Personalised Health Monitoring and Decision Support Based on Artificial Intelligence and Holistic Health Records

D4.10 – Big data platform and knowledge management system II

WP4 Knowledge Management and Utilisation in the iHelp Platform

Dissemination Level: Public
Document type: Report
Version: 1.0
Date: October 31, 2022

The project iHelp has received funding from the European Union's Horizon 2020 Programme for research, technological development, and demonstration under grant agreement no 101017441.

Document Details

Project Number	101017441
Project Title	iHelp - Personalised Health Monitoring and Decision Support Based on Artificial Intelligence and Holistic Health Records
Title of deliverable	Big data platform and knowledge management system II
Work package	WP4 Knowledge Management and Utilisation in the iHelp Platform
Due Date	October 31, 2022
Submission Date	October 31, 2022
Start Date of Project	January 1, 2021
Duration of project	36 months
Main Responsible Partner	LXS
Deliverable nature	Report
Authors names	Pavlos Kranas, Javier Pereira, Alejandro Ramiro, Rogelio Rodriguez, Pablos Spencer, Jesus Manuel Gallego (LXS)
Reviewers names	Pencho Stefanov (KOD), Ainhoa Azqueta (UPM)

Document Revision History

Version History			
Version	**Date**	**Author(s)**	**Changes made**
0.1	2022-09-01	Pavlos Kranas (LXS)	Initial ToCs
0.2	2022-09-20	Pavlos Kranas, Javier Pereira, Alejandro Ramiro, Rogelio Rodriguez, Pablos Spencer, Jesus Manuel Gallego (LXS)	Section 4, 5
0.3	2022-09-27	Pavlos Kranas, Javier Pereira, Alejandro Ramiro, Rogelio Rodriguez, Pablos Spencer, Jesus Manuel Gallego (LXS)	Additions in section 6
0.4	2022-09-29	Pavlos Kranas (LXS)	Updates on the plan
0.5	2022-09-30	Pavlos Kranas (LXS)	Updates on intro/conclusions
0.6	2022-09-30	Pavlos Kranas (LXS)	Submission for internal review
0.7	2022-10-20	Pencho Stefanov (KOD)	1st Internal review
0.8	2022-10-22	Ainhoa Azqueta (UPM)	2nd Internal review

0.9	2022-10-25	Pavlos Kranas (LXS)	Revision and updates based on the internal reviews. Quality review
1.0	2022-10-31	Dimosthenis Kyriazis (UPRC)	Final version

Table of Contents

Table of Figures

Executive summary

The iHelp integrated solution aims at providing personalised health monitoring and decision support based on artificial intelligence using datasets coming from a variety of different and heterogeneous sources that will be integrated into a common data model: the Holistic Health Records (HHRs). The integrated solutions consist of various technology building blocks that are related firstly with the data ingestion process that is responsible to capture data from external sources, transform them and store them to the Big Data Platform. Secondly with the data analytics layer that makes use of these data to feed their internal AI algorithms. Finally, with the platform level components that provide the runtime execution environment and the data management activities of the integrated solution. As a result, the last category of building blocks is central to the iHelp platform and interacts with all other components.

This deliverable reports the work that has been recently carried out under the scope of T4.4 – "Big Data Platform and Knowledge Management System", which is responsible for the data management activities of the platform. The outcome of this task, the Big Data Platform of iHelp will be used from i) the data ingestion processes that store data and ii) the data analytics functions that read data. As a result, it firstly needs to allow for data ingestion in very high rates and at the same time, to enable data analytics over the operational data that are being ingested at the same time. Moreover, the Big Data Platform needs to be integrated with various popular processing frameworks that are being used by the iHelp or the analytical functions, like Apache Spark or Apache Kafka Therefore, it provides various means of data connectivity mechanisms. Both the runtime execution environment and the data ingestion pipelines make use of intermediate Kafka queues, while Apache Spark is the popular analytical processing framework used by many developers of analytical tools.

Another important requirement for the Big Data Platform is its ability to combine and aggregate data coming from different data sources. An important requirement however is the need for the data not to be moved from outside the organisation they belong to, due to EU and country level regulations. The Big Data Platform provides support for polyglot query processing, which will be a key factor on the implementation of this requirement.

At this phase of the project, a prototype of the outcome of this task has been provided and made available to project partners. It includes the Big Data Platform itself along with the definition of the HHR relational schema, an enhanced Kafka Broker that is currently being used by the data ingestion pipelines It includes also the datastore connector, and a set of several microservices that have been developed during the second phase of the project. This deliverable includes a separate section that demonstrates the deployment, installation, and use of each of the aforementioned components, giving concrete examples with code snippets that can be used by all partners of the project as development and operational guidelines.

The T4.4 – "Big Data Platform and Knowledge Management System" has three development phases, and this deliverable reports the work that has been carried out until the second phase of the project (M22). Therefore, it also includes a section with next steps and a roadmap for the implementation and prototype delivery of the Big Data Platform until the end of the project. The last version of this report is planned to be delivered in M32.

1 Introduction

The Big Data Platform and Knowledge Management System of iHelp is the central data repository of the overall integrated solution and provides all data management activities that are needed from it. It is based on the LeanXcale's database, which is a relational distributed database with advanced capabilities, bringing a list of interesting innovations that are relevant to the requirements of the project. It is a central building block of the overall integrated solution, as it will be used during the data ingestion process, where data are being collected, assured their quality, transformed to the common data model of iHelp, the HHR, and eventually persistently to be stored by other components for further processing. When stored, the analytical functions hosted by the platform need to efficiently retrieve and process this data, while also store intermediate or final results that will be later visualized by the Clinical DSS Suite and its Visual Analytic Tools. Therefore, the Big Data Platform interacts with the majority of the building blocks offered by the iHelp platform. This introduces a number of challenges that need to be tackled by the Big Data Platform and will be reported in this document.

The common data model of iHelp, the HHR, consist of various *entities* along with their relations. Its conceptual model is given as an E-R (Entity-Relational) diagram, while analytical tools will need to make use of different entities. As a result, a relational data management system is required to store this information and retrieve data efficiently. This introduces a great challenge, as traditional relational database systems, in order to ensure database transactions, often implements a mechanism that is commonly known as the *two phase locking* protocol. It makes use of *shared* and *exclusive locks* over the data items of a data table. Shared locks are being put when a read operation access a data item, allowing other read operations to perform in parallel, but forbids write operations to access shared locked items concurrently. On the other hand, exclusive locks prevent any other operation to access the specific item that has been previously exclusively locked. This happens in order to avoid data access anomalies raised by the phenomena that occur when concurrent transactions try to access the same data items. The result of this protocol is that write transactions block read-only ones, as the latter usually need to perform a full scan over a data table, but they cannot, as many data items are exclusively locked. On the other hand, a last-running read transaction will eventually put shared locks all records of a data table, which will prevent any write operation to perform on that table. As a result, it is evident that read and write transactions are competitive and one block the other. In order to apply data analytics, data engineers often migrate periodically the latest snapshot of the operational data to a data warehouse and use the latter to perform analytics. The drawback of this architecture is that data that feed the analytical algorithms are outdated and analytics cannot rely on fresh or live operational data. This is important in the healthcare domain, where the healthcare professional might need to have the current picture of how a disease is being developing at a current point in time.

To avoid this issue, the Big Data Platform of iHelp makes use of a different approaches: it relies on the implementation of the *snapshot isolation* paradigm, that instead of shared and exclusive locks, it maintains different versions of the data items. Now, each transaction has the visibility of a specific version of each data item, and concurrent transactions can access the same data items, without blocking each other. This is crucial, as it allows the data analysts to execute their AI algorithms on the fresh data, that is the initial requirement. Moreover, due to its scalable transactional manager and the internal data structure used by its storage engine, it allows for data ingestion on very high rates, thus combining the benefits of both words: the SQL and the NoSQL ecosystem.

This section introduces the document. We first provide the main objectives of this deliverable (§1.1). Next, we describe how the work that has been carried out under the scope of T4.4 – "Big Data Platform and Knowledge Management System" is strongly related with the majority of the technical tasks of the iHelp project, and provide insights into the dependencies with all related activities performed under these tasks (§1.2). Finally, the overall structure of this deliverable is given (§1.3), while at the end we provide information about what additional content has been included in this second version of the deliverable.

1.1 Objectives of this deliverable

The objective of this deliverable is to report on the work that has been recently done under the scope of the T4.4 – "Big Data Platform and Knowledge Management System" at this phase of the project (M22). The main focus of this work at this phase was to provide the design principles and capabilities of the Big Data Platform and the design and requirements for the development and implementation of federated scenarios involving different instances of the platform itself. Then, examples of how to deploy and make use of the platform have been provided, along with code snippets highlighting the different data connectivity mechanisms that are provided and allow the integration of the datastore with popular processing frameworks used by the iHelp integrated solution. Additionally, a set of newly developed microservices have been documented, which were needed for the integration of the Big Data Platform with other components of the iHelp solution that are now delivered, in order to implement the remaining of the identified scenarios. Finally, a list of next steps and roadmap for implementation for the next period is described, highlighting the status at this phase and the basic design principles that will guide our work in the last period of the project.

1.2 Insights from other tasks and deliverables

The work that has been carried out under the scope of the T4.4 – "Big Data Platform and Knowledge Management System" has dependencies with the activities of WP2, WP3 and WP5 and other tasks of WP4. In fact, the Big Data Platform is a central building block of the overall integrated solution of iHelp and interacts with the majority of the components in the platform. More specifically, it gets input from T2.1 – "Requirements, State of the Art Analysis and User Scenarios in iHELP" and has dependencies with T2.2 – "Reference Architecture Specifications" and T2.3 – "Functional and Non-Functional Specifications". Then, it has dependencies with WP3, in particular T3.1 – "Data Modelling and Integrated Health Records" as this task defines the common conceptual data model of iHelp, the HHR, which needs to be translated into the corresponding relational schema. T3.2 – "Primary Data Capture and Ingestion" defines the data ingestion pipelines that eventually will interact with the Big Data Platform to store the data, using the Kafka queues. From what concerns WP4, this work has dependencies with the technology provided by T4.2 – "Model Library: Implementation and Recalibration of Adaptive Models" as the latter will provide the runtime execution environment for the analytic tools. It makes use of Kafka queues to interact with the datastore and will make use of the Big Data Platform to store data models provided by T4.1 – "Data Modelling and Integrated Health Records" and intermediate or final results by tools provided by T5.1 – "Techniques for Early Risk Identification, Predictions and Assessment". These results will be further visualized by the tools developed in T4.3 – "Clinical DSS Suite with Visual Analytic Tools". These tools will also need to interact with the Big Data Platform, as they provide UI tools for the data analysts to explore the data stored into the datastore. Finally, it also interacts with various components developed under the scope of WP5 – "AI for Early Risk Assessment and Personalised Recommendations" via the newly developed microservices.

As it is evident, the work that is being carried out under this task and reported in this document is strongly related with the majority of the technical tasks of the project.

1.3 Structure

This document is structured as follows: Section 2 gives the overview of the Big Data Platform focusing on its internal architecture, its parallel OLAP engine and how it is being integrated with the overall solution of the iHelp platform. Section 3 gives insights about its polyglot query processing engine andhow it works. The polyglot capabilities will be used when a data user will need to combine data coming from different sources that cannot be moved outside the premises of an organization. It will allow the data analyst to make use of federated queries in order to feed his or her AI algorithms with aggregated information. The newly added Section 4 describes the HHR relational schema of the datastore, while Section 5 documents the newly implemented microservices that supports various scenarios and interacts with other components of the overall iHelp solution. Section 6 has been now extended and demonstrates the installation, deployment and use of the Big Data Platform, along with the additional microservices and the enhanced Kafka broker. Section 7 has been updated in order to describe the next steps and the roadmap for implementation for the final phase. Finally, Section 8 concludes the document.

1.4 Changes from previous version

In this second version of this deliverable, we introduced Section 4 that defines the current HHR relational model. We also added Section 5 to document the newly designed and implemented microservices that supports various scenarios and interacts with other components of the overall iHelp solution. Section 6 has been extended to include the installation, deployment and use of the enhanced Kafka Broker and the new microservices, while the plan and roadmap for the last phase of the project has been updated accordingly and included in Section 7.

2 The iHelp Big Data Platform

The iHelp Big Data Platform is the central data management system of the overall integrated solution. It is a relational distributed database with extended and innovative capabilities that has been built upon the LeanXcale datastore. The following sections will give a wide overview of the background technology and how the iHelp project will benefit from it.

2.1 Internal Architecture

The Big Data Platform is a scalable distributed SQL database management system with OLTP and OLAP support and full ACID capabilities. The overall architecture of this database solution is depicted in Figure 1 and can be considered as having three main subsystems: the relational query engine, the transactional manager, and the storage engine which provides multi-versioning, all three can be deployed in a distributed manner and can be highly scalable independently (i.e., to hundreds of nodes).

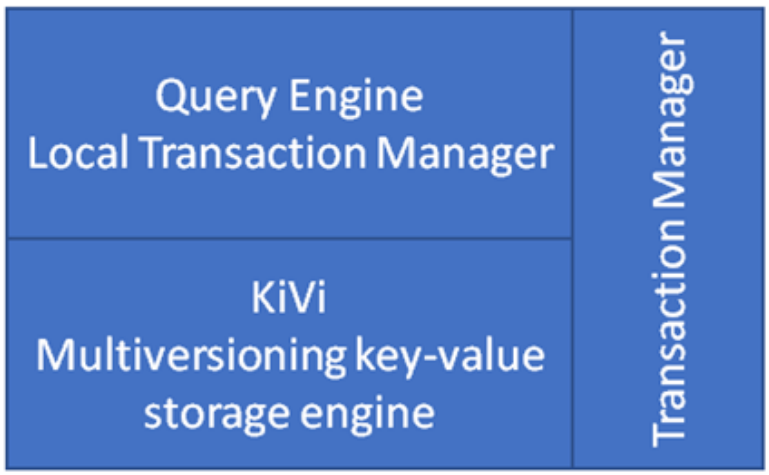

Figure 1: Architectural layer of the datastore

The system applies the principles of Hybrid Transactional and Analytical Processing (HTAP) and addresses the hard problem of scaling out transactions in mixed operational and analytical workloads over big data, possibly coming from different data stores (HDFS, SQL, NoSQL, etc.). LeanXcale solves this problem through its patented technology for scalable transaction processing. The transactional engine provides snapshot isolation by scaling out its components, the ensemble of which guarantees all ACID properties: local transaction managers (atomicity), conflict managers (isolation of writes), snapshot servers (isolation of reads), and transaction loggers (durability).

Let's first focus on the transactional engine of LeanXcale. This ensures online transactional processing (OLTP) and enforces database transactions, but being able to scale out efficiently to hundreds of nodes. The main differentiator that allows this component to scale horizontally is a novel distributed algorithm for transactional processing. Scaling out database transactions is a problem that has been more than three decades without being solved. The reason for this is that traditional relational database management systems often rely on the two-phase commit protocol (2PL) to ensure transactions. This protocol requires the use of shared and exclusive locks on data items according to the selected granularity. This means that locks can be put on a data table level, on a leaf node of the index tree or to the data items itself. Read operations places shared locks on the data items that are trying to access, while data modification operations places exclusive locks on their corresponding accessed data rows. Using shared and exclusive

locks, the transactional engine can ensure the desired level of isolation, the 'I' property of the ACID. For instance, putting shared locks forbids the concurrent modification of the value of a data item that has been previously accessed by a read operation, and therefore, removes the anomaly of non-repeatable reads. At the same time, putting exclusive locks, the transactional engine forbids a read operation to access a data item whose value has been recently modified, but the transaction that modified this value has not yet been committed. This removes the anomaly of dirty reads.

The adaptation of the two-phase commit paradigm enables the transactional engines of traditional relational datastores to ensure ACID properties as it guarantees the desired levels of isolation. However, this comes with two important drawbacks: The implementation requires a specific process or component to manage the distribution and the acquirement of such locks. This process or component must be centralized by design and therefore, it is very hard to efficiently scale horizontally the corresponding transactional managers. Most implementations only allow vertical scalability, which requires very powerful and thus expensive computer mainframes that also have a specific limitation and not endless computational power. The latter is required in modern ecosystems where most application components have been moved to the cloud.

Another drawback of the two-phase commit paradigm is that read locks, being placed by analytical queries, will block exclusive locks that are being placed by operational workloads and vice versa. In fact, those two different types of workloads are competitive and the one blocks the other. An analytical query will most likely involve a full scan operation over a data table, which would have placed shared locks over the whole dataset, thus blocking all other data modification operations on that table until the analytical query finished. To make things worse, data modification operations from on-going concurrent transactions would have already placed exclusive locks over this data table, thus blocking the execution of the analytical query. As result, OLTP and OLAP workloads are very difficult to be supported at the same time, and modern integrated data management systems tend to use traditional operational datastores for OLTP processing and periodically migrate data to analytical data warehouses or data lakes that will provide OLAP support. The drawback of such solutions is that the data users will perform their analysis over obsolete and outdated data.

On the other hand, the transactional engine of the LeanXcale database is based on a different approach: it implements the snapshot isolation paradigm which removes the need for placing shared and exclusive locks over data items and instead, relies on the use of versioned data items. Using this approach, a concurrent transaction that includes data modification operators, adds newer versions of data item using monotonically incremented timestamps. On-going read operations on the other hand will access the last committed version of data items before the time that the corresponding transactions began. As a result, read and write operations cannot block each other now, as each one access different versions of data items and Hybrid Transactional and Analytical Processing (HTAP) can be supported. This is crucial as it enables analytical query processing over real data, as being concurrently modified in the operational data store and removes the constraint of migrating data to external data warehouses where the analytical processing would have been applied over obsolete and outdated data.

A consequence of the fact that the transaction protocol of LeanXcale does not rely on a central component to manage the shared and exclusive locks is that the lifecycle of the transaction can be split in various phases, as it can be depicted from Figure 2. When a transaction starts, it receives the current timestamp in

order to read the most recent versions of the data items. While committing, it sends a request to the transaction manager, and the latter stores the updates in its persistent logs, so that updates can be recovered in case of failures, thus enforcing the durability property of the transactions. When the durability is being confirmed, the transaction finally finishes, while in parallel, the transaction manager of LeanXcale increments the value of the timestamps so that the forthcoming transactions can now have the visibility of the newly modified data items. It is important to be mentioned that each of these phases is being resolved by different components, which allows to decouple them and instead of having a centralized mechanism for managing transactions, the distributed approach allows for the components to be scaled out independently.

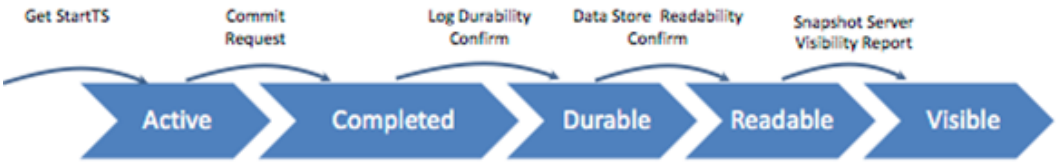

Figure 2: Transaction phases

The transactional engine of the LeanXcale database has been designed in such a manner that can be horizontally scalable. The differentiator of other approaches is that the enforcement of the ACID properties has been split into independent components that are responsible for them. For instance, the responsibility to ensure atomicity, the A property, has being moved to the client level, and therefore, it can be scaled out as much as the client application does. The durability, the D property has been implemented by the corresponding loggers that can be scale horizontally, while the enforcement of the consistency, the C property, has been moved to the storage layer and the relational query engine. Finally, the isolation, the I property, is being implemented by the following components: the snapshot server that is responsible to distribute the corresponding timestamps to be used by the read operations, the conflict managers that checks for write-write conflicts by concurrent transactions that contain data modification operations, and the commit sequencer that is responsible to advance and distribute new timestamps during the commit phase. Both the snapshot server and the conflict managers are solving problems that can be solved in a distributed manner, while the commit sequencer is the only component that must be centralized. However, the amount of work that requires is very low as it only needs to periodically increment a counter, and as such, it cannot be a bottleneck even in a vast amount of millions of concurrent transactions. As a result, the whole transactional engine can scale out to hundreds of nodes, while allowing on the same time for hybrid transactional and analytical processing.

The architecture of the transactional manager of LeanXcale can be depicted in Figure 3. Each instantiation of the query engine creates its own instance of the local transaction manager, whose role is to ensure the atomicity and communicates with the transaction manager. It is evident that the local transaction manager can scale out to as many instances of the query engine are necessary. The transaction manager also contains the conflict manager whose role is to ensure that there are no write-write conflicts while two concurrent transactions try to update the same data item. As it keeps only the list of data identifiers that are being currently accessed, it can be distributed. Both the local transaction and conflict managers make use of loggers which can be also distributed. The only components that must be centralized, are the ones that are included in the Mastermind. However, the unit of work that needs to execute is minimum as they hold information about the configuration of the current deployment, they increment a counter or they

periodically distribute the current timestamp. Therefore, the Mastermind cannot be considered as a bottleneck.

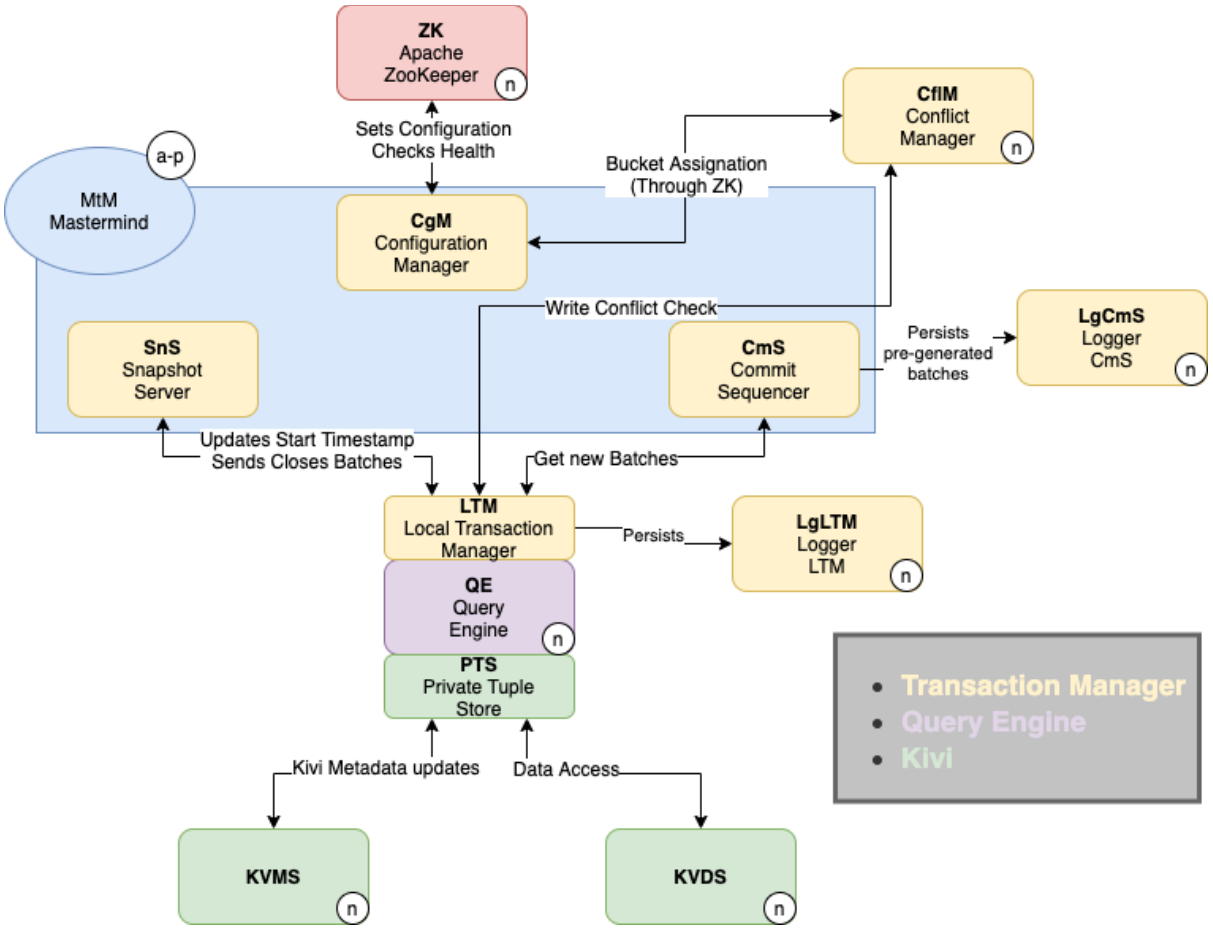

Figure 3: Transaction Manager

The second important architectural component of the LeanXcale database is its internal storage engine, which is called KiVi. KiVi can be considered as a standalone relational key-value store that provides an additional set of innovative features. It manages all operations that are related with the storage level, like persistently store data items to the storage medium, provide data access, and make use of data structures for indexing that allows for efficient query processing. KiVi is a distributed datastore, and consists of a data meta-server node and a list of data server nodes. It allows for you to split data tables in various data regions, move these data regions to other data nodes and finally merge the data regions. The split, move and merge operations over data regions can be executed in real-time, under heavy operational processing. This allows the elastic scalability of the storage engine itself. Under heavy workloads, KiVi allows the deployment of additional data nodes. Once the new nodes are available, a load balancing algorithm solves the resource allocation problem and decides how to redistribute the data regions across the data nodes, by splitting and moving these regions if necessary. This can happen in real time, ensuring database transactions as these actions take place. As a result, this removes the two important obstacles when scaling out a database that can be found by traditional solutions: NoSQL solutions can scale out in real time, but they lack of database transactions, while SQL solutions cannot scale online, and they have to suffer from a period of downtime during the data movement required by the scalability action. On the contrary, the storage engine of LeanXcale provides the benefits of the two worlds.

As it has been already mentioned, KiVi is a relational key-value datastore that provides a rich interface, which supports all relational algebra operations but the *join* operation. It allows for the definition of a schema that supports all SQL data types, and in fact, the values of the data items can have multiple columns. It allows scan operations, with or without filters, projections, ordering operations over indexed columns and aggregation operations. This allows to the third important component of the LeanXcale database, its relational query engine, to interact with the storage engine using its native interface and push down query operations down to the storage level. As a result, the query processing can be executed more efficiently as it does not have to access and transmit the whole data items to the level of the query engine. By doing the filtering on the storage level, pushing down projections and even aggregations, it minimizes the overall size of the data that need to be transmitted and processed in the upper level. What is more, the distributed nature of the KiVi also allows parallel processing of queries, intra-operator parallelism, which will be explained in the following section. As it has been mentioned, data tables can be split to a number of data regions, and the scan operator can be executed in parallel. Moreover, aggregation operators can be also supported by KiVi and implemented in a distributed manner: for instance, having the count operator as an example, each data region calculates its local count and the final result that will be sent to the relational query engine will be the summary of all local counts. The minimum, maximum and summary operators can be also implemented in a distributed manner, while the average operator is transformed to the overall summary divided by the overall count, and both of those can be executed in parallel.

Finally, the LeanXcale database consists of a third component that is the relational query engine that provides support for analytical processing. The query engine has derived its OLAP query engine from Apache Calcite[1], a Java-based open-source framework for SQL query processing and optimization. LeanXcale's distributed query engine (DQE) is designed to process OLAP workloads over the operational data, exploiting the capabilities of its transactional engine, so that analytical queries are answered over real-time data. This enables to avoid ETL processes to migrate data from operational databases to data warehouses by providing both functionalities in a single database manager. The parallel implementation of the query engine for OLAP queries follows the single-program multiple data (SPMD) approach, where multiple symmetric workers (threads) on different query instances execute the same query/operator, but each of them deals with different portions of the data. In the next section a brief overview of the query engine distributed architecture will be provided.

2.2 OLAP Queries

In this subsection, additional information will be given regarding the background technology that this work was based upon, the parallel query engine of the LeanXcale's database that allows for the parallel execution of analytical queries over real operational data.

Figure 4 illustrates the architecture of LeanXcale's Distributed Query Engine (DQE). Applications connect to one of the multiple DQE instances running, which exposes a typical JDBC interface to the applications, with support for SQL and transactions. The DQE executes the applications' requests, handling transaction control, and updating data, if necessary. The data itself are stored on a proprietary relational key-value store, KiVi, which allows for efficient horizontal partitioning of LeanXcale tables and indexes, based on the primary key or index key. Each table partition corresponds to a range of the primary/index keys and it is the unit of distribution. Each table is stored as a KiVi table, where the key corresponds to the primary key of

[1] https://calcite.apache.org/

the LeanXcale table and all the columns are stored as they are into KiVi columns. Indexes are also stored as KiVi tables, where the index keys are mapped to the corresponding primary keys. This model enables high scalability of the storage layer by partitioning tables and indexes across KiVi Data Servers (KVDS). KiVi implements all relational operators but the join operator, so any relational operator below a join can be pushed down to KiVi. However, as this is a limitation of the storage engine, join operators can be only supported by the relational query engine of the Big Data Platform.

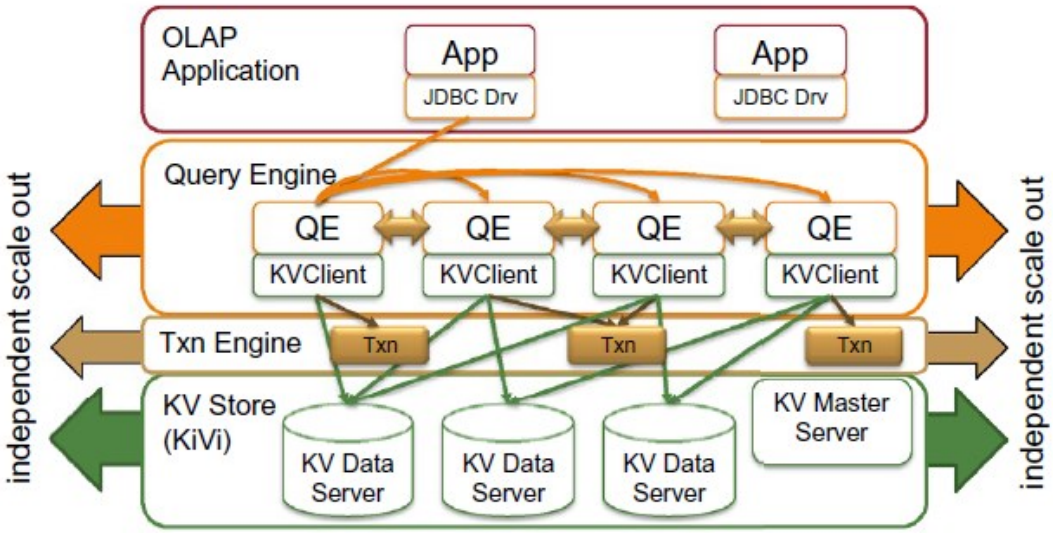

Figure 4: DQE Distributed Architecture

This architecture scales by allowing analytical queries to execute in parallel, based on the master-worker model using intra-query and intra-operator parallelism. For parallel query execution, the initial connection (which creates the master worker) will start additional connections (workers), all of which will cooperate on the execution of the queries received by the master.

When a parallel connection is started, the master worker starts by determining the available DQE instances, and it decides how many workers will be created on each instance. For each additional worker needed, the master then creates a thread, which initiates a transmission control protocol (TCP) connection to the worker. Each TCP connection is initialized as a worker, creating a communication endpoint for an overlay network to be used for intra-query synchronization and data exchange. After the initialization of all workers the overlay network is connected. After this point, the master is ready to accept queries to process.

The master includes a state-of-the-art [19] query optimizer that transforms a query into a parallel execution plan. The transformation made by the optimizer involves replacing table scans with parallel table scans, and adding shuffle operators to make sure that, in stateful operators (such as Group By, or Join), related rows are handled by the same worker. Parallel table scans will divide the rows from the base tables among all workers, i.e., each worker will retrieve a disjoint subset of the rows during table scans. This is done by scheduling the obtained subsets to the different DQE instances. This scheduling is handled by a component in the master worker, named DQE scheduler. The generated parallel execution plan is broadcast to be processed by all workers. Each worker then processes the rows obtained from subsets scheduled to its DQE instance, exchanging rows with other workers as determined by the shuffle operators added to the query plan.

Let us consider the query Q1 below, which we will use as a running example throughout the paper to illustrate the different query processing modes. The query assumes a TPC-H[2] schema.

```
Q1: SELECT count(*)
    FROM LINEITEM L, ORDERS O
    WHERE L_ORDERKEY = O_ORDERKEY
    AND L_QUANTITY = 5
```

This query is transformed into a query execution plan, where leaf nodes correspond to tables or index scans. The master worker then broadcasts to all workers the generated query plan, with the additional shuffle operators (Figure 5a). Then, the DQE scheduler assigns evenly all database shards across all workers. To handle the leaf nodes of the query plan, each worker will do table/index scans only at the assigned shards. Let us assume for simplicity that the DQE launches the same number of workers as KVDS servers, so each worker connects to exactly one KVDS server and reads the partition of each table that is located in that KVDS server. Then, workers execute in parallel the same copy of the query plan, exchanging rows across each other at the shuffle operators (marked with an S box).

To process joins, the query engine may use different strategies. First, to exchange data across workers, shuffle or broadcast methods can be used. The shuffle method is efficient when both sides of a join are quite big; however, if one of the sides is relatively small, the optimizer may decide to use the broadcast approach, so that each worker has a full copy of the small table, which is to be joined with the local partition of the other table, thus avoiding the shuffling of rows from the large table (Figure 5b). Apart from the data exchange operators, the DQE supports various join methods (hash, nested loop, etc.), performed locally at each worker after the data exchange takes place.

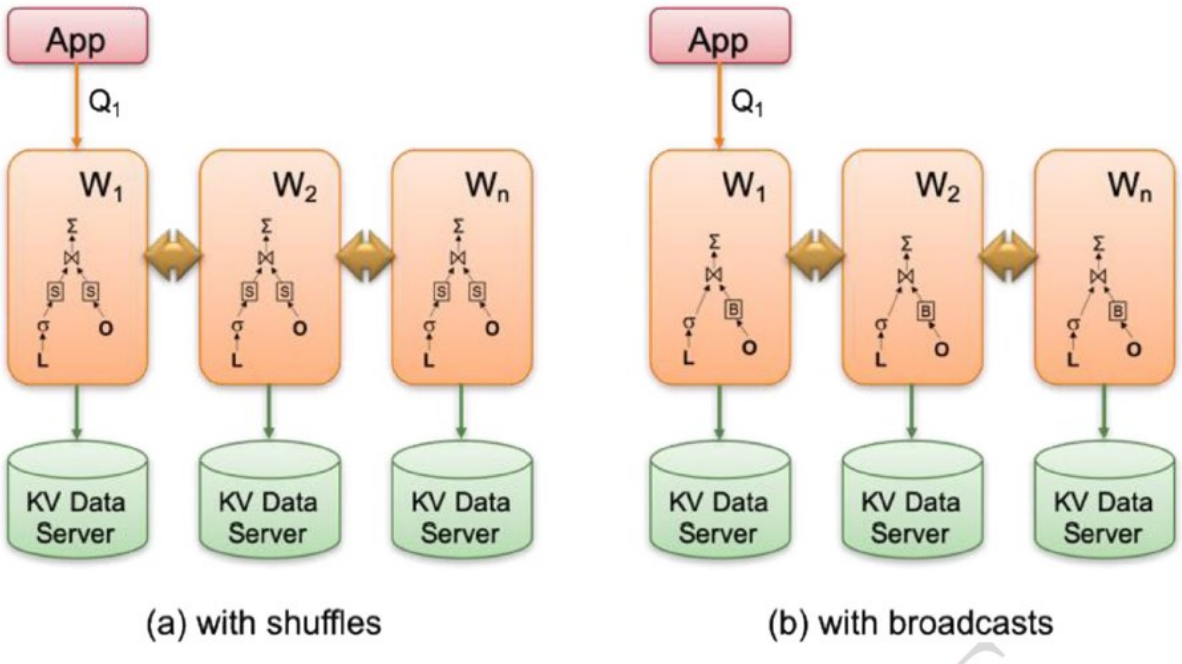

Figure 5: Query processing in parallel mode

[2] http://www.tpc.org/tpch/

2.3 iHelp Integrated Solution

Having described in the previous subsections the fundamentals of the Big Data Platform, this subsection will give more details of the overall integrated solution for iHelp, which can be depicted in Figure 6.

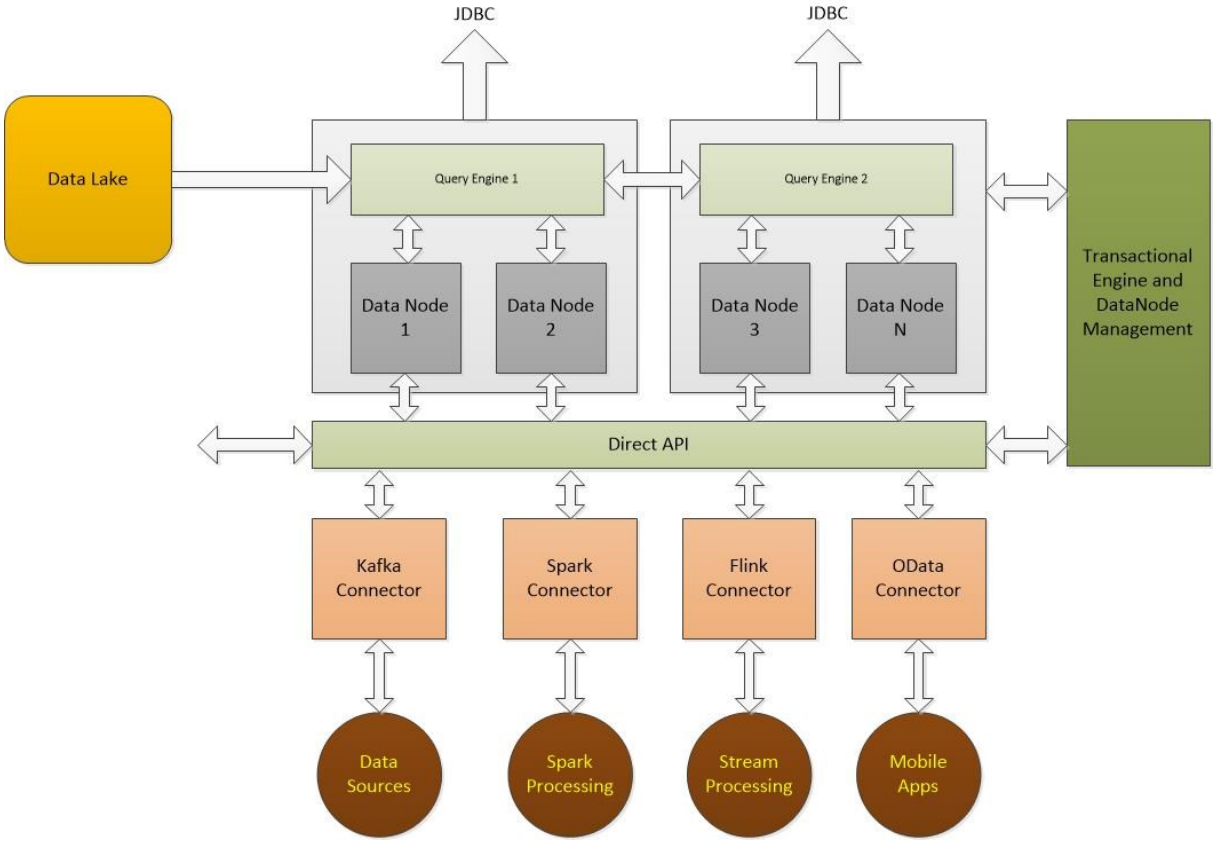

Figure 6: iHelp Big Data Platform

As we can see, the Big Data Platform consists of various components and provides data connectivity mechanisms, which allow the integration of the big data platform with different and diverse data providers, data consumers or other analytical processing frameworks.

The key components of the Big Data Platform are the internal components of the LeanXcale distributed datastore: its query engine, its storage engine and the transactional management. These components are illustrated in Figure 6 as well. The distributed storage engine is depicted by the various data node components of Figure 6 while the distributed query engine is depicted by the various *query engine* nodes in the figure. A typical installation of the datastore will require one query engine to be responsible for accessing a pair of data nodes, all deployed in the same physical or virtual machine. Moreover, the transactional engine of LeanXcale can be considered as a vertical component that is accessible by both the query engine and the storage element. It is deployed on separate machines and if needed, its internal components can also scale out independently, as it has been described in detail in a previous subsection.

The Big Data Platform exposes various means for data connectivity; with the most important ones will be the connection to the query engine itself and the direct connection to its internal storage. Regarding the connection to the relational distributed query engine, the Big Data Platform provides support for the following:

- JDBC: This is the standard implementation of the Java's JDBC interface, as defined by the Java JSR 221[3], which allows for java applications to establish data base connections with a relational database. It is provided as a java library packaged in a jar file.
- ODBC: This is the open database connectivity standard that allows each application to connect to a relational database. It is provided as a *Ruby Gem* package, while there is also the provision for Microsoft Windows installation using an exe program.
- Python driver: This is a driver to be used by application developers or data analysts that make use of python source code, with the use of SQLAlchemy[4].

It is important to be mentioned that all these three means are connected to the distributed query engine of the big data platform. The query engine itself is responsible for communicating with the transactional manager of the datastore in order to ensure the data base transactions and the corresponding ACID properties. This means that when a database transaction opens in the client level, it is always opened in the scope of an existing connection. Therefore, by opening a database transaction in the client level, this is being communicated via this open connection to the specific instance of the query engine, which will request a new *snapshot timestamp* by the transactional manager. Upon commit, the client (the driver) sends the commit request to the query engine, via the open connection, and the query engine forwards the request to the transactional manager. The latter decides if and when it is possible for the transaction to be safely and durably committed, and if this is the case, it informs the query engine. If the transaction fails to commit, then the query engine will receive an exception from the transactional manager, which will be further propagated to the driver as a SQLException, which will cause the client program to fail. The latter now needs so wisely handle these exceptions and perform accordingly.

Apart from these three means of data connectivity, a client program or a data analyst can also connect directly to the storage engine, using its direct API. The storage engine can be seen as a key value store, with extended capabilities. It provides the majority of the operations defined in the relational algebra. Apart from the support for selections (get or scan) and insertions (insert, update, delete or upsert), there is also the support from more complicated operations like filters, orderings, projections, while there is also the support for aggregations (min, max, sum, count, avg) under group by clauses. Only the join operation cannot be executed by the direct api, as this is an operation that requires the combination of two different relations (tables), and therefore, this can be executed only by the relational query engine of the big data platform.

The benefit for using the direct API of the platform instead of the query engine itself, is that the execution completely bypasses the latter, and thus, it can be done much more efficiently. It completely avoids the footprint of the query engine, while the connection targets the storage engine directly. This increases the throughput of ingestion data workload that can be supported, allowing the big data platform to support data ingestion in very high rates. The direct API also communicates with the transactional manager of the big data platform when the user requests to start or commit/rollback a transaction, thus also ensuring database transactions and ACID properties. In fact, the transactional manager ensures database transactions even if two concurrent transactions occur from different means: one from the relational query engine and the other from the direct API. This means that big data platform allows from one part to connect to a high rate data ingestion pipeline, and on the other hand, for data analysts to execute sophisticated

[3] https://jcp.org/en/jsr/detail?id=221
[4] https://www.sqlalchemy.org/

queries and do analytics using the relational query engine at the same time. The snapshot isolation and the provision for Hybrid Transactional and Analytical Processing (HTAP) of the datastore allows the analytics to be executed in the real data, without the need to migrate the data that is being ingested to a different analytical database management system.

The drawback of using the direct API is that it cannot support all the operations of the relational algebra, with most important the *join* operation, and therefore this must be implemented in the client application level. Moreover, it does not implement a well-known standard, and therefore, it cannot be used by other popular frameworks.

For popular frameworks to take advantage of the benefits of the direct API of the big data platforms, specific connectors must be implemented for each of those. The big data platform implements connectors for the following frameworks, which allows the use of the platform from a wide variety of different applications:

- Spark connector: This is the connector to the Apache Spark[5], which is the world's dominant parallel analytical processing framework. The majority of AI algorithms are fed with analytics that make use of this framework to do the data retrieval and pre-processing. Using the spark connector, the Apache Spark can now push all its operations down to the storage engine, so it can retrieve the minimum amount of data items that it needs.
- Kafka connector: This is the connector to the Apache Kafka[6], which is the world's dominant event streaming platform. It allows data being pushed to a Kafka topic to be transparently stored into the big data platform in a predefined data table, using this connector. Apache Kafka is an important platform element of the overall iHelp integrated solution. In fact, the data ingestion pipelines defined under the scope of the T3.2 – "Primary Data Capture and Ingestion" makes use of Kafka topics to move data from one function to the following in the pipeline, and at the end, to store them into datastore. Moreover, Apache Kafka is being used by the DRyICE serverless platform that is the fundamental pillar of the iHelp integrated solution, provided by the T4.2 – "Model Library: Implementation and Recalibration of Adaptive Models". The serverless platform is responsible to provide the execution environment of the data analytics and models defined under T4.1 – "Personalized Health Modelling and Predictions" and T5.1 – "Techniques for Early Risk Identification, Predictions and Assessment" and it makes use of Kafka topics to push data down to the datastore.
- Flink connector: This is the connector to the Apache Flink[7], which is the world's dominant streaming processing framework. It allows for data connections using standard JDBC in order to combine streaming data with data *at-rest*, stored in a persistent medium like a database. Moreover, it also provides its own interface to query data *at-rest*, and therefore, the Flink connector is the bridge between the streaming processing framework and the distributed database of the big data platform. This could be used in possible scenarios that would require streaming processing and rely on the Apache Flink. Possible scenarios could be defined in T5.3 – "Delivery Mechanisms for Personalised Healthcare and Real-time Feedback", however, at this point, it has not been foreseen.

[5] https://spark.apache.org/
[6] https://kafka.apache.org/
[7] https://flink.apache.org/

- OData: This is an implementation of the Open Data Protocol[8], which is an OASIS standard. It allows data providers or consumers to connect to a database using a REST interface. This will allow data connectivity with other types of applications that would require a direct access to the database, like the visualization components of the query builders implemented under the scope of T4.3 – "Clinical DSS Suite with Visual Analytic Tools".

Apart from the aforementioned methods for data connectivity, the Big Data Platform includes a parallel polyglot query engine that allows the query processing and combination of data that are stored externally to the iHelp platform. This for instance would allow the query processing and combination of data stored internally to the platform, as HHR records, and data that are stored in an external data lake. The benefit of having this is that the data analyst can push down queries directly to the Big Data Platform and let the latter execute them, instead of using an analytical framework like Apache Spark for that. Moreover, this will allow the data exchange between different installations of the iHelp platform, as it will be explained in the following section.

[8] https://www.odata.org/

3 Polyglot Query Processing

One of the key requirements for the integrated iHelp solution is to be portable and to be deployed in various premises. At the very beginning of the project, a centralized environment was envisioned by the technical partners of the consortium. This centralized environment would have stored all the data coming from the various use cases: from the hospitals in Spain and Italy, the research institutions of UK and Sweden, and the individual secondary data coming from the individual persons. These different datasets would have been categorized per use case, so each AI algorithm could target the specific dataset that the data analyst is interested in. The AI algorithms on the other hand would have been extendable and applicable to the different datasets, which in fact would have been compliant to the common data model of the iHelp integrated solution, the HHR.

This approach however was violated when the first discussions with the pilot use cases started in order to collect the user requirements under the scope of T2.1 – "Requirements, State of the Art Analysis and User Scenarios in iHELP" and to define the data management plan under the scope of T1.4 – "Data and Ethic Management". It was understood by the consortium that the data that will have to be stored and processed by the iHelp platform and tools are related with clinicians and therefore, there are very sensitive. This makes very difficult the deployment and maintenance of a centralized environment where all data could be accessible by all tools that could be used by any data analyst. Moreover, specific pilots have put restrictions on the movement of the data: data cannot be moved outside of the country, or the organization itself. This means that the iHelp platform, along with its Big Data Platform, which is an integral part of the overall integrated solution, must be deployed on premise, inside the organization, that will be responsible to provide the corresponding infrastructure and resources.

When it comes to the tools or the building blocks of the solution, this can be facilitated with the use of virtualization technologies: the platform components and AI tools can be containerized and the overall integrated solution can be deployed and instantiated using virtualization deployment orchestrators like Kubernetes or docker compose. However, the deployments cannot move data outside, and therefore, restricts the applicability of the tools: even if an AI solution is generic and can be applicable both to a dataset coming from UK or Spain, it cannot combine both datasets in order to get better insights. An example could be the evolution of the pancreatic cancer on a population living in the northern parts of Europe (i.e., UK or Sweden) compared to population living in the southern part of Europe (i.e., Spain or Italy). In such scenario, the AI algorithm would need to access datasets coming from different sources. However, as there is no centralized environment, and data cannot be moved from one place to another, this cannot be feasible.

The solution to this problem is given by the Big Data Platform and more precisely, by its polyglot query processing engine. The Big Data Platform can be able to access data that are stored in other instances of the platform. This will enable a peer deployment of the iHelp integrated solution, that each deployment could access the other, as Figure 7 illustrates.

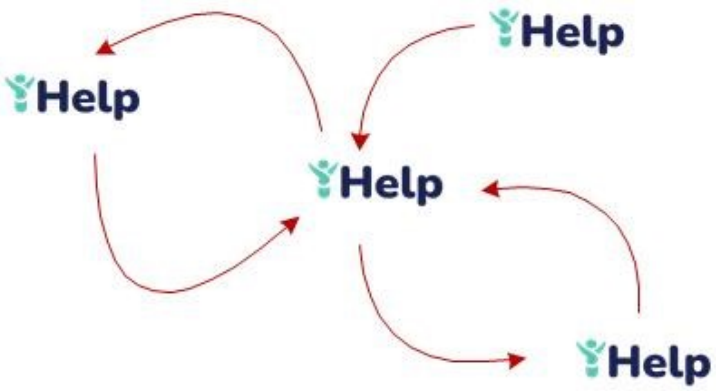

Figure 7: iHelp peered deployments

It is important to highlight at this point that in the envisioned scenarios, when data can be accessed by other deployments, the data itself will not be moved outside of the organization, and the organization itself, as the data provider, can define the type of access is allowed to its data. To access data, as mentioned, we will rely on the polyglot query engine of the Big Data Platform. As a result, taking into account what is depicted in Figure 7, then Figure 6 can be extended as follows:

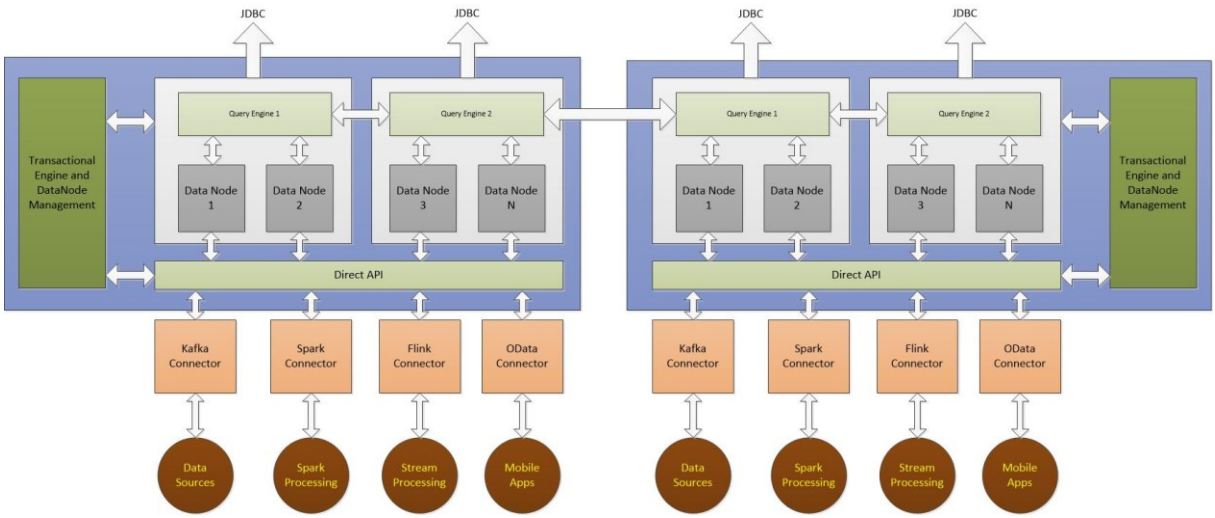

Figure 8: Polyglot Access on Peer Deployments

As depicted now in Figure 8, we can have different instances of the Big Data Platform, each one deployed inside the premises of the organization as part of the corresponding peer. However, each big data platform can communicate with each other execute remotely query statements. This makes use of the polyglot query engine of the LeanXcale internal database. More technical details of the polyglot query engine can be found in (P., P., K., +21). To summarize, each query engine takes a string as the query input and returns back data. The input string is being parsed by the query *compiler* that transforms the input string into a data structure that can be programmatically used by the query engine itself. The data structure is most often a data tree, where each node of the tree is a query *operator* and each leaf of the tree corresponds to a *scan* operation. The latter is the one responsible for the actual data access, which commonly requires an I/O operation to the disk or the corresponding persistent storage medium.

The query *operators* in the Big Data Platform can be any of the relational algebraic operations and their corresponding implementations by the platform's query engine itself. What happens in practice is that after

the *compiler* produces the query tree of operations, this is pushed to the query *optimizer,* which is responsible for exploring the space of equivalent query execution plans in order to find a more efficient plan. Internally, it uses a *dynamic programic* type of algorithm while investigating the space, while the space has been generated by applying a set of query transformation rules. This process is out of the scope of this report, however, the important outcome is that the query operations can be re-ordered and placed in any level of the query execution plan. As a result, an important requirement for the query engine is that these operators must be generic, and in fact, they need to implement a common interface. In most relational query engines, and in fact, in the LeanXcale datastore, this interface extends an *iterator*, which will have to return an array of objects. The array of objects can be seen as the data row of a dataset, and the *iterator* iterates through dataset that needs to be returned by each operator. This leads to the *volcano iterator model* where data is being pulled from the leaf of the data tree to the upper layers of the tree and finally, to the data user.

In the polyglot query engine, data needs to be accessed by both the internal I/O of the storage engine of the Big Data Platform, and the external data source. Due to the *volcano iterator model,* it is irrelevant from where the data will be retrieved from. For the query engine perspective, everything is an operator. As a result, the polyglot query engine needs to implement specific operators for accessing data from the corresponding type of data source. This means that in the scope of the iHelp integrated solution, an *operator* needs to be implemented that can submit statements from other instances of the Big Data Platform. As it accesses data, this *operator* will always be in the leaf of the query execution tree. However, it can be able to accept a query statement that will be pushed down to the external datastore.

As it has been described in the previous section, the Big Data Platform provides JDBC connections. Therefore, the *operator* that will enable the de-centralized access will open JDBC connections and submit relational SQL statements. The result of a query execution in JDBC is a *ResultSet,* which in fact, extends the *Iterator.* So, it perfectly fits to our design.

At this point, we need to highlight the fact that the important requirement from the pilot use cases is that data must not be moved outside of the organization. With our approach, data indeed remain stored at where each instance of the big data platform is deployed. External instances only access data of other peers, but they never move this data. However, there is a tricky point that needs to be further clarified. Let's assume that the data user is allowed to execute the following statement:

```
Q2: SELECT *
    FROM Patients
```

This will open a remote JDBC connection to the target datastore, will execute this statement and will retrieve the results by iterating over the returned dataset. This implies that all data stored in the *Patients* data table, can be available to the instance of the query engine of the data user that submitted this statement. In correspondence, all data of this data table can be available now to the data user. And this implies that one person is capable of executing such a statement, access the data that are stored externally, retrieve everything and finally store this information elsewhere. At the end, even if the Big Data Platform only accesses but never moves data, the data in fact can be moved by exploiting the capabilities and functionalities of the Big Data Platform. This means that these types of queries must be forbidden.

To solve this issue, we go back to the basics of the relational database management systems. We need to recall that the Big Data Platform is a relational database itself, and therefore, it provides the support of relational *views*. A *view* is a virtual data table whose rows, at one point in time, are the result of the execution of a predefined query statement at that time. This allows the data owner to define such *views* and only allow remote data access via these *views.* As each *view* is the result of the execution of a predefine query statement, these statements could only expose aggregated information of the data, and not the raw data themselves. In fact, the Big Data Platform will request data from external deployments via only these *views*, which resembles the pushing of a query statement down to the target data source. Each Big Data Platform can create specific users that can access these data and prohibit the access to all other data tables. Each user is authenticated via a token that is provided to the datastore through the corresponding connection URL, each time the client application or data analyst tries to open a connection. Then, the Big Data Platform checks if the user is authorized to access the data sources, which in our case will be the *views,* and if yes, it will execute the corresponding query and return the aggregated information that will be further processed by the query engine of the client.

At this phase of the project, the polyglot query engine and the remote data access has been currently designed, but the implementation has not started yet, as it is not the main focus of for the first prototype of the overall solution. The third version of this report will provide more details about the implementation of the aforementioned *operation*. At this phase, the target is only on the implementation of the *operation*. We will assume that the data user or analyst knows beforehand where to connect. In a more sophisticated scenario, we might need to introduce an additional centralized service, which will act as a *broker*. The latter will publish to each of the deployments the available instances so that they can connect automatically. However, this mechanism will be defined and designed at a later phase, if needed.

4 HHR relational schema

One of the outcomes of WP3 – "Personalised Holistic Health Records" and more precisely of T3.1 – "Data Modelling and Integrated Health Records" is the definition of the *Holistic Health Record* (HHR). This has been documented in D3.1 – "Data Modelling and Integrated Health Records I" and it is currently under further development.

The D3.1 – "Data Modelling and Integrated Health Records I" defines the *entity-relational* (E-R) model of the HHRs. For all of the available datasets coming from the different pilot use cases, an extensive analysis has been performed in order to map the raw datasets to HHR entities. This is still work in progress for all datasets, however, an initial version of this model has been defined based at this phase of the project and it has been validated based on the dataset coming from the "*Study of Lifestyle Choices on Elevating the Risk Factors for Pancreatic Cancer*" scenario of the Hospital de Dénia-MarinaSalud.

According to this scenario, there have been identified 7 distinctive entities: i) patient, ii) encounter, iii) procedure, iv) medication and medication statements, v) observation, vi) measurements and vii) conditions. These entities are dependent to others with the E-R model to define several one-to-many relationships. One of the responsibilities of this T4.4 – "Big Data Platform and Knowledge Management System" is to define the equivalent relational schema for the HHR. As a result, T4.4 – "Big Data Platform and Knowledge Management System" was waiting for input coming from T3.1 – "Data Modelling and Integrated Health Records" that defines the E-R model of the HHR, and provides output to T3.2 – "Primary data capture and ingestion", in order for the later task to develop the *HHR Importer*. This component is responsible for receiving HHR resources from the data ingestion pipelines, and needs to store them to Big Data Platform. As the datastore is a relational database, the *HHR Importer,* as also described in D3.4 – "Primary data capture and ingestion II", needs to transform these HHR resources to data beans or POJOs, that are mapped to the data tables of the relational schema. As a result, T3.2 – "Primary data capture and ingestion" was waiting for the definition of the relational schema to be able to proceed with its implementation.

The process for transforming an E-R model to a relational schema is the following: for each of the entities in the E-R model, we need to create a new data table. Then, for each one-to-many relations in the E-R model, we need to define a foreign key relationship, from the child table (the *many* in the relation) to the parent table (the *one* in the relation). Then, for all many-to-many relations, we need to create a new data table, and to create two foreign key relationships to the two tables involved in the many-to-many relation of the E-R model. According to the definition of the HHR and its validation and after its validation with the "*Study of Lifestyle Choices on Elevating the Risk Factors for Pancreatic Cancer*" scenario of the Hospital de Dénia-MarinaSalud, the relational schema for the HHR which is equivalent to the E-R model defined under T3.1 – "Data Modelling and Integrated Health Records" is depicted in the following Figure 9:

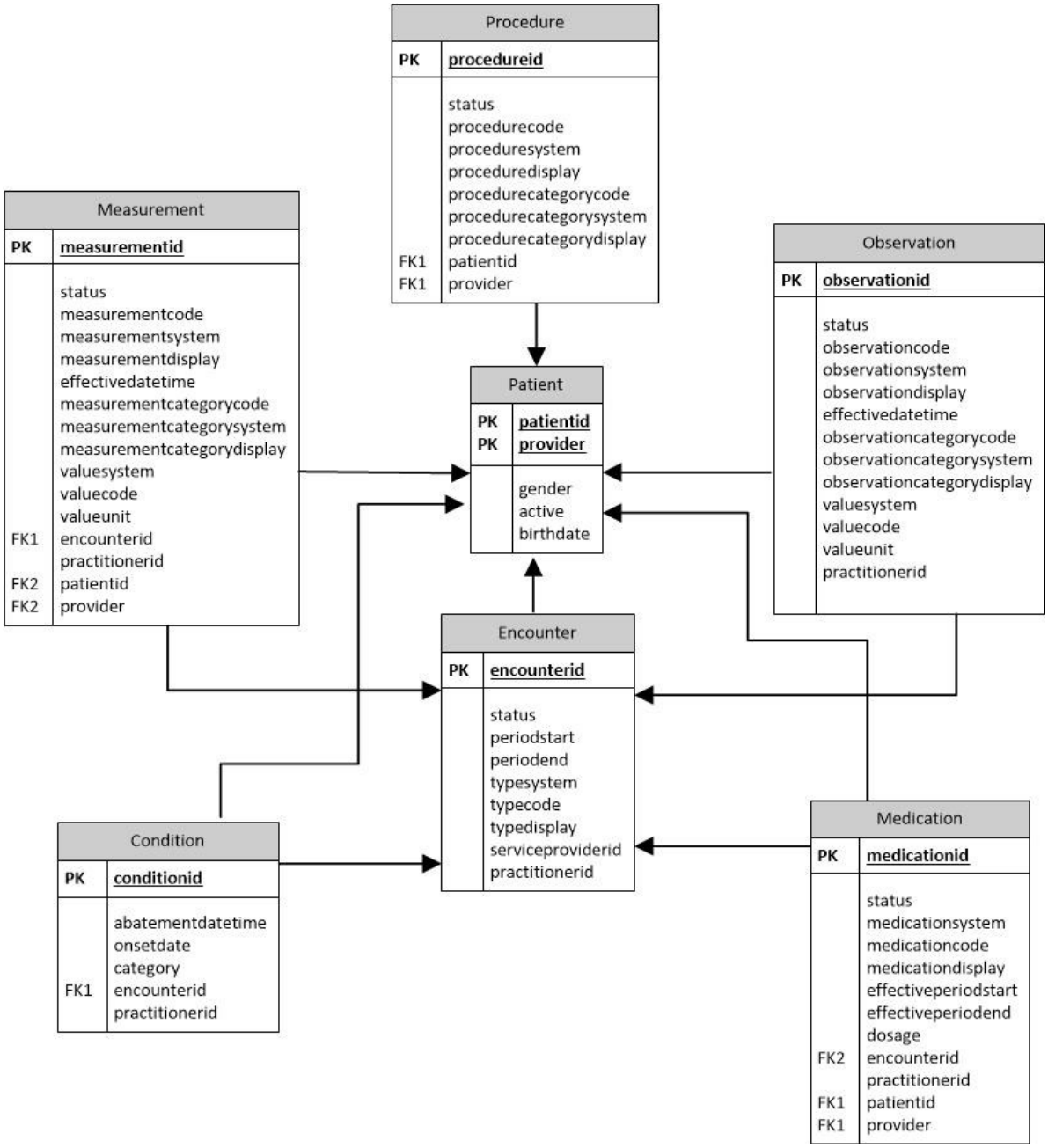

Figure 9: HHR relational schema

In the above figure we can see seven data tables that correspond to the seven entities that we observed in the E-R model of the HHR. Following the procedure previously described, we created all foreign key relationships for the one-to-many relations found in the E-R model. At this phase of the project, no many-to-many relation has been identified, and therefore, there was no need to define additional data tables that can *break* these relations.

In the next phase of the project, as new datasets will be eventually incorporated into the overall HHR model, the latter is expected to be further extended adding new entities or extending the attributes of the existing ones. Therefore, in the last version of this deliverable, we will include all the final description of the relational schema of the HHR, as the work that is currently being developed under T3.1 would have been finished by then.

5 Additional functionalities

The main outcome of the work that has been carried out under the scope of the T4.4 – "Big Data Platform and Knowledge Management System" is the delivery of the first version of the Big Data Platform of the overall integrated iHelp solution. Its core functionalities and innovations have been described under section 2, while section 3 describes its polyglot capabilities and these can be brought and provide added value to the overall solution. Moreover, as the big data platform is a relational database where the raw data transformed to HHR need to be stored, we put some additional effort to define this relational schema of the HHR that will be provided with the installation and deployment of the Big Data Platform.

Apart from the work that has been performed to implement these activities, there has been the need to implement additional functionalities in order to support other use cases and scenarios. These use cases and scenarios require the implementation of a set of functionalities to be provided by the iHelp platform. Thus, at most of the cases, these components require the interaction with the Big Data Platform in a non-standardized manner, or at least, in a way that is not supported by the datastore. For instance, an intermediate functionality might be needed between the big data platform and the core component of a scenario, or, as in the case of the DSS Suite, the component cannot interact via standard database connectivity mechanisms and requires the provision of a set of REST APIs. As a result, some additional activities are required by the T4.4 – "Big Data Platform and Knowledge Management System" which are not direct related with the big data platform, however, they are supportive activities for the latter to be integrated with other components into the overall iHelp solution.

For each functionality required, a new microservice has or will be designed and implemented, as a java process to be deployed using the Kubernetes container orchestrator platform. As a microservice, each of these components will expose a set of REST APIs, or a later phase an additional set of websockets, to allow the communication among the various subcomponents of the platform.

At this phase of the project, we have developed a microservice needed for the DSS Suite, to integrate its SQL Node Pallet with the big data platform, and additionally, use the same API to retrieve information regarding the patients. We have also design the *user enrolment* scenario, along with its extensions in the relational schema and the definition of the interfaces required by the microservice that is supposed to be implemented under the scope of this task in the forthcoming period. For the last version of this deliverable, all other functionalities and microservices will be included in this section.

5.1 iHelp-Store-Rest

The first microservice developed under this task is the one that is currently known as *ihelp-store-ret.* This is consumed by the DSS Suite and its SQL Node Pallet to allow the end-users to define from a graphical interface relational algebraic operations, and combine them to produce a query execution tree that will be translated to a complicated SQL statement. The latter will be sent to the database and it will return back the results. As such, we named this microservice as *ihelp-store-rest* due to the fact that it exposes a REST API to access the raw data stored into the big data platform, allowing the end users to explore them.

This microservice defines two different REST *resources,* each one of those defines a set of different *web methods*. The first REST *resource* is used by the SQL Node Pallet to retrieve meta-information regarding the relational schema definition provided by the big data platform. This is used to illustrate to the end user the

name of the existing data tables, their columns etc. That way the end-user can use the Pallet to start designing his or her relational algebraic operations.

The second REST *resource* is responsible for the query processing of the incoming SQL statement. It returns back a list of the values related with the result of this execution.

Their interfaces are defined in the following subsection.

5.1.1 Interface

5.1.1.1 MetaInfoResource

1. GET /metainfo: Returns a list of table names

Example of response:

```
[
  "tableA", "tableB", "tableC"
]
```

2. GET /metainfo /{table}/columns: Returns a list of columns along with their types

Example of response:

```
[
  {
    "name": "string",
    "type": "string"
  }
]
```

3. GET /metainfo/{table}/indexes: Returns a list of indexes along with their types

Example of response:

```
[
  {
    "name": "string",
    "unique": true,
    "fields": [
      {
        "name": "string",
        "type": "string"
      }
    ]
  }
]
```

4. GET /metainfo/{table}/primaryke: Returns a list of fiels along with their types that consists the table's primary key

Example of response:

```
[
  {
    "name": "string",
```

```
    "type": "string"
  }
]
```

5.1.1.2 Process

1. GET: /process?query={query}: Returns a list of data rows according to the input query

Example of response:

```
[
  {
    "row": [
      {
        "name": "string"
      }
    ]
  }
]
```

2. POST: /process: Returns the data row with its id. It is used to additionally store data items, but it has been deprecated at the current version of this microservice

Example of response:

```
{
  "ihelpID": 0,
  "gender": "string",
  "date": "string",
  "pilotID": "string",
  "healthentiaID": "string"
}
```

5.2 User Enrolment

This microservice is needed for the DSS Suite to implement the dashboards to allow the clinicians to *enrol* patients into the system. We have distinguished raw data into two major categories: primary and secondary data. Primary data are usually provided by the hospitals and include clinical information of a patient. Secondary data are usually provided by external applications, like the Healthentia, which collects behavioural information from individuals, like habits, food consumption etc.

During the data ingestion pipelines, there is no distinction if the incoming data are primary or secondary, as the process is the same. Eventually, raw data are transformed to HHR resources which will be persistently stored to the big data platform, following the HHR relational schema that was described in a previous section. According to the definition of the relational schema, there is the data table called *Patient*, where the information related with the individuals (being patient or not) will be stored. The primary keys of this table are the patient ID, as produced by the external provider of the dataset, and the name of this provider. This is due to the fact that data coming from different providers might have the same identifier, as its uniqueness is ensured by the provider. However, if we put data from different providers into the same data table, this might create a conflict on the uniqueness of the key. That is why we used a multi-column primary key in that table, which is composed by the identifier and the name of the provider.

Let's consider now the *Interventional Monocentric Study based on Patient Reported Outcomes* scenario of the Agostino Gemelli University Policlinic in Italy. Primary data exported by the hospital are being ingested into the Big Data Platform. At the same time, the patients are also using the Healthentia application to participate to its studies, so daily, secondary data produced by the individuals are also being stored. Both data are stored into the *patient* data table, with a different provider name and their associated identifier. However, with this design of the schema, it is impossible to correlate a patient's primary data with his or her secondary, unless this information is somehow provided.

We define the *user enrolment* use case, that allows clinicians to enrol patients to the iHelp platform, thus being assigned a unique identifier of the platform that will correlate all data coming from the different sources.

5.2.1 Design

In order to support such scenario, we need firstly to extend our relational schema to be able to hold this type of information. As it is depicted from our relational schema, data concerning an individual will be entered as much times as the individual is included in the ingested datasets. That is, for our *Interventional Monocentric Study based on Patient Reported Outcomes* scenario, it will appear twice. This is valid, as the individual might have different information coming from the primary data ingestion and other information coming from the secondary data ingestion. In any case, there are two distinct identifiers that are related to the two different data source providers.

In order to correlate these data records that relevant to the same individual, we will include an additional table for this purpose. The *ihelpidentitu* data table will have a foreign key relationship to *patient*, as the following Figure 10 depicts.

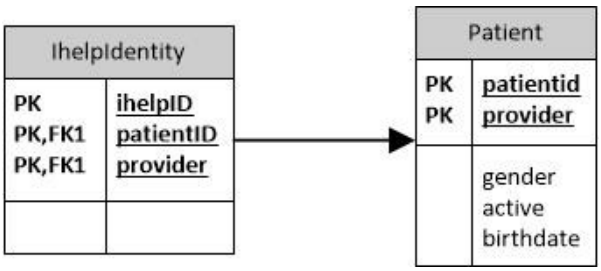

Figure 10: iHelp Identifier

With this design, we allow the data ingestion pipelines defined under the scope of T3.2 – "Primary data capture and ingestion" to remain the same. In fact, as data is being ingested, it will be added in the *patient* table as previously. However, the clinician now has the ability to correlate this information that might be ingested from different sources. For this, a new microservice has been designed with the interfaces that are documented in the following subsection.

5.2.2 Interface

The definition of the interface of this microservice is currently in progress, at the time that this report was written. We include here its initial version, however, it is expected that it might be modified. The final version of the interface will be documented at the third version of this deliverable.

1. POST: /userenrolment: It creates a new iHelp identifier for the specific patient

Example of response:

```
{
  "ihelpID": 0,
  "patientID": "string",
  "provider": "string"
}
```

2. PUT: /userenrolment: It adds a new patient record to be assigned with the given iHelpID
3. GET: /userenrolment?id={id} It returns the list of all patient records related with this iHelpID

Example of response:

```
[
   {
     "patientID": 0,
     “provider” : “string”
     "gender": "string",
     "active": true,
     "birthdate": "string",
   }
]
```

4. GET: /userenrolment?id={id}&provider={provider} It returns the patient record from the given provider having the given iHelpID

Example of response:

```
{
  "patientID": 0,
  “provider” : “string”
  "gender": "string",
  "active": true,
  "birthdate": "string",
}
```

6 Deployment and use

In this section we will discuss the deployment and run-time phase of the Big Data Platform and its additional microservices that have been developed under the scope of this task, along with some examples for its use. We will give examples on how to install and deploy in a local machine for testing purposes and how we can use a container orchestration platform like Kubernetes to facilitate the deployment of the Big Data Platform to different infrastructures. Finally, example of its use will be given at the end of this section.

6.1 Big Data Platform

In this subsection, we will give information about the core component provided by this task: the big data platform. This section remains the same as of the first version of this document.

6.1.1 Local installation using docker

Firstly, the released distribution for the iHelp project has been uploaded to the project's private Gitlab[9]. Each member of the consortium should have access to this code repository and can issue the following command, adding his or her username and password in order to download the Big Data Platform locally:

```
git clone https://gitlab.ihelp-project.eu/pkranas/ihelp-store.git
```

This GitLab project contains all binaries, scripts and additional configuration files, along with a Dockerfile, which allows the data administrator to build a *docker* image locally that can be later used to deploy and install the Big Data Platform in a containerized environment. In order to build the image locally, having already cloned the GitLab project, the data administrator needs to execute the following command:

```
docker build –t ihelp-store .
```

This will take some time as the Dockerfile is using the base Ubuntu 20.04 image and needs to pre-installed firstly various packages that are necessary for the Big Data Platform to run. At the end of the process, the docker image is built and available locally at the machine where this command was issued. In order to deploy and run the Big Data Platform, the data administrator needs to issue the following command:

```
docker run -d -p 2181:2181 -p 1529:1529 --name datastore --env KVPEXTERNALIP='datastore!9800' ihelp-
store
```

This command creates a container that will start in the background, giving the name *datastore* while it exposes the ports 2181 and 1529. This is necessary, as these are the ports that are being targeted by the data connections, when the data user or application developer needs to connect to the query engine of the datastore in order to submit and execute SQL statements. The data administrator can also connect to the container from a command line tool, by executing the following:

```
docker exec –it datastore bash
```

This command will open an interactive secure shell that will remain opened and will execute the *bash* script. The latter will allow him or her to issue any command is provided by the Ubuntu Linux distribution on the

[9] https://gitlab.ihelp-project.eu/pkranas/ihelp-store

container, thus, he or she can navigate and monitor the status or the various logs of the installation of the Big Data Platform.

6.1.2 Remote installation using Kubernetes

As we have previously mentioned, at the very beginning of the project, the technical partners of the consortium had foreseen the need for a centralized deployment of the integrated iHelp solution in a common infrastructure. This means that there would have been the need for only one instance of the Big Data Platform to be deployed that would store all datasets coming from the different pilot use cases of the project, at the place where the infrastructure is available. A centralized deployment can be achieved by installing all different building blocks of the integrated iHelp solution one by one. As a result, the process described in the previous subsection with the use of a docker container would have been sufficient.

However, the fact that the data are related with clinical information of patient, which characterizes them as very sensitive, along with the EU and each country's regulations prohibit the movement of these data outside the country or even the organization that owns these data. As a result, the integrated iHelp solution needs to be deployed in different premises, following a peer approach. This increases the level of difficulty for deploying all building blocks and makes more complex its overall maintenance. It is now required by each technology provider to manually install and deploy his or her components to each pilot organization, apart from only having the centralized environment. To make things worse, it would be required for the technology providers to be granted access inside the network and infrastructure of each of these pilot organizations, which is not always feasible and usually requires a heavy bureaucracy procedure to be followed.

In order to tackle this problem, we have decided to force the use of a container orchestration tool to handle the deployment. All components that consist of the iHelp platform need to be containerized and be available as docker images. Then, Kubernetes, as the container orchestration tool, will be responsible to deploy all necessary components one by one and establish network connectivity among them. Having done this, each system administrator inside the organization can now follow a typical procedure and deploy everything on his or her own. In this subsection, we will give details on the steps that need to be followed in order to install and deploy the Big Data Platform using Kubernetes.

Firstly, as the Big Data Platform is a stateful component that needs to store data to a persistent storage medium, we will need to define a persistent volume claim that will be later mounted to the running container of the datastore. The following code snippet requests from the infrastructure 100GB of storage space to be available, so that it can be later mounted. We give the name of datastore-datasets-pvc to this persistent volume claim that will be later used by the datastore.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: datastore-datasets-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 100Gi
```

Secondly, in order for the Big Data Platform to be reachable by other components inside the network that will be established by the Kubernetes, we will also need to define a service that will be responsible for this. This service will expose the various ports that need to be accessible by other components. We can do this as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: ihelp-store-service
  labels:
    app: ihelp-store
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: ihelp-store
```

Here, we defined the *ihelp-store-service* kind that lists a number of ports that need to be opened and exposed, namely the 2181, 1529 that we saw previously, along with 9876, 9992, 14400 and 9800 that are needed in cases where the direct API of the datastore will be used instead of the standard JDBC. This is the cases for instance where an analytical tool will make use of the Apache Spark analytical processing framework, which in fact will make use of the direct Spark Connector, as we described in a previous section. An important thing to highlight at this code snippet is the definition of the selector application that will make use of this service, named *ihelp-store*. This will be the name of the actual deployment of the Big Data Platform that will see right next.

Having now the basic elements defined, our persistent volume claim and service, we now need to define the Big Data Platform deployment. The latter, as a stateful component, will make use of a Stateful Set. The reason for this is that it consists of various internal components that need to maintain some attributes like IP address etc, in cases of restarts. The following code snippet illustrates the definition of such a stateful set that can be used as a guideline for the deployments:

```
apiVersion: v1
kind: StatefulSet
metadata:
  name: ihelp-store
  labels:
```

```
    app: ihelp-store
spec:
  serviceName: ihelp-store-service
  replicas: 1
  selector:
    matchLabels:
      app: ihelp-store
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: ihelp-store
    spec:
      initContainers:
      - name: ihelp-store-home-fix
        image: busybox:1.30.1
        command: ["/bin/sh", "-c", "chown -R 999:999 /datasets"]
        volumeMounts:
          - name: datastore-datasets-storage
            mountPath: /lx/LX-DATA
      containers:
        - image: {name of the docker registry}/ihelp-store:latest
          name: infinistore
          ports:
            - containerPort: 2181
            - containerPort: 1529
            - containerPort: 9876
            - containerPort: 9992
            - containerPort: 14400
            - containerPort: 9800
          volumeMounts:
            - name: datastore-datasets-storage
              mountPath: /lx/LX-DATA
          startupProbe:
            exec:
              command:
                - /bin/sh
                - -c
                - python3 /lx/LX-BIN/scripts/lxManageNode.py check QE
            timeoutSeconds: 5
            failureThreshold: 30
            periodSeconds: 10
          resources:
            limits:
              cpu: 4000m
              memory: 8Gi
            requests:
              cpu: 2000m
              memory: 4Gi
          env:
          - name: USEIP
            value: "yes"
          - name: KVPEXTERNALIP
            value: "ihelp-store-service!9800"
      restartPolicy: Always
      imagePullSecrets:
        - name: registrysecret
      volumes:
        - name: datastore-datasets-storage
```

```
          persistentVolumeClaim:
            claimName: datastore-datasets-pvc
```

Here there are various things that need some special attention. Firstly, we gave the name *ihelp-store* to our stateful set, which must be the same as the selector application name in the definition of our service. Also, in our stateful set we reference this service giving its name, *ihelp-store-service*. Secondly, we did make use of the *datastore-datasets-pcv*, which we defined earlier, and we mounted this to the /lx/LX-DATA folder of the created container. The incoming data and all meta-information that needs to be persistently stored will be found. We give the resource requirements for our deployments, which in this example will require at minimum 2 virtual CPUs and 4GB of memory. These values can be further increased in case we need a deployment with more powerful resources, in cases we need distributed deployments that must be able to handle a significant volume of data. Last but not least, we need to provide the URL of the docker image that Kubernetes will make use in order to grab the docker image that we built earlier and create the corresponding container. We left this value generic in our example on purpose, as a centralized private docker registry has not been yet deployed for the needs of the project, when this document was firstly written.

Having these three elements applied using Kubernetes tools, we can now have a running deployment of the Big Data Platform of iHelp inside the organization's premises. The Big Data Platform however can only be accessed from within the internal network established by Kubernetes, and not from other tools or application deployed on premise, but outside the internal Kubernetes network. In case the data administrator wants to allow this, we would also need to define a nodeport that will expose a list of our desired ports outside the internal network. The following code snippet illustrates how to do this:

```
apiVersion: v1
kind: Service
metadata:
  name: ihelp-store-np
spec:
  type: NodePort
  selector:
    app: ihelp-store
  ports:
    - name: "1529"
      protocol: TCP
      port: 1529
      targetPort: 1529
      nodePort: 30201
```

Here, we expose the 1529 of the *ihelp-store* selected application to the outside 30201. This means that the data user or application developer will need to connect to the public URL of the Kubernetes deployment and target 30201 port. Then the Kubernetes will forward this request to the pod named *ihelp-store*, our Big Data Platform, to its internal 1529 port that has been opened by defining previously the corresponding service.

6.1.3 Examples of use

Having the Big Data Platform up and running, in these subsections we will provide examples of its use. We will focus on three different data connectivity mechanisms: using standard JDBC, using the direct API and using Apache Spark along with the provided Spark Connector. In the next versions of this deliverable,

additional examples will be given using Apache Kafka queues. As the integration with Kafka and the implementation of the corresponding data connectors are currently in progress, this information will be given in the second version.

6.1.3.1 Using standard JDBC

JDBC is a standard connectivity mechanism for java application, and as such, it is out of scope of this report to describe its use. It allows the data user or application developer to submit SQL statements via standard interfaces. The Big Data Platform is based on the LeanXcale datastore which supports an SQL dialect that is very similar to the one supported by PostGres. However, there is a complete documentation[10] for what is currently supported. As it is out of the scope of this report the complete description of how to connect using JDBC, we will only provide an indicative example in the following code snippet:

```
String connectionString = "jdbc:leanxcale://127.0.0.1:1529/IM;user=APP"
Properties props = new Properties();

try {
   Class.forName("com.leanxcale.client.Driver").newInstance();
   Log.info("Loaded the appropriate driver");
} catch (ClassNotFoundException | InstantiationException | IllegalAccessException cnfe) {
   cnfe.printStackTrace(System.err);
   throw cnfe;
}

try(Connection conn = DriverManager.getConnection(connectionString, props);
   PreparedStatement statement = conn.prepareStatement("SQL STATEMENT");) {
   ResultSet resultSet = statement.executeQuery();
   while(resultSet.next()) {
      //do stuff
   }
   resultSet.close();
}
```

6.1.3.2 Using the Direct API

The Direct API is a secondary data connectivity mechanism, which allows the data user or application developer to direct connect and interact with the storage engine of the Big Data Platform. The following code snippets will illustrate some examples of its use. Firstly, we need to open a session which will keep a connection open. The data user needs to define the connection url, the name of the logical database and the user credentials:

```
import com.leanxcale.kivi.database.Database;
import com.leanxcale.kivi.database.Index;
import com.leanxcale.kivi.database.Table;
import com.leanxcale.kivi.session.Session;
import com.leanxcale.kivi.session.SessionFactory;
import com.leanxcale.kivi.session.Credentials;
import com.leanxcale.kivi.session.Settings;

Settings settings = new Settings()
.credentials(user, pass, "tpch")

Session session = SessionFactory.newSession("kivi:lxis//lxserver:9876", settings);
Database database = session.database();
```

[10] https://docs.leanxcale.com/leanxcale/1.5/apis/sql-api.html

The result is an instance of the *database* that we need to interact. Now, let's try to add a new row (tuple). We assume we have defined a table named *Persons*. We need to create a tuple for this table, and interact with the table's interface to insert the new record, as the following code snippet illustrates:

```
Table people = database.getTable("person");
Tuple person = people.createTuple();

//Fill in tuple fields
person.putLong("id", 1L).
      .putString("name", "John")
      .putString("lastName", "Doe")
      .putString("phone", "555333695")
      .putString("email", "johndoe@nowhere.no")
      .putDate("birthday",Date.valueOf("1970-01-01"))
      .putInt("numChildren", 4);

//Insert tuple
people.insert(person);

//Tuples are sent to the datastore when COMMIT is done
session.commit();
```

In case the table has an auto-generated primary key, we need to get first the corresponding *sequence* that generates these values for us and use this instead. The previous code needs to be slightly changed as follows:

```
Table people = database.getTable("person");
Tuple person = people.createTuple();

long personId = database.getSequence("personId").nextVal();

person.putLong("id", personId)
      .putString("name", "John")
      .putString("lastName", "Doe")
      .putString("phone", "555333695")
      .putString("email", "johndoe@nowhere.no")
      .putDate("birthday",Date.valueOf("1970-01-01"))
      .putInt("numChildren", 4);

people.insert(person);
session.commit();
```

In order to read records from the database, there are two different options, whether we need to do a get operation over the primary key or perform a scan operation over the table. In case we need to get a tuple using the get operation over the primary key, we first need to define a *TupleKey* that will guide the storage engine to get the corresponding row, having this key, from the corresponding data table as follows:

```
import com.leanxcale.kivi.tuple.Tuple;
import com.leanxcale.kivi.tuple.TupleKey;

Table table = database.getTable("person");

Tuplekey key = table.createTupleKey();
key.putLong("id", 0L);

Tuple tuple = table.get(key);
```

In case we need to perform a scan, we can do it like this:

```
import static com.leanxcale.kivi.query.aggregation.Aggregations.*;
import static com.leanxcale.kivi.query.expression.Constants.*;
import static com.leanxcale.kivi.query.expression.Expressions.*;
import static com.leanxcale.kivi.query.filter.Filters.*;
import static com.leanxcale.kivi.query.projection.Projections.*;

Table people = database.getTable("person");
TupleKey min = people.createTupleKey();
min.putLong("id", 20);

TupleKey max = people.createTupleKey();
max.putLong("id", 30L);

people.find()
        .min(min)
        .max(max)
        .foreach(tuple->processTuple(tuple));

// Max 20 results
people.find()
        .first(20)
        .foreach(tuple->processTuple(tuple));
```

Here, we performed a scan over a primary key. We first defined the boundaries of the range that we need to scan, by defining two *TupleKey* objects, and then we invoked the find operation using those two keys. It is important to highlight that the actual retrieval of the data will happen when the foreach method will be invoked.

Now, the following examples show how to define a set of filters that will be pushed to the storage engine of the datastore to get a result matching a set of conditions.

```
import static com.leanxcale.kivi.query.aggregation.Aggregations.*;
import static com.leanxcale.kivi.query.expression.Constants.*;
import static com.leanxcale.kivi.query.expression.Expressions.*;
import static com.leanxcale.kivi.query.filter.Filters.*;
import static com.leanxcale.kivi.query.projection.Projections.*;

Table people = database.getTable("person");

// Basic comparisons
people.find()
      .filter(gt("numChildren", 4).and(eq("name", string("John"))))
      .foreach(tuple->processTuple(tuple));

// Between
Date minDate = Date.valueOf("1900-01-01");
Date maxDate = Date.valueOf("2000-01-01");

people.find()
      .filter(between("birthday", date(minDate), date(maxDate))
      .foreach(tuple->processTuple(tuple));

// Using a Expression with operators
people.find()
      .filter(gt("numChildren",sub(field("numRooms"),int(1))))
      .foreach(tuple->processTuple(tuple));
```

Then, the concept of the projection is the set of fields from the table that you want to get from the table. The project may also include operations over the fields to be retrieved:

```
import static com.leanxcale.kivi.query.aggregation.Aggregations.*;
import static com.leanxcale.kivi.query.expression.Constants.*;
import static com.leanxcale.kivi.query.expression.Expressions.*;
import static com.leanxcale.kivi.query.filter.Filters.*;
import static com.leanxcale.kivi.query.projection.Projections.*;

Table people = database.getTable("person");

// Basic inclusion
people.find()
      .project(include(asList("name", "lastName", "birthdate")))
      .foreach(tuple->{
            String name = tuple.getString("name");
            String lastname = tuple.getString("lastName");
            Date date = tuple.getDate("birthdate");
      });

// SELECT name, Lastname, weight/height^2 AS imc FROM person
// Alias and expression usage
people.find()
      .project(compose(Arrays.asList(
        alias("name"),
        alias("lastName"),
        alias("imc",div(field("weight"),pow(field("height"),2)))
        )))
      .foreach(tuple->{
        String name = tuple.getString("name");
        String lastName = tuple.getString("lastName");
        float imc = getFloat("imc");
});
```

Finally, the Direct API provides a rich interface that allows the data user or application developer to perform aggregation operators over a data table. We need to define a first array with the fields to be used as the group by key (if any), and then a list of aggregation expressions over some fields, as the following code snippet illustrates:

```
import static com.leanxcale.kivi.query.aggregation.Aggregations.*;
import static com.leanxcale.kivi.query.expression.Constants.*;
import static com.leanxcale.kivi.query.expression.Expressions.*;
import static com.leanxcale.kivi.query.filter.Filters.*;
import static com.leanxcale.kivi.query.projection.Projections.*;

// Simple aggregation
int numPeople =
    people.find()
      .aggregate(emptyList(), count("numPeople"))
      .iterator.next().getLong("numPeople");

// Group By aggregation
people.find()
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height"),2)))
```

```
        ))
      .foreach(tuple->{
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });

// Filtering before aggregate
Date date = Date.valueOf("1970-01-01");

people.find()
      .filter(gt("birthdate", date(date)))
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height),2)))
        ))
      .foreach(tuple->{
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });

//Filtering after aggregate
people.find()
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height),2)))
        ))
      .filter(gt("averageIMC",float(26)).or(ge("numPeople",100)))
      .foreach(tuple->{
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });
```

6.1.3.3 Using Apache Spark

The Big Data Platform provides additional means for data connectivity, having been integrated with popular frameworks used by data analysts or data engineers. One of the most common analytical processing frameworks is the one provided by Apache Spark. This allows the data analysts to create data frame from a target datastore and then apply relational algebraic operations using Spark's native interface. Having a relational datastore, as the Big Data Platform, one can use a standard JDBC mechanism to connect to the datastore, submit relational queries that Spark pushes them down to the database and retrieves the results. Having however our direct API, Spark can avoid using the relational query engine of the Big Data Platform, and instead, connecting directly to the storage engine of the integrated solution. In order to do this, the data user or application developer will additionally need to import the corresponding Spark Connector implementation, which serves as the bridge between the direct API and the Spark processing framework itself. In fact, the Spark Connector implements the Spark's interface for driving Spark on how to use the API. However, from the data user or application developer point of view, everything is being done transparently for him or her. It is the implementation of the Spark Connector that is being provided by this task and the

Big Data Platform that does all the necessary work. The following code snippet illustrates the use of Apache Spark with the Direct API:

```
import com.leanxcale.spark.LeanxcaleDataSource;
import java.io.IOException;
import java.util.List;
import org.apache.spark.SparkConf;
import org.apache.spark.sql.Column;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.functions;


public class TestSpark {
    public static final String SPARK_APP = "sparkApp";
    public static final String CONNECTION_URL = "
kivi:lxis//lxserver:9876/tpch@APP;KVPROXY=datastore!9800";
    public static final String TABLE_NAME_SIMULATION =  "PERSONS";

    public static void main(String [] args) throws IOException {
        SparkConf conf = new SparkConf().setAppName(SPARK_APP).setMaster("local[*]");

        SparkSession spark = SparkSession.builder()
                .appName(SPARK_APP)
                .config("spark.master", "local")
                .config(conf)
                .getOrCreate();

        //connect and get load from table " PERSONS "
        Dataset<Row> simulationDF = spark.read()
            .format(LeanxcaleDataSource.SHORT_NAME)
            .option(LeanxcaleDataSource.CONNECTION_PROPERTIES, CONNECTION_URL)
            .option(LeanxcaleDataSource.TABLE_PARAM, TABLE_NAME_SIMULATION)
            .load()
            .repartition(5);

        //print the schema and show data
        simulationDF.printSchema();
        simulationDF.show();

        //do a scan and project the column PERSON_NAME
        System.out.println("Testing");
        Dataset<Row> result = simulationDF.select(functions.col("PERSON_NAME"));
        result.printSchema();
        result.show();
    }
}
```

In this code snippet, we create a Spark session and then we define our data frame. It is important to mention here that the *format* of the data frame will not be “JDBC”, but instead, the name of our Spark Connector, as it has been highlighted. Spark will use Java reflection to instantiate the corresponding class, which means that the connector must be available in the classpath of the java process. Then, we put the corresponding parameters as the connection URL, the table name etc. Finally, we use the Spark’s interface for do a scan over the table and project the corresponding columns that we are interested.

6.2 Kafka Broker

One of the requirements for the design of the *data ingestion pipelines* of the overall integrated iHelp solution, documented in D3.4 – "Primary data capture and ingestion II", is the use of the *service choreography* pattern. This requires the developed functions and microservices to be loosely coupled and exchange messages via data queues. As a result, it was decided to use Kafka broker to provide these types of data queues to be used by the functions and microservices involved in the data ingestion pipelines.

On the other hand, as it has been mentioned in subsection 2.3, the Big Data Platform itself comes with a variety of different connectors to popular tools and frameworks. One of those is the Kafka, by providing a specific Kafka connector, allowing the direct ingestion of data coming from a specific topic to the corresponding data table using the Big Data Platform's direct API. This could facilitate the data ingestions developed and provided by the work that has been carried out under the scope of T3.2 – "Primary Data Capture and Ingestion". At the end of such pipelines, it is requested that the data coming as HHR resources to be transformed to data beans or POJOs relevant with the equivalent relational schema of the HHR, as described in a previous section. Those POJOs must now be persistently stored to the Big Data Platform. Instead of having to develop a new functionality that would take care of opening and maintaining data base connections, and store the data in an efficient way, we decided to take advantage of the Kafka connector of the Big Data Platform, and instead, serialize and send these POJOs to a specific Kafka topic, letting the connector itself to take care of the final ingestion to the datastore. This has been also described in the *HHR Importer,* documented in the aforementioned D3.4 – "Primary data capture and ingestion II".

As a result, it was finally decided that the delivery of the Kafka broker would be the responsibility of this T4.4 – "Big Data Platform and Knowledge Management System", as the integrated iHelp solution does not only need the common distribution of the Kafka broker, but instead, an enhanced distribution that contains the connector to the Big Data Platform. Due to this, a demonstration of its installation, deployment and use is included in this document.

6.2.1 Local installation using docker

Firstly, the released distribution for the iHelp project has been uploaded to the project's private Gitlab[11]. As explained in the previous subsection, each member of the consortium should have access to this code repository and can issue the following command, adding his or her username and password in order to download the Kafka Broker locally:

```
git clone https://gitlab.ihelp-project.eu/pkranas/ihelp-kafka.git
```

This GitLab project contains all binaries, scripts and additional configuration files, along with a Dockerfile, which allows the data administrator to build a *docker* image locally that can be later used to deploy and install the Kafka Broker in a containerized environment. In order to build the image locally, having already cloned the GitLab project, the data administrator needs to execute the following command:

```
docker build –t ihelp-store .
```

[11] https://gitlab.ihelp-project.eu/pkranas/ihelp-store

This will take some time as the Dockerfile is using the base Ubuntu 20.04 image and needs to pre-installed firstly various packages that are necessary for the Kafka broker to run. At the end of the process, the docker image is built and available locally at the machine where this command was issued. In order to deploy and run the Kafka Broker, we would also need an instance of the Big Data Platform to be previously deployed, as our enhanced Kafka would try to connect to that instance. Due to this, we will make use of the *docker compose* that allows us to create an environment with 2 or more containers and establish network connectivity among them.

For this, we have created a docker-compose.yml file that will drive the deployment of both the big data platform and our enhanced Kafka broker. This configuration is depicted in the following code snippet

```
version: '3.1'

services:
  ihelp-store-service:
    image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-store:latest
    container_name: ihelp-store-service
    restart: unless-stopped
    ports:
      - 9876:9876
      - 9992:9992
      - 14400:14400
      - 1529:1529
      - 8900:8900
    environment:
      - KVPEXTERNALIP=ihelp-store-service!9800
      - USEIP=yes
  ihelp-kafka:
    image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-kafka:latest
    container_name: ihelp-kafka
    restart: unless-stopped
    ports:
      - 8081:8081
      - 9092:9092
    depends_on:
      - ihelp-store-service
    links:
      - ihelp-store-service
    environment:
      - advertised_url=192.168.2.5
      - advertised_port=9092
```

The important thing to be highlighted is the environment variable of the *ihelp-kafka*. When connecting to Kafka, the latter advertises to its client a list of its deployed brokers, so that the client may choose where to connect. This includes their URL, and as a fact, we need to explicitly define this *advertised url*. For local installations and deployments, in the configuration file, the system administrator needs to put there the IP of the machine where the deployment will take place.

Now the data administrator needs to issue the following command:

```
docker-compose up
```

This command will create the two containers that will start in the background, giving them the names *ihelp-store-service* and *ihelp-kafka*, while exposing the corresponding ports. For Kafka, the 9092 is used to

connect to its brokers, while 8081 is used to connect to the Apache Avro Registry that is included in the container. The data administrator can also connect to the container from a command line tool, by executing the following:

```
docker exec –it ihelp-kafka bash
```

This command will open an interactive secure shell that will remain opened and will execute the *bash* script. The latter will allow him or her to issue any command is provided by the Ubuntu Linux distribution on the container, thus, he or she can navigate and monitor the status or the various logs of the installation of the Kafka broker.

6.2.2 Remote installation using Kubernetes

As already explained in a previous subsection, apart from a centralized deployment of the integrated iHelp solution, due to data sensitivity and privacy constraints, we identified that there will be the need for installations and deployments of the iHelp solution to the premises of each of the hospital or clinical organization of the project. Due to this, we would need our solution to be portable, so that the system administrators and operation managers can be facilitated to do this process. As a result, we decided to make use of Kubernetes for the container orchestration system in order to automate the software deployment, scaling, and management.

To do so, firstly we would need to create the *service* kind, in order for Kubernetes to attach a network to the container that will host Kafka, so that the latter can be accessible by other components. The following code snippet illustrates how we can define this:

```
apiVersion: v1
kind: Service
metadata:
  name: ihelp-kafka-service
  labels:
    app: ihelp-kafka
spec:
  ports:
    - name: "9092"
      port: 9092
      targetPort: 9092
    - name: "8081"
      port: 8081
      targetPort: 8081
  selector:
    app: ihelp-kafka
```

We need to pay attention here that our *selector* will be named as *ihelp-kafka,* which will be the name of the container that we will create soon after. We exposed the ports 9092 and 8081 for the Kafka broker and the Avro Schema Registry.

Now, we need also to allow access from outside the cluster, and for this, we will create a *node port* for this container, as the following code snippet illustrates:

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: ihelp-kafka-np
spec:
  type: NodePort
  selector:
    app: ihelp-kafka
  ports:
    - name: "9092"
      protocol: TCP
      port: 9092
      targetPort: 9092
      nodePort: 30003
    - name: "8081"
      protocol: TCP
      port: 8081
      targetPort: 8081
      nodePort: 30004
```

Same as before, our *selector* is named *ihelp-kafka,* that will be the name of the container that we will create soon after, and we map the external ports 30004 and 30003 to the Avro Schema Registry (listening to port 8081) and to the Kafka Broker (listening to port 9092).

Now, before the definition for the deployment of our Kafka broker, we would need to define some properties that will be injected to the container as environment variables. In order to do this, we create a configuration mapping, known as *configmap* in Kubernetes terminology, as the following code snippet illustrates:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: ihelp-kafka-configmap
data:
  advertised.url: 147.102.230.182
  advertised.port: "30003"
```

Here, we need to define the environment variables that were previously defined in the docker-compose.yml file for local installations. In our example, we set the url to 147.102.230.182, which is the external IP of our Kubernetes cluster. Moreover, we set the port to 30003, which is the port that is exposed to the outside, as defined previously in our *node port.*

Finally, now we are ready to define our *Stateful Set* for the deployment of our enhanced Kafka. This is defined as follows:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: ihelp-kafka
  labels:
    app: ihelp-kafka
spec:
  serviceName: ihelp-kafka-service
  replicas: 1
  selector:
    matchLabels:
      app: ihelp-kafka
  updateStrategy:
```

```
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: ihelp-kafka
    spec:
      containers:
        - image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-kafka:latest
          name: ihelp-kafka
          ports:
            - containerPort: 8081
            - containerPort: 9092
          resources:
            limits:
              cpu: 2000m
              memory: 2Gi
            requests:
              cpu: 1000m
              memory: 1Gi
          env:
            - name: advertised_url
              valueFrom:
                configMapKeyRef:
                  name: ihelp-kafka-configmap
                  key: advertised.url
            - name: advertised_port
              valueFrom:
                configMapKeyRef:
                  name: ihelp-kafka-configmap
                  key: advertised.port
      restartPolicy: Always
      imagePullSecrets:
        - name: regcred
```

The important thing to be highlighted here is the definition of the environment variables. For instance, we define the environment variable called *advertised_url,* whose value will be retrieved from the previously defined *configmap,* and more precisely, from the one that we named ihelp-kafka-configmap, as of the previous example, using the value of the attribute called *advertised.url,* which in our example is now 147.102.230.182.

Our Kafka broker is now up and running at this URL and we can now start experimenting by sending messages to its topics and let the connector to store them transparently to the Big Data Platform. In order to do so, we need to explicitly guide our connector on how to do so, using a set of configuration files, one per each data table, as the next subsection will explain in more details.

6.2.3 Configuring the Kafka Connector

Inside the container/pod of the Kafka connector, there is a java process that takes the role of the Kafka connector to the big data platform. It will listen to data items coming to a specific topic and will store them to a data table inside the big data platform. Upon initialization, this process will read 1 or more configuration files that will guide it how to do this. In the integrated iHelp solution, we need HHR resources to be stored to specific tables. For that, we have defined the relational schema to be equivalent with the HHR E-R model, as explained in a previous section.

The HHR relational schema contains several data tables, each one of those has a list of columns and primary keys. In this subsection, we will document how we can configure our connector to put data to each one of those tables.

Patient

For the entity *patient,* we will use the following configuration:

```
name=hhr-patient
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=patient

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=patient
pk.mode=record_key
pk.fields=patientid
fields.whitelist=gender, active, birthdate
```

Here we define that our connector will create a new thread called *hhr-patient* that will make the LXSinkConnector class, to connect. It will listen for data items in the topic called *patients* and will rely on the AvroConverter to deserialize the key and values of the data items placed under the topic *patients*. We further need to provide the connection URL for the connector to know how to connect to the Big Data Platform. This is the connection URL that the direct API will make use. We need to bear in mind here that we used the *ihelp-store-servive* as the URL, which is the one we defined earlier using Kubernetes. Data coming from this topic will be stored into the table called *patient,* whose primary key is the column *patientid*, while it contains three additional columns: gender, active and birthdate. Finally, we won't use transactions here, as we only append new records, and we make use of the *upsert* instead of the *insert* option. This allows us to update the records: newly added records will replace the old records that contain the same primary key.

Condition

For the entity *condition,* we will use the following configuration:

```
name=hhr-condition
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=condition

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
```

```
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=condition
pk.mode=record_key
pk.fields=conditionid
fields.whitelist=abatementdatetime, onsetdate, category, encounterid, practitionerid, patientid
```

Encounter

For the entity *encounter,* we will use the following configuration:

```
name=hhr-encounter
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=encounter

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=encounter
pk.mode=record_key
pk.fields=encounterid
fields.whitelist=status, periodstart, periodend, typesystem, typecode, typedisplay, serviceproviderid,
practitionerid, patientid
```

Measurement

For the entity *measurement,* we will use the following configuration:

```
name=hhr-measurement
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=measurement

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
```

```
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=measurement
pk.mode=record_key
pk.fields=measurementid
fields.whitelist=status, measurementcode, measurementsystem, measurementdisplay, effectivedatetime,
measurementcategorycode, measurementcategorysystem, measurementcategorydisplay, valuesystem, valuecode,
valueunit, valuequantity, valuesystemlow, valuecodelow, valueunitlow, valuequantitylow,
valuesystemhigh, valuecodehigh, valueunithigh, valuequantityhigh, encounterid, practitionerid,
patientid
```

Medication

For the entity *medication,* we will use the following configuration:

```
name=hhr-medication
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=medication

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=medication
pk.mode=record_key
pk.fields=medicationid
fields.whitelist=status, medicationsystem, medicationcode, medicationdisplay, effectiveperiodstart,
effectiveperiodend, dosage, encounterid, practitionerid, patientid
```

Observation

For the entity *observation,* we will use the following configuration:

```
name=hhr-observation
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
```

```
topics=observation

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=observation
pk.mode=record_key
pk.fields=observationid
fields.whitelist=status, observationcode, observationsystem, observationdisplay, effectivedatetime,
observationcategorycode, observationcategorysystem, observationcategorydisplay, valuesystem, valuecode,
valueunit, valuequantity, valuesystemlow, valuecodelow, valueunitlow, valuequantitylow,
valuesystemhigh, valuecodehigh, valueunithigh, valuequantityhigh, encounterid, practitionerid,
patientid
```

Procedure

For the entity *procedure,* we will use the following configuration:

```
name=hhr-procedure
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=procedure

connection.properties=lx://ihelp-store-service:9776/ihelp@APP;KVPROXY=ihelp-store-service!9800
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

insert.mode=upsert

table.name.format=procedure
pk.mode=record_key
pk.fields=procedureid
fields.whitelist=status, procedurecode, proceduresystem, proceduredisplay, procedurecategorycode,
procedurecategorysystem, procedurecategorydisplay, patientid
```

6.3 Big Data Platform microservices

As it has been already described in a previous section, apart from the delivery of the Big Data Platform of the iHelp integrated solution, along with the Kafka Broker enhanced with the data connector of the datastore, the role of this task T4.4 ("Big Data Platform and Knowledge Management System") is also to provide some additional functionality that is required by other components of the platform. Such functionalities have been developed as independent microservices that implement a set of REST APIs to be accessible. In this subsection will give information about how to install locally or remotely using Kubernetes.

6.3.1 iHelp REST Interface

This microservice exposes a set of REST APIs to the DSS Suite, and most precisely to its SQL Node Palette. This will allow the end-user to define using a graphical interface algebraic relational operations, combine them and finally produce a complicated SQL statement, without actually having to by a data user.

6.3.1.1 Local installation using docker

Firstly, the released distribution for the iHelp project has been uploaded to the project's private Gitlab[12]. As explained in the previous subsection, each member of the consortium should have access to this code repository and can issue the following command, adding his or her username and password in order to download the iHelp rest locally:

```
git clone https://gitlab.ihelp-project.eu/pkranas/ihelp-store-rest.git
```

This GitLab project contains the source code and various scripts and additional configurations files. The source code has been developed in Java and the user would additionally need the installation of the *maven* software project management and comprehension tool. Having everything in place, the user firstly needs to compile the source code using the following command:

```
mvn clean install
```

Maven will download all dependencies, compile the source code a create a *fat jar* container all dependencies. Having done that, we should make use of the provided Dockerfile, which allows the data administrator to build a *docker* image locally that can be later used to deploy and install the microservice in a containerized environment. In order to build the image locally, having already complied the GitLab project, the data administrator needs to execute the following command:

```
docker build –t ihelp-store-rest .
```

This will take some time as the Dockerfile is using the base Ubuntu 20.04 image and needs to pre-installed firstly various packages that are necessary for microservice to run. At the end of the process, the docker image is built and available locally at the machine where this command was issued. In order to deploy and run the microservice, we would also need an instance of the Big Data Platform to be previously deployed, as our microservice would try to connect to that instance. Due to this, we will make use of the *docker*

[12] https://gitlab.ihelp-project.eu/pkranas/ihelp-store

compose that allows us to create an environment with 2 or more containers and establish network connectivity among them.

For this, we have created a docker-compose.yml file that will drive the deployment of both the big data platform and our microservice. This configuration is depicted in the following code snippet

```
version: '3.1'

services:
  datastore:
    image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-store:latest
    container_name: datastore
    restart: unless-stopped
    ports:
      - 2181:2181
      - 1529:1529
    environment:
      - KVPEXTERNALIP=datastore!9800
  backend:
    image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-store-rest:test
    container_name: datastore-rest
    restart: unless-stopped
    ports:
      - 54735:54735
    depends_on:
      - datastore
    links:
      - datastore
    environment:
      - DATASTORE_HOST=datastore
      - swagger_path=/tmp/
      - USEIP=yes
```

Now the data administrator needs to issue the following command:

```
docker-compose up
```

This command will create the two containers that will start in the background, giving them the names *datastore* and *datastore-rest*, while exposing the corresponding ports. The data administrator can also connect to the container from a command line tool, by executing the following:

```
docker exec –it ihelp-store-rest bash
```

This command will open an interactive secure shell that will remain opened and will execute the *bash* script. The latter will allow him or her to issue any command is provided by the Ubuntu Linux distribution on the container, thus, he or she can navigate and monitor the status or the various logs of the installation of the microservice.

6.3.1.2 Remote installation using Kubernetes

The use of Kubernetes was decided to be necessary, as it will facilitate the portability of our integrated solution. For this, we first need to create the network that will be attached to the container, using the following code snippet:

```
apiVersion: v1
kind: Service
metadata:
  name: ihelp-store-rest-service
  labels:
    app: ihelp-store-rest
spec:
  ports:
    - name: "54735"
      port: 54735
      targetPort: 54735
  selector:
    app: ihelp-store-rest
```

As always, we need this microservice to be accessible from outside of the cluster, so we will also need to create a *node port* for this:

```
apiVersion: v1
kind: Service
metadata:
  name: ihelp-store-rest-np
spec:
  type: NodePort
  selector:
    app: ihelp-store-rest
  ports:
    - name: "54735"
      protocol: TCP
      port: 54735
      targetPort: 54735
      nodePort: 30005
```

Here, we will map the external port 30005 of our cluster to the internal one, 54735, which we have exposed with our *service.* Finally, the following configuration will create our container. As our microservice is stateless, we will use a *Deployment* kind, instead of a *Stateful Set* that we were using before:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ihelp-store-rest
  labels:
    app: ihelp-store-rest
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ihelp-store-rest
  template:
    metadata:
      labels:
        app: ihelp-store-rest
```

```
  spec:
    containers:
      - image: gitlab.ihelp-project.eu:5050/pkranas/ihelp-store-rest:latest
        imagePullPolicy: Always
        name: ihelp-store-rest
        env:
         - name: DATASTORE_HOST
           value: "ihelp-store-service"
         - name: swagger_path
           value: "/tmp/"
        ports:
          - containerPort: 54735
        resources:
          limits:
            cpu: 500m
            memory: 512Mi
          requests:
            cpu: 500m
            memory: 512Mi
    restartPolicy: Always
    imagePullSecrets:
      - name: regcred
```

7 Next Steps and Roadmap

The main objective of this report is to present the current status of the work that has been carried out in the scope of the T4.4 – "Big Data Platform and Knowledge Management System" at this phase of the project (M22) and to give technical details about the background technology that has been adopted, its advancements and the design of the overall solution related with the Big Data Platform of the iHelp integrated solution. This section describes the next steps and open issues that currently are under investigation, and they will be tackled and reported in the final version of this deliverable.

Firstly, the definition of the common data model of the iHelp took place while this document was initially written and is currently further developed and extended. The HHR is being developed under the scope of T3.1 – "Data Modelling and Integrated Health Records". This provides the *entity-relational* (E-R) conceptual model of the holistic health records, which are the raw data that will be stored into the datastore. After the delivery of its first version, the role of the T4.4 – "Big Data Platform and Knowledge Management System" was to define the corresponding relational schema in the database, which will store the actual data. Regarding the current version, this has been documented in the respective section of this document. The relational schema will be updated in the forthcoming period of the project, following the outcomes of the work that is currently being done under the scope of T3.1 – "Data Modelling and Integrated Health Records" and it will be reported in the final version of this document. At this final phase, the corresponding data ingestion functions will be also extended to support the final version of the HHR relational schema. As described in the D3.4 – "Primary data capture and ingestion II", these are responsible to transform data coming as HHR resources, into the corresponding relational entities that will be pushed to the Kafka data queue. From there, the Kafka connector transparently adds the data into the Big Data Platform, as explained in the previous section. The need for this functionality has been defined under the corresponding deliverables of the T3.2 – "Primary Data Capture and Ingestion", however, it is the responsibility of T4.4 – "Big Data Platform and Knowledge Management System" to provide the Kafka broker and its configuration, as explained in the previous section.

Secondly, another important activity that currently takes place is the integration of the Big Data Platform with the Analytical Workbench, which is being developed under the scope of T4.2 – "Model Library: Implementation and Recalibration of Adaptive Models". The latter provides the runtime execution environment that all analytical AI algorithms and models will run and use to exchange data with the datastore. The Analytical Workbench uses internally a database to store meta-information related with the execution of the models and the models themselves. The objective is to replace its internal database with the Big Data Platform of the overall iHelp solution.

Another important objective of the T4.4 – "Big Data Platform and Knowledge Management System" is to provide the necessary functionalities for the data visualization components of the DSS suite and other components developed under the scope of WP5 – "AI for Early Risk Assessment and Personalised Recommendations". At this phase of the project, we have developed a set of microservices to the *SQL Node Pallet* of the DSS to use and for the latter to be able to retrieve base raw information (primary data) of a selected patient. Moreover, we have designed and define the interfaces for the *user enrolment* scenario. For the next period, its implementation will take place, along the design and implementation of all other microservices needed to provide the functionalities for the remaining scenarios: the i) communication of risk, the ii) risk mitigation/treatment planning, the iii) plan a visit or control, (iv) advice review and (v) risk

mitigation. The implementation of these scenarios took place during the first half of the project, and now, during its second half, the integration with the overall iHelp solution is planned to take place. Therefore, this integration requires the development of this series of microservices under the scope of this task that will be documented in the last version of this deliverable.

Moreover, following the need for visualizing the results of a polyglot query that targets different peers of Big Data Platform deployments, the corresponding query operator needs to be implemented. At this phase, the consortium has foreseen the need to combine data coming from different sources and the prohibition for moving sensitive data outside of an organization. The result is for T4.4 – "Big Data Platform and Knowledge Management System" to propose the solution described in a previous section. The overall design and definition of the programming interfaces has been started, and in the forthcoming period, the implementation of the corresponding operators and its validation will follow, if there is such a need from a pilot use case.

Finally, after the delivery of the first prototype of the iHelp integrated solution along with the Big Data Platform and the validation of its functionality, the next focus can be the provision of a secure access to the data user. Regarding the security aspects, there are two ways for secure access:

- Encrypt the data transmission over the network
- Encrypt the data themselves when stored in the database

Regarding the first aspect, this is currently supported by the Big Data Platform. A broker mechanism in order to discover other peers and automatically handle the corresponding tokens when connecting to an external datastore might be needed, however this is not yet a requirement for the Big Data Platform. Regarding the second aspect, this is currently under validation: we test that the query processing over encrypted and non-encrypted data return always equivalent results. The second phase of the validation will be to benchmark the performance overhead when querying data that are encrypted in the disk. Currently, as data will be stayed inside the premise of the organization, it seems that this functionality is not of a high priority, so the focus has been given in other activities of this task.

8 Conclusions

This deliverable reports the work that has been currently done under the scope of the T4.4 – "Big Data Platform and Knowledge Management System" at this phase of the project (M22). The main objective of this task is to provide a big data platform to efficiently manage the data generated by different sources and provide the ground for the envisioned HHRs and iHelp analytics. Towards this direction, we relied on the background technology of the LeanXcale database, which offers a list of innovative characteristics. First, it allows for hybrid transactional and analytical processing that allows firstly data ingestion on very high rates and secondly for performing data analytics over the same dataset. This removes the need to migrate data to different analytical datastores and thus allows to perform analytics over the operational data. The Big Data Platform also provides a parallel OLAP engine that will be exploited by the data analytics.

In the scope of the iHelp, the background technology is being further developed to support the needs of the integrated solution. Firstly, it has defined a first version of the relational schema of the HHR records that will be used to import raw data coming from the pilot use cases, transformed to the conceptual common model. After being ingested, the parallel engine and advanced capabilities of its internal storage, will allow the data analytic tools to efficiently retrieve and process data stored into the database. What is more, the Big Data Platform provides various needs for data connectivity and is being integrated with popular frameworks used by data analysts and data engineers. For instance, it has been integrated with Apache Kafka that will be used in various cases in the iHelp platform. Regarding the data ingestion, data analytical functions deployed into the data ingestion pipelines communicate using Kafka queues. At the end, data will be pushed to a data queue that will make use of the Kafka connector of the Big Data Platform to transparently store data to the database. Moreover, the Big Data Platform is being integrated with Apache Spark, the dominant analytical processing framework often used by many data analysts. The Spark connector of the Big Data Platform allows for efficient data retrieval, bypassing and removing the integral overhead of the footprint of its query engine.

Another important feature of the Big Data Platform necessary for the needs of the project, is its polyglot query processing capabilities. This allows the data user to submit a query statement to the database, that will be further sent to an external data source in order to retrieve data from a remote data provider. The Big Data Platform then will combine and join the results and return them back to the user via the single opened connection. Having the restriction from the data owners not to move data outside of their institutions and the requirement of the project to combine data from various sources, this feature is a catalyst and enables the combination of data without moving them. We envisioned peer deployments of the integrated iHelp solution, where each instance of the Big Data Platform can remotely connect to another and grab aggregated information, without actually moving the raw data themselves. Then, it can combine the aggregated results retrieved remotely with the internal data. This can be used by analytical tools to get more insights by using aggregated datasets.

What is more, during this second phase of the project, additional activities needed to be performed by this task, in order to support the integration of the other components of the overall iHelp platform with the Big Data Platform. As a result, a set of different functionalities have been designed and implemented, as microservices, that will allow the communication of the currently delivered components with the datastore. Towards this direction, a microservice that provides a REST API to the SQL Node Pallet of the DSS Suite has been delivered, while currently under development is a novel microservice responsible to support the *user*

enrolment process. Additional microservices are planned to be designed and implemented during the last phase of the project and will be documented in the last version of this deliverable.

After a successful demonstration of the outcomes of this task during the first Project Review, we decided to include an additional section in this deliverable that can serve the purposes of a *demonstrator*, giving guidelines and examples on how to install and deploy the artefacts of this tasks, both in a local environment for testing purposes, or in a distributed environment with the use of Kubernetes. The use of Kubernetes facilitate the portability of our solution that is necessary, as we need to deploy the overall iHelp integrated solution to the premises of each of the hospital or clinical organizations of the project.

The progress of the task at this phase is as planned, with the design of the Big Data Platform delivered, along with the implementation of its prototype, the definition of the HHR relational schema and the design and implementation of a set of additional microservices, while we have identified the goals and next steps that need to be done in the next phase. This document also reports a demonstrator of its use, having examples on how to deploy and install all involved components locally, or in a virtualized environment using container orchestrator tools like Kubernetes. Then, examples of its use have been given, based on different data connectivity mechanisms: JBDC, the Direct API and Apache Spark. The next version of this document is planned to be released in M32 and will include additional information regarding the implementation of the next items identified in our roadmap that is presented in this document.

Bibliography

P. Kranas, B. Kolev, O. Levchenko, E. Pacitti, P. Valduriez, R. Jiménez-Peris, and M. Patiño-Martinez, “Parallel query processing in a polystore.”, Distributed and Parallel Databases (2021): 1-39.

List of Acronyms

2PL	2 Phase Locking
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Interface
CPU	Central Processing Unit
DQE	Distributed Query Engine
DSS	Decision Support System
ETL	Extract, Transform, Load
EU	European Union
HDFS	Hadoop Distributed FileSystem
HHR	Holistic Health Record
HTAP	Hybrid Transactional Analytical Processing
I/O	Input/Output
JDBC	Java Database Connectivity
KOD	KODAR Systems
LXS	LeanXcale
NoSQL	Non Structured Query Language
ODBC	Open Database Connectivity
OLAP	Online Analytical Processing
OLTP	Online Transactional Processing
REST	Representational State Transfer
SQL	Structured Query Language
TCP	Transmission Control Protocol
UI	User Interface
UPM	Universidad Politécnica de Madrid
UPRC	University of Piraeus Research Centre
URL	Uniform Resource Locator
WP	Work Package