Federated decentralized trusted dAta Marketplace for Embedded finance

D5.2 - Energy Efficient Analytics Toolbox . I

Title	D5.2 - Energy Efficient Analytics Toolbox . I
Revision Number	3.0
Task reference	T5.3 T5.4 T5.5
Lead Beneficiary	LXS
Responsible	Pavlos Kranas LXS
Partners	ATOS, UPRC
Deliverable Type	DEM
Dissemination Level	PU
Due Date	2024-03-31 [Month 15]
Delivered Date	2024-03-22
Internal Reviewers	AL ECO
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	2024-02-01	LXS	ToC Version
0.3	2024-02-28	JSI	Input on CO2 monitoring
0.4	2024-02-29	ATOS	Input on SD and FML
0.5	2024-03-01	LXS	Input on Section 2
0.6	2024-03-08	LXS	Executive Summary, Intro and Conclusions
0.7	2024-03-11	LXS	Finalize the documents
1.0	2024-03-12	LXS	First version and internal review
1.1	2024-03-18	AL ECO	Internal review
1.2	2024-03-19	GFT	Internal review
2.0	2024-03-21	UPRC	Internal QA
0.2	2024-03-22	LXS	Input on Incremental Analytics
3.0	2024-03-22	LXS GFT	Submission

Definitions

Acronyms	Definition
ACID	Atomicity, Consistency, Isolation, Durability
AI	Artificial Intelligence
API	Application Programming Interface
ATOS	Atos It Solutions And Services Iberia Sl
AWS	Amazon Web Services
CD	Continuous Development
CI	Continuous Integration
CPU	Central Processing Unit
DB	Data Base
DDL	Data Definition Language
FAME	Federated decentralized trusted dAta Marketplace for Embedded finance
FDAC	Federated Data Assets Catalogue
FML	Federated Machine Learning
GUI	Graphical User Interface
IBM	International Business Machines
ID	Identity
IEEE	Institute (of) Electrical (and) Electronic Engineers
IN	Intelligent Network
JDBC	Java Database Connectivity
KPI	Key Performance Indicator
ML	Machine Learning
MVCC	Multi Version Concurrency Control
MVP	Minimum Viable Product|Platform
ODBC	Open Database Connectivity
OS	Operating System
RAM	Random Access Memory
REST	Representational State Transfer
SA	Supervisory Authority
SAX	Situation Aware Explainability
SQL	Structured Query Language
SSD	Solid State Drive
UI	User Interface
XAI	eXplainable Artificial Intelligence

Executive Summary

This deliverable presents the work that has been carried out during the first 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 at the edge, using the **smart deployment** component presented in this deliverable, while enabling federated machine learning techniques via the **FML orchestrator**. 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 first version of the deliverable, accomplishing the target objectives of milestone MS09. According to the level of maturity of the aforementioned components, it provides more details about their internal architecture, describing the related work and baseline technologies that have been used, while also providing technical specifications and interface documentation, when applicable. These technical specifications are accompanied by a first version of demonstration of their use. In the second and final version of this deliverable, the final specifications and demonstrators will be presented, along with the evaluation of the technology outcomes with respect to the relevant KPIs, taken from the DoA's *Objective 5.* This second version is planned to be released in M27.

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 at this 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 Orchestrator*.

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 before going into more details about their design and specifications. After positioning into the FAME SA, we provide further deep details per component, starting with a brief description of its purpose and then giving more information about the related work towards accomplishing the objectives of each one of those. We accompany this description 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. Finally, we provide some representative demonstrators regarding the installation, deployment and use of the components.

It is important to highlight though that at this phase of the project (M15), different focus has been given to different components, according to their importance and priority to fulfil the MVP user stories of the overall FAME project. This had as a result that different components have different level of maturity at this phase, thus some might be presented with more details, while for others their design or implementation is still in progress. Moreover, due to the change of leadership of T5.5 ("*FML for Privacy Friendly and Energy Efficient Data Markets*") that took place soon before the writing of this deliverable, this task has been restarted and the majority of the effort will be spent during the second phase of the project. As a result, the *FML Orchestrator* related technological outcome is currently under design. The final design and implementation of all components presented in this deliverable will be included in the next and final version of this deliverable, which will include the final demonstrators of the *Energy Efficient Analytics Toolbox* architectural layer of the overall FAME integrated solution.

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 will evaluate our prototypes against the corresponding KPIs of this objective. However, the evaluation phase will take place during the second and last phase of the project and will be included in the second and final version of this deliverable.

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.
- *Section 3* provides the technical specification per component.
- *Section 4* demonstrates the use of each component.
- *Section 5* concludes the document.

2 Positioning into FAME SA

This section describes where the *Energy Efficient Analytics Toolbox* is positioned with regards to the overall FAME SA. Figure 1 depicts the overall architecture at its current version described with more details in the D2.2 ("Technical Specifications and Platform Architecture"). In the following figure, 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.

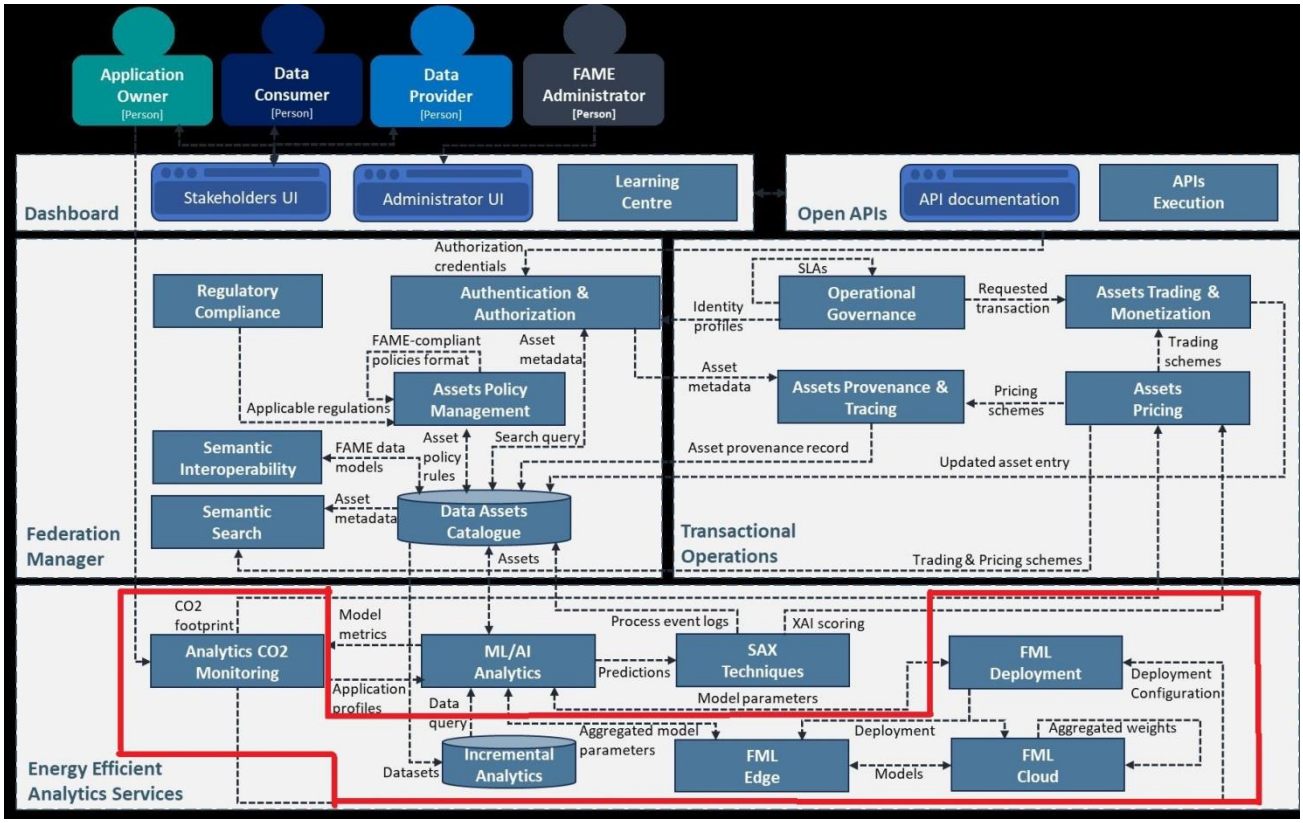

Figure 1: FAME SA C4 container diagram

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 sections provide more details regarding the specification of each of the aforementioned components, with the addition of demonstrators of their use. It is important to

highlight however, that the task responsible for *FML Orchestrator* has been recently restarted and therefore, this document gives some initial details of its design and specification. The next version of this deliverable will include the complete and final description of all the components, along with their demonstrators.

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.

The *Incremental Analytics* component relies on the LeanXcale database, which can be considered as the background technology that the T5.3 ("*Energy Efficient Incremental Analytics"*) is built upon. The LeanXcale database can be classified in the NewSQL family of databases that combine the benefits of two worlds: the traditional relational databases that are compliant with the SQL query language and force transactional semantics, ensuring the so called ACID properties, and the NoSQL databases that are meant to be used in cases where data is supposed to be ingested at very high-rates (from batch processing of very high volume to streaming processing of very high velocity) where database transactions are not a first-class citizen and can lead to unnecessary bottlenecks. As LeanXcale DB can provide complementary capabilities of both SQL and NoSQL worlds, it has been used as the fundamental pillar of the *Incremental Analytics* component, enhanced with the innovations developed under the T5.3 task in order to provide both incremental and energy efficient analytics. It is important to highlight that with regards to this component, the term *analytics* is referred to the database analytic operations that are requested from the high-level ML/AI analytic processing.

3.1.2 Related Work

In this subsection, we give more details regarding the innovations developed under the scope of T5.3 ("*Energy Efficient Incremental Analytics"*) and currently merged into the *Incremental Analytics* component. We start by describing how we tackle the energy efficient part of the query processing and then we give more information regarding the incremental analytic query processing. For the former, we rely on a novel indexing mechanism currently introduced into the database and additionally the ability of its query engine to push down query operations to the data nodes, exploiting this mechanism. For the latter, we make use of one additional novelty, the *online (or real time aggregations).*

3.1.2.1 Energy Efficient Query Processing

In order to comply with the requirement for energy efficient query processing, we firstly introduced a novel indexing mechanism that was developed during the first period of the project. We started by observing the need that distributed databases need to partition tables into fragments in order to be able to distribute the workload across nodes. Leanxcale DB, which the *Incremental Analytics* is built upon, is a distributed database that shares the same need. Let us discuss how distributed databases handle partitioned tables and in particular what implications have for indexed access.

Tables are distributed according to some criterion, the most common one is to use the primary key to partition the table into fragments corresponding to different primary key ranges.

Let us assume the primary key of a table, T, can have values from 0 to 255 and that we have 4 servers. In order to partition it across four servers we need to have at least 4 partitions. So let us partition it into 4 fragments. Let us assume the access of the different rows has the same probability, so we just need to partition into 4 fragments of the same size: [0, 64), [64, 128), [128, 192), and [192, 256). Let us call the fragments p0, p1, p2, p3. Now each fragment can be located on each of the four servers. Since all rows have the same probability of access, each server will get a fourth of the workload for that table.

But what happens if we need one or more secondary indexes to query efficiently the table? How will be the secondary indexes partitioned and distributed? The traditional way used by distributed databases is to treat the secondary index as another table. We call this approach global index, because it is an index for the whole table, which is why it is called global as it is meant to be used for the full table. Let us assume that the secondary key is an integer column with values between 0 and 64535 and that all secondary keys have the same access probability. This means that we can partition the global index in the following 4 fragments: [0, 16384), [16384, 32768), [16384, 49152), and [49152, 64536). Let us call the fragments s0, s1, s2, and s3. Now we can distribute the secondary index across the four servers. However, we didn't think about a crucial aspect. Since the secondary index is not collocated with the actual rows, we need a way to get from a secondary key to the associated row. The only way since there is no collocation of secondary keys with the rows because they have different partitioning criteria is to associate the primary key to the secondary key. Thus, we can search through a secondary index using a range of secondary keys and we will get the primary keys of the secondary keys we searched. But now, each primary key has to be searched individually to get the row. If we are looking for 100 rows with secondary keys stored in s0 and s1, we will get 100 primary keys associated to the 100 secondary keys. Now we have to search each of the primary keys that can go to any server and read the full and select the set of columns required by the query. This will require 2 accesses for doing a search scan on two fragments of the secondary index, s0 and s1, plus 100 accesses to any of the four servers to get the full row from any of the fragments p0, p1, p2, and p3. If instead of 100 rows we would be reading 10 million, we would require 10 million individual accesses and each individual access has to cross the network from the query engine to the target storage server to get the full rows. This is way too expensive and one of the weaknesses of distributed databases that result in high energy consumption for indexed searches that are the most common ones.

Under the scope of T5.3, we have developed two additional techniques in the context of the FAME project to improve energy efficiency by reducing the cost of indexed accesses: *local* indexes and *global covered* indexes. The first technique, local indexes, targets small deployments with few servers. The second technique, *global covered* indexes, targets medium and large deployments with any number of servers.

Local indexes are secondary indexes that are local to a fragment. That is, they are indexes that have the secondary keys contained by the rows of the fragment. This means that a table with four fragments will have four local indexes for each secondary key, local to each fragment. Since there is no collocation between rows and secondary keys, to search a secondary key is necessary to send the scan over the secondary index to each local index. Each local index will perform the search and then provide the address of the row that then will be locally as part of the scan recovering for each row the requested columns. For a couple of storage servers, it is pretty efficient since it will perform two scans and as many local reads as rows accessed. So in terms of network, two requests will be

sent and replied to, and in terms of operations, two scans will be performed on each local index, and then as many reads as rows recovered. As can be noticed as the number of storage servers grows, the access through the secondary index becomes less and less efficient since more storage servers are impacted (all containing fragments of the table).

Global covered indexes are global indexes that are extended with the columns required to answer the queries that will use the secondary index. As traditional global indexes they are partitioned according to ranges of the secondary key. However, the secondary keys do not just have associated the primary key, but also all columns required to answer the queries relying on the secondary index. This means that a search through a secondary index that involves only one fragment of the secondary index will simply read the range and get automatically the columns it needs to answer the query without any other further accesses. In this way, the access is extremely efficient, and the technique keeps its efficiency independently the system size in number of servers. The main trade off of the technique is with respect update operations. Any update on the columns stored in the secondary index will require to update the secondary index with an additional update operation. However, inserts, and deletes will preserve its cost.

Thus, with our implementation we have greatly improved energy efficiency by improving indexed accesses that are the most commonly used and what mean the highest fraction of the workload of most databases.

Another innovation of the *Incremental Analytics* component that contributes to increase the energy efficient of the query processing is the ability of its relational query engine to push down operation to the data nodes. The LeanXcale database that this component is built upon consists (among others) with a list of different data nodes that the data are persistently stored, we call them KiVi nodes, and the relational query engine nodes that provide the SQL interface to the data user or ML/AI analytical processes, compile the SQL statement into a query execution plan and retrieve the data from those data nodes. Typically, the SQL statement is provided as a String and is compiled by the query engine to a tree data structure that can be later programmatically manipulated by the engine. To have a concrete example, let's take a look at the following query:

```
SELECT t1.name, t2.account_number
FROM Persons as t1 INNER JOIN Accounts as t2 on t1.person_id = t2.person_id
WHERE t1.age > 60
```

This query involves a JOIN operation over two tables, *Persons* and *Accounts*, a filter operation on the *Persons* table to retrieve data for persons having a particular age, and a projection operation to return only the corresponding columns. After being compiled by the query engine, this statement could be transformed to the following query execution plan.

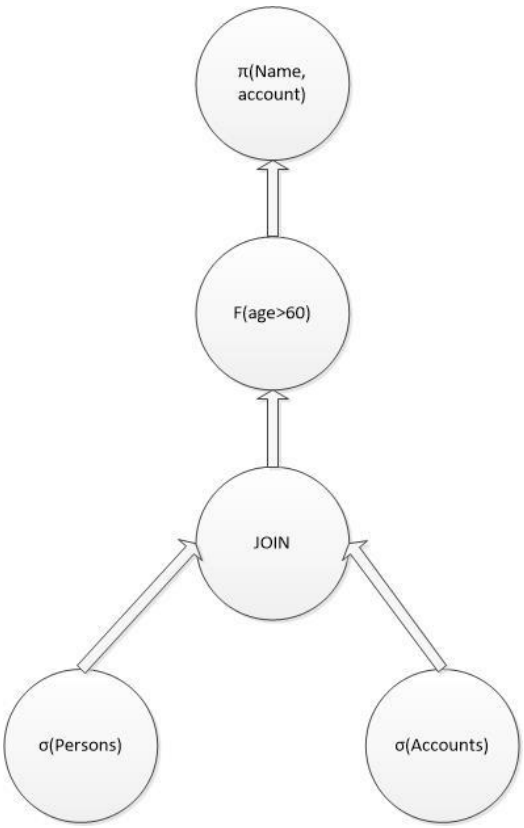

Figure 2: Example of a query execution plan

This query plan implies that two *select* operations will be executed, σ(Persons) and σ(Accounts) to get data from the data tables stored into the data nodes, then the result of these operations will be further forward to the upper tree node of the query execution plan, in order to apply the JOIN operation. The result of the latter will be sent to the *filter* operation, to remove all those tuples that do not meet the corresponding condition. Finally, the *project* operation will discard all columns that are not included in the projection list, and it will only return the *name* and *account_number* ones.

Even if the query optimizer could decide a different but equivalent query plan (probably pushing down the filter operation before the JOIN), a traditional approach would imply to first retrieve all data from the corresponding tables with the σ operators and then apply the remaining ones. With the advancements developed under the scope of T5.3 ("*Energy Efficient Incremental Analytics"*), the KiVi data nodes are able to provide enriched query processing directly to the data storage, thus giving the ability to the query engine to push down operations as closest as the storage itself. This creates several benefits, from reduced latency to reduced energy consumption. We highlight this with two concrete examples.

Firstly, we will examine one of the most common query operators, the filter condition. For the sake of simplicity (but without downgrading the generality or our design), let's assume we have a single table that contains 1 billion rows and a filter criterion over an index. Without pushing the filter operation down to the data nodes, it would have required all data nodes to be activated to execute this query and send all 1 billion of rows to the query engine itself. That would require 1 billion I/O access, network transmission of such amount of data and increased CPU cycles (apart from memory consumption) on the query engine itself to filter out the data tuples that do not satisfy the query criteria. By having the query engine capable of pushing this operation down to the data nodes, and taking advantage of our novel indexing mechanism, we can activate only the data nodes that actually stores data that meet the filtering criteria and transmit only the valid records. This increases significantly the effectiveness of our solution, reducing the carbon footprint related with this

operation, as we reduce all I/O access, network transmission, memory consumption and CPU cycles in the query engine related with very commonly used query operation.

Secondly, we will examine a more complex, still recurrently appeared family of operations related with aggregations. We also assume the same single table that contains 1 billion rows we want to get the summary of the value of a specific column. Without pushing down to the data nodes, the query engine would have to activate all the data nodes in order to retrieve the overall dataset (1 billion rows) and instead of filtering out undesired ones, it would have calculated the aggregated operation of summarizing that value. Again, this would have required the I/O access of 1 billion rows, their data transmission over the network and a similar consumption of memory and CPU power. Being able to push this family of operations down to the storage, we force the storage to do this processing on behalf of the query engine. We still need to activate all data nodes, which will need to perform 1 billion I/O accesses, but instead of transmitting the whole dataset to the query engine, each data nodes calculate the local summary and transmit only 1 value. By doing that, we only transmit as much values as the number of the data nodes and let the query engine summarize only these few values in order to provide the global summary. In fact, the global summary is the summary of the local ones, the global counting is the summary of the local counting operations, and the global maximum and minimum is the maximum and minimum of the local respective ones. The only difference is the global average that is not the average of the local averages. However, the global average is the division of the global summary by the global counting, and these operations can be executed in a distributed manner.

We saw how both our innovations developed under the scope of T5.3 ("*Energy Efficient Incremental Analytics"*) can lead to a significant increase of energy efficient query processing. In the second version of this deliverable, after the experimentation with various use cases of the project we plan to provide more details about the different cost estimation of the query plans generated by this component. Moreover, we plan to provide benchmark results comparing before and after the introduction of our novelties and show the different usage of different resources: cpu, network transmission, memory consumption, etc.

3.1.2.2 Incremental Query Processing

One of the most recurrent *pains* in the industry that can appear in a variety of diverse verticals (from the finance and insurance section, to the telecom providers and the manufacturing and maintenance) is the need to compute specific KPIs taking into account all historical data associated with a particular KPI and provide the results in real time, visualizing them in a dashboard or trigger specific alerts or actions when some conditions or rules are met. For instance, a forex trading analyst might be interest to monitor the average deviations of a given currency over a period of time, and once he or she notices some abnormalities, to take specific actions (i.e. transfer some money in a different currency). A different use case might be relevant for the manufacturing section and for the maintenance of equipment. They might be interested in monitoring the average and maximum or minimum aggregated values collected from a set of sensors related with the equipment in order to predict when some parts are most likely to fail in the short future.

All these different scenarios fall into the same use case: Data being collected from external sources need to be stored in a data management system and be part of what is called historical data. Usually, the primary key of these data is composed by the identified of the source (i.e. the sensor that generated the data or the specific currency) plus a timestamp of the point in time when the data was acquired. Having a continuous acquisition of such information into the data management system, the data users or ML/AI analytic processes require the calculation of aggregation operators in order to take decisions by observing the results of such operations that might be relevant with the average,

summary, minimum, maximum or counting of a specific value over a given period of time, or even worse, over the overall historical data that exists in the system.

There might be several ways on how to design such scenario. We start with the most naïve. The simplest approach would be to rely on the ML/AI analytic processes to do this calculation on itself. That would imply that the ML/AI processes need to access the whole historical data in order to perform this type of operations. In order to do so, they would request from the data management system to send to the processes all the historical data and do the processing in memory. That would imply I/O access of all the individual data records, network transmission of GBs or even TBs of data and finally, similar requirements for memory consumption in addition to the huge amount of CPU cycles spent to calculate the aggregated operations. To avoid such waste of resources, system architects and data scientists typically push down these operations to the data management systems (i.e. a database or data warehouse) to make these calculations on behalf of the ML/AI analytic processing.

For a database, executing an aggregated operation requires to compute all the corresponding values found in every historical data record. For instance, to find the maximum value, it needs to traverse all historical data, comparing the current maximum with the value of the data record being currently examined, and finally return the global maximum of a given historical dataset. The same applies to all database analytic operations. That means that for an aggregated operation, the database requires running large analytical queries to compute the corresponding aggregates over the historical data, whose response arrives with certain latency, usually proportional to the overall space of the historical dataset. That is, the bigger the dataset, the bigger the latency. As a result, it is not the best solution when the scenario requires these calculations to take place in real time and the end user needs to continuously monitor the deviation of such values.

Taking into consideration the current *pains* recurrently observed in the industry, our *Incremental Analytic* component follows a different approach. Instead of forcing the database to always calculate the aggregated values per request, we pre-calculate these values in advance, and thus, provide incremental analytic processing as data is being ingested in the database at the same time. We call these operations *online (or real-time)* aggregations. We have extended the DDL dialect of the database in order to allow the data user to explicitly define such cases. Internally, the database creates a *derived table* where the aggregated data can be stored. We call these tables as *delta tables*, which contains a *primary key and delta column.* The latter can be of numeric data types. When data is ingested into the database, the data is persistently stored to the corresponding table, while updating the value of the corresponding *derived table* at the same time. As a result, with any update of the historical dataset (i.e. ingestion of a new record), the *online aggregate* is updated in real time, or else, incrementally. Once the data user needs to get that value, the database does not have to traverse the overall historical dataset stored in the *parent table*, but instead, it only accesses the corresponding data record of the *derived table* and immediately returns the now incrementally updated value. The gain of performance in that case is extremely high. As we mentioned in the previous paragraph, using traditional approaches the latency of the execution is proportional to the overall space of the historical dataset, as it requires to traverse all data records, which implies a computational complexity of *O(n)*. With the incremental processing, the database requires to only access the particular key, which implies a computational complexity of *O(1).*

It is important to highlight that due to the introduction of our novel *delta columns and tables,* our approach continues to satisfy transactional semantics, without the bottleneck introduced by the traditional transactional management. In traditional relational databases, in order to perform this pre-calculation of the aggregated value the use would have to create a multi-statement transaction,

which first adds the new record in the parent table and then read, set and update the value of the derived one. However, as the updates on the derived table happen over the overall historical dataset (or over the granularity of the *group by* clause), each concurrent insert operation will create a *write-write conflict* on the derived table, which will make the transaction to abort. In our approach, instead of directly updating the values of the derived tables, we add the *delta* with respect to the previous value. For instance, if a new data record is added with a given value XYZ, the database transparently adds the delta value of +XYZ. As deltas are only inserted, there is no *write-write conflict* in the derived tables and concurrent transactions can perform without having to be aborted. When a read operation takes place, the database calculates the current list of unresolved delta values and returns the incrementally updated value instantly, instead of having to traverse the overall historical dataset. From the perspective of the query engine, the data user still continues to rely on the standard SQL query language. It is the query engine itself that holds meta-information of such tables upon their definition with our extended DDL dialect, and instead of relying on the parent table to calculate the aggregated value, it makes use of its internal operations that attack the derived table.

In the second version of this deliverable, additional information will be given related with the analysis of the different query plans that can be performed by the *Incremental Analytics* component when such type of queries is requested by the database. At the time when this first version of the deliverable was written, the integration with concrete use cases of the project was just started. After the integration and further experimentation with the Pilot #7: *("Assessing the Quality and Monetary Value of Data Assets"),* we will include an analysis of the query plans that can be decided by our component, with or without the use of our innovation presented in this subsection. Our roadmap is to firstly provide an extensive benchmark analysis that validates our approaches and highlights the benefit gained in terms of latency. Moreover, as the query plans are accompanied with a cost estimation of the query operators involved in the execution of a query statement, we will include an analysis of the query plans that can be decided by our component, with or without the use of our innovation presented in this subsection. This will highlight even more how our approach further contributes to the energy efficient query execution.

3.1.3 Technical Specification

3.1.3.1 Component-level C4 Architecture

In this subsection, we give more details about the internal architecture of the *Incremental Analytics* component. 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 *Incremental Analytics* can be depicted in Figure 3.

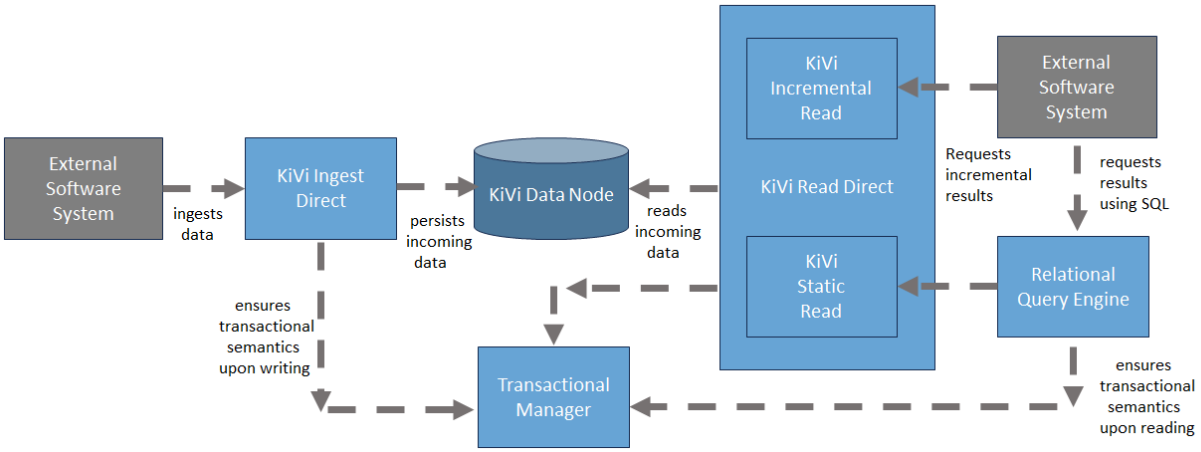

Figure 3: Incremental Analytics 3rd level of C4 architecture

In this figure we can see the various internal subcomponents of our solution. As previously mentioned, the *Incremental Analytics* relies on the background technology of the LeanXcale datastore, enhanced with the foreground technology implemented under the scope T5.3 ("*Energy Efficient Incremental Analytics"*). As a fact, it mainly follows the internal architecture of the database itself, while the innovations developed during the project have been already incorporated in the corresponding subcomponents and will be further evaluated during the next phase of the project as a whole.

In the heart of our solution lies the list of the *Kivi Data Nodes*, which contain the actual data that are persistently stored into the database. These data nodes provide specific interfaces that allows to both read (retrieve data) or provide data modification operations (DMOs). In that sense, we have conceptually split these two types of operations in different sub elements. The Kivi Data Nodes have been enhanced with the implementation of the novel *local indexes* that has been explained in the previous section.

The *Kivi Ingest Direct* is the sub-element responsible for storing data into the database. It provides a list of different connectors to the external software systems and is capable of ingesting data at very high rates. This implies either periodic batch ingestion of data of significant volume, or streaming ingestion of data streams that arrive with significant velocity. The deployment of this component can be further instantiated by the FAME marketplace in order to trigger such data ingestion scenarios when a FAME user wants to exploit the *Incremental Analytics* component and feed the database with some data. Moreover, as the LeanXcale DB can be considered as a relational database, the *Kivi Ingest Direct* interacts with the *Transactional Manager* to ensure transactional semantics and force the compliance with the ACID properties of a database.

The *Relational Query Engine* subcomponent, as its name implies, provides the relational query processing capabilities of the entire solution. LeanXcale DB is a relational and fully SQL compliant database, and as a fact, it provides standard SQL interfaces for the external software systems to interact. These are both JDBC and ODBC implementations, with the addition of a list of different data connectors for popular analytical processing frameworks that can be often found in the industry (i.e. Apache Spark, Apache Flink connectors, etc.). This subcomponent receives query statements, compiles them into a tree data structure, and includes the *query optimizer* and *query execution* whose purpose is to explore a cost efficient and equivalent query plan to be executed. Internally, it holds meta-information of the data schema for the *query optimizer* to rely on and various implementations of the query operations that can be used during the runtime when the query plan is being executed. These query operations are responsible for pushing the operations down to the data

nodes for an energy efficient query processing, while others make use of the operations developed to exploit the *incremental online aggregates* described in the previous section. As a relational database, the *relational query engine* also interacts with the *transactional manager* subcomponent to ensure that data that are read meet the *Isolation* ACID property when concurrent transactions co-exist.

The *Kivi Read Direct* is the subcomponent responsible for the actual retrieval of the data from the data nodes. Conceptually, it can be split into two different elements: the incremental and the static read. The static read includes the global and covered indexes of our novel indexing mechanism, while the incremental read is responsible for pre-calculating the results of the aggregated operations incrementally, as described in the previous subsection. Both these sub-elements directly interact with the *Kivi Data Nodes* and provide the interfaces to the *relational query engine* to push down these operations to that subcomponent. As a result, the *Kivi Read Direct* can be considered as an internal part of the entire solution.

Last but not least, the *Transactional Manager* is yet another internal component of the solution, whose purpose is to ensure transactional semantics and force the ACID properties of the database. It relies on the *Snapshot Isolation* paradigm for dealing with transactions, which removes the need of distributing shared or exclusive locks on the data items when a transaction is performing operations on the data nodes. Instead, this paradigm relies on the *Multi-Version Concurrency Control (MVCC)* that relies on having different snapshots of data items, distinguished with a labelled timestamp. This component interacts with all sub-elements that are responsible to execute read or write operations to the data nodes, however, its functionality is outside the scope of this project.

3.1.3.2 Baseline Technologies and Tools

As it has been mentioned in the previous subsection, the *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"*).

Table 1: 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.

3.1.4 Interfaces

The *Incremental Analytics* component relies on the LeanXcale DB, which can be considered as a relational, full SQL compliant database. All innovations developed for the purposes of this component have been already integrated into the core (or internal) subcomponents of the database itself. Therefore, in order to exploit the innovations presented in this section, the data user or the ML/AI analytic process need to use the same interfaces that LeanXcale DB exposes. As a relational database, it provides standard data connectivity mechanisms, which are standard JDBC and ODBC implementations. Additionally, this component has been already integrated with popular analytical frameworks (i.e. Apache Spark, Apache Flink, etc.) and popular ORM frameworks (i.e. Apache OpenJPA, Hibernate, etc.). By doing that, the data user or ML/AI analytical process does not even

need to know how to explicitly connect to our component, as they only need to make use of these frameworks that takes this responsibility on behalf of the users.

3.2 Analytics CO2 Monitoring

3.2.1 Description

The component is targeted at monitoring and prediction of CO2 emissions for artificial intelligence and machine learning algorithms applicable for FAME data, techniques, and solutions.

3.2.2 Related Work

A number of related methods have been focused on tracking energy consumptions and CO2 emissions accounting. In [1], Budennyy et al. introduce an open-source package eco2AI to help data scientists and researchers to track the energy consumption and equivalent CO2 emissions of their models. Other relevant solutions for carbon footprint estimation are: CodeCarbon [2], Cloud Carbon Footprint [3], Carbontracker [4].

The specific contribution of the component is the adaptation of CO2 monitoring and analytics tasks for the FAME environment and available FAME scenarios. In the component we are testing a wide range of classical and novel artificial intelligence and machine learning algorithms that could be utilised within project datasets.

3.2.3 Technical Specification

3.2.3.1 Component-level C4 Architecture

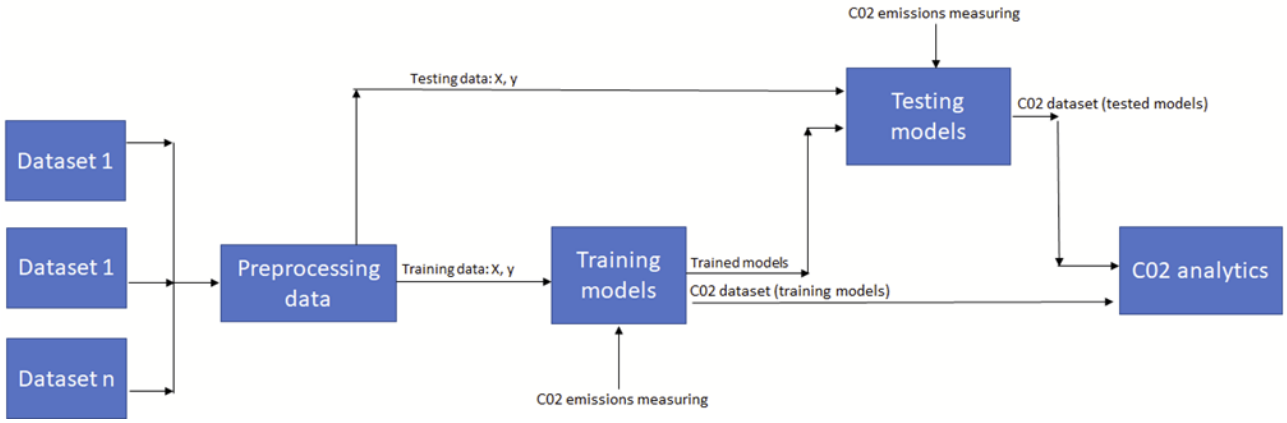

Figure 4: Initial Analytics CO2 Monitoring Architecture diagram

The main elements of the CO2 monitoring and analysis include a number of important steps, such as:

- Collection of relevant datasets (available internally and externally);
- Data preprocessing/data cleaning;
- Generation of CO2 emission data for training (experimentation with different artificial intelligence and machine learning algorithms);
- Generation of CO2 emission data for testing (experimentation with models);
- CO2 analytics (in FAME context);

Additionally, a number of XAI techniques might be used for explaining CO2 predictors (in FAME context).

The features for generated CO2 emission data currently include:

- id,
- project_name,
- experiment_description,
- epoch,
- start_time,
- duration(s),
- power_consumption(kWh),
- CO2_emissions(kg),
- CPU_name,
- GPU_name,
- OS,
- region/country,
- cost

Additional features (to be considered): dataset name, problem complexity measure, number of features, separate model specific parameters from the experiment_name-number of classes, number of missing values, number of categorical_features, number off continuous_features, binary regression/classification characterization, number of instances.

3.2.3.2 Baseline Technologies and Tools

As a baseline tool for CO2 monitoring, we are currently using eco2AI [1]. The eco2AI is an open-source library capable of tracking equivalent carbon emissions while training or inferring Python-based AI models accounting for energy consumption of CPU, GPU, RAM devices.

Tested algorithms include:

- kNN(with k dependency, to be separated into a different feature later on): the k-nearest neighbours algorithm (k-NN) is a non-parametric supervised learning method frequently used for classification and regression;
- ANN (with a test in different dropout layer and hidden layer sizes): artificial neural networks, a branch of machine learning models;
- Random Forest (for different number of estimators): an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time;
- DecisionTree: a decision support hierarchical model that uses a tree-like model of decisions and their possible consequences;
- Logistic regression: estimating the parameters of a logistic model (the coefficients in the linear combination);
- NaiveBayes: family of linear "probabilistic classifiers" which assumes that the features are conditionally independent, given the target class;
- XGBClassifier: eXtreme Gradient Boosting is an open-source software library which provides a regularising gradient boosting framework;
- AdaBoosing: a statistical classification meta-algorithm;
- GradientBoosting: a machine learning technique based on boosting in a functional space, where the target is pseudo-residuals rather than the typical residuals used in traditional boosting;

- LGBMClassifier: short for light gradient-boosting machine, is a free and open-source distributed gradient-boosting framework for machine learning;
- CatBoostClassifier: a gradient boosting classifier which among other features attempts to solve categorical features using a permutation driven alternative compared to the classical algorithm.

3.2.4 Interfaces

Interfaces have not yet been defined at this phase of the project.

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 and services.

Nowadays, almost every human action is somehow recorded, indicating that there is a vast amount of data available for exploitation. Machine Learning, 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 machine learning project: the Data Scientist and the MLOps Engineer. While the former develops the model, the latter focuses on its deployment.

The SD component plays a role akin to that of an MLOps engineer, ensuring that a given model is made available to the end users. It is important to note that it does not cover the training phase. Given an ML model, the component can read it from the repository, encapsulate it in a Docker image and deploy it on a specified infrastructure.

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 need to be 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 and downloading the model, to making it available to users/clients.
- A REST API architecture is required to serve the model.

The following sections describe this component in detail.

3.3.2 Related Work

The state of the art related to this component focuses on automating the deployment of Machine Learning models. The Machine Learning Operations (MLOps) paradigm addresses all aspects related to bringing ML models into production, as highlighted in [1]. In recent years, numerous open-source technologies addressing various aspects of MLOps have been developed, as detailed in [2]:

- **TensorFlow Extended**[1]. A framework that provides libraries for every task in a machine learning pipeline, including a comprehensive configuration system and a machine learning model serving component.
- **Airflow**[2]. A tool for orchestrating tasks and workflows, also suitable for managing machine learning pipelines.

[1] https://www.tensorflow.org/tfx

- **Kubeflow**[3]. A machine learning platform built on Kubernetes, covering model development, deployment, and serving.
- **MLFlow**[4]. A machine learning platform designed for end-to-end lifecycle management, offering advanced experiment tracking, a model registry, and a model serving feature.
- **Jenkins**[5]. A CI/CD server that plays a crucial role in the MLOps paradigm by enabling the automation of machine learning workflows. Jenkins facilitates continuous integration (CI) and continuous delivery (CD) for machine learning projects.

Many studies aim to use one or more of these tools to implement end-to-end ML pipelines. In [3] and [4], workflow orchestrators such as Airflow, Kubeflow pipelines, and Jenkins are integrated with MLFlow to build ML platforms. [5] describes a complete end-to-end MLOps platform based on Kubeflow, MLFlow, Keycloak (for security purposes) and MinIO.

Cloud providers offer their own commercial solutions to these challenges, including Amazon SageMaker, Azure ML, and IBM Watson Studio, among others, as mentioned in [2].

The main contribution of this work will be the development of a deployment component that provides a plug-and-play solution to this challenge. Additionally, the word "Smart" in the name of the component refers to the optimization of the process based on inputs received from the CO2 monitoring service. It will be configurable, allowing users/customers to decide what and where to deploy the Machine Learning tools available to them within the FAME marketplace.

3.3.3 Technical Specification

3.3.3.1 Component-level C4 Architecture

This component is integrated within the FAME architecture to provide customers with a user-friendly manner to deploy the models and services developed in WP5. The customer will be able to decide which components to purchase and the SD will act accordingly.

A high-level schema is shown in Figure 5. The SD component (in the center of the figure) is an asset that provides all the capabilities described at the beginning of this section. In order to implement all of them, this module must interact with other parts of the system. As shown, the SD component connects with the rest of the FAME modules as follows:

- **Provider**. Responsible for developing the SD component.
- **Model's Repository**. All models will be stored and registered here. The SD component will read the models from this repository.
- **CO2 Analytics**. This will predict the CO2 emissions of ML algorithms. The SD component will use this information as input for deployment.
- **FDAC**. The SD will be indexed in this component, making the asset available to clients.
- **Model Serving**. The SD component will deploy the model on client infrastructure. Note that "infrastructure" here refers to a Kubernetes cluster.

[2] https://airflow.apache.org/
[3] https://www.kubeflow.org/
[4] https://mlflow.org
[5] https://www.jenkins.io/

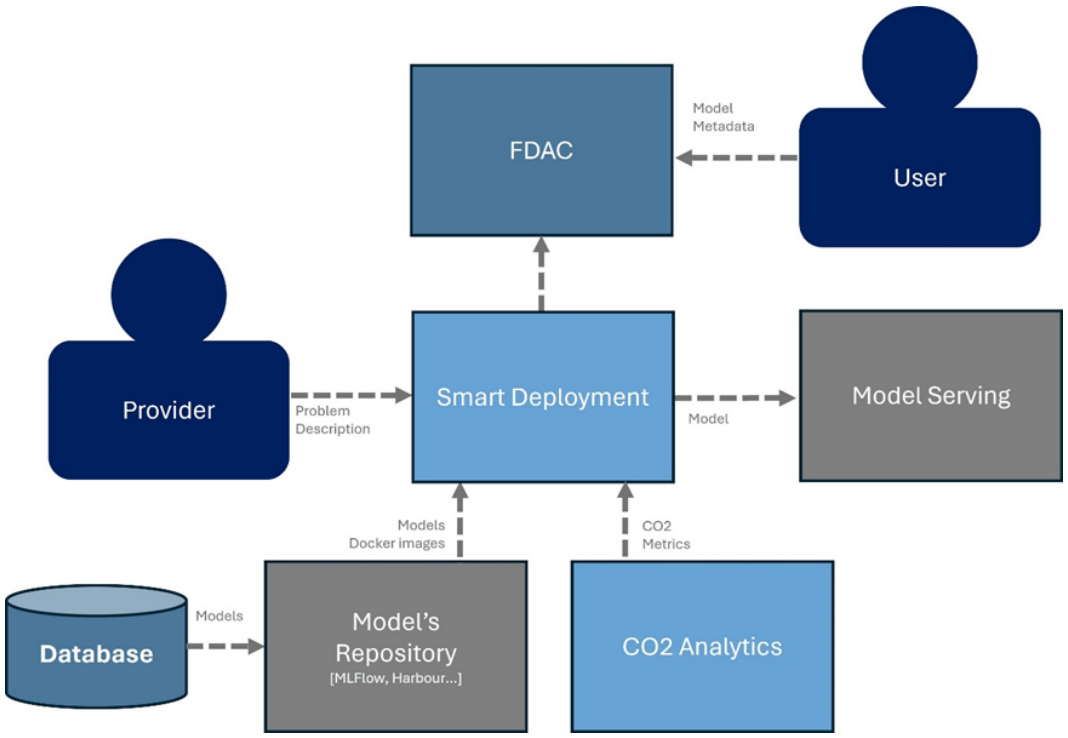

Figure 5: Smart Deployment C4 component diagram

Thus, with an understanding of what the component is and how it works, the following sections provide a detailed look at the technology behind it.

3.3.3.2 Baseline Technologies and Tools

The following table provides an overview of key components and technologies used in the orchestration and deployment of machine learning models. It outlines the specific requirements such as model registry, orchestrator, and REST API, alongside the technologies used such as MLFlow, Jenkins, MLFlow Serving, and KServe. Each entry includes a brief description of the technology's function and its role in managing the machine learning lifecycle.

Table 2: Baseline Technologies and Tools for Smart Deployment

Requirement	Technology	Description
Model Registry	MLFlow	MLFlow is a platform that enables tracking experiments, managing machine learning models, and orchestrating the machine learning lifecycle, from model development to deployment
Orchestrator	Jenkins	Jenkins automates the various stages of software development and delivery, from building and testing code to deploying it across different environments
REST API	MLFlow Serving	MLFlow Serving allows for the deployment and serving of machine learning models, enabling easy access and integration into applications via a REST API

	KServe	KServe enables the serving of machine learning models in a scalable and serverless manner, facilitating easy deployment and management of models in production environments

In Figure 6, the technologies are depicted. The diagram can be split into four areas: Orchestration, Model Registry, Model Serving and Model Monitoring, where the underlying technology is Kubernetes.

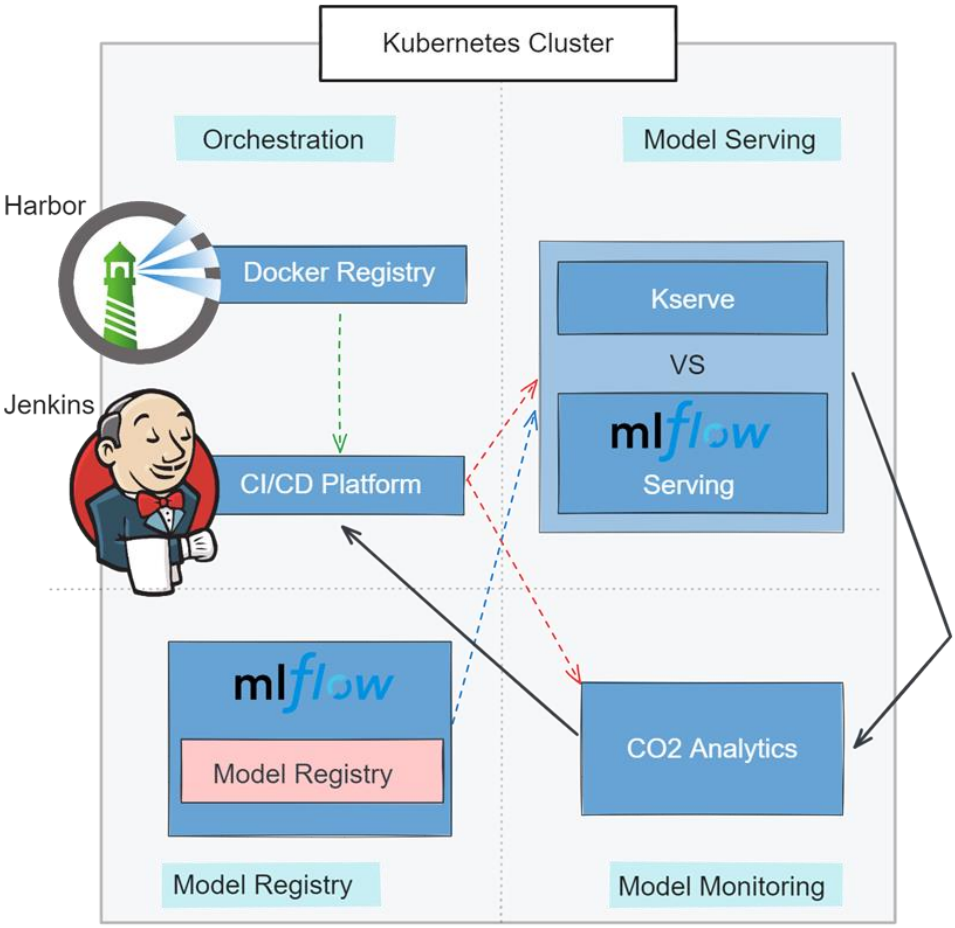

Figure 6 : Smart Deployment architecture

The orchestration includes a Docker Registry (Harbor) and an orchestrator (Jenkins). All baseline Docker images are stored in Harbor. Jenkins will use these baseline images to generate model serving solutions, executing several steps sequentially to achieve the common goal.

The model, which is the principal added value for clients, is retrieved from the Model Registry. This is an MLFlow Server that will contain all ML models offered within the FAME project. Different versions of the models can coexist.

The main outcome of the SD component is the model serving container, ready to accept requests from users/customers. As shown above in Figure 6, at this point in the project, we are exploring technologies to achieve this goal. As an MLFlow Server is already deployed, the MLFlow Serving feature will be analyzed. Alternatively, KServe, a native KubeFlow component, can be used. A comparison between the two technologies will be conducted to determine the best fit for the proposed solution. Additional technologies may also be included in the study.

In addition to the ML model, several components developed in WP5 will also be available for deployment through SD. For example, CO2 Analytics is an eligible component that monitors energy emissions. This information can serve as an input to SD to adapt the deployment based on target infrastructure resources.

3.3.4 Interfaces

As demonstrated in Figure 5, the Smart Deployment component has several connections to other components of the system. It will require connection to the MLFlow Server to download models, connection to the Harbor repository in order to download and execute docker images, and connection to the target infrastructure. Because of the stage of the project, it is not possible to provide specific details about these interfaces.

3.4 FML Orchestrator

3.4.1 Description

The Federated Machine Learning (FML) Orchestrator component brings to the FAME project the necessary tools to enable customers to train purchased models in a federated manner using data hosted on different servers.

To properly orchestrate this process, it is necessary to establish communication between customers and providers so that the parameters of the AI assets are shared, as shown in Figure 7. This requires customers to download the FML Client asset and install it on the machines where the data resides. On the other hand, providers need to set up a FML Server to manage the AI asset aggregation and broadcasting processes.

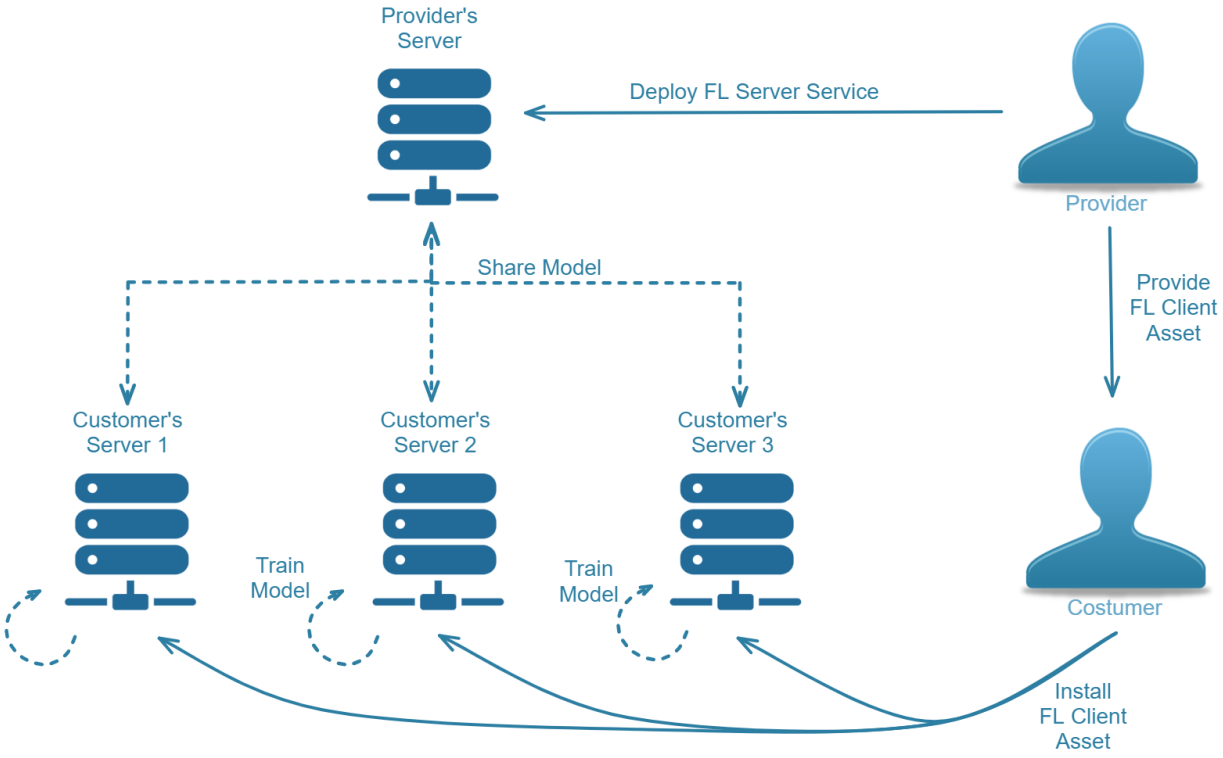

Figure 7: FLM orchestrator overall example.

3.4.2 Related Work

The constant growth of the financial industry in terms of data provides new opportunities to develop and improve the industry [10]. However, obtaining and using this data is challenging for the following reasons:

1. With the growing importance of data privacy around the world, it is becoming increasingly difficult to comply with all legal requirements to obtain and retain financial data [11].
2. Due to customer anonymity policies, the financial industry avoids sharing customer data, resulting in "data silos" [12].

Federated Learning is an effective solution to these problems, meeting existing regulatory requirements and ensuring the anonymity of customer data [13] [14], enabling collaboration across departments, institutions, or industries.

The federated training concept was first introduced by Google in 2016 [15] as a distributed machine learning framework. Here, multiple clients collaboratively train a model under the coordination of a central server, with the training data kept locally, without being uploaded to a data centre. Thus, and unlike general machine learning models, the confidentiality of the data and of the clients to which it belongs is guaranteed by default [16]. However, depending on the heterogeneity of the data hosted on the different clients, 3 different types of federated learning scenarios are considered, also show in Figure 8:

1. **Horizontal Federated Learning [17]:** This is a system where all parties share the same feature space. However, their sample space may be significantly different. Such parties can learn a model collaboratively through a server. Each party computes a training gradient locally, and all parties send their individual gradients to the server. The server aggregates them and sends the aggregated result to all parties. The parties update their model with the aggregated gradient and in this way all parties share the final model parameters. An example of horizontal federated learning is a set of banks that are in the same city or region and therefore share the same set of attributes. Horizontal federated learning can be used to learn a common model by agglomerating models learnt in individual banks. Horizontal federated learning can be implemented with different machine learning algorithms without changing the main framework.
2. **Vertical Federated Learning [17] [18]:** In vertical federated learning, participating parties do not expose users that do not overlap with other parties. Overlapping users are found by using encryption-based user ID matching. As each party has different attributes corresponding to the common users, vertical federated learning aggregates the different attributes from each of the parties and computes the training loss and gradients. The computed gradients are then used to train the model. An example of vertical federated learning is the case of cooperation between online retailers and insurance companies. They have their own feature space (and labels). However, they have a significant number of common users. In such cases, vertical federated learning exploits such situation by federating the features to create a larger feature space for machine learning tasks.
3. **Federated Transfer Learning [19]:** Federated transfer learning is a special case of federated learning and differs from both horizontal and vertical federated learning. In federated transfer learning, two data sets differ in the feature space. This applies to datasets collected from organisations of different but similar nature. Due to the differences in the nature of the business, such companies have only a small overlap in the feature space. Thus, in such scenarios, the datasets differ both in the sample and in the feature space. Transfer learning techniques aim to build an effective model for the target domain while leveraging

knowledge from the other (source) domains. Federated transfer learning takes a model trained on samples and feature space from the source domain. Subsequently, federated transfer learning orientates the model for reuse in the target space, so that the model is used for non-overlapping samples, using the knowledge gained from the non-overlapping features of the source domain.

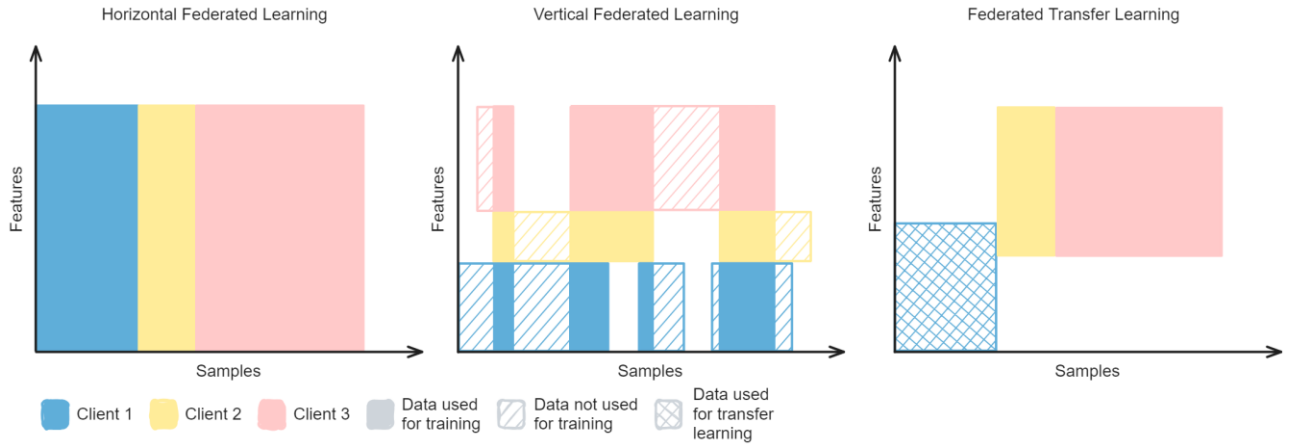

Figure 8: Leveraging customer-owned data for different FL scenarios.

The *FML orchestrator* will provide the FAME project with the means to implement the abovementioned federated learning schemes whenever they fit the use cases proposed by pilot partners. The proposed solution will use the storage technology provided by the *incremental analytics* component in order to enable efficient and rapid exchange of model parameters.

3.4.3 Technical Specification

3.4.3.1 Component-level C4 Architecture

Following recent technical discussions on the approach to be taken by WP5 as a whole, and in view of the new change in leadership at T5.5 regarding the FML Orchestrator component, this task has been restarted recently and the component has been changed from that shown in Section 7.3.15 of D2.2 - Technical Specifications and Platform Architecture.

Figure 9 shows the new C4 architecture diagram proposal. It consists of three main modules that allow customers to train an AI asset using data hosted in different locations (e.g. servers, machines, data spaces...).

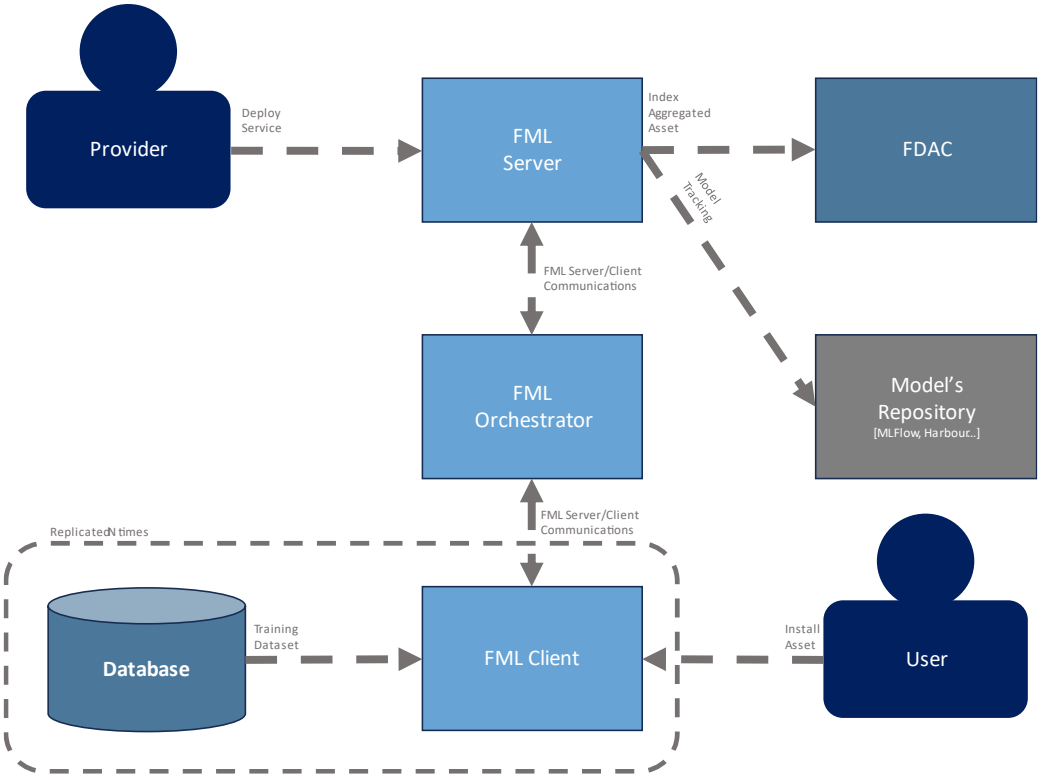

Figure 9: FML Orchestrator C4 component diagram.

FML Server module

This block manages all the functions necessary to carry out federated training. Specifically, the FML Server block implements the following functionalities:

1. define the AI asset to be used.
2. broadcast it to all participating FML clients.
3. gather the local training results from each of them.
4. aggregate all models into a global model.
5. distribute them to the FML clients.

An important feature is that once the FML training is complete, the resulting model is indexed in the FDAC and then stored in the model repository.

FML Client module

This module is designed to make transparent to the customer all the logic required to train models in a federated manner. The main functionalities implemented in this block are:

1. receiving the model parameters from the FML server.
2. training the model with the locally stored data.
3. sending the trained model back to the FML server.

It is important to note that a federated learning process requires at least 2 FML clients to train the models in this way, exploiting all the advantages of this technique (e.g. no sharing of local data).

FML Orchestrator module

This is a simple module responsible for implementing the necessary technologies to enable a communication channel (in particular, for sharing the parameters of the AI asset), as well as implementing an orchestrator that synchronises the different steps to be performed in the exchange

of information between the FML server and the FML clients. This ensures that no component remains idle for any longer than required.

3.4.3.2 Baseline Technologies and Tools

The technologies and/or tools to be used for the implementation of the FML Orchestrator component have not yet been investigated, as this task has recently been taken over by ATOS and the development process has been restarted.

3.4.4 Interfaces

The interfaces required for the correct behaviour of the FML Orchestrator component have not yet been defined, as this task has recently been taken over by ATOS and the development process has been restarted.

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 Figure 3, the *incremental analytics* component consists of several sub-components that interact with each other internally in order to provide the overall functionality. All of them can be considered as different processes that can be containerized and deployed separately in different data servers, while each one of those can scale out independently. That is, we might have one instance of the *transaction manager*, 1 instance of the *relational query engine*, and as much number *data nodes* as required to serve a big data set. However, this is transparent from the user's point of view as the internal installation process that is triggered upon the initialization of the overall deployment of the component handles these details. As a result, we should consider the *Incremental Analytics* as a single box. In the following subsections we will provide more details about prerequisites and installation guidelines.

4.1.1 Prerequisites and Installation Environment

There are several ways that someone could install the overall environment to run the *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 component 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. On the contrary, at this phase of the project, this component is an integral part of the FAME integrated solution, and we can only consider it being deployed within FAME's environment over a cluster having Kubernetes deployment orchestrator. As a result, the prerequisites are the same as of any other component within FAME. Even if at a later phase we could consider the possibility to offer this functionality as-a-Service through the FAME marketplace, the requirements will be the same, as we only envision a Kubernetes installation.

4.1.2 Installation Guide

The first step towards the installation of the *Incremental Analytics* component is to create a docker image for our solution. Given that a user can have access to the binaries 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

RUN apt-get update \
 && apt-get -y install screen iputils-ping netcat \
    vim vsftpd net-tools python3 python3-dev jq \
    python3-pip libssl-dev silversearcher-ag bc \
    libc6-dbg gdb openssh-server lsof psmisc binutils \
    virtualenv telnet apt-transport-https \
    openjdk-11-jdk iproute2 curl gcc python-dev \
    libffi-dev rustc locales git build-essential cargo \
 && apt-get clean \
 && curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py \
 && python3 get-pip.py \
 && rm get-pip.py \
 && pip3 install pyjolokia flask psutil
```

```
#add environment variables for LXS
ENV BASEDIR="/lx" \
    LX_HOME="/lx" \
    LX_DIST="/lx_dist" \
    LX_BIN="${BASEDIR}/LX-BIN" \
    LX_DATA="${BASEDIR}/LX-DATA" \
    ADMIN_DIR="${BASEDIR}/admin" \
    KIVI_HOME="${LX_BIN}/exe" \
    LX_LOGS="${LX_DATA}/logs" \
    IS_RUNNING_ON_DOCKER_CONTAINER="true"

COPY conf/inventory /tmp/inventory

##<--------------------here we need to grab it from the artifactory
RUN cd ${LX_DIST} \
 && curl --header 'Private-Token:XXXXXXXXXX' -L -O https://gitlab.infinitech-
h2020.eu/api/v4/projects/33/packages/maven/eu/fame/leanxcaledb-bins/1.9.13/leanxcaledb-bins-
1.9.13.tgz

RUN groupadd -r appuser \
  && cd ${LX_DIST} \
  && tar vfzx leanxcaledb-bins-1.9.13.tgz && rm leanxcaledb-bins-1.9.13.tgz \
  && mv /tmp/inventory ${LX_DIST}/conf \
  && echo "source $BASEDIR/env.sh" >> /lx/.bashrc

COPY --chown=appuser:appuser bin/* ${BASEDIR}/LX-BIN/scripts/
USER appuser

WORKDIR ${BASEDIR}
USER appuser
CMD cd ${LX_DIST} \
  && ./postscript.sh
```

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 11, python 3, etc.) and creating the corresponding environment variables, the whole distribution package is being copied from a remote repository and finally installed into the image. For the purposes of the FAME project, we have used the internal maven artifactory to upload the while binary distribution of our implementation (marked with bold). 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 common artifactory and the CI pipeline is triggered, using the aforementioned *dockerfile* to create the image that will be now available for the FAME integrated solution.

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. First, we need to define the network resource to enable access from different components or ML/AI analytical processes. The following code snippet depicts this:

```
apiVersion: v1
kind: Service
metadata:
  name: leanxcaledb-service
  labels:
    app: leanxcaledb
spec:
```

```
  ports:
    - name: "2181"
      port: 2181
      targetPort: 2181
    - name: "1529"
      port: 1529
      targetPort: 1529
    - name: "9876"
      port: 9876
      targetPort: 9876
    - name: "9992"
      port: 9992
      targetPort: 9992
    - name: "14400"
      port: 14400
      targetPort: 14400
    - name: "9800"
      port: 9800
      targetPort: 9800
  selector:
    app: leanxcaledb
```

We enable access to several ports for debugging purposes, so that we can connect to all internal components of our implementation. In a production environment, we should only enable the access to the *Relational Query Engine* (port 1529) and to *Kivi Ingest Direct* (configurable port) which are the two components that interact with external software systems, according to the 3rd level view of the C4 Architecture.

Having set up the network, we need to define a persistent storage to be later mounted to our running container. This is crucial, as the *Incremental Analytics* component serves as a datastore and as a result, it needs to persistently store the data that the analytic query processing will be applied upon. The following code snippet provides an example on how to define this:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: leanxcaledb-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 10Gi
```

Here, we make use of the available storage class *ebs-sc* and we requested 10GBs of storage for our component. It is important to highlight that the storage class name might be subject to change in cases of remote deployments, as it needs to match the name of the available *persistent storages* in the target cluster environment. The size of the storage can be increased according to the requirements for data volume.

Having defined both the network and the persistent storage, we now need to define a *stateful set* that will host our running container, as the following code snippet indicates:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: leanxcaledb
  labels:
    app: leanxcaledb
spec:
```

```
serviceName: leanxcaledb-service
replicas: 1
selector:
  matchLabels:
    app: leanxcaledb
updateStrategy:
  type: RollingUpdate
podManagementPolicy: OrderedReady
  spec:
      volumeMounts:
        - name: leanxcaledb-storage
          mountPath: /lx/LX-DATA
    containers:
      - image: harbor.infinitech-h2020.eu/fame/leanxcaledb:latest
        name: leanxcaledb
        ports:
          - containerPort: 2181
          - containerPort: 1529
          - containerPort: 9876
          - containerPort: 9992
          - containerPort: 14400
          - containerPort: 9800
    restartPolicy: Always
    imagePullSecrets:
      - name: registrysecret
    volumes:
      - name: leanxcaledb-storage
        persistentVolumeClaim:
          claimName: leanxcaledb-pvc
```

The important elements that need to be taken under consideration are the following:

- Image: the name of the image that has been created during the CI process
- Image Secret: the *secret* that has been defined to allow Kubernetes to pull the image from the private docker registry (we make use of the Harbor in FAME)
- Service name: the name of the service resource that was defined earlier
- Volume: the name of the persistent volume claim that was defined earlier
- Volume Mount: we need to mount this new volume to the /lx/LX-DATA directory

Similarly to the creation of the docker image, for the deployment of our component we make use of the CD pipelines provided by the FAME project. Once a new development cycle has been finalized and we have merged the outcomes to the master distribution, and after the completion of the CI pipeline, then this CD pipeline is triggered that creates the prerequisite resources and deploys the overall solution to the FAME's cluster.

4.1.3 User Guide

As the novel enhancements of this component related with the i) secondary covered indexes and the ii) online aggregations described in the previous section are currently under validation, the SQL dialect needed is continuously updated, taken as input the recommendations by the pilot experimentation. More details and a complete user guide will be provided in the next version of this deliverable.

4.2 Analytics CO2 Monitoring

The initial experiments for Analytics CO2 Monitoring have been performed with available open data (specifically, on a dataset from the healthcare domain).

The following figures present examples of collected CO2 emissions data with different experimental setups (Random Forest Regression, Decision Trees, SVM, Linear Regression et al.)

At the current stage, the component gathers the CO2 emissions data from the training perspective - the future developments include CO2 monitoring analytics model development and usage.

The project involves working with large amounts of data that can be analyzed using various machine learning models. Our system can recommend the model with the lowest CO2 emissions for current datasets.

index	id	project_name	experiment_descriptor	duration(s)	power_consumption(kWh)	CPU	TDP	GPU	OS	Country_code	cost	model_type	num_estimators	kernel_type	dataset_size	CO2_Emission(kg)
0	22e496da-	CO2_Emmission	Random Fore	5.81131315	4.50E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Random For	20	None	101	**1.11E-06**
1	9a5ffc82-1	CO2_Emmission	Decision Tree	5.80678988	4.50E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Decision Tre	None	None	101	**1.11E-06**
2	2114b1d1	CO2_Emmission	Support Vecto	5.87084103	4.70E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	rbf	101	**1.15E-06**
3	986c0d59-	CO2_Emmission	Support Vecto	5.79380679	4.36E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	linear	101	**1.07E-06**
4	a69e8b95-	CO2_Emmission	Support Vecto	5.84428263	4.67E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	poly	101	**1.15E-06**
5	6b8091fc-	CO2_Emmission	Support Vecto	5.86776376	4.64E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	rbf	101	**1.14E-06**
6	00fead0a-	CO2_Emmission	Support Vecto	5.82303333	4.60E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	sigmoid	101	**1.13E-06**
7	e4f341ab-	CO2_Emmission	NN Regressio	7.20167232	6.97E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Neural Netw	None	None	101	**1.71E-06**
8	1667ce41-	CO2_Emmission	Linear Regres	5.70837569	4.44E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Linear Regre	None	None	101	**1.09E-06**
9	b3b21aa5	CO2_Emmission	Random Fore	5.79616284	4.64E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Random For	20	None	101	**1.14E-06**
10	6a032c64-	CO2_Emmission	Decision Tree	5.68434215	4.38E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Decision Tre	None	None	101	**1.08E-06**
11	cfc9b1d0-	CO2_Emmission	Support Vecto	5.81925917	4.68E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	rbf	101	**1.15E-06**
12	d0f9134f-(	CO2_Emmission	Support Vecto	5.76422858	4.55E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	linear	101	**1.12E-06**
13	6f4d3b24-	CO2_Emmission	Support Vecto	5.81303239	4.60E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	poly	101	**1.13E-06**
14	b0f81608-	CO2_Emmission	Support Vecto	5.74993801	4.59E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	rbf	101	**1.13E-06**
15	c332ce33-	CO2_Emmission	Support Vecto	5.76971745	4.51E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Support Vect	None	sigmoid	101	**1.11E-06**
16	c4383125-	CO2_Emmission	NN Regressio	7.02840352	6.57E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Neural Netw	None	None	101	**1.61E-06**
17	d28ff448-(	CO2_Emmission	Linear Regres	5.7485342	4.55E-06	AMD Ryze	TDP:45.0	NVIDIA Ge	Windows	SI	0	Linear Regre	None	None	101	**1.12E-06**

Figure 10: Example of collected CO2 emissions data (training AI models on numerical data)

id	project_name	experiment_description	epoch	start_time	duration(s)	power_consumption (kWh)	CPU_name	GPU_name	OS	region/country	cost	CO2_emissions(kg)
23cbe1e6-	CO2_Emmission	NN_text_Em_Flat_Cat		05-01-24 4:25	225.09673	0.0006113	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**0.00015019**
d196b9ba	CO2_Emmission	RFC_100		05-01-24 4:35	93.209903	0.0001819	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**4.47E-05**
707fb674-	CO2_Emmission	RFC_100		05-01-24 4:37	33.237058	5.82E-05	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**1.43E-05**
44456b45-	CO2_Emmission	RFC_100		05-01-24 4:38	113.32789	0.0002229	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**5.48E-05**
15b29afc-	CO2_Emmission	RFC_100		05-01-24 4:40	20.867061	3.04E-05	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**7.48E-06**
bc405c80-	CO2_Emmission	NN_text_Em_Flat_Cat	N/A	06-02-24 12:02	104.77869	0.0001219	AMD Ryzen 7	NVIDIA GeF	Windows	SI	0	**3.00E-05**

Figure 11: Example of collected CO2 emissions data (training AI models on text data)

4.2.1 Prerequisites and Installation Environment

The development is done in Python programming language, the required libraries/additional technologies include:

- pandas: is a software library written for the Python programming language for data manipulation and analysis;
- sklearn: is a free software machine learning library for the Python programming language;
- numpy: is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- matplotlib: is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits.
- eco2ai: is a python library which accumulates statistics about power consumption and CO2 emission during running code.
- tensorflow: is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.
- keras: is a powerful and easy-to-use free open-source Python library for developing and evaluating deep learning models.

At the end of the experiments our software will be packed in a Docker container and will not require installation of third-party libraries.

4.2.2 Installation Guide

As this is currently work in progress, the installation guide is yet to be defined in the next version of this deliverable.

4.2.3 User Guide

As this is currently a work in progress, the complete user guide is yet to be defined in the next version of this deliverable.

4.3 Smart Deployment

As it is shown in a previous section, Smart Deployment (SD) component is focused on offering a user-friendly and powerful solution to deploy rest of WP5 modules. From a technical point of view, it is based on MLFlow Server (model registry), Jenkins (pipelines) and KServe or MLFlow Serving (model serving). In this demonstration, a complete guide to deploy MLFlow Server in Kubernetes is explained.

4.3.1 Prerequisites and Installation Environment

The SD will be able to deploy components in Kubernetes. It is the chosen baseline technology because it efficiently manages and scales containerized applications, ensuring high availability, automated deployment, and resource optimization, making it essential for modern, dynamic, and scalable application infrastructure.

For this demonstration, the underlying cluster has the following features:

- Kubernetes version is 1.26.13.
- It is a four-node cluster:
    - Master node. This node has three roles: controlplane, etcd and worker. Resources: 12 virtual CPUs, 12 Gb of RAM and SSD of 150 Gb.
    - Worker-1 node. It acts as a worker. Resources: 32 virtual CPUs, 48 Gb of RAM and SSD of 150 Gb.
    - Worker-2 node. It acts as a worker. Resources: 32 virtual CPUs, 50 Gb of RAM and SSD of 150 Gb.
    - NFS node. It is the storage system. Resources: 10 virtual CPUs, 10 Gb of RAM and SSD of 1.15 Tb.

The cluster is set up using Rancher. It is an open-source container management platform that simplifies Kubernetes cluster deployment across any infrastructure. It offers an intuitive UI, making cluster setup straightforward.

4.3.2 Installation Guide

MLflow requires both a structured database for metadata storage and an artifact storage to manage machine learning experiments and model' registry efficiently. The structured database tracks experiment details, parameters, and metrics, while the artifact storage saves models, data, and other files. It can be integrated with SQLite, SQL Server, PostgresQL, AWS S3, MinIO and Azure Blob Storage, among others.

The deployment will be done using Helm. It is a package manager for Kubernetes that allows developers and operators to easily package, configure, and deploy applications and services onto Kubernetes clusters. It uses a packaging format called charts to describe a set of Kubernetes

resources, simplifying the management and deployment of complex applications. The Bitnami Helm repository is a collection of pre-packaged Helm charts provided by Bitnami, which simplify the deployment of popular open-source software applications and development stacks on Kubernetes. These charts are tested, maintained, and regularly updated, ensuring an easy and reliable way to deploy applications across any Kubernetes environment.

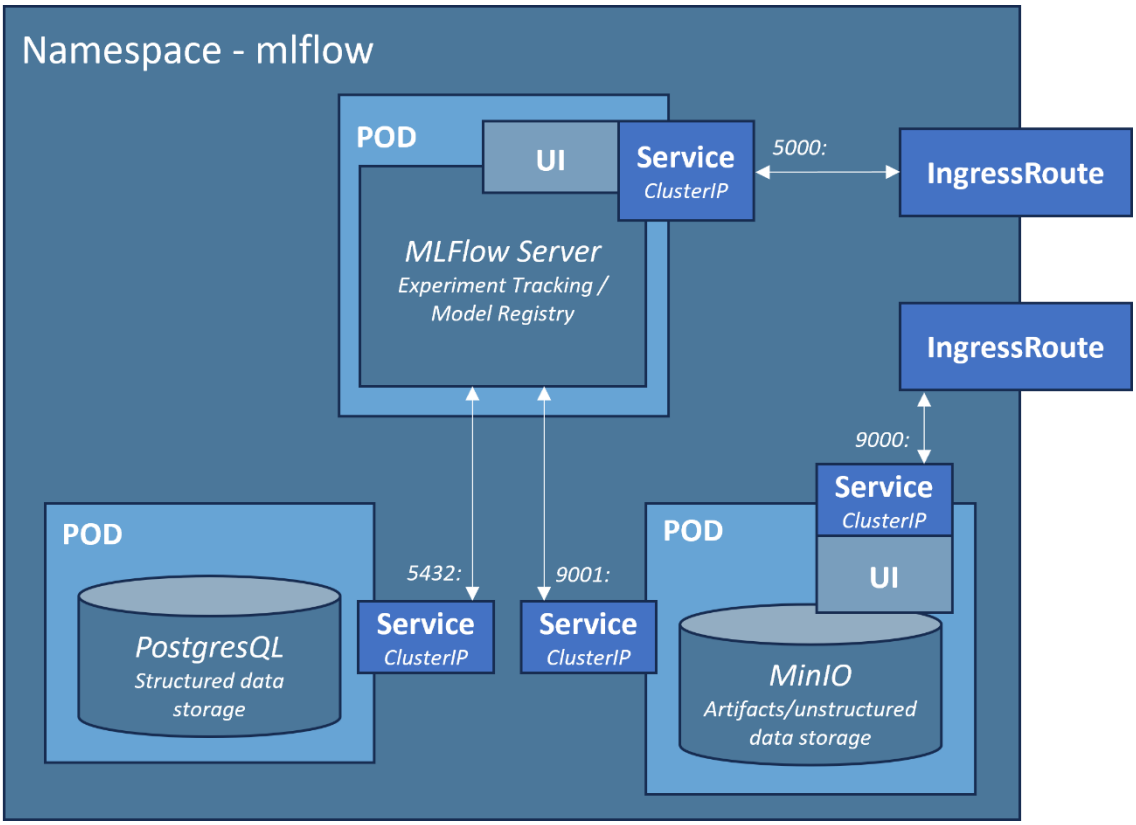

Figure 12: Smart Deployment Kubernetes components

In Figure 12, the architecture of the solution is presented. As shown, PostgreSQL and MinIO will serve as the backend for MLFlow. Three pods and corresponding services are deployed to ensure communication. To reproduce the steps, ensure that the machine can connect to the cluster. Once this is achieved, follow the steps below:

1. MinIO. First, add the Bitnami repository to our local Helm tool. Then, print the chart content to a file so it can be adapted to the target infrastructure. Override the rootPassword field with “minio-password” and defaultBuckets with “mlflow”. Finally, install MinIO using the Helm install command.

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm show values bitnami/minio > minio.yaml
helm install minio bitnami/minio -n mlflow --create-namespace --values minio.yaml
```

Figure 13: MinIO Helm commands

2. PostgreSQL. Follow the same approach to install this tool. The Helm repository has already been added, so there's no need to do it again. Override auth.username with “postgres-user”, auth.password with “postgres-password”, and auth.database with “mlflow”.

```
helm show values bitnami/postgresql > postgresql.yaml
helm install postgresql bitnami/postgresql -n mlflow --values postgresql.yaml
```

Figure 14: PostgreSQL Helm commands

3. MLFlow Server. This will be installed manually. Generate YAML files and deploy them directly with *kubectl apply* commands as follows:
   a. Generate a Secret containing PostgreSQL and MinIO credentials, then create it in the "mlflow" namespace.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mlflow-secret
  namespace: mlflow
  labels:
    app: mlflow-deployment
type: Opaque
stringData:
  MLFLOW_S3_ENDPOINT_URL: http://api.mlflow.svc.cluster.local:9000
  AWS_ACCESS_KEY_ID: admin
  AWS_SECRET_ACCESS_KEY: minio-password
  POSTGRES_USER: postgres-user
  POSTGRES_PASSWORD: postgres-password
```

Figure 15: MLFlow Server secret

   b. Generate and deploy the MLFlow Server and the corresponding service. It will be configured based on credentials defined in the secret.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mlflow-deployment
  namespace: mlflow
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mlflow-deployment
  template:
    metadata:
      labels:
        app: mlflow-deployment
    spec:
      containers:
      - name: mlflow-deployment
        image: ${MLFLOW_DOCKER_IMAGE_TAG}
        imagePullPolicy: Always
        command: ["/bin/bash"]
        args: ["-c", "pip install psycopg2-binary; pip install boto3; sed -i 's/=
password/= ${MLFLOW_ADMIN_PASSWORD}/g' /usr/local/lib/python3.10/site-
packages/mlflow/server/auth/basic_auth.ini; mlflow server --app-name basic-auth --host
0.0.0.0 --port 5000 --backend-store-uri
postgresql://${MLFLOW_POSTGRES_USER}:${MLFLOW_POSTGRES_PASSWORD}@postgres-
service.mlflow.svc.cluster.local:5432/mlflow --artifacts-destination s3://mlflow/ --
workers 2"]
        envFrom:
            - secretRef:
                name: mlflow-secret
        ports:
        - name: http
          containerPort: 5000
          protocol: TCP
```

Figure 16: MLFlow Server deployment

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mlflow-service
  namespace: mlflow
spec:
  selector:
    app: mlflow-deployment
  type: ClusterIP
  ports:
    - port: 5000
      targetPort: 5000
      protocol: TCP
      name: http
```

Figure 17: MLFlow Server service

At this point, the MLFlow Server is up and running. Note that no Ingress Routes are deployed, as it is beyond the scope of this demonstration. To access the MinIO UI and MLFlow Server UI, port forwarding will be used.

4.3.3 User Guide

In Figure 18, MLFlow' user interface is presented. As it is shown, we access to it through port forwarding. To be more specific, port 5000 of the pod is mapped to port 5000 of the local host. By default, experiment tracking tab is shown. It creates a Default experiment and corresponding folders in MinIO to save artifacts. It is a new deployment, so no experiments are logged yet.

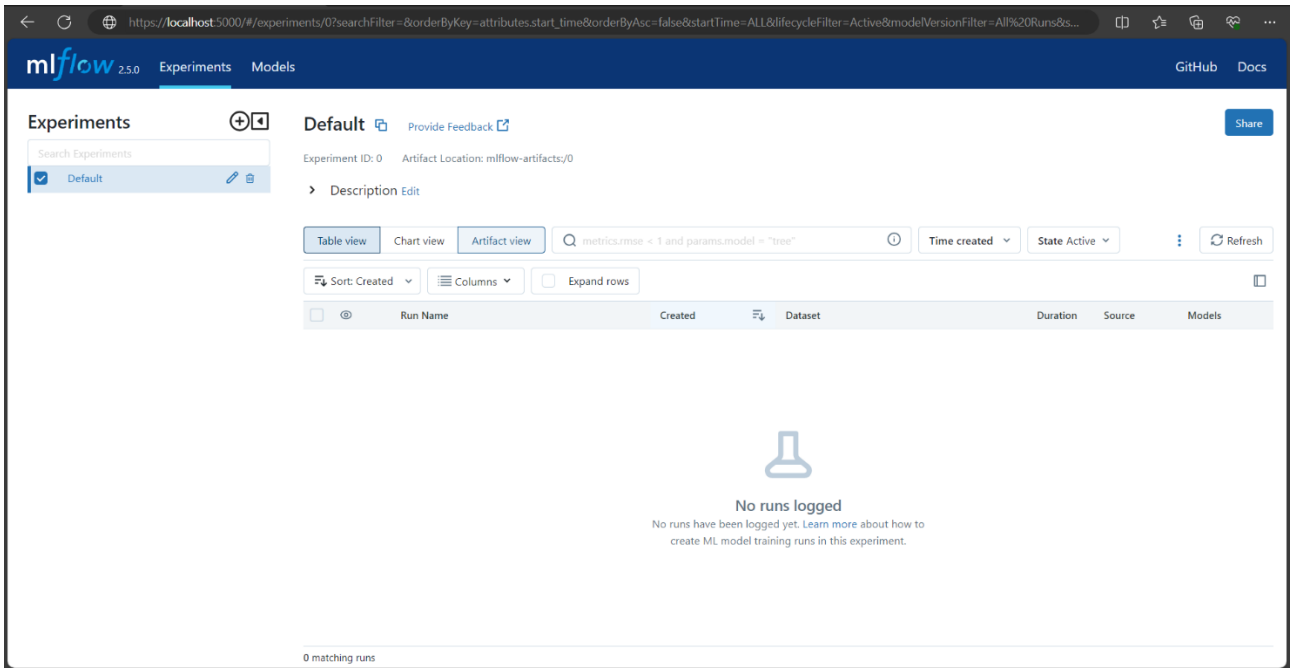

Figure 18: MLFlow Server UI screenshot

The main capability that will be used from MLFlow to implement Smart Deployment component is the model registry (Figure 19). It has its own tab in the user interface. Registered models will be shown here. Additionally, it provides a centralized hub for managing the lifecycle of ML models, including their versioning, stage transitions (such as from staging to production), and annotations.

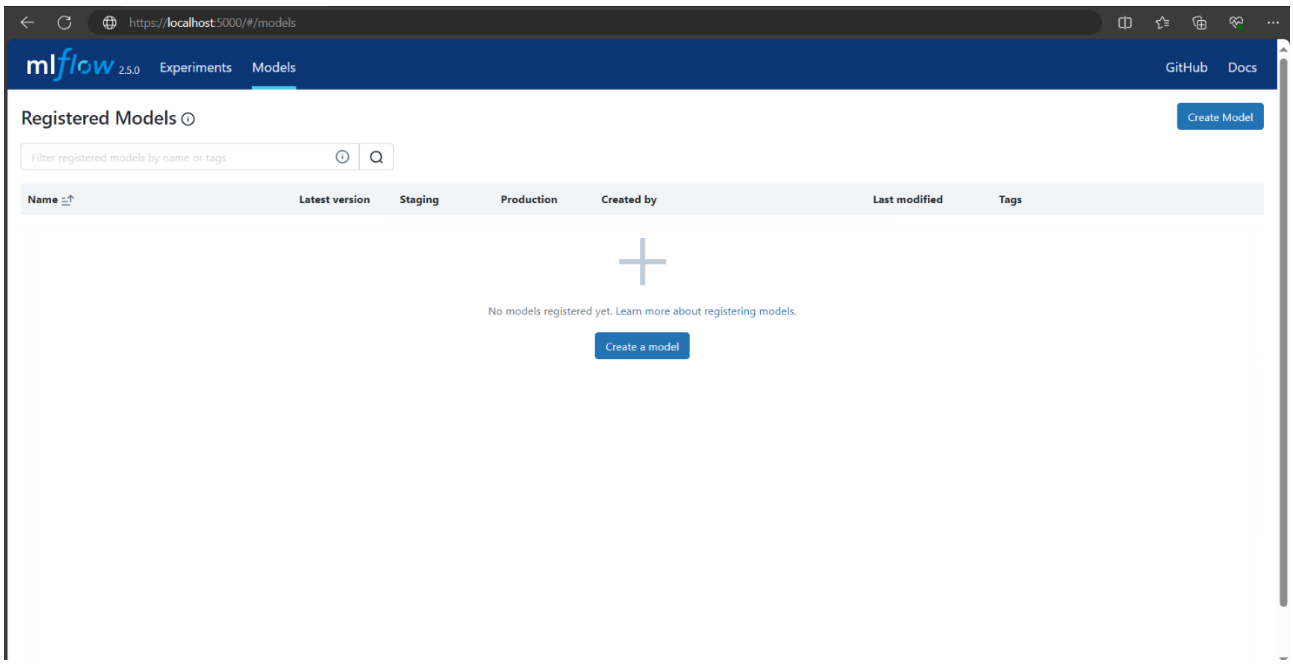

Figure 19: MLFlow Server UI screenshot

At this point, the MLFlow Server has been deployed and is ready for use. In this demonstration, we have explored the deployment of the MLFlow Server and its user interface, focusing on its model registry capabilities. These capabilities are essential for managing the lifecycle of ML models, including versioning and stage transitions, thereby facilitating efficient and scalable model deployment solutions.

5 Conclusions

This deliverable summarized the work that has been currently carried out at this 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 Orchestrator* 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, followed by the related work that has been currently carried out and the description of the baseline technologies that the components relied on. It also included a detailed technical description, according to the level of maturity of each component, accompanied by the 3rd level C4 architecture, 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, while a separate section demonstrated the installation, deployment and use of the technology.

As this is the first version of this deliverable, it included preliminary design and implementations of the respected technological outcomes. At the time when this document was written, these outcomes are being verified by the use case scenarios identified by the pilot partners of the consortium. As a result, updates and finalization of this first version of prototypes, along with concrete demonstrations will be included in the next version of the document. This last version will additionally include the prototype evaluation with regards to the former KPIs that the *Energy Efficient Analytics Toolbox* is concerned.

References

[1]. S. L. V. Z. N. e. a. Budennyy, «eco2AI: Carbon Emissions Tracking of Machine Learning Models as the First Step Towards Sustainable AI,» ADVANCED STUDIES IN ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING, vol. 106, pp. 118-128, 2023.

[2]. «Codecarbon,» [En línea]. Available: https://github.com/mlco2/codecarbon.

[3]. «cloud-carbon-footprint,» [En línea]. Available: https://github.com/cloud-carbon-footprint/cloud-carbon-footprint.

[4]. «Carbontracker,» [En línea]. Available: https://github.com/lfwa/carbontracker.

[5]. D. e. a. SCULLEY, «Hidden technical debt in machine learning systems,» Advances in neural information processing systems, vol. 28, 2015.

[6]. D. KREUZBERGER, N. KÜHL y S. HIRSCHL, «Machine learning operations (mlops): Overview, definition, and architecture,» IEEE Access, 2023.

[7]. K.-M. RATILAINEN, «Adopting Machine Learning Pipeline in Existing Environment,» 2023.

[8]. H. HAPKE y C. NELSON, «Building machine learning pipelines,» O'Reilly Media, 2020.

[9]. A. C. COB-PARRO, Y. LALANGUI y R. LAZCANO, «Fostering Agricultural Transformation through AI: An Open-Source AI Architecture Exploiting the MLOps Paradigm,» Agronomy, vol. 14, nº 2, 2024.

[10]. T. S. A. K. T. A. &. S. V. Li, «Federated learning: Challenges, methods, and future directions,» IEEE signal processing magazine, pp. 50-60, 2020.

[11]. A. O. P. A. M. G. I. &. K. K. Jolfaei, «Guest editorial special issue on privacy and security in distributed edge computing and evolving IoT,» IEEE Internet of Things Journal, pp. 2496-2500, 2020.

[12]. A. N. I. X. Y. H. G. &. G. S. Mehmood, «Protection of big data privacy,» IEEE access, pp. 1821-1834, 2016.

[13]. C. X. Y. B. H. Y. B. L. W. &. G. Y. Zhang, «A survey on federated learning,» Knowledge-Based Systems, 2021.

[14]. H. S. N. M. H. &. S. N. Miyajima, «Federated learning with divided data for BP,» In Lecture Notes in Engineering and Computer Science: Proceedings of the International MultiConference of Engineers and Computer Scientists, pp. 20-22, 2021.

[15]. J. M. H. B. Y. F. X. R. P. S. A. T. &. B. D. Konečný, «Federated learning: Strategies for improving communication efficiency,» arXiv preprint, 2016.

[16]. B. M. E. R. D. H. S. &. y. A. B. A. McMahan, «Communication-efficient learning of deep networks from decentralized data,» In Artificial intelligence and statistics, pp. 1273-1282, 2017.

[17]. P. M. H. B. A. B. B. A. B. M. B. A. N. .. &. Z. S. Kairouz, «Advances and open problems in federated learning,» Foundations and Trends® in Machine Learning, pp. 1-210, 2021.

[18]. A. t. T. M. &. W. A. Khan, «Communication-efficient vertical federated learning,» Algorithms, 2022.

[19]. S. X. C. L. Y. &. K. Y. Sharma, «Secure and efficient federated transfer learning,» IEEE international conference on big data (Big Data), pp. 2569-2576, 2019.