PoC. Each machine consists of several sensors that monitor specific functionalities of a particular machine. In the meantime, pilot#7 has provided with a list of rules that defines when a specific value of one of the machine's attribute should be considered abnormal. Eventually, pilot#7 needs to perform KPI monitoring on the sensors of all its machines, and identify abnormalities, while be able to perform predictive maintenance.

The *energy efficient incremental analytics* component is built on top of the baseline technology of the LeanXcaleDB, which is a relational database with advanced innovative features. One of those is its capability for fast data ingestion. This means that the database can be massively loaded with data of high volume, while being able to ingest sensor IoT data coming with high velocity. Using the database's standard connectivity mechanism, we developed within the T5.3 a data ingestion application for the purposes of the component evaluation, whose purpose is to take as an input extracted dataset of the Pilot#7 in a csv format, and using the standard connectivity mechanisms to exploit the LeanXcale's background technology and ingest the data to our component highly efficiently. Given that each machine of Pilot#7 is unique and consists of different types of sensor, our client application has been developed in a generic way. Having done so, it is able to read any type of sensor data, and generate the corresponding data schema on the fly. At the second phase of the project, this client application that was developed for evaluation reasons, was further extended

in order to take into account the thresholds of each sensor and given these rules, to decide whether a particular value should be considered as abnormal or not.

In the following figure, it is illustrated a part of a data table automatically created by our client application to store data related with the K2201 machine:

Figure 20: Pilot#7 K2201 Data Table

From this figure, we can see that each data record contains a specific timestamp, along with a list of numeric values. Each of this value is associated with a specific sensor of the K2201 machine, while it is accompanied with a boolean flag to further indicate if this value is abnormal (true) or not (false). Figure 21 demonstrates some of the values of the K2201, after being ingested into the LeanXcale database.

SELECT "TIMESTAMP", COL22SI101, COL22SI101_I Enter a SQL expression to filter results (use Ctrl+Space)

Grid Text Record

	TIMESTAMP	COL22SI101	COL22SI101_ISVALID	COL22TI120	COL22TI120_ISVALID
1	2017-01-01 00:00:00.000	9,799.456771	0	57.68009949	0
2	2017-01-01 00:05:00.000	9,799.782656	0	57.59870021	0
3	2017-01-01 00:10:00.000	9,799.466908	0	57.58242035	0
4	2017-01-01 00:15:00.000	9,799.571429	0	57.58242035	0
5	2017-01-01 00:20:00.000	9,799.26	0	57.58242035	0
6	2017-01-01 00:25:00.000	9,800.123333	0	57.5205543	0
7	2017-01-01 00:30:00.000	9,799.942422	0	57.4847374	0
8	2017-01-01 00:35:00.000	9,799.440885	0	57.55669698	0
9	2017-01-01 00:40:00.000	9,799.086667	0	57.57875193	0
10	2017-01-01 00:45:00.000	9,799.166667	0	57.48482422	0
11	2017-01-01 00:50:00.000	9,799.416667	0	57.44917784	0
12	2017-01-01 00:55:00.000	9,799.933333	0	57.57865891	0
13	2017-01-01 01:00:00.000	9,800.005547	0	57.43256588	0
14	2017-01-01 01:05:00.000	9,799.021107	0	57.45892871	0
15	2017-01-01 01:10:00.000	9,798.18	0	57.39333143	0
16	2017-01-01 01:15:00.000	9,799.287337	0	57.27619244	0
17	2017-01-01 01:20:00.000	9,800.027669	0	57.25665661	0

Figure 21: Sample values of the K2201 data table

One of the main goals of Pilot#7 was to be able to do predictive maintenance of their machines. For that, AI models and algorithms have been implements, whose common characteristic is that they all rely on aggregated information of that particular datasets. For instance, they need to calculate the minimum, maximum and average value of the sensors of interest, per a specific time unit (i.e. hour, day, etc). Moreover, they need to *count* the number of anomalies of the sensors of interest, per time unit again. This is where the *energy efficient incremental analytics* comes into play, whose key innovation is the ability to perform *online aggregates* in an incremental fashion, while being energy efficient at the same time.

Now, our client application is not only able to generate the schema of the target tables on the fly, based on the input dataset, but also to generate automatically the *online aggregates*. Having that, we have defined to generate the aggregates per hour. As a result, Figure 22 depicts the derived data table that contains the online aggregations for the machine K2201.

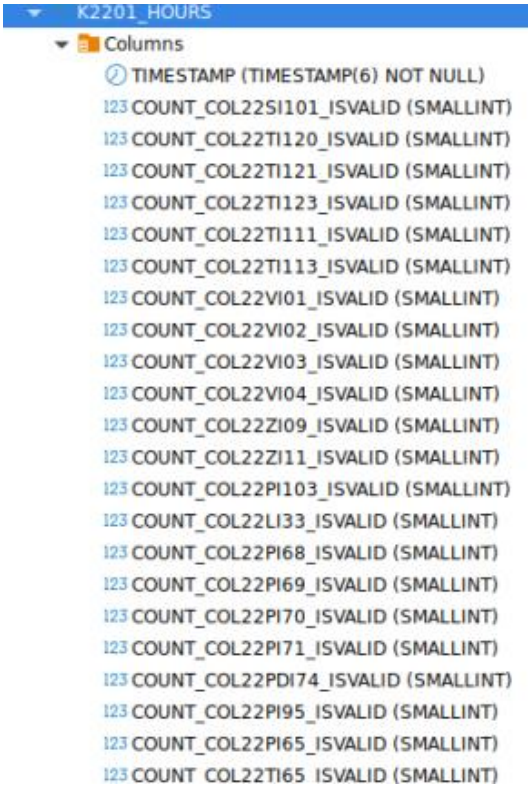


Figure 22: Aggregated information regarding abnormalies of K2201 machine per hour

Similarly, Figure 23 illustrates data coming from the auto-generated derived tables that were created on-the-fly to hold aggregated information for the K2201 sensors per hour.

SELECT "TIMESTAMP", COUNT_COL22SI101_ISVALI | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text | Record

	TIMESTAMP	123 COUNT COL22SI101 ISVALID	123 MAX COL22SI101	123 MIN COL22SI101	123 SUM COL22SI101
1	2017-01-01 00:00:00.000	0	9,800.123333	9,799.086667	117,594.647738
2	2017-01-01 01:00:00.000	0	9,800.027669	9,798.18	117,592.341647
3	2017-01-01 02:00:00.000	0	9,802.376468	9,798.058887	117,593.66976
4	2017-01-01 03:00:00.000	0	9,799.984727	9,798.389036	117,591.717572
5	2017-01-01 04:00:00.000	0	9,800.321224	9,798.549284	117,593.663925
6	2017-01-01 05:00:00.000	0	9,800.251667	9,798.607513	117,594.084343
7	2017-01-01 06:00:00.000	0	9,801.290951	9,798.634287	117,595.645197
8	2017-01-01 07:00:00.000	0	9,800.311816	9,798.7475	117,593.162396
9	2017-01-01 08:00:00.000	0	9,800.961875	9,798.773665	117,596.980335
10	2017-01-01 09:00:00.000	0	9,799.749837	9,798.928945	117,592.814688
11	2017-01-01 10:00:00.000	0	9,800.522487	9,798.4975	117,594.829206
12	2017-01-01 11:00:00.000	0	9,799.836126	9,799.105	117,592.400859
13	2017-01-01 12:00:00.000	0	9,800.707292	9,798.651667	117,595.518348
14	2017-01-01 13:00:00.000	0	9,800.56	9,799.205	117,595.855847
15	2017-01-01 14:00:00.000	0	9,800.146667	9,798.338333	117,593.238336
16	2017-01-01 15:00:00.000	0	9,800.38444	9,799.402272	117,596.646431
17	2017-01-01 16:00:00.000	0	9,800.587272	9,798.867852	117,595.110535
18	2017-01-01 17:00:00.000	0	9,799.858516	9,798.600667	117,599.88787

Figure 23: Sample values of the online aggregates for the K2201 table per hour.

To evaluate our implementation, we did a performance benchmarking based the functionalities needed by the AI tools and models developed for the PoC of Pilot#7. As previously mentioned, all AI tools require aggregated information from the dataset, like the minimum, maximum, average values of the sensors of interest per time unit, and the count of anomalies per the same time unit. This can be interpreted in the following SQL query for the database:

```
SELECT min(COL225I101), max(COL225I101), sum(COL225I101)/count(COL225I101) as “avg”, “TIMESTAMP”

FROM K2201

GROUP BY CTUMPLE(“timestamp”, INTERVAL ‘1’ HOUR, TIMESTAMP ‘1970-01-01 00:00:00’) as “TIMESTAMP”
```

This query returns the statistical information (min, max, sum and count, which can be combined to calculate the average) about the COL225I101 sensor, per time unit. In the code snippet, we calculate this information per hour. These aggregated relational operations are one of the most demanding (along with the joins) to be implemented in terms of resource consumption, as they often require the full scan of the data table. This implies a vast amount of I/O access to the disk to read all the information, vast amount of network data transmission between the data nodes and the relational query engine that is responsible to compute the final result, and final lot of resources needed for both CPU and memory. This query will be used as a reference to benchmark our implementation both in terms of overall acceleration of the response time and the efficiency of resource consumption.

For all types of experimentation, we will firstly rely on the energy efficient incremental analytics component, which means the LeanXcaleDB enhanced with the novelties introduced or further developed under the scope of the project. Then we will repeat the experimentation with a vanilla LeanXcaleDB, meaning the prototype released before the beginning of the project. We will observe the overall response time, the energy consumption and the computational resources needed to execute the aforementioned query. All experiments made use of separate threads, each one of those handling an already open connection and prepared statement, so that the measurements could only observe the overall time and resources needed for the query processing. This removed the overhead for having to deal with transaction management (for instance repeatedly open and close connections or start and commit transactions) and query optimization (for instance to compile the submitted query each time and explore the optimal query plan to be executed).

Overall response time

In the first series of our performance evaluation, we focused on evaluating the overall performance in terms of response time. For that, we used the dataset available from Pilot#7, used for experimentation internally in the project, and as a PoC for the pilot. We relied on the client application that was described before to ingest the data to the database, and create on-the-fly the relevant data schema, along with the online aggregates. Then, we started ingesting data and performed the evaluation by starting with 1K rows ingested in both a vanilla LeanXcaleDB and the ones enhanced with the energy efficient incremental analytics. Then we executed the queries in both. Each query has been executed five times and we took the average response time. Then, we scaled the dataset, adding an additional 1K of data and repeat the measurements. We continued scaling the datasets up to 7697K rows.

Figure 24 compares the overall response time of the vanilla LeanXcaleDB with the *energy efficient incremental analytics* component, without forcing the ordering.

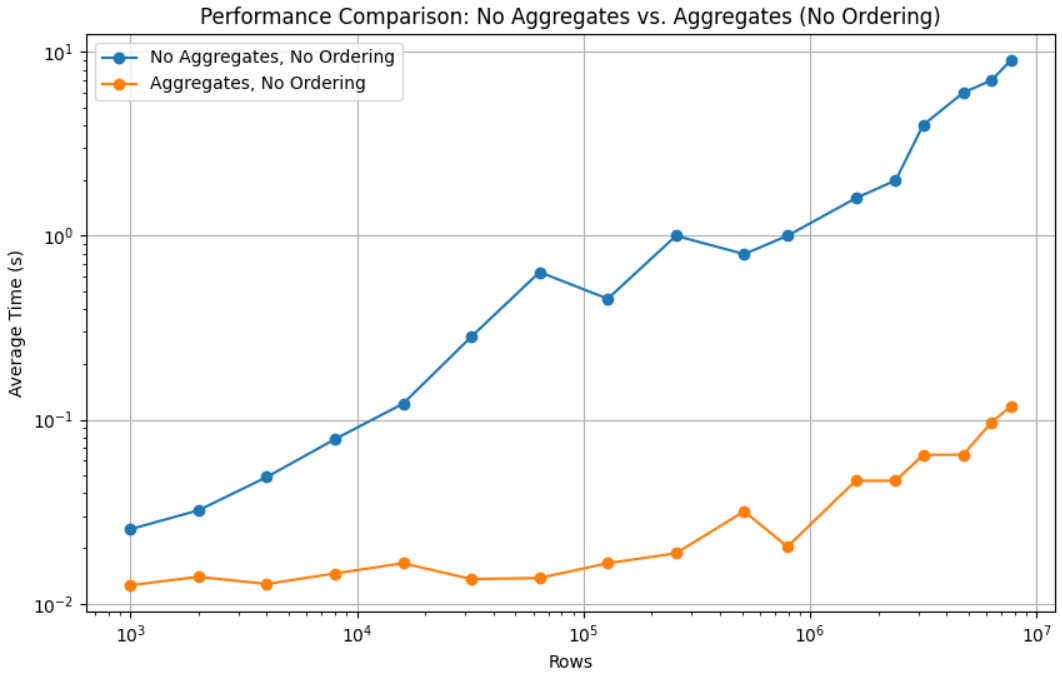


Figure 24: Overall response time with no ordering

From this figure, it is plotted with blue color the response time of the vanilla LeanXcaleDB, while with the orange color the response time of the *energy efficient incremental analytics*. We used logarithmic scale in the axis. From the figure it is depicted that the vanilla implementation scales linearly as the data volume grows. This was expected, as the aggregated operations requires the full scan of the target table, which implies a computational complexity of O(n). In the orange plot, we can observe the same, however the response time increases much slower. Here again, the bigger the data volume, the higher the demand to scan, however instead of scanning the whole data table (which includes values per second), now we only access the aggregated information per time unit. What is depicted is that we have a performance acceleration of orders of magnitude, and in fact, the bigger the data volume, the greater the acceleration gained in absolute, and logarithmic, numbers.

We repeated our benchmark evaluation, however this time we enforce the ordering operation. The results are depicted in the following figure:

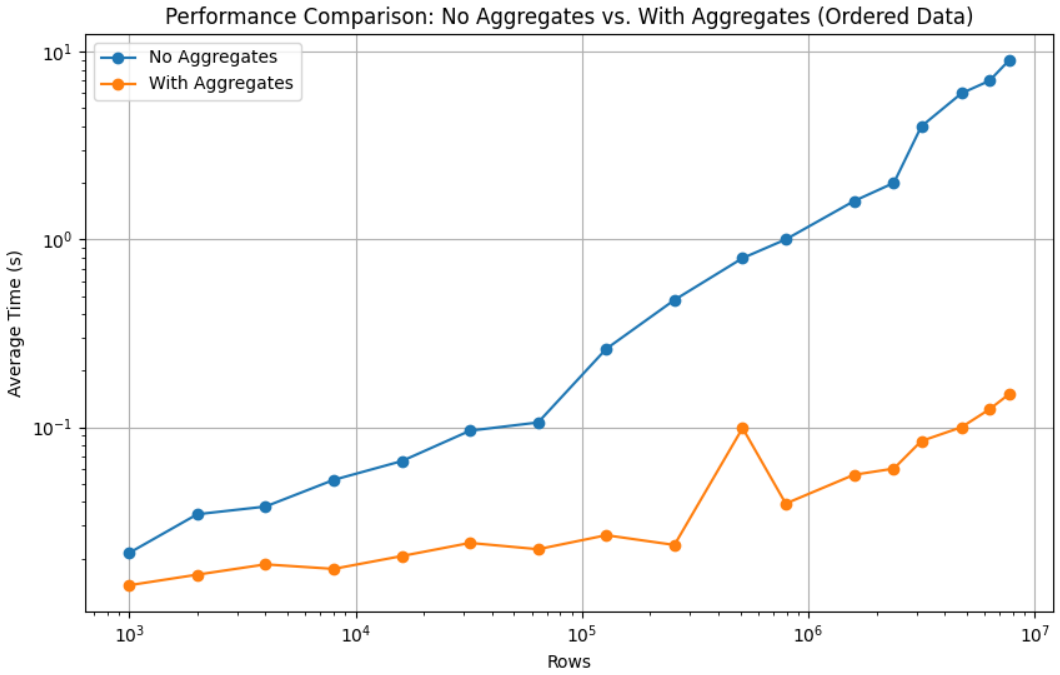


Figure 25: Overall response time with ordering

The ordering forces the query engine to return the results ordered by the specific columns of interest. Internally, this implies that the query engine needs to hold all the intermediate results in memory, and then apply the ordering operation, unless a specific index is well defined that it can make use of. The reason why we repeated the experimentation having the additional ordering operation was to be fair, and not let the energy efficient incremental analytics component to get any advantage of underlying indexing structure while calculating the results. Having the ordering operation, both vanilla and our prototype will wait until the complete processing of the aggregated operations finishes, before returning the results. Once again, it is depicted that the response time of the vanilla implementation grows linearly, however our implementation benefits from the significant data items that need to be accessed, transmitted and processed in the query engine side.

Energy consumption

For this experimentation, we made use of the scaphandre[7] open source tool used for monitoring the energy consumption of different processes being running in a machine. This tool can get metrics related with CPU cycles and memory space used, disk access etc, in order to calculate. Additionally to standard metrics, the scaphandre open source tool can calculate the power consumption in watts (W). This power consumption monitoring agent helps system engineers to track how much energy various processes or virtual machines are using on a system. This power consumption in watts is reported per process, so we can further estimate the power usage of the various subcomponents of our implementation.

[7] https://github.com/hubblo-org/scaphandre

In this experimentation, we monitored the execution of the same queries under a fixed rate targeting both the vanilla and the our enhanced implementation based on the *energy efficient incremental analytics* component. We didn't want to repeatedly send the same query once the previous had finished, as the our implementation would impose high throughput due to its significant lesser response time we observed in the previous experimentation. That would end having 100s of times more workload to our implementation, so that the results would not be objective. For that, we use several threads that execute the same query with a fixed rate, enough to be give some time to the database to return the result.

Diagrams in Figure 26 and Figure 27 depict the overall memory and CPU consumption of the processes of LeanXcaleDB, along with its energy consumption, in an idle state. It is similar to both the vanilla and our newly developed prototypes, as they don't need to process any data in the idle state.

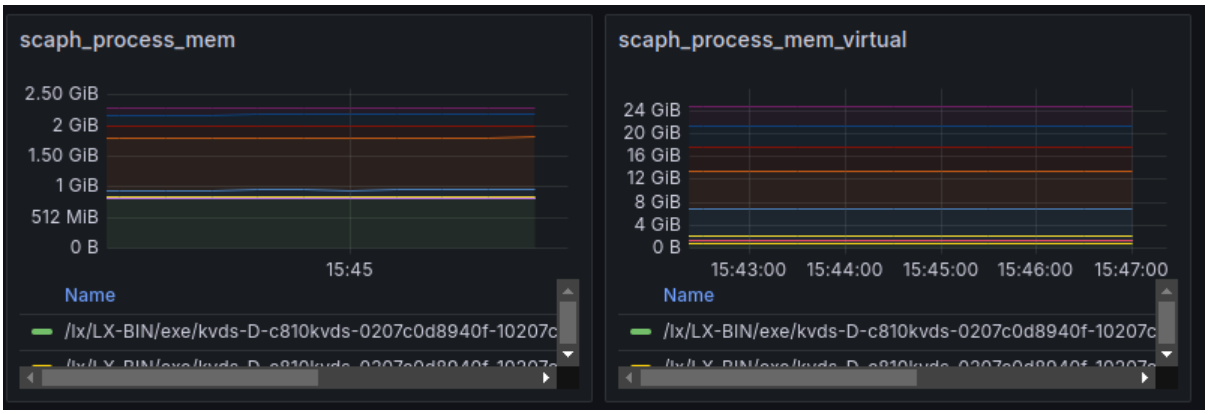


Figure 26: Vanilla LeanXcaleDB memory consumption in idle state

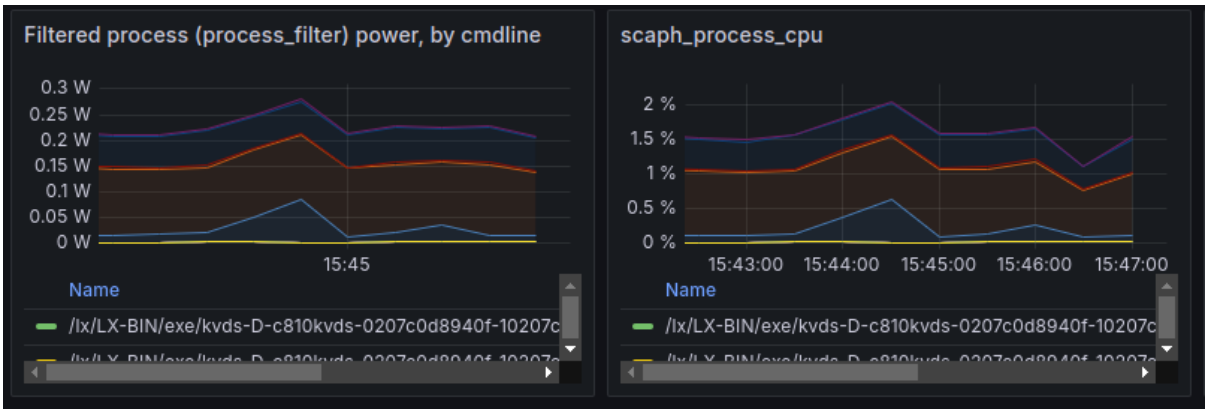


Figure 27: Vanilla LeanXCaleDB cpu and energy consumption in idle state

We can see that in an idle state, it requires less than 1% of CPU cycles and less than 0.25W. This slight resource consumption is caused by the query engine (in the red plot) which interacts with the end-users. For that, it keeps internally a pool of connections that need to be maintained, along with an internal thread that is always up and running and serves the needs of a servlet container. In an idle state, the data nodes remain fully inactive, as the yellow line depicts. Regarding memory consumption, the database pre-reserves the memory it needs from the operating system, in order to perform better when there is the need to execute queries. As a result, it does not ask the operating system to assign more memory, unless it is really needed and feasible. Due to that, the overall observed memory consumption is always the same, no matter if we choose the vanilla or the MobiSpaces implementation. For that, we will focus on the CPU and energy consumption.

Figure 28 depicts the CPU and energy consumption of the vanilla LeanXcaleDB, while continuously processing the aforementioned query.

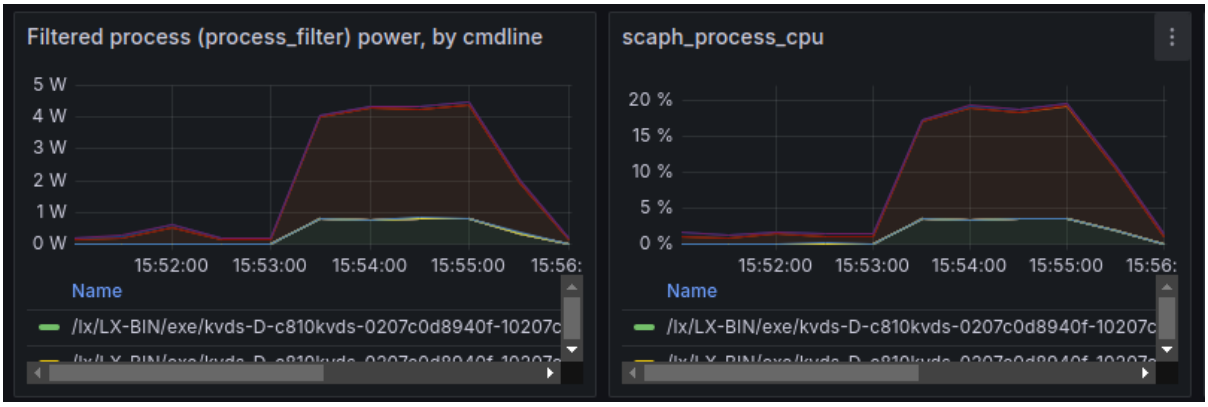


Figure 28: Energy consumption of vanilla LeanXcaleDB

From this figure we observe an average of almost 20% of CPU consumption, while the overall energy consumption is increased to more than 4Watts, compared to 0.2Watts consumed in its idle state.

Figure 29 now depicts the CPU and energy consumption of the *incremental analytics* component, while continuously processing the aforementioned query.

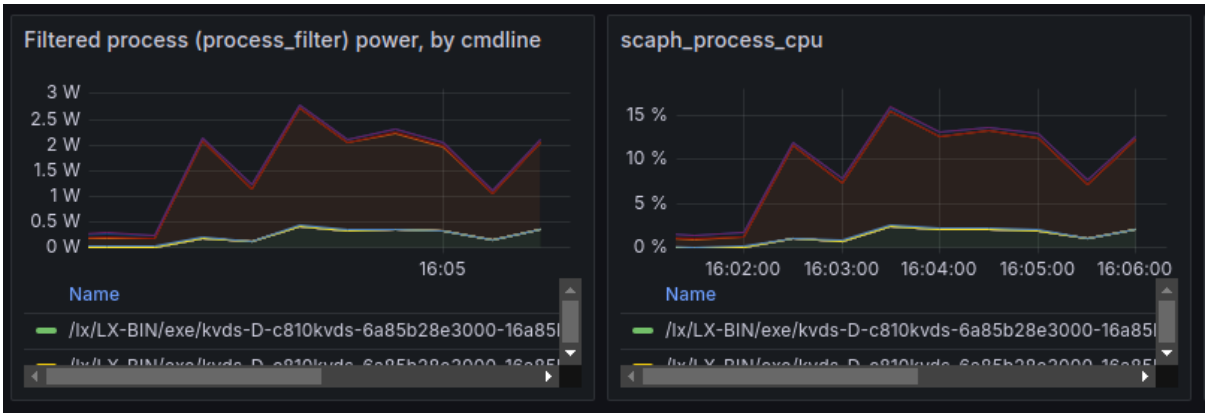


Figure 29: Energy consumption of incremental analytics

In this figure, we can note a significant decrease of CPU and energy consumption required for executing the same queries, at the same fixed rate, while executed in the *incremental analytics* prototype. The average CPU consumption is 12.5%, which is 37.5% less than the vanilla implementation. On the other hand, the average energy consumption is 2Watts, which is 67% lesser than what is required in the vanilla one. It is also important to highlight the energy and CPU consumption of the data nodes themselves, depicted with the yellow plot. As the *incremental analytics* component implementation requires the data nodes to access and transmit lesser data than the vanilla ones, they are lesser active and require for lesser resources.

4.2 Analytics CO2 Monitoring

4.2.1 Prerequisites and Installation Environment Documentation

Kepler: installation on the cluster.

Specialized Electricitymaps scraper requirements: Selenium and beautifulsoup4.

(Optional) Database support prerequisites: A timescaleDB database and psycopg2.

Environment Documentation, as well as setup tutorial, to be provided with the final Dockerized component.

Data sample (in csv format):

timestamp	pod_id	container_name	namespace	country_iso2	# energy_consumption_jps	# co2_emissions_gps
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	HR	52.84578357	0.002745044869
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	SK	52.84578357	0.002055113806
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	BG	52.84578357	0.005402013432
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	EE	52.84578357	0.0007780073692
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	LT	52.84578357	0.001203709515
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	LV	52.84578357	0.0008807630595
2025-08-04 7:28:48	busybox-pod-5b4d7c6f78	busybox	default	IS	52.84578357	0.0004257021454
2025-08-04 7:29:18	busybox-pod-5b4d7c6f78	busybox	default	DE	49.01057797	0.002600283442
2025-08-04 7:29:18	busybox-pod-5b4d7c6f78	busybox	default	FR	49.01057797	0.0002314388404
2025-08-04 7:29:18	busybox-pod-5b4d7c6f78	busybox	default	IT	49.01057797	0.002927020629

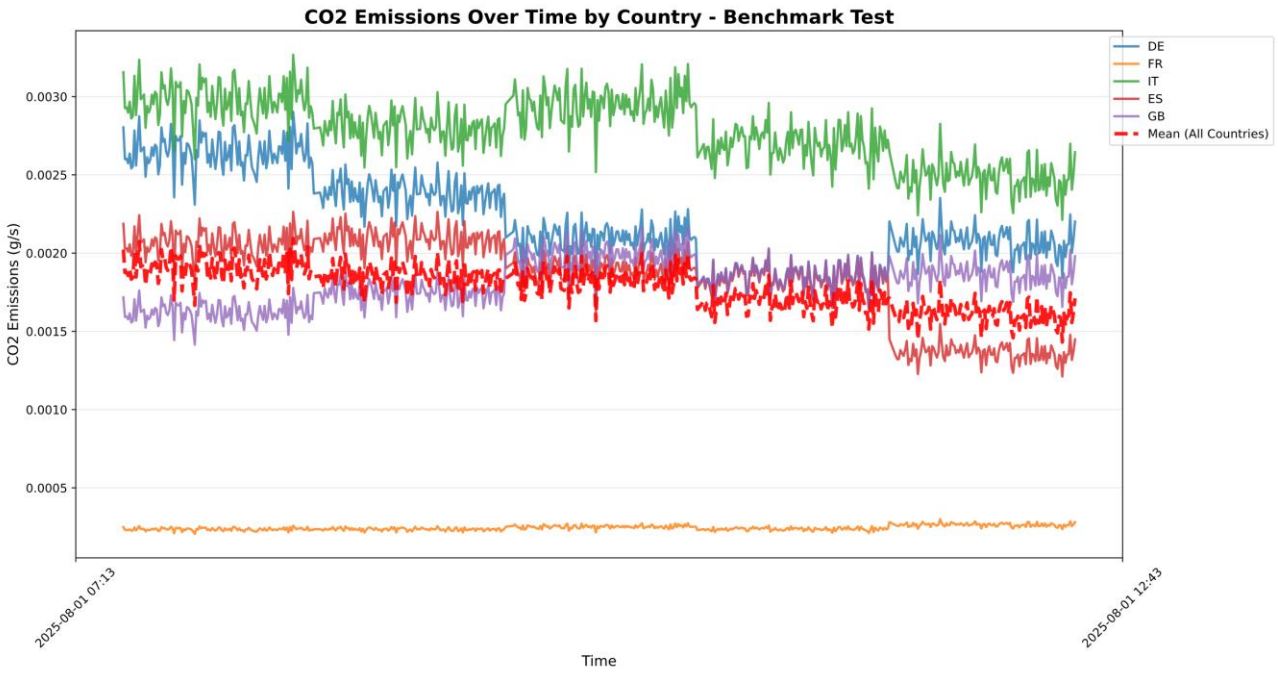
4.2.2 Component Evaluation

As for reaching KPI goals, simple benchmark testing leads us to the following result. For a simple test using **busybox**—a minimal, lightweight Linux distribution commonly used as a placeholder or dummy container—the measured optimal CO2 emissions in g/s were significantly better than the mean observed emissions.

The KPI goal was to achieve a **200% improvement**, which in our case corresponds to a **66.6% reduction** in CO2 emissions compared to the baseline.

Assuming optimal resource availability and a broad deployment country range, there should be no issue in factoring in CO2-awareness to the desired degree. Results of the busybox benchmark test can be seen in the following line chart for a limited number of countries.

It should be noted that the component is constrained both by the electricity grid and available resources upon migration. As a result certain cases fall short of the KPI, however these rarely appear. See example scenario 5.



An SVG version of the image will be available in the official documentation for closer examination.

4.2.2.1 Scenario definition

Here, a *scenario* describes the subset of countries available for workload migration within a 5-hour window. These are defined by randomly selecting 4–7 countries from a pre-measured pool of 28 European countries, simulating situations with restricted resource availability. For each scenario, the system attempts to minimize CO2 emissions by optimally migrating the busybox container.

The **mean emissions** across all allowed countries (per time step) are compared to the **optimal emissions** possible within the restricted country subset. This comparison is used to assess whether the scenario meets or exceeds the KPI threshold.

Below is a brief overview of the scenarios tested (countries mentioned by ISO2 codes):

- **Scenario 1**: 6 countries — IS, CZ, BG, RO, AT, SE
- **Scenario 2**: 4 countries — DE, PL, GR, LV
- **Scenario 3**: 6 countries — GB, LT, SI, DE, AT, GR
- **Scenario 4**: 7 countries — ES, FR, GB, PL, HU, LT, SE *(Best case)*
- **Scenario 5**: 4 countries — LV, ES, HU, LT *(Worst case)*
- **All Countries**: Full access to all tracked countries*("ideal" scenario)*

Despite the differing country availability, each scenario successfully performed **at least one migration** during the window, maintaining a **uniform migration rate of 16.7%**. The average reductions observed were:

- Scenario 1: 88.2% ± 2.1%
- Scenario 2: 72.8% ± 5.6%
- Scenario 3: 78.0% ± 1.7%

- Scenario 4: **89.6% ± 1.1%** *(Best-performing subset)*
- Scenario 5: **32.4% ± 12.7%** *(Worst-performing subset)*
- All Countries: 87.7% ± 1.7%

This shows how country selection impacts the outcome, but that even with limited resources, dynamic migration can yield significant emission savings.

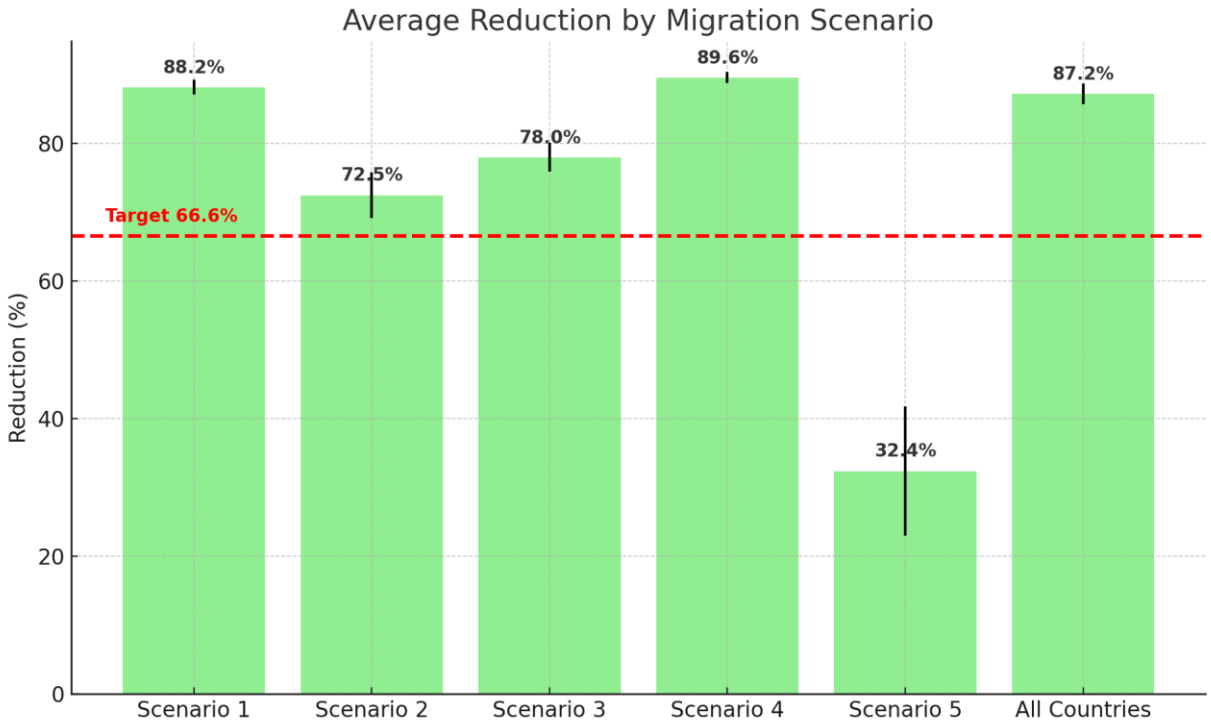
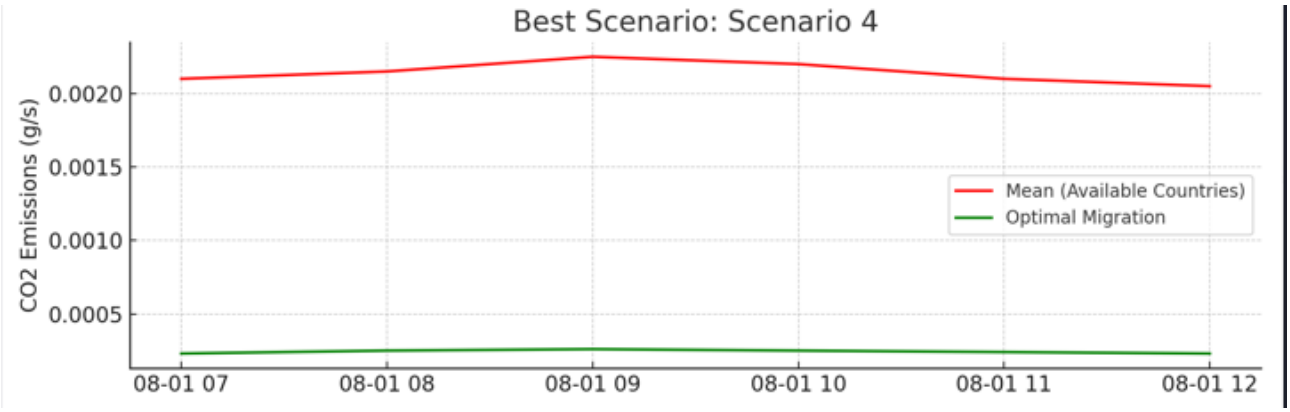


Figure 30: Average reduction by migration scenario

4.2.2.2 Chart Commentary and Insights

The top chart shows the percentage reduction in emissions across each scenario. Most scenarios surpass the 66.6% reduction threshold, indicating that even under limited country availability, significant improvements can be achieved.

Scenario 4 stands out with the highest reduction, demonstrating that careful selection of even a few countries can yield results comparable to unrestricted deployments. Scenario 5, however, shows a much lower improvement, highlighting the impact of reduced flexibility.



The above time-series chart compares average emissions versus optimal emissions in the best-performing scenario. It illustrates a consistent advantage in applying CO2-aware migration decisions.

4.2.2.3 Overall Analysis

The results show that our approach can successfully exceed the defined KPI targets, even under varying and limited conditions. Busybox, while a lightweight workload, clearly demonstrates the feasibility and effectiveness of the proposed CO2-optimizing strategies.

From the six defined migration scenarios, all demonstrated a reduction in emissions above 20%, with an **average reduction of 74.8% across all cases**. This highlights the robustness of the dynamic migration mechanism.

In conclusion, scenario-based evaluation proves to be a powerful method to simulate realistic deployment environments and validate the environmental impact of smart migration strategies. The clear success of the simulations provides a strong foundation for future extensions toward more compute-intensive workloads.

4.3 Smart Deployment

4.3.1 Prerequisites and Installation Environment Documentation

The execution environment for the SD component is Kubernetes (K8s), thus it expects an operating K8s cluster to be available for its own operation as well as node deployment. Moreover, each worker node of the cluster must be labeled with the country where it is located, according with the ISO 3166-1 alpha-2 code:

```
smartdeployment/countrycode: XX
```

The code snippet above is substituted by the corresponding country code. The correct labeling of the nodes is vital to locate the models in the country based on their CO_2 emissions.

```
apiVersion: v1
kind: Namespace
metadata:
  name: fame-smartdeployment
  labels:
    smartDeployment.environment: fame
```

All the resources will be deployed in its own namespace. For instance, the label *smartDeployment.environment: fame* is used by the SD to control the resources controlled by itself.

An example of YAML files for the SD component is given in the annex Smart Deployment YAML.

4.3.1.1 RBAC

For the correct management of the different modules, they must have been granted with the correct privileges, allowing them to list, watch, create, etc, different resources. This is done through the following resources:

- *ServiceAccount*: it provides an identity for pods to authenticate with the Kubernetes API.
- *Role*. Declares which actions in which namespaced resources are allowed.

- *ClusterRole*: it is the same than the Role, but for non-namespaced resources or to give privileges over all namespaces.
- *RoleBinding*: it binds a Role to a SeviceAccount.
- *ClusterRoleBinding*: it binds a ClusterRole to a ServiceAccount.

For this reason, the manager must create the Role, ClusterRole, and its own RoleBinding and ClusterRoleBinding related with a ServiceAccount (SA) that allows to the SD modules the enough privileges to perform the necessary operations. This difference between role and ClusteRrole is made because the SD system does not need access to all the resources from all namespaces, which avoids possible troubles or the model deployment in other namespaces.

Once access has been granted to the SA fame-smartdeployment, the credentials to access to the MLFlow instance must be granted, saving them in a Kubernetes secret:

- MLFLOW_TRACKING_URI
- MLFLOW_TRACKING_USERNAME
- MLFLOW_TRACKING_PASSWORD
- MINIO_API_URL
- MINIO_ACCESS_KEY
- MINIO_SECRET_ACCESS_KEY
- MINIO_BUCKET_NAME

With these credentials, the manager can deploy the corresponding YAML files for each module:

- Deployment
- Service

The YAML samples in the annex are the base for the following modules:

- API-Server
- WebAPP
- Node-Resolver

Modifying the variables:

Table 5: Environment variables for the smart deployment

Variable	API-Server	WebAPP	Node-Resolver
MODULE_NAME	api-server	webapp	node-resolver
APP_LABEL	api-server	webapp	node-resolver
CONTAINER_NAME	api-server	webapp	node-resolver
REPOSITORY_IMAGE_URL	REGISTRY /api-server	REGISTRY /webapp	REGISTRY /node-resolver
COMMAND_FROM_IMAGE	["-c", "fastapi run main.py --port 8000"]	["-c", "streamlit run app.py"]	["-c", "fastapi run main.py --port 9000"]
PORT	8000	8000	9000
PORT_NAME	app	app	app

If each module's image is in a private repository, the variable IMAGE_REGISTRY_CREDENTIALS must be saved with the corresponding credentials.

Once every module is up and running, the access to the SD API-Server can be done by port-forwarding the service running the following command:

```
kubectl port-forward svc/api-server 8000:8000 -n fame-smartdeployment
```

Once port-forwarding is enabled, the user can deploy models making requests with curl or any tool able to make requests to a REST API.

Accessing to the SD WebAPP module can be made in a similar manner, which deploys the WebAPP under http://localhost:8000 (accessible in the browser):

```
kubectl port-forward svc/webapp 8000:8000 -n fame-smartdeployment
```

4.3.2 Component Evaluation

The ML model management is done in the SD API-Server, a REST API built for this task. All the calls to the SD API-Server, including the petitions done through the SD WebAPP module, are done according with the REST API standard[8].

4.3.2.1 Model Management

As indicated in section 3.3.2.1.1, the SD API-Server module can perform some model actions.

4.3.2.1.1 Deploy Model

When any user wants to deploy any model, must give to the SD AS the following information:

- name: this is the model that will have inside all the deployment.
- modelName: if it is given, it provides the name that appears on the MLFlow instance.
- image: the image of the model server, this image can be the SD MS or another one given by the provider.
- portName: the name of the port inside the deployment.
- portNumber: the number where the deployment exposes the server.
- credentialsSecret: the credentials needed by the to deploy the model.
- imagePullSecret: the credentials from the image repository if they are needed.
- namespace: the namespace to deploy the model

This information must be received by the SD AS in a JSON through a HTTP request:

```
curl -X 'POST' \
  'http://localhost:8000/api/v1/model/deploy' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "modelName": "KMeans",
  "image": "newregistry.evidenresearch.eu:18522/repository/fame/smartdeployment/modelserver",
  "name": "model-test-1",
  "portName": "port",
  "portNumber": 9000,
```

[8] https://restfulapi.net/

```
  "credentialsSecret": "fame-secret",
  "imagePullSecret": "nexus-registry",
  "namespace": "fame-smartdeployment"
}'
```

When it is received, the SD module performs the following connections between all the modules, as they can see in the figure below:

- The user makes a request to the SD AS.
- The SD AS connects with the MLFlow instance and checks if the model exists and its available.
- The SD AS connects with the K8s API checking if the namespace requested exists.
- The SD AS connects with the SD NR to receive the node where the model will be deployed.
    - The SD NR connects with the Analytics CO_2 Monitoring to receive the CO_2 emissions per country.
    - The SD NR decides which node is the best option to receive the model and returns the node to the SD AS.
- The SD AS connects with the K8S API checking if the resources can be deployed.
- If all its successful, the model is deployed.

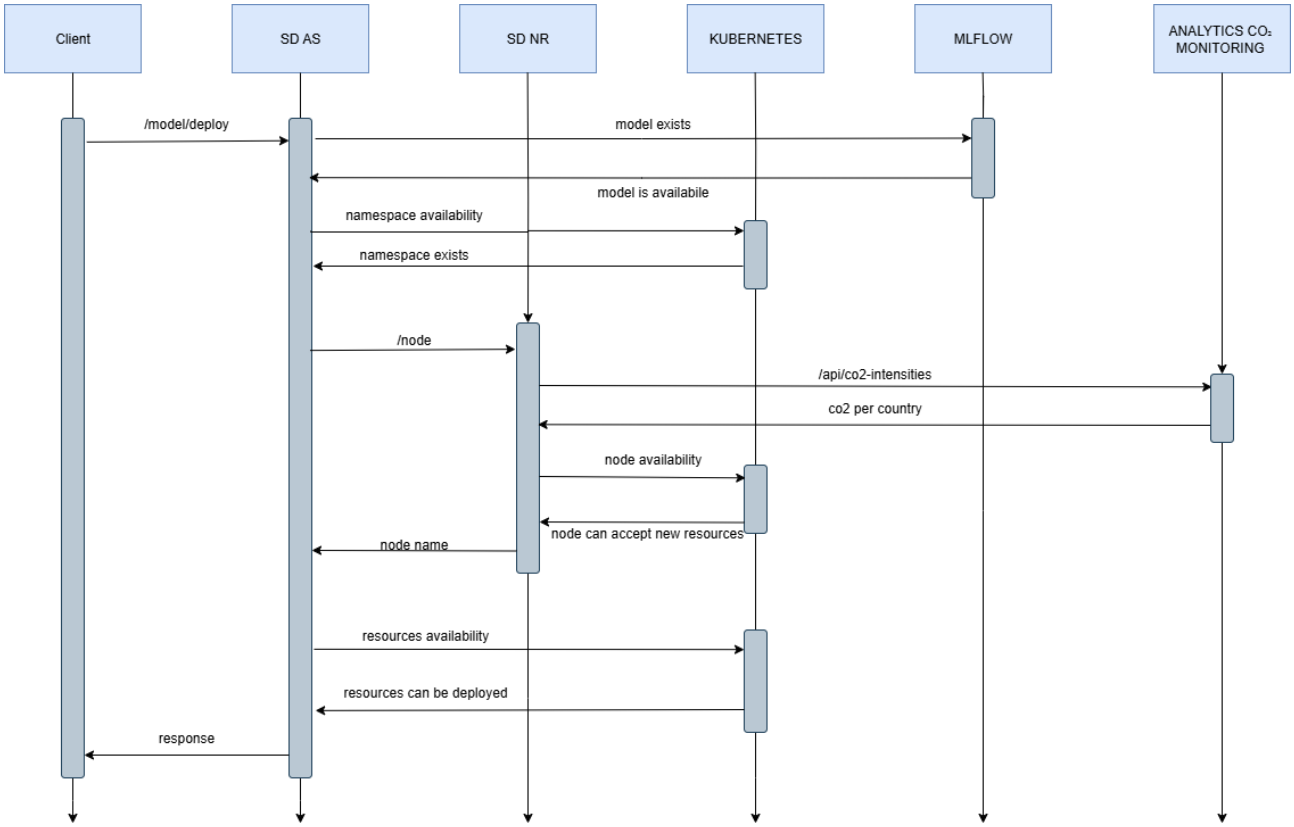


Figure 31: Smart Deployment connections schema for model deployment

After a few seconds, the SD API-Server returns a response, with all the information related with the model status:

- The namespace status.
- The model on MLFlow status (if needed).
- The credentials status.
- The node where the model is deployed

The following snippet depicts the configMap, service and deployment status.


```
{
  "namespace": {
    "name": "fame-smartdeployment",
    "status": "ok"
  },
  "model": {
    "name": "KMeans",
    "status": "model available"
  },
  "secretCredentials": {
    "name": "credentials",
    "status": "secret credentials exists"
  },
  "node": {
    "status": "available",
    "name": "server-stuart"
  },
  "configMap": {
    "name": "model-test-1-model-data",
    "status": "created"
  },
  "service": {
    "name": "model-test-1",
    "status": "created"
  },
  "deployment": {
    "name": "model-test-1",
    "status": "created"
  }
```


If something went wrong (for instance, a credentials secret that does not exist), the model is not deployed, and the system returns an error response with all the information. For example, if the credentials are not given correctly or does not exist, the following is returned:


```
{
  "namespace": {
    "name": "fame-smartdeployment",
    "status": "ok"
  },
  "model": {
```

```
    "name": "KMeans",
    "status": "model available"
  },
  "secretCredentials": {
    "name": "fame-secret",
    "status": "secret credentials does not exist"
  },
  "general status": "error"
}
```


4.4 FML Framework

4.4.1 Prerequisites and Installation Environment Documentation

The FML framework is indexed in the FAME marketplace and can be purchased in five different assets. The first two correspond to the root server and client applied to Pilot 1, while the other three correspond to the root and center servers, and the client applied to Pilot 7. The application of the FML framework to Pilots 1 and 7 is described below in more detail. Each individual asset is provided as a set of Docker containers that can run standalone or on a Kubernetes cluster. An optional component is the *incremental analytics* that was described in a previous section. Moreover, a Custom data loader pod has been implemented to support this. This was described in the 4.1 section. Otherwise, if the *incremental analytics* is not used, the data is expected to be passed as a file.

4.4.2 Component Evaluation

This section presents an evaluation of the FML framework on pilot data. The following table provides an applicability analysis to the different pilots. Two different characteristics have been considered in this analysis, namely the need for privacy, which depends on the kind of collected data, and the limited communication capabilities due to the deployment. On the one hand, specific privacy requirements are only present in Pilots 1 and 3, as they comprise data banking information that is highly sensitive. On the other hand, limited communication is only present in pilot 7, which comprises distributed sensors in industrial machinery.

Table 6: Pilot suitability analysis for Federated Learning

Pilots	Privacy	Limited Comm.
Pilot 1	✓	✗
Pilot 2	✗	✗
Pilot 3	✓	✗
Pilot 4	✗	✗
Pilot 5	✗	✗
Pilot 6	✗	✗
Pilot 7	✗	✓

No data was available for Pilot 3 at the time of developing FL assets, and hence the FML framework has not been evaluated for this use case. The developments for Pilots 1 and 7 are presented in the following.

4.4.2.1 Pilot 1

The customer Profiler developed in D5.3 Section 3.1.1.1 applies KMeans to Pilot 1's dataset to extract customer information based on Recency-Frequency-Monetary (RFM) metrics. RFM segmentation enables the analysis of customers' interaction patterns with the service, thereby providing insightful information to tailor the company-customer relations.

Pilot 1 data consists of highly sensitive banking data, which is protected by GDPR, which limits how data can be shared between companies. Nevertheless, multiple companies may still want to perform a joint analysis of their customer base. For example, a bank with a common customer base with several insurance companies could obtain common clusters across them for a joint product offering campaign.

This section showcases how the Federated KMeans pod presented in Section 3.4.2.2 can be leveraged to perform KMeans across several companies in a federated manner. For this, we have split Pilot 1 dataset using the *generation* column. As its name implies, this column contains the generation every customer belongs to. Two different datasets have been created with the following distribution:

Table 7: Pilot 1 federated split percentages

	Lucky Few	**Baby Boomers**	**Mature**	**Millennials**	**Centennials**
Dataset 1	90%	70%	50%	30%	10%
Dataset 2	10%	30%	50%	70%	90%

The split percentages have been selected to produce a generation bias on purpose, so that the non-IID data split that is common in federated scenarios (McMahan, Moore, Ramage, Hampson, & Agüera y Arcas, 2017) is replicated.

Table 8: Total cluster assignations per client

	Cluster 0	**Cluster 1**	**Cluster 2**	**Cluster 3**
client-one	6716	1191	121	21
client-two	6111	1041	105	28

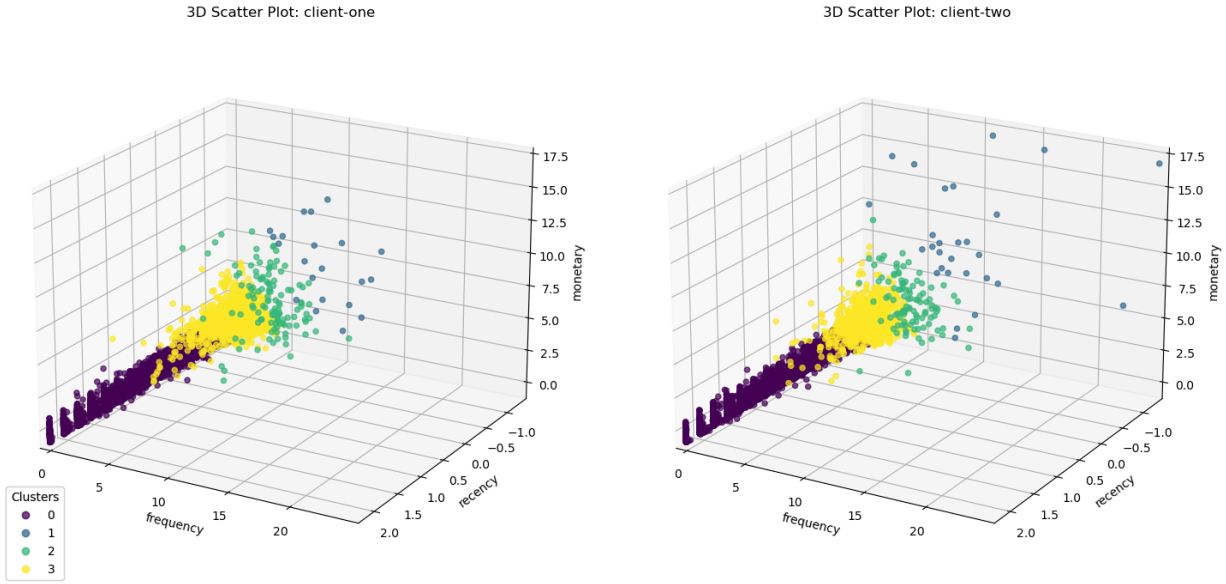


Figure 32: Client one and two scatter plots

After training for two rounds, client 1 obtains a silhouette score of 0.4347 and client 2 obtains 0.435, which are in line with the 0.43 score reported for the centralized training in D5.3, Section 3.1.1.1. The above figure depicts the clusters obtained by *client-one* and *client-two* with datasets 1 and 2, respectively. It can be observed that the two datasets present a similar distribution despite the generational bias. The table above also corroborates this intuition, as the total number of clients associated to every cluster in both clients is rather similar. Nevertheless, it can be observed that e.g., client-two's cluster 1 has a wider distribution than client-one's data, which Federated KMeans is still able to learn despite the training occurring in a distributed manner.

4.4.2.2 Pilot 7

This pilot is a paradigmatic example of limited communication resources: the deployment consists in industrial machinery monitored by a network of distributed sensors measuring temperature, pressure, etc. Based on these sensors' measurements, machinery anomalies are detected. The sensors collect data every 10 seconds, and transmit an aggregation of these data (i.e., a single f32 value) to a central database every 5 minutes.

The objective of applying the FML framework in this context is to enable the pilot to train models with the fine granularity of 10 seconds in order to improve the anomaly prediction capabilities, without a significant increase in the transmitted data rate. For this, the anomaly detection model developed in D5.3, Section 4.1.4.1 is reused here as the base model architecture. The following figure depicts the deployment at the refinery level, which consists in a hierarchical FL deployment (see Section 3.4.1). This deployment suits the refinery distribution: it consists of centralized refinery server, 5 different compressor machinery (i.e., compressor central servers), and each compressor is monitored by different sensors (i.e., sensor clients). The distributed deployment has been simulated in a Kubernetes cluster and all the manifests have been uploaded to the marketplace.

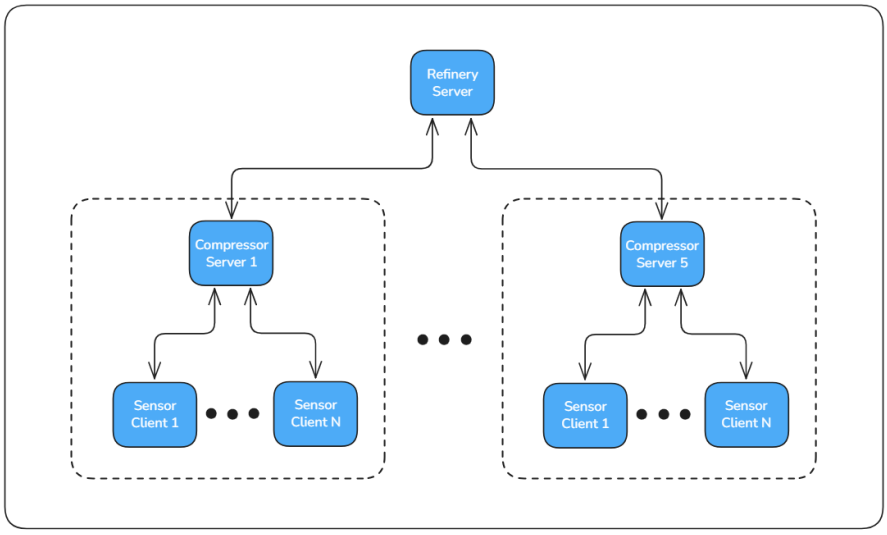


Figure 33 - Pilot 7 refinery FL deployment

The blue boxes in Figure 33 above are Kubernetes pods running FL pipelines on a Docker container. The pipelines connect different pods from the FML framework to obtain the desired behavior. In particular, the *CustomTrainer* and *CustomLoader* pods have been used to replicate the centralized pipeline in D5.3, Section 4.1.4.1 in a federated manner. These custom pods showcase two of the key features of the FML framework, namely its flexibility and ease of use, as the centralized pipeline was converted to a federated pipeline by just encapsulating the code of the former into Python functions for the latter.

The model has been trained for all temperature sensors in the 5 compressors, which results in a total of 33 sensors (i.e., clients). The data corresponds to 4 full years, from January 2017 to December 2020. The training is configured with a patience value of 3, i.e., every client trains until the model performance has not improved for three consecutive epochs, and then it transmits its model to its compressor.

The following figure shows the loss evolution for a training session with 3 FL rounds and different client selection rates at the compressor server, which means that all compressors are selected in every round, and every compressor selects a subset of its client according to the configured selection rate. The number of rounds has been configured to 3 because it provides the best trade-off between achieved loss and training time. As the random selection of clients introduces noise in the results, four different models have been trained for every client selection rate. Figure 34 shows the mean loss in every round, and the shaded area represents the standard deviation.

As expected, the most predictable performance is achieved when all clients are always selected, and the unpredictability is inversely proportional to the selection rate. On average, the lowest loss is achieved with a rate of 1.0, which permits to capture relevant information from every available temperature signal. However, the lowest overall loss is obtained with a selection rate of 0.25 (0.44 vs 0.58 obtained with a rate of 1.0).

Unlike other Machine Learning deployments, in FL the data distribution across clients is non-IID and unbalanced (McMahan, Moore, Ramage, Hampson, & Agüera y Arcas, 2017). In other words, a client's dataset may have simple structure that can be quickly learned, while other client's data may require more iterations. Selecting all clients in every round can lead to overfitting for the former and underfitting for the latter. Client selection prevents this issue.

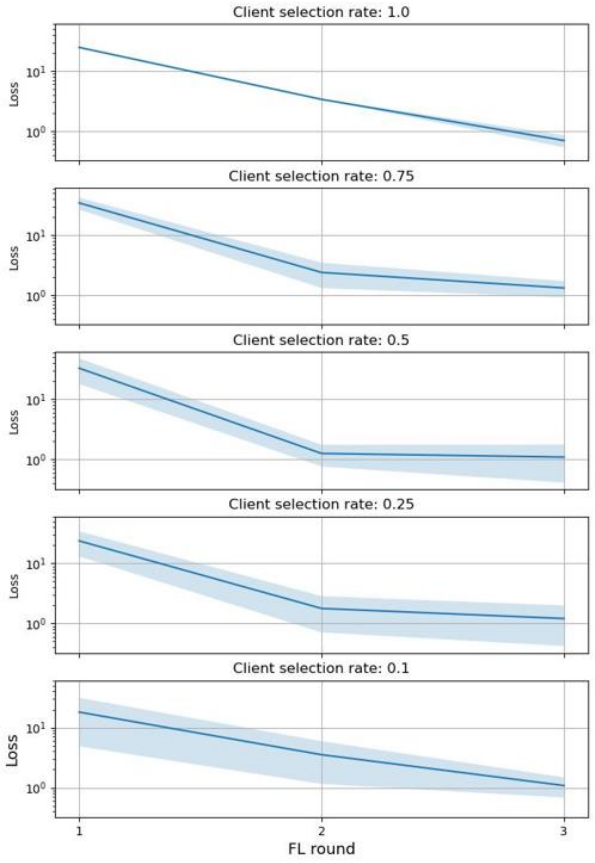


Figure 34 - Loss evolution by client selection rate

The next figureFigure 35 depicts the loss evolution when compressor selection is used instead of client selection. As soon as compressor selection is introduced, the predictability of the results decreases, which shown by the larger standard deviation of the results than when only client selection is used. Moreover, only when all compressors are selected does the loss go below 0.8. While sensor temperatures on the same compressor are expected to be highly correlated, thereby allowing a lower selection rate without a significant impact on the model's performance, different compressors may operate at different temperature ranges. Therefore, a more conservative selection rate must be selected for the compressors than for the clients.

Every year, a sensor collects 48MB of data (one f32 every 5 seconds) and transmits 1.6MB (one f32 every 10 minutes). Therefore, in the four years used for training, the 33 temperature sensors used in this training generate 1.55GB of data and transmit 52.8MB. Based on the results above on client and compressor selection, we have trained a model with compressors selection rate 0.8 and client selection rate 0.25. The obtained loss is 0.52 and transmitted 197.8, which considers the data transmitted from clients to compressors and compressors to the root server. Although this is 3.74 × the currently transmitted data, this model enables anomaly detection with 6,000 × lower granularity than the centralized training.

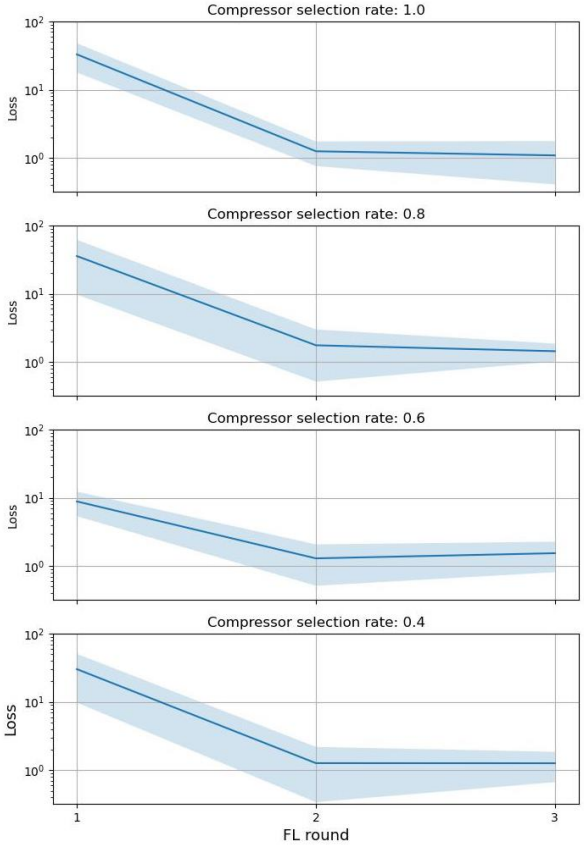


Figure 35: Loss evolution by compressor selection rate

Moreover, once the model is trained, the continual learning component (see Section 3.4.2.1) allows the system to cherry pick statistically significant data, thereby limiting the data transmission from 2020. **Errore. L'origine riferimento non è stata trovata.** shows the number of data drifts detected for the period 2021-2022. The average number of data drift detections is 8.2, with compressor K-7502 presenting the most detections with 12.

Table 9: Detected data drifts

Compressor	K-3201	K-5701	K-7502	K-3301	K-2201
Data drifts	7	8	12	7	7

The continual learning system adds the weeks with a data drift to the training dataset. Given the low average number of data drift detections per year, we configured the retraining period to one year. Moreover, as all sensors in a compressor measure the temperature, they show a high correlation, which results in the same data drift being detected by several sensors at the same time. Therefore, the client selection at production time is constant to a single sensor per compressor. As a result, the continual learning system transmits a single model per compressor per year, resulting in a transmission of 30MB. In contrast, the continual data transmission without continual learning produces a total of 52.8MB for all temperature sensors, as show above. Therefore, the continual learning system produces a throughput reduction of 76% per year.

5 Conclusions

This deliverable summarized the work that has been currently carried out at the last phase of the project (M15) with what concerns the *Energy Efficient Analytics Toolbox*. The latter consists of a number of building blocks that are being currently developed under the scope of WP5 ("*Trusted and Energy Efficient Analytics"* and more precisely T5.3 ("*Incremental Energy Efficient Analytics"),* T5.4 ("*Edge Data Management and EdgeAI Optimization")* and finally T5.5 ("*FML for Privacy Friendly and Energy Efficient Data Markets*").

This family of components involves firstly the *Incremental Analytics* component, which is responsible to provide energy efficient query processing on one hand, and on another hand to implement database operators that can calculate the result incrementally as the database is transited to a different state, as the result of a data modification operation. Secondly, the *Analytics CO2 Monitoring* provides monitoring and prediction of CO2 emissions of ML/AI algorithms, testing a wide range of classical and novel artificial intelligence and machine learning algorithms that could be utilised within the project datasets. Thirdly, the *Smart Deployment* component aims to provide a powerful way to deploy models and services on a specified infrastructure. Its aim is to establish a system capable of deploying a model given an infrastructure optimization of the process based on inputs received from the *Analytics CO2 Monitoring* service. Last but not least, the *FML Framework* is designed to be used in federated machine learning use cases, bringing to the overall FAME solution the necessary tools to enable customers to train purchased models in a federated manner using data hosted on different servers.

This deliverable provided a brief description of each of the aforementioned components, their final design and specifications, accompanied by the 3rd level C4 architecture. Then it followed a detailed technical description, highlighting the internal subcomponents, and the interactions among them and also among the other sub systems of the FAME integrated solution. The specification of the interfaces was provided when applicable. Later, a separate section demonstrated the installation, deployment and use of the technology, providing the documentation of the demonstrator, followed by an extensive evaluation of all prototypes.

In this last version of the deliverable, we have included how the outcomes of the work carried out by the three aforementioned tasks are now indexed in the FAME Marketplace. Last but not least, we provided the KPIs related with the successful criteria of our prototypes, and how we addressed them.

References

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., Patel, S., Patel, S., . . . Seth, K. (2017). *Practical Secure Aggregation for Privacy-Preserving Machine Learning.* Dallas, TX, USA: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.

Garst, S., & Reinders, M. (2024). *Federated K-Means Clustering.* Proceedings of the International Conference on Pattern Recognition.

McMahan, H., Moore, E., Ramage, D., Hampson, S., & Agüera y Arcas, B. (2017). *Communication-Efficient Learning of Deep Networks from Decentralized Data.* For Lauderdale, Florida, USA: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics.

McSherry, F., & Talwar, K. (2007). *Mechanism Design via Differential Privacy.* Providence, RI, USA: 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07).

Reddi, S. J., Charles, Z., Zaheer, M., Garrett, Z., Rush, K., Konečný, J., . . . McMahan, H. B. (2021). Adaptive Federated Optimization. *International Conference on Learning Representations.*

Annexes

Smart Deployment YAML

In this section are specified the YAML needed to deploy the Smart Deployment.

- ClusterRole. This is to allow the SD to get the nodes

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: smartdeployment-get-nodes
rules:
- verbs: ["get"]
  resources: ["nodes"]
  apiGroups: [""]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: fame-smartdeployment
  name: smartdeployment-deploy-models
rules:
- apiGroups: [""]
  resources: ["secrets", "services", "configmaps"]
  verbs: ["get", "create", "list", "update", "patch", "delete"]
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "list"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "create", "list", "update", "patch", "delete"]
```

- Role. This is to allow the SD to get access to the namespaced necessary resources.

- RoleBinding/ClusterroleBinding. With these bindings, the role/ClusterRole are bound to the serviceaccount of the Smart Deployment

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: smartdeployment-get-nodes-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: smartdeployment-get-nodes
subjects:
  - kind: ServiceAccount
    name: fame-smartdeployment
    namespace: fame-smartdeployment
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: smartdeployment-deploy-models-binding
  namespace: fame-smartdeployment
subjects:
  - kind: ServiceAccount
    name: fame-smartdeployment
    namespace: fame-smartdeployment
roleRef:
  kind: Role
  name: smartdeployment-deploy-models
  apiGroup: rbac.authorization.k8s.io
```

- Service Account. Definition of the Smart Deployment service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fame-smartdeployment
  namespace: fame-smartdeployment
```

- Service template. Template of the different SD services, giving external access to the modules.

```
apiVersion: v1
kind: Service
metadata:
  name: <MODULE_NAME>
  namespace: fame-smartdeployment
  labels:
    smartDeployment.environment: fame
    smartDeployment.type: <APP_LABEL>
spec:
  selector:
    app: <APP_LABEL>
  ports:
  - port: <PORT>
    targetPort: <PORT>
    name: <PORT_NAME>
```

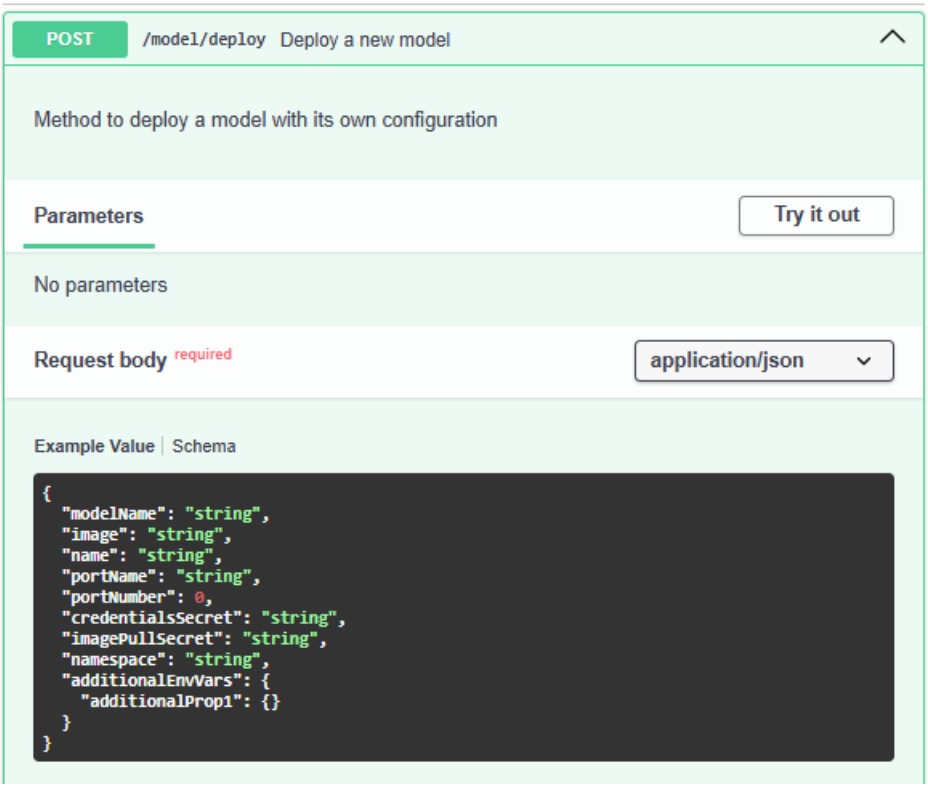
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <MODULE_NAME>
  namespace: fame-smartdeployment
  labels:
    smartDeployment.environment: fame
    smartDeployment.type: <APP_LABEL>
spec:
  selector:
    matchLabels:
      app: <APP_LABEL>
  template:
    metadata:
      labels:
        app: <APP_LABEL>
    spec:
      serviceAccountName: fame-smartdeployment
       # This is only needed if the image is in a private repository
      imagePullSecrets:
        - name: <IMAGE_REGISTRY_CREDENTIALS>
      containers:
      - name: <CONTAINER_NAME>
        image: <REPOSITORY_IMAGE_URL>
        imagePullPolicy: Always
        command: ["/bin/bash"]
        args: <COMMAND_FROM_IMAGE>
        envFrom:
          - secretRef:
              name: credentials
        ports:
        - containerPort: <PORT>
          name: <PORT_NAME>
```

- Deployment template. Template of the different SD deployment of modules.

SD Interfaces

SD API-Server

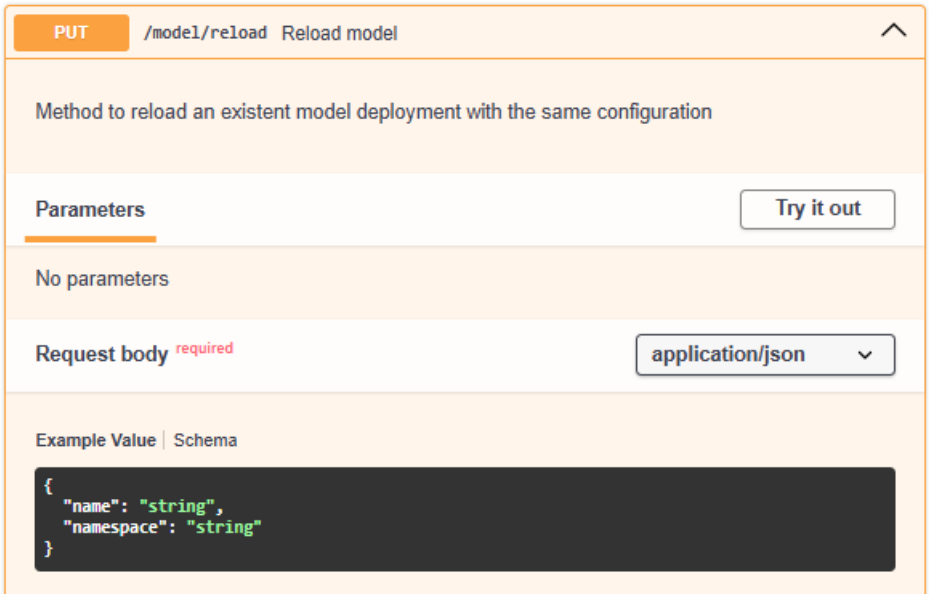
- Model Deployment



The user must provide the following information:

- *modelName*: Model name from inside MLFlow (optional),
- *image*: Image from the repository that will serve the model,
- *name*: Name given to the model that will be deployed,
- *portName*: Name given to the port,
- *portNumber*: Port number from which the image exposes the service,
- *credentialsSecret*: Secret with the credentials for the image (optional),
- *imagePullSecret*: Secret with the credentials to download the image if it is stored in a private repository (optional),
- *Namespace*: Namespace where the model will be deployed,
- *additionalEnvVars*. Additional environment variables to give to the container.

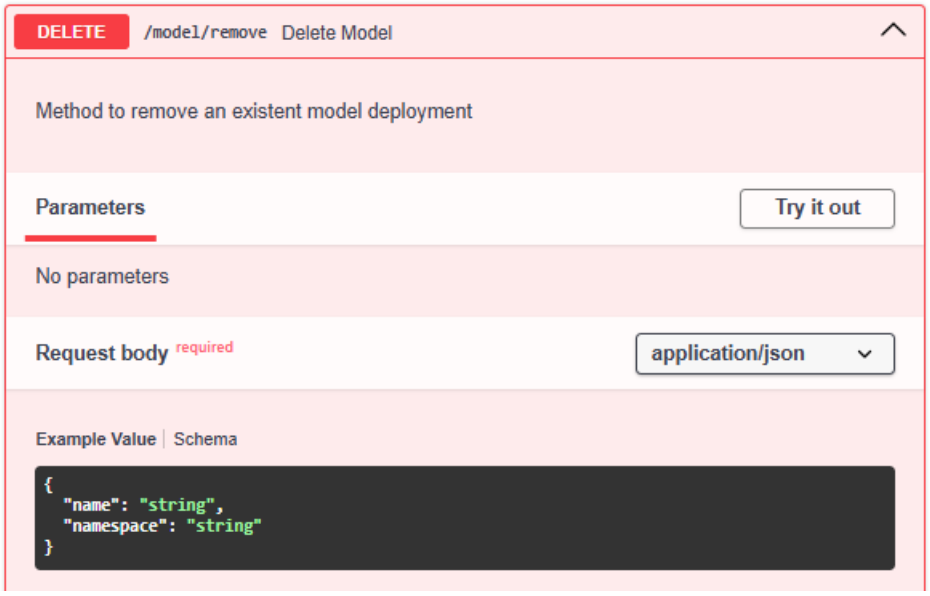
- Reload Model



The user must provide the following information:

  - *name*: Name given to the model that will be reloaded,
  - *namespace*: Namespace where the model will be reloaded.

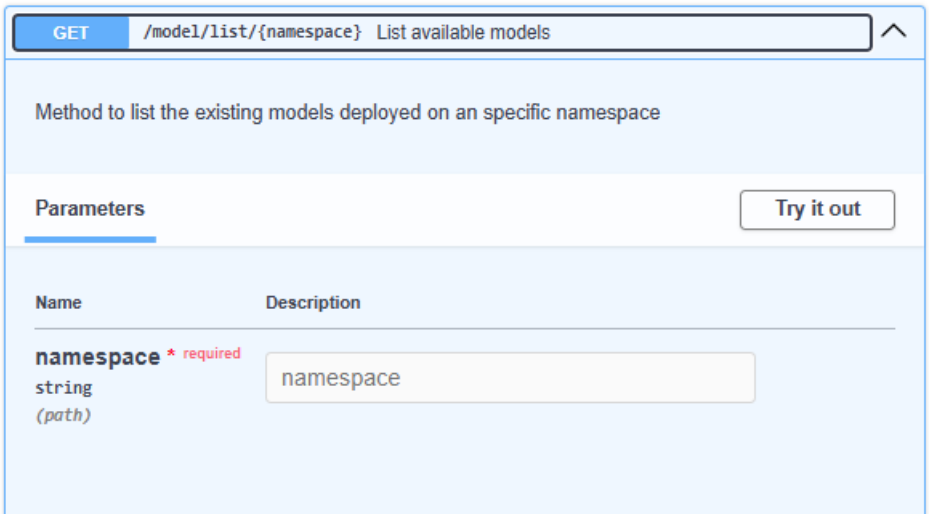
- Delete Model



The user must provide the following information:

  - *name*: Name given to the model that will be deleted,
  - *namespace*: Namespace where the model will be deleted.

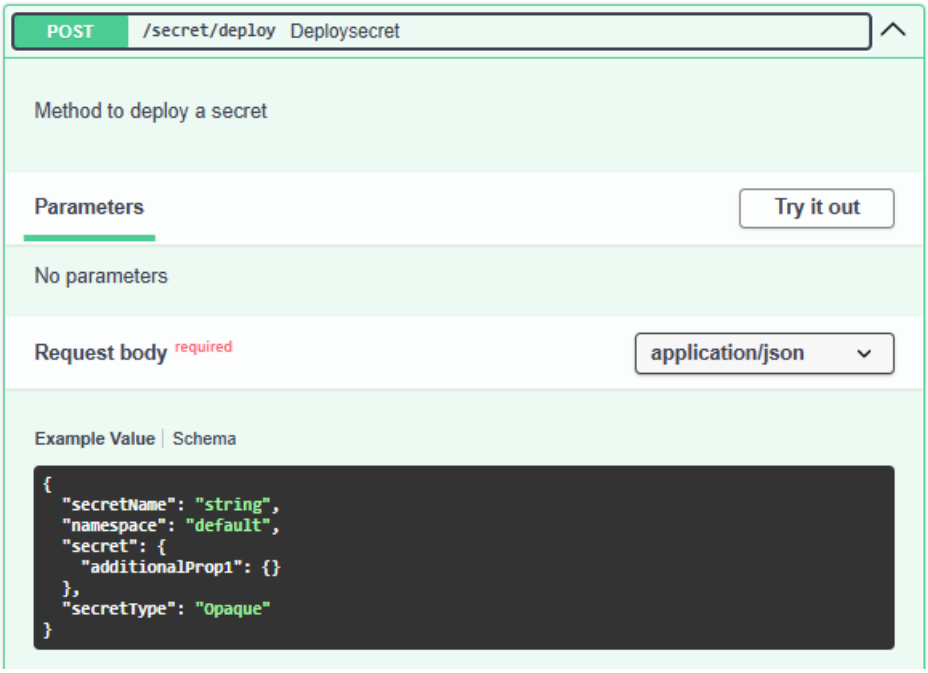
- List Model



The user must provide the following information:

  - *namespace*: Namespace where the models are deployed.

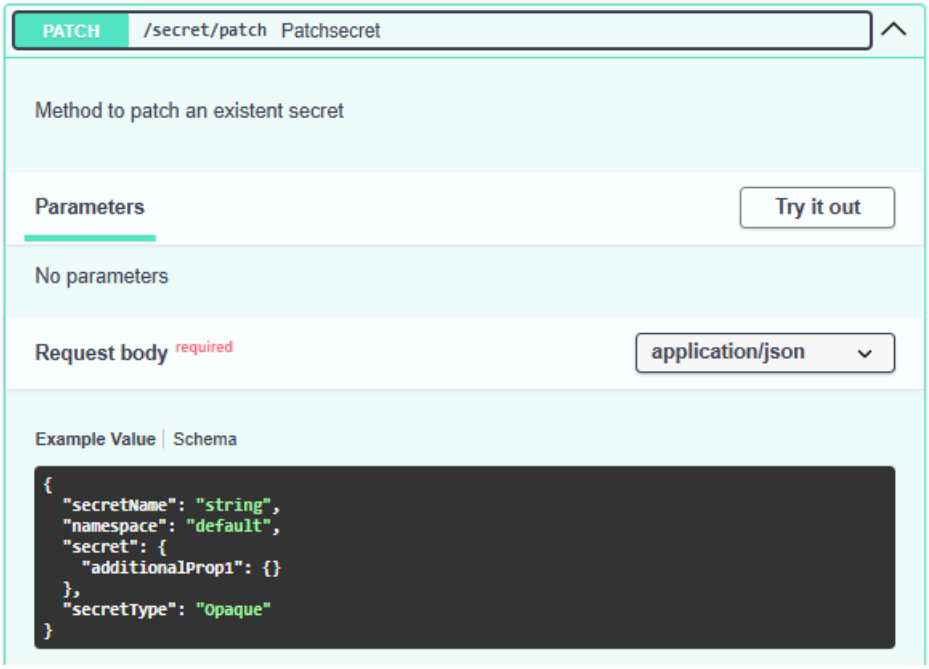
- Deploy Secret



The user must provide the following information:

  - *secretName*: Name given to the secret that will be deployed,
  - *namespace*: Namespace where the secret will be deployed,
  - *secret:*
    - *additionalProp1.* Dictionary with the corresponding key-value secrets to be stored,
  - *secretType.* The type of the secret.

- Patch Secret



The user must provide the following information:

- *secretName*. Name given to the secret that will be patched,
- *namespace*: Namespace where the secret will be patched,
- *secret:*
  - *additionalProp1*. Dictionary with the corresponding key-value secrets to be stored,
- *secretType.* The type of the secret.

FL Client Subscription

The client subscription mechanism implemented in the FML framework involves the following steps:

1. Upon execution start, the clients subscribe to the server. As soon as enough clients have subscribed, the server begins the federated rounds.
2. The server selects the clients that will participate in the current federated round from the pool of subscribed clients and broadcasts the initial model to them.
3. Each client trains this model with its local data. The locally trained model is forwarded back to the server.
4. The server aggregates the model from all selected clients and updates the global model. This step, which resembles the optimization step in centralized Machine Learning (ML), may apply different optimization techniques as proposed in (Reddi, y otros, 2021)

Figure 36 shows the steps mentioned above. Text in capital letters refers to pod's wires, whereas *req* represents a dictionary that is transmitted with the request.

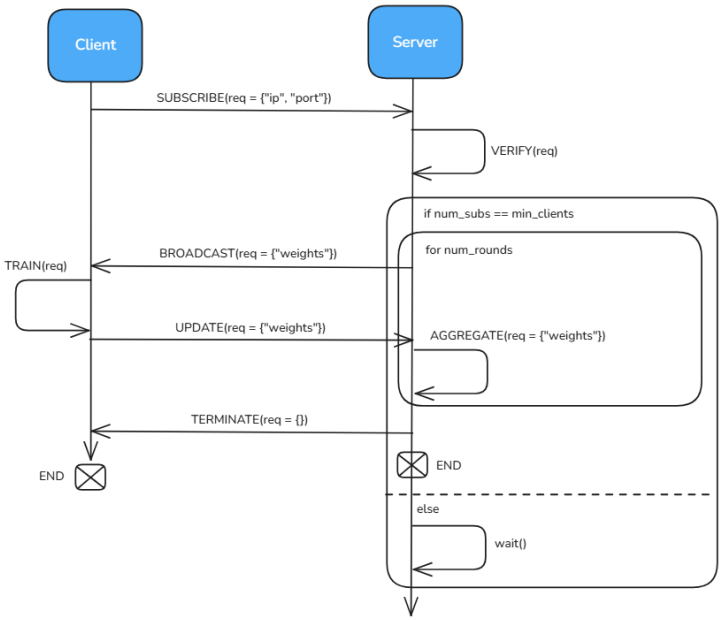


Figure 36 - Federated Learning Round