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

D3.2 - Primary data capture and ingestion I

WP3 Personalized Holistic Health Records

Dissemination Level: Public
Document type: Report
Version: 2.0
Date: October 29, 2021

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	Primary Data Capture and Ingestion I
Work package	WP3
Due Date	October 31, 2021
Submission Date	October 29, 2021
Start Date of Project	January 1, 2021
Duration of project	36 months
Main Responsible Partner	LXS
Deliverable nature	Report
Authors' names	Pavlos Kranas (LXS), Patricio Martínez (LXS), Javier Pereira (LXS), Luis Miguel Garcia (LXS), George Manias (UPRC), Eleftheria Kouremenou (UPRC), Marta Patiño (UPM), Ainhoa Azqueta Alzúaz (UPM)
Reviewers' names	Harm op den Akker (iSPRINT), Usman Wajid (ICE)

Document Revision History

Version History			
Version	**Date**	**Author(s)**	**Changes made**
0.1	2021-09-06	Pavlos Kranas (LXS)	Initial ToC
0.2	2021-09-07	Pavlos Kranas (LXS), Patricio Martínez (LXS), Javier Pereira (LXS), Luis Miguel Garcia (LXS), George Manias (UPRC), Ainhoa Azqueta Alzúaz (UPM)	Input in sections 1, 2, 3, 5
0.3	2021-09-09	George Manias (UPRC), Ainhoa Azqueta Alzúaz (UPM)	Input in section 4
0.4	2021-09-14	Eleftheria Kouremenou (UPRC), Ainhoa Azqueta Alzúaz, Marta Patiño (UPM)	Input in section 5
0.5	2021-09-14	Pavlos Kranas (LXS)	Finalizing the document. Submitted for internal review
1.0	2021-09-22	Harm op den Akker (iSPRINT)	Internal Review
1.1	2021-10-25	Usman Wajid (ICE)	Internal Review

2.0	2021-10-26	Pavlos Kranas (LXS)	Finalize and submit the document

Table of Contents

Executive summary 5
1 Introduction..... 6
1.1 Objectives of this deliverable 6
1.2 Insights from other tasks and deliverables..... 6
1.3 Structure 6
2 Overview of the Data Capture and Ingestion Pipeline 8
3 Data Ingestion Gateway 13
3.1 Overview..... 13
3.2 Basic Principles 15
3.3 Initial Design 16
3.4 Interface 18
4 Data Ingestion Functions..... 21
4.1 Requirements 21
4.1.1 Domain/Schema Agnostic Functions..... 21
4.1.2 Domain/Schema Specific Functions 22
4.2 Data Cleaner 23
4.3 Data Qualifier 23
4.4 Data Harmonizer..... 24
4.5 Raw-to-HHR Converter 24
4.6 HHR Data Importer 24
5 Deployment of Data Ingestion Pipeline 26
5.1 Fully automated and dynamic deployments 27
5.2 Fully automated and dynamic deployments using a serverless platform..... 27
5.3 Configurable static deployments..... 28
5.4 Static deployments..... 28
5.5 Planning for deployment in the future steps 29
6 Conclusions..... 31
List of Acronyms 33

Table of Figures

Figure 1: Conceptual Data Capture and Ingestion Pipeline. 10
Figure 2: Data Ingestion Gateway Overview. 13
Figure 3: Gateway Class Diagram. 16

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 (HHR). As such, the software components that make up the integrated platform of iHelp can be categorized in three major layers: (1) those components that are involved with the business intelligence – using artificial intelligence algorithms that consume data from the data management layer, (2) the building blocks that make up the data management layer itself and provide the runtime environment for these analytics to be executed along with central data repository, and (3) the building blocks that are responsible for capturing data from the external sources, and eventually ingest it to the central data repository after applying various functions for data quality assurance during the data ingestion process.

Regarding the third category of software components, which is the one related with the ingestion of data, there can be identified the need to ingest both primary and secondary data. The distinction between those two is that primary data concerns static information of clinical data, while the secondary data accumulates information that has been initially captured during the runtime and is available to the iHelp platform after a pre-processing. This deliverable reports on the work that has been carried out for the activities related with Task 3.2: “Primary Data Capture and Ingestion”. These activities concern the capture of data from the external sources (either primary data that will need to be captured directly from an external source, or secondary data that will be captured by the output of the corresponding activities of the Task 3.3: “Secondary Data Extraction and Interoperability”. The capture of these two types of data contributes towards the establishment of the data ingestion pipelines so that the data can be eventually stored into the big data platform, converted to the common data model provided by the activities of the Task 3.1: “Data Modelling and Integrated Health Records”. Other functions implemented as part of the data ingestion pipeline are related with the data quality assurance that are researched and realized in the scope of Task 3.4: “Standardisation and Quality Assurance of Heterogeneous Data”, and reported in D3.7: “Standardisation and Quality Assurance of Heterogenous Data I”.

This deliverable reports on the work carried out under Task 3.2: “Primary Data Capture and Ingestion”, and as such, it will only focus on its objectives. We start by providing the overview of data pipelines, and then it will drill down to the details of the design and implementation of its three main aspects that they concern. Firstly, the software components related with the data capture, secondly on the requirements that the involved data ingestion functions must comply with, and finally, how the deployment and establishment of such data pipelines will take place.

As this is the first version of this report, at this phase of the project the initial design of the involved components will be provided, along with different design approaches for the overall integration of the components that will be involved in the data capture and ingestion pipelines. Detailed discussions about the benefits and drawbacks of each of the proposed designs will be given, along with the requirements that each design brings along. The second version of this document will include the decisions about which approach will be followed, along with the final design of the involved components towards their updated implementation.

1 Introduction

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 task T3.2 ("Primary Data Capture and Ingestion") 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).

1.1 Objectives of this deliverable

The objective of this deliverable is to report on the work that has been currently done under the scope of the task T3.2 ("Primary Data Capture and Ingestion") at this phase of the project. The main focus of this work at this early phase was to provide the basic design principles of the data capture and ingestion pipelines and the initial design and requirements for the development and implementation of all the involved components. This includes the components related with the data capture and also the ones related with the data ingestion functions. Having the initial design and requirements, this document can be used as the basis for the implementation of these (data ingestion pipeline related) functionalities. Finally, different types of deployments have been identified at this phase with the objective to define a plan with next steps for the overall deployment of the data pipelines.

1.2 Insights from other tasks and deliverables

The work that has been carried out under the scope of the task T3.2 ("Primary Data Capture and Ingestion") has dependencies with the activities of WP2 and WP4 and the other tasks of WP3. More specifically, it gets input from task T2.1 ("Requirements, State of the Art Analysis and User Scenarios in iHELP") and has strong dependencies with T2.2 ("Reference Architecture Specifications") and T2.3 ("Functional and Non-Functional Specifications"). Moreover, it considers T3.3 ("Secondary Data Extraction and Interoperability") as an external type of data source. What is more, the majority of the data ingestion functions that are involved in the data pipelines are implemented and provided under the scope of T3.4 ("Standardisation and Quality Assurance of Heterogeneous Data"). The *HHR Importer* function that will be implemented under this task, will rely on the specification of the common data model, the HHR, that will be defined under the scope of T3.1 ("Data Modelling and Integrated Health Records"), and will store data into the relational data schema in the Big Data Platform of the iHelp platform, provided by the T4.4 ("Big Data Platform and Knowledge Management System"). Finally, it will be investigated at a later phase whether or not the serverless platform currently developed under the scope of T4.2 ("Model Library: Implementation and Recalibration of Adaptive Models") will be used for the deployment of the data pipelines, however, the design of the data capture and ingestion functions have been designed in order to support the requirements for such type of deployments.

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 data capture and ingestion pipelines, discussing its scope and basic principle concepts of the design. Then the next sections discuss

the three main aspects of the data pipelines: Section 3 covers the data capture aspect focusing on the gateway component, Section 4 focuses on the data ingestion functions while Section 5 focuses on the different deployment scenarios for establishing the data pipelines. Finally, Section 6 concludes the document.

2 Overview of the Data Capture and Ingestion Pipeline

One of the most important technological building blocks of the iHelp platform is the data pipeline that captures the various types of data and ingests the information into the big data management platform of the iHelp platform. This section provides the overview of such data pipelines and provide the basic design decisions and principles of their implementation. To start, we need to distinguish among the different aspects related with the data pipelines. On one hand, there are the software components that take care of the actual capture of the data that resign in different sources, such as external database management systems, exported files or data that are stored in external systems or applications that can be retrievable via well-defined application programming interfaces (APIs). Secondly, there are the software components or functions that do a pre-processing or transformation and are part of the data ingestion pipeline. These implement a specific business logic that can be packaged into a function or micro-service, and are used to enhance or transform the raw data that has been initially pushed to the deployed data pipeline by the data capture components. They either provide data quality assurance by cleaning or harmonizing the ingested datasets, or transform the latter to the common HHR data model. Finally, the third aspect of the data capture and ingestion pipeline is the interaction and interconnectivity of the aforementioned software components in a way that can actually deploy and establish such data pipeline. These can be either static or can be dynamically deployed during run-time, while at the same time, different types of data pipelines might need to involve different types of data ingestion functionalities in different steps of the pipeline. This section will give the overview of these concepts, while the next sections will dive into more details of these three different aspects.

Regarding the software components related with the data capture aspects, there is the need to access, retrieve and finally push raw data into the data pipeline. The raw data in the eHealth domain can be in different structures (i.e. a CSV file which contains comma separated values, a relational database where each data item can be stored in a separate data tables, JSON files, etc) and in different data schemas. Moreover, there are two types of data: primary and secondary data. With the notion of primary data, we identify static information that is mostly related with clinical data. This can be historical data of patients coming from their electronic patient records, research studies, official treatment guidelines across different European countries etc. On the other hand, with the notion of secondary data, we identify dynamic information that can be gathered and is mostly related with the behavioural activities of patients that can be collected via external sources or devices such as wearables and IoT platforms such as Fitbit, or by collecting data from citizens via the newly developed mobile app that will be implemented in the scope of the iHelp project. The actual capture of the secondary data is part of the activities related with task T3.3 (“Secondary Data Ingestion and Interoperability”) and will not be reported in this report. From what concerns the activities of T3.2 (“Primary data capture and ingestion”), the secondary data are considered as an additional external source. As a result, it should be transparent from the deployed ingestion pipelines whether or not the data are related with the primary or the secondary category and therefore, the data capture related software components that are being reported here will have to deal with both types of data: primary data being retrieved from their sources or secondary data that have been initially captured by external devices or applications and their accumulated information will be further captured by the components described in this document.

Regarding the software components or functions that will formulate the data ingestion pipelines, we also distinguish them in two major categories: The ones that are domain/schema agnostic and those that are

not. An additional type of function is the one that is related with the transformation of the raw data into the common data model, the HHR. However, this can be considered as a subcategory of the domain/schema specific functionalities. This categorization is important as it will define whether each data function will be placed inside the deployed data ingestion pipeline. The reason for this is evident from the different schemas of the raw data. As it has been described, various data sources can be retrieved from various sources, and each of the provided dataset can has its own schema, which is valid as the data capture components need to access data that can be related with clinician information stored in electronic health record that can contain the values of the vital organs of a patient, or data related with socio-demographic, medication adherence or other behavioural data of a person. As a matter of fact, it is evident that all these aforementioned type of dataset are heterogeneous and therefore they will be compliant with a different data schema.

Let's see now how schema agnostic or specific functions are usually implemented and why there is the need to be placed in a different position inside a formulated data pipeline. Schema agnostic functions or microservices can be applied in whatever type of an incoming dataset. An example can be a rule-based cleaning algorithm that checks if the values of specific metrics are valid or dirty. For instance, such a rule could be that the value of the heartbeat of a person must always be positive. To generalize, the algorithm can check the value of a specific column of a dataset against a pre-defined rule (i.e. column A must include numeric values that are always positive). The implementation of such functionality can accept as input parameters the list of rules along with the schema of the dataset that it will be applied to. Upon receiving each data item, it can check for the specific value in the raw data based on the definition of its schema, and after retrieving the specific data element (i.e. the column in our example) it can check if the rule is valid or not, which will lead to the acceptance or rejection of the specific column or the data item as a whole. This implementation can be considered as domain/schema agnostic and can be applied to each of the aforementioned datasets, such as the electronic health records that contain primary data related with the clinicians, socio-demographic information of the related persons etc.

On the other hand, an implementation of a function that provides quality assurance might be more sophisticated and might need to correlate information from different columns over a specific period of time. For instance, there might be a user scenario that requires the identification of possible failures of a sensor that collects information from patients. Let us consider the following example: When the systolic and diastolic blood pressures of persons are high, then it is expected that the heartbeat will be also high. As a fact, the function that implements the quality assurance will have to check that if a specific person has high blood pressures for more than a minute, his or her heartbeat will have to start increasing. If it continues giving low values, then it might be an indication that the corresponding sensor is malfunctioning. For such an implementation, there might be the need to know the schema of the incoming dataset in order to be able to retrieve the corresponding information to feed the AI analytical algorithm. As a result, it cannot be applied to whatever dataset that contains raw data, as opposed to our previous example.

In order to cope with these aspects, the iHelp project defines a common data model that will be used: the Holistic Health Record (HHR) that is being defined under the scope of T3.1 ("Data Modelling and Integrated Health Records"). Having a common data model in place will allow for all data analytical algorithms, which includes those involved in the data ingestion pipelines, to rely on a common schema. Therefore, they can be implemented once and make use of the global data model of iHelp, instead of

having the need to implement such functionalities differently for each incoming dataset. With iHelp's integrated health records that defines the HHR, each data analyst or data developer can know beforehand which is the schema and can have a single implementation for all types of incoming datasets. However, it is required that there must be a data transformation of the incoming datasets that contains raw data to the common data model of iHelp. These functionalities are domain specific and need to be written for each of the supported datasets.

Having provided the basic principles of the components that are related with the data capture and ingestion pipelines, the following figure depicts the overview conceptual diagram of our solution.

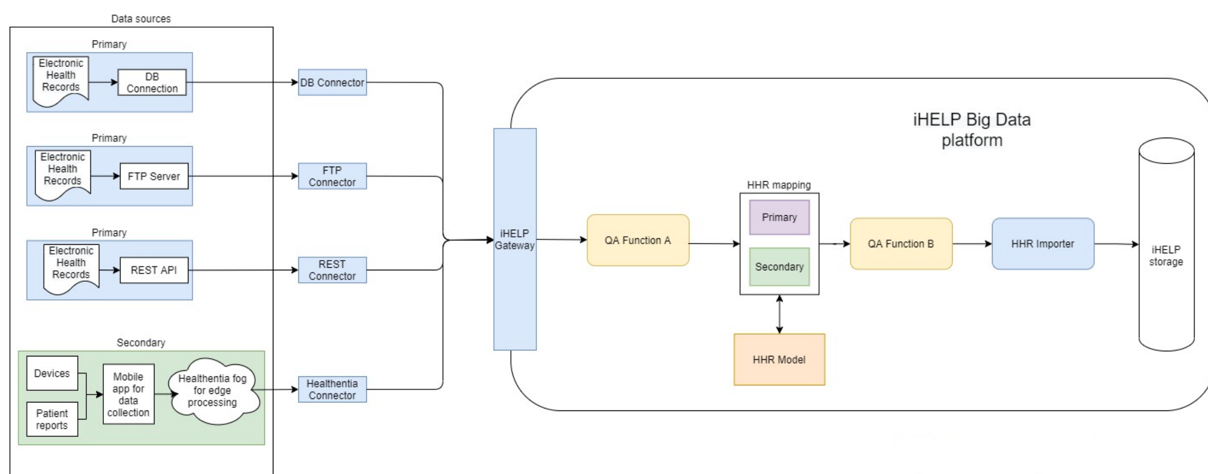

Figure 1: Conceptual Data Capture and Ingestion Pipeline.

We can divide Figure 1 in two parts: The part that is placed left to the iHelp Gateway and can be considered as external to the integrated solution, and the part that is placed right to the Gateway, and can be considered as part of the iHelp Big Data Platform itself. In the left part, we can see the various types of data sources that contain the raw data that need to be ingested inside the iHelp platform and eventually reach the persistent storage. In Figure 1, some examples of data sources are listed that might be supported through the data ingestion pipeline. Regarding the primary data sources, there might be related to clinician info that has been stored into electronic health records and being provided by different types of sources: i) from inside a data management system such as a relational database, ii) from csv files that can be placed into an FTP server, or iii) from a well-defined application programming interface. Regarding the secondary data, these may be collected through wearable devices or IoT platforms and will be primarily collected by a the iHelp mobile application that will be implemented during the scope of the project, under the activities of task T3.3 ("Secondary Data Ingestion and Interoperability"). This raw data will be further processed locally by the Healthentia platform (on which the iHelp mobile application will run) and the newly developed AI algorithms that will be implemented under the scope of the WP5, in order to provide anonymized and consolidated information. This information will be accessible via a REST API which will be requested this secondary data.

The iHelp Gateway is the frontier of the overall integrated solution and is responsible for capturing both types of data, primary raw data or secondary data after their initial pre-processing by the Healthentia application. The gateway implements all functionalities related to the data capture aspects of the

integrated data pipelines and it must provide different types of connectors to each of the supported storage mediums. As can be seen in Figure 1, there is a need to connect to a relational datastore, an FTP server and some web services that expose a REST API. As a result, different types of connectors will be available from the Gateway to be used in order to establish data connectivity with the different types of sources. It is important to highlight that it is irrelevant from the Gateway perspective whether the data is characterized as primary or secondary, as it will be treated equally. In both cases where there is the need to consume REST web services, the data connectivity mechanism will be the same. This is the reason why in this report we provide a holistic approach for capturing data that will be further ingested by the data ingestion pipeline, and leave the focus on how to primarily capture the secondary data in T3.3.

Regarding the functionalities that will formulate the data ingestion pipelines, they are considered as internal to the overall integrated solution of the iHelp platform, and have been placed right next to the gateway. An important category of such functions are the ones that are responsible for the transformation of the raw data to the common data model. In Figure 1, they are listed as primary and secondary mappers. These mappers will take input from the previous functions in a raw data format and by consulting the definition of the common HHR data model, they will convert the raw data to HHR data entities. As the big data repository is using a relational schema that is compliant with the HHR conceptual model, it is an obligation for the data pipeline to contain one function of this category. Each supported dataset will require the implementation of a dataset specific HHR mapper that will be responsible for the data conversion task.

Between the Gateway and the HHR Mappers, there are the functions that implement algorithms that are domain/schema agnostic. These functions can take as an input raw data being ingested in the data pipeline by the Gateway and they will send as an output the same raw data with respect to their initial schema. Therefore, an important requirement is that they do not change the schema of the raw data, as the initial schema will be important for the HHR Mappers. As it has been described, these functions do not need to know about the schema of the data that they are being applied to and might only need to accept some input parameters in order to appropriately instantiate themselves so that they can be in place to know how to treat the incoming data.

Moreover, after the transformation of the raw data to the common HHR model that has been done by the corresponding HHR Mappers, there will be deployed all the functions that implement algorithms that are domain/schema specific. As it has been described, they must rely on the common HHR data model and they must accept data items transformed into this schema and send as an output data also compliant with the HHR. As they will be placed inside the established data pipeline after the HHR Mappers, this ensures that they will always receive data objects that are compliant with the HHR and therefore, they do not need to accept any schema specific information to instantiate themselves, apart from function specific parameters.

Finally, the last part of the deployed data pipelines will always be the HHR Importer, whose responsibility is to accept data items in the HHR common data model and store this information to the datastore. This will hide all internal complexities of the implementing database specific data connectivity mechanisms, and as a matter of fact, it can accept data items in the same fashion as all the aforementioned functions and place them into the Big Data Platform to persistently store them.

Last but not least, there are aspects related with the deployment of the data capture and ingestion pipelines and their establishments and interconnectivity during the runtime. At this phase of the project, it has been decided that the basic design principle will be the service choreography[1], which allows for loosely-coupled microservices that can communicate with each other according to the data pipeline they are involved in. The data will be exchanged using Kafka queues. This means that after the initial capture of the raw data from the Gateway, the data items will be placed inside a specific Kafka topic. Then, each of the data functions involved into the data pipeline that needs to be established will read data from one Kafka topic and will place the output into another Kafka topic, so that the next involved function can retrieve it. At the end of the data pipeline, the HHR importer will consume data from a specific data topic in a similar way, and will connect to the Big Data Platform to persistently store the data. At this phase of the project, it has not been decided yet whether we make use of static microservices that listen to pre-defined Kafka topics, or we will take decide for dynamic deployments of such data pipelines, probably taking advantage of the serverless platform provided under the scope of T4.2 ("Model Library: Implementation and Recalibration of Adaptive Models"). Section 5 will give more details about the deployment aspects of the data capture and ingestion pipelines.

[1] https://en.wikipedia.org/wiki/Service_choreography

3 Data Ingestion Gateway

As T3.2 ("Primary Data Capture and Ingestion") deals with data capture and ingestion pipelines, the work that needs to be carried out under its scope is to first implement the software components that are related with the data capture aspects, to support the data ingestion functionalities and to finally deploy and establish the integrated data pipelines. This section will give information about the data capture aspects of such pipelines that will be provided by the Data Ingestion Gateway. It will first provide the overview of this component and then will report about the basic principles. As the work that is being carried out under the scope of task T3.2 is currently in progress, the initial design of our solution will be discussed along with the programming interfaces that have been foreseen to be necessary at this early phase. In the second version of this report, these subsections will be further extended to cover all updated aspects, design decisions and implementation details.

3.1 Overview

The Data Ingestion Gateway is the component that can be considered as the barrier between the iHelp integrated solution and the external data sources, from which it will capture the data to be pushed into the established data ingestion pipeline. At this phase of the project, it will be a standalone Java process that will take care of connecting to the various external data sources and send the data to an intermediate Kafka topic, so that the data can be retrievable from the later functions that consist of each of the data ingestion pipeline. It will also provide a REST API which will be used in order to initiate data capture activities, or schedule them for a later or a periodic execution. The REST API will be deployed into a servlet container, however it will make use of the core functionalities of the Data Ingestion Gateway and therefore, both the REST API and the code implementation will be inside the single Java process. A high-level overview of the different software elements of this initial design can be depicted in Figure 2.

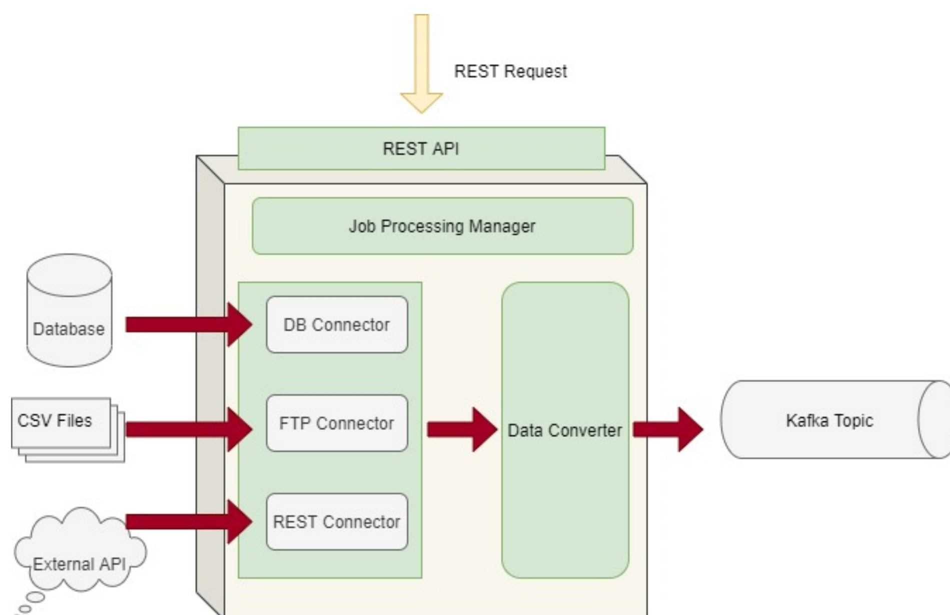

Figure 2: Data Ingestion Gateway Overview.

Figure 2 illustrates the components that the Data Ingestion Gateway interacts with. As the data capture element of the iHelp integrated solution, it will interact directly with all supported external data sources. At this phase of the project, three different means for data connection have been identified: (1) CSV files

uploaded to an FTP server, (2) a relational database and (3) external REST APIs. As additional types of data sources might be identified and needed during the scope of the project, the design is not limited to those three but rather can be extendable to support other means for connection. An example for instance could be the download of data from a file that has been uploaded into a Google drive or needs to be fetched from a versioning system or a git repository.

As the Data Ingestion Gateway captures data from the supported sources, it will forward them into a common Kafka topic to be retrieved by the functions involved in the established data pipeline. As it has been described in the previous section, all data functions that are involved in such data pipelines they will inter-exchange data from. Therefore, a requirement for the gateway is to be able to push data items in a topic. As the first functions of the pipelines are considered domain/schema agnostic, a single topic can be used where all these functions can listen to and retrieve data. However, this implies that the data sent to this topic must additionally contain their corresponding schema, so that the data functions can be aware and execute their internal algorithms respectively. This might be expensive in terms of the data that needs to be transmitted over the network, as every batch of data items must include (and thus replicate) the corresponding data schema. An alternative approach is to serialize the data that will be transmitted and sent to the Kafka topic via Avro. This will require the use of an Avro Schema Registry that allows to only transmit the amount of bytes that concerns the data itself, thus minimizing the overall size of the data elements. With such transmission, only the identifier of the schema is being passed along with the data and the deserializer can retrieve the original schema from the Avro Schema Registry and cache it locally. However, this would require all data functions involved in the data pipelines to be able to (de)serialize data items in such a manner, and this would require an additional overhead for their implementation. Due to this, it will be decided in a later phase of the project if such approach will be applied. Another alternative is for the gateway to send data to different topics and each one of those will be responsible to store data items of a specific schema. By doing this, each analytical data ingestion function can be instantiated with the schema definition of each dataset and the topic that needs to listen to. Accordingly, the gateway can accept such an information while starting the data ingestion process, so that it can forward the captured data to the corresponding Kafka topic. However, these design options have not been decided yet at this early phase of the project, and the next version of this document will be extended to include such information. By all means, whatever the design decision that will be agreed, it does not break the general applicability of the gateway.

The Data Ingestion Gateway itself consists of 4 main software elements: the REST API, the Job Processing Manager, its connectors and the converter. The REST API is the software module that allows to programmatically configure the gateway to start, stop or schedule a data ingestion pipeline. It can accept HTTP/HTTPs requests from an external application, or it can be used by a front-end element to facilitate its use from the data provider. However, the provision of such a front-end is out of the scope of the task T3.2, so no additional information will be given. The Job Processing Manager module keeps track of all submitted jobs to be executed. This will be the main point of interaction with the REST API layer, whose purpose is to submit and maintain jobs for execution. Then a job consists of the execution of the actual data capture, and makes use of the other two software modules of the gateway. The connector module encapsulates the details for establishing the data connectivity with the external resources, and forwards data to the corresponding converter. Then the latter, retrieves the data from the connectors, without the actual knowledge of what how each connector managed to capture the raw data items, and then, after

putting them in a common data placeholder, it implements the logic for sending raw data to the corresponding Kafka topic, so that the next data functions involved in the established data pipeline can access and process them.

3.2 Basic Principles

For the initial design of the data ingestion gateway, the following basic principles have been taken into consideration:

- All end-user functionalities need to be exposed via a REST API. The end-user functionalities are the ones that are related to the submission of a job for ingesting data, its scheduling to be executed periodically or in a latter time, its monitoring and the ability to stop already submitted jobs. Therefore, the end-user should be able to check ingestions that are currently taking place and all scheduled ones so that he or she cancel them later, if there such a need.
- There must be an abstraction between the software elements that are responsible to send data items/rows to the kafka topic and how these elements are being originally captured by the external source.
- Different connectors need different types of attributes/arguments that they need to know in order to instantiate themselves accordingly.
- Each connector must encapsulate the details on how it establishes data connectivity with the external sources and should only need to pass a generic data item to the corresponding converters.
- A data item that is being passed to a converter must be agnostic to the source where the datum has been originally retrieved.
- As a consequence of the above, the schema of the dataset that is going to be retrieved must be defined, and must be passed along the instances of the converters and the connectors.
- Source code should be maintained in separate classes and not replicated among different ones.
- There will be only one implementation of the data converter, that will be schema agnostic and will serve the needs for all different types of dataset schemas and connectivity mechanisms to external sources. However, it might be later decided to have different implementations according to specific types of datasets. The design must be easily extendable and allow for polymorphism in the terms of object-oriented programming, so that it will not break.
- Data items that will be passed to a Kafka queue will be in JSON format, having a String as the identifier of the data item. Later on this might change and the data items might be required to be serialized using the Avro Schema Registry. The design should allow to easily change the corresponding implementation.
- Data ingestions must be executed in parallel and one must not block the other. Therefore a separate thread should take the responsibility to instantiated the relevant classes and software elements and execute the corresponding code.
- At this phase of the project, the gateway must be a single standalone Java process that uses multi-threading in order to keep track and execute the corresponding jobs in parallel. However, at a later phase it might be the requirement for the gateway to be used in a Function-as-a-Service fashion, and this function will be submitted for execution from an underlying serverless platform that is currently being developed under the scope of the task T4.2. Therefore, the whole design must be adaptable to be used in such a way with no changes. This drove the previous

requirement that each job must be executed in a separate thread and design of the source code must isolate the part that will be used by the serverless function.

3.3 Initial Design

Having taken into consideration the basic principles that were presented in the previous subsection, the initial design of the data ingestion gateway has been agreed. Its class diagram is depicted in the following Figure 3.

Figure 3: Gateway Class Diagram.

A basic requirement at this phase is that the end-user functionality must be exposed via a REST API. Therefore, the frontier class is the RESTResource that implements the web service with its web methods that can be invoked programmatically or via a REST client. This class is packaged in the REST web module of the whole solution that will be deployed in a servlet container. All other classes are part of the gateway backend and will be used by this module. The RESTResource implements the functionalities for starting/stopping a job, schedule its execution for a later phase and monitoring their status. It makes use of a reference to the JobManager.

The JobManager keeps track of all jobs that are currently being executed or have been scheduled for execution. It is a singleton, therefore only one instance of this class can be instantiated. It makes use of the Job, which contains all information that is related with the execution of a data ingestion process, along with meta-information such as its time of submission, its current status, etc. Internally, it places the Jobs inside a local cache implemented by a HashMap, and additionally keeps track of the schedulers for periodic job submissions, so that it can provide administrator actions to the end user. All its public functionalities are being consumed by the RESTResource that exposes them through REST to the end-user.

The Job is a central class in the design, as it contains all information required for the execution of a job, like the name of the dataset and the definition of its schema, the type of the connector that must be used, along with other connector or converter specific information. As a result, it is composed of both the ConverterParams and ConnectorParams. It must be noted at this point that each connector expects different types of input parameters, therefore the ConnectorParams is an abstract class that can be extended to support different types of connectors. Apart from the input arguments related with the execution of a Job, the latter also contains information about the actual execution; therefore it holds a reference to the JobExecutor instance that is responsible for establishing, starting, monitoring and gracefully finishing the overall execution of the data ingestion process. When a job is being submitted to the gateway, a new instance of the JobExecutor is being created in a separate thread. The JobExecutor implements the Java *Callable* interface, so when a new thread is created, it returns back a Future object that allows for monitoring the execution of the corresponding code. That way, this object can know when the job has finished or failed, in order to update the corresponding status of the job that is related to. In case of scheduled executions, the Java ScheduledExecutorService that implements the ExecutorService is used, and it is added to the list of executors of the JobManager. Therefore, the end-user can stop such a scheduling via the the JobManager's related method, exposed via the RESTResource, that has access the scheduler. Moreover, each time that the latter starts a new thread, the thread executes the same logic as it has been called to execute the job once: it creates a new Callable thread of the JobExecutor, retrieves the corresponding Future object and places it to the HashMap of the JobManager, so that the latter can monitor its activity.

The JobExecutor contains all arguments related with the execution of the data ingestion process. It creates an instance of the Converter and the corresponding instance of the Connector and invokes the latter to start capturing data from the external data source that is related with. Both objects share a common BlockingQueue object to communicate. The data items captured by the connector are being stored in a DataRowItem object that will be put into the queue, so that the converter can retrieve them from there. When the connector finishes capturing the data, it places a zombie object in the queue that will signal the converter that the data ingestion has finished, and it can now gracefully shut down itself. Upon initialization of the converter, the latter instantiates an instance of the KafkaConnector, whose role is to send JSON objects to a specific Kafka topic. Therefore, we separated the code responsible for converting data into those JSON objects, taken into account the schema of the corresponding dataset, with the code responsible for establishing a connection to the Kafka queue and handle the data transmission of these objects. At a later phase, the KafkaConnector could be extended in order to make use of the Avro Schema Registry and serialize the data items accordingly before transmitting them to the network. Therefore, our design is clean in the sense that there will be no change in the source code, apart from extending the KafkaConnector class that will be used instead. Finally, it is important for the connector to send the zombie object in the queue when the data capture is finished, so that the converter can be notified and shutdown its KafkaConnector instance that will further close the connections to Kafka in order for the JobExecutor to gracefully release its resources.

Last but not least, an important requirement is for the data ingestion gateway to be used as a Function-as-a-Service at later phase. Our design allows for that as the whole source responsible for the execution of the data capture is encapsulated under the JobExecutor, which expects a JSON object with the data capture parameters. Therefore, our implementation can be used as it is by the underlying serverless

platform. The latter will now have to instantiate (usually via Java Reflection) the JobExecutor class and pass the JSON object in its constructor. In case the instantiation and pass of parameters is being done differently, a separate method can be implemented in the JobExecutor that will take the input parameters, instantiate itself and kick off the data capture process.

3.4 Interface

This subsection provides the end-user interface that will be exposed via the REST interface, according to the requirements that have been decided at this early phase of the project. As all other functionalities of the data ingestion gateway are internal to the gateway itself and not of interest for the data providers that are the end-users of this component, only the REST web methods will be given here. However, the parameters of these methods will be the same as if the gateway was invoked in a Function-as-a-Service manner. Moreover, as the implementation of the gateway has started recently and it is yet under development, while it has not been validated by an end-user, the input and output parameters are subject to be changed or extended and will be reported at the next version of this deliverable.

The following web methods define the interface of the data ingestion gateway:

- POST: It submits a job to immediately start a data capture process. It accepts a JSON body similar to the code snippet that will be listed below:
- POST: /schedule: It submits a job to schedule a data capture process. The JSON body for both web methods should be the following:

```
{
  "dataset": "DatasetXYZ",
  "connectorType": "REST/DB/FTP",
  "topic": "topic name, or can be ommited in case of the default",
  "schema": "the schema of the dataset in an avro compatible format. This must be a JSON
object",
  "arguments": "the connector specific arguments. This must be a JSON object",
  "schedule": {
    "future": {
      "time": 2,
      "unit": "hours"
    },
    "periodic": {
      "time": 1,
      "unit": "week"
    }
  }
}
```

The body message of this requests must contain the name of the dataset, the type of the connector to be used, the schema definition of the dataset in an Avro format and the connector specific arguments to guide the connector how and where to connect to the external data source. The name of the topic can be explicitly provided, if there is the need to use of different Kafka topics per datasets, or it can be omitted in case there will be a single topic where all data ingestion functions must listen to. In case there is a scheduled job that has been submitted, the schedule JSON element must be included with optionally the future point in time when the process should start and optionally, the period of time for the job to be periodically executed.

The arguments element of the inputs parameter is relevant to the type of the connector that will be used for the data capturing process. As the connectors are currently being implemented, these arguments have not been finalized yet. The following code snippet provides an indicative example of what has been foreseen to be expected by the REST connector:

```
{
  "domain": "http://rest-domain-app:8080",
  "method": "retrieve",
  "type": "GET/POST",
  "queryParams": {
    "key1": "value1",
    "key2": "value2"
  },
  "body": "a JSON that contains the body request of a POST invocation"
}
```

Regarding the schema of the dataset, this should be provided in an Avro Schema compatible format, in order to boost the interoperability and have a well-known standard to be further used by other functions involved in the data ingestion process. An example can be the following:

```
{
  "namespace" : "my.com.ns",
  "name": "myrecord",
  "type" :  "record",
  "fields" : [
     {"name": "uid", "type": "int"},
     {"name": "somefield", "type": "string"},
     {"name": "options", "type": {
        "type": "array",
        "items": {
            "type": "record",
            "name": "lvl2_record",
            "fields": [
                {"name": "item1_lvl2", "type": "string"},
                {"name": "item2_lvl2", "type": {
                    "type": "array",
                    "items": {
                        "type": "record",
                        "name": "lvl3_record",
                        "fields": [
                            {"name": "item1_lvl3", "type": "string"},
                            {"name": "item2_lvl3", "type": "string"}
                        ]
                    }
                }}
            ]
        }
     }}
  ]
}
```

Finally, the response of the POST invocation depends on the type of data capture process has been requested. In the case of a request to immediate start the process, the response will be the JSON object related with this job, as the following code snippet indicates:

```
{
  "id": "UUID-XXX-XXX-XXX",
```

```
  "submitted" : "1970-01-01 00:00:00",
  "finished" : "1970-01-01 00:00:00",
  "status": "STARTED/FINISHED/FAILED",
  "message": "error message",
  "dataset": "DatasetXYZ",
  "connectorType" : "REST/DB/FTP"
}
```


In case of a scheduled invocation, there is no submitted job yet, and the response will be information relevant to the scheduler itself, as the following code snippet indicates:


```
{
  "id": "UUID-XXX-XXX-XXX",
  "submitted": "1970-01-01 00:00:00",
  "dataset": "DatasetXYZ",
  "connectorType": "REST/DB/FTP",
  "schedule": {
    "future": {
      "time": 2,
      "unit": "hours"
    },
    "periodic": {
      "time": 1,
      "unit": "week"
    }
  }
}
```


- GET: returns all submitted jobs. The result will be a list of all submitted jobs
- GET: /?ud=UUID-XXX-XXX: returns information for the job with the given id. It has the same information as the one returned above, with the addition of the input arguments of the data schema and the connector specific ones.
- GET: /stop?id=UUID-XXX-XXX-XXX: stops a currently running job, according to its id. It returns the same element as above.
- GET: /schedule: it returns a list of all scheduled jobs
- GET: /schedule?id=UUID-XXX-XXX: it returns information the scheduler with the given id. It has the same information as the aforementioned, with the addition of the input arguments of the data schema and the connector specific ones
- GET: /schedule/stop?id=UUID-XXX-XXX: stops a currently submitted scheduler, according to its given id. It returns the JSON object related with the scheduler.

4 Data Ingestion Functions

The data capture and ingestion pipelines consist of the data capture part that has been covered in the previous section, and the data ingestion part that will be covered in this section. The data ingestion part consists of the data ingestion functions that are being involved after the data has been captured and pushed to the now created data pipeline.

It is important to be highlighted that the implementation and provision of the data ingestion functions of these data pipelines are not the responsibility of the task T3.2 ("Primary Data Capture and Ingestion") and that this document reports the work that has been carried out under its scope, therefore design decisions and implementation details will be given at the corresponding deliverables. However task T3.2 is responsible for designing the data ingestion process and establish the corresponding data pipelines, therefore, in this section will provide information about the requirements that all involved data ingestion functions must cover, along with a brief explanation of the functionality of each of these functions.

4.1 Requirements

As mentioned in Section 2, the data ingestion functions can be categorized as follows: domain/schema agnostic, domain/schema specific and the HHR Mappers that are responsible to transform raw data into the HHR common data model. Moreover, even if at this early phase we target static deployments of the involved functionalities in order to accelerate the development process and validate their implementations, it is not clear at this point whether or not the iHelp integrated solution will rely on a serverless platform to be used in order to automate a dynamic deployment and establishment of these data ingestion pipelines. To make things worse, as this serverless platform is being currently developed under the scope of T4.2 and its first prototype will be reported at a later phase, it is also unclear what types of requirements or restrictions will be put by the platform. Another important uncertainty at this early phase is the way that the involved data ingestion functions will inter-exchange raw data, apart from the fact that they will rely on a Kafka queue and will listen to and push data to specific topics. Taken all these into account, this subsection will discuss the various requirements that will be raised per each design decision that will be taken in the forthcoming period. We will separate the requirements per category of the data ingestion functions.

4.1.1 Domain/Schema Agnostic Functions

This category of functions is placed after the gateway and before the HHR mappers in the data pipeline. They can be included or not in a data pipeline, according to the needs of the proposed to be established pipeline, while their order in the chain of the functions in the pipeline is irrelevant. That means that they could consume data from any topic and they will need to send the output data to a not-known a priori topic. Therefore, an important requirement for this category is to be aware of which topics they should listen and send data to. As they will be standalone processes, this information must be passed through during their initialization. As a result, in static deployments, they could be configured with the domain of the Kafka Queue and the relevant topics or in cases of an automated dynamic deployment, they should allow this information to be given upon initialization. In the second case, it will be the serverless platform itself that will send this type of information, so the functions must be compliant to what the platform demands.

Secondly, the main characteristic of this category of functions that distinguishes them from the others is that they are domain/schema agnostic. This means that they can be applied to any type of dataset, but most likely, they should be aware on the definition of the schema during runtime, so they would need this type of information when they receive raw data from a Kafka topic. Here there are two design options: The first is to start a separate instance of these functions per different type of dataset that they would need to be applied on. In this case, the functions should be configured upon their serialization with the definition of the schema, so that they can be aware of. This is similar to the previous requirement when the functions need to know the topics that they need to connect during the deployment phase. Then, they will receive and will need to send a list of data elements from and to the topics. This implies that each instance of the functions (that corresponds to a specific dataset) will need to connect to its own topics. The second option is for the functions to start in a single instance and retrieve the schema definition from topic itself. In this option, the schema will need to be passed during the runtime and there are two further options to do this. The first is to use the Avro Schema Registry. This means that the avro libraries will take the responsibility to (de)serialize the data items that will be passed through the Kafka topics, while the ingestion functions can retrieve the schema definition from the registry itself and adapt their functionality accordingly. The second option is to manually pass the whole schema to the kafka topics. In this scenario, the data that will be inter-exchanged among the involved data ingestion functions will have a format similar to the following:

```
{
  "dataset": "DatasetXYZ",
  "schema": "the schema of the dataset in an avro compatible format. This must be a JSON
object",
  "total": 100,
  "data": [
    { ... JSONElement ...},
    { ... JSONElement ...},
    { ... JSONElement ...},
    { ... JSONElement ...},
    { ... JSONElement ...}
}
```

We include in this category of functions the HHR Mappers, as they can be domain/schema agnostic, with the only difference is that instead of sending raw data ensuring their input schema, they transform the data and send lists of HHR data items.

4.1.2 Domain/Schema Specific Functions

This category of functions is placed after the HHR Mappers and before the HHR Importer. The first requirement for this type of functions is similar to the domain/schema agnostic ones. Both types would need to know Kafka the topic names to connect. As a fact, this is the same requirement as above, so they would need to instantiate themselves with this information, either being passed using a configuration file, or via the serverless platform.

The main differentiator with the previous type of data ingestion functions however, is that the domain/schema specific ones do not need to know the schema during the deployment or the runtime phase, rather than during the implementation phase. However, as iHelp defines the common data model, and these functions are placed after the HHR Mappers where the raw data will have been already transformed to HHR, there is no need to pass this type of information regarding the data schema. As a

matter of fact, as they will always consume HHR objects, they can be instantiated in a single instance that will always listen to a specific topic and send data to another one. They will also need to inter-exchange lists of HHR data elements with no additional information as an overhead.

In this category of functions we include the HHR Importer that shares common requirements as a domain/schema specific function, with the only difference that this function does not need to send data to another topic, rather than connect to the big data platform in order to persistently store the data.

4.2 Data Cleaner

The Data Cleaner function will be utilized as an integrated microservice in the overall iHelp project, and its main objective is to facilitate the management and federation of the data. The Data Cleaner should be placed at the beginning of the data processing pipeline as it seeks to provide the assurance that the provided data coming from several heterogeneous data sources will be cleaned and completed. This microservice will be designed to minimize and filter the non-important data, thus improving the data quality and importance. To this end, the Data Cleaner aims to provide all the processes that will detect and correct (or remove) inaccurate or corrupted datasets containing incomplete, incorrect, inaccurate, or irrelevant data elements with the purpose of replacing, modifying or deleting these data elements. On top of this, the main goals of this sub-component are to (i) assure the incoming data's accuracy, integrity, and quality, (ii) investigate and develop mechanisms that ensure information non-repudiation and provision, and (iii) design and develop prediction mechanisms for predicting and/or correcting erroneous data. Moreover, this microservice will be integrated with the project's provided message bus mechanism, i.e. Kafka, and it will inter-exchange data from specific Kafka topics. In deeper details, raw incoming data will be consumed from one Kafka topic, these data will be further cleaned and finally the cleaned data, which are the output of the will be sent as a new message to another Kafka topic. A more extensive description and reference to the overall functionality, the specifications and the implementation steps of this specific function, the Data Cleaner, will be provided in the context of D3.7 "Standardisation and Quality Assurance of Heterogenous Data I", which will be delivered also in M10.

4.3 Data Qualifier

The data qualifier component classifies data sources and datasets as reliable and non-reliable. For instance, a sensor providing the temperature of a room may produce values between 4 and 40 celsius degrees. If the sensor provides values higher than 50 degrees for a period (e.g., half an hour), the sensor (data source) and the last readings (dataset) are considered as faulty. This component receives the dataset schema, the dataset and the frequency of observation, in addition to information generated by the Data Cleaner component about the percentage of cleaned data. The information is received from different Kafka topics and the output is sent through a different Kafka topic.

The reliability of a data source is calculated based on the information related to the data schema (value ranges) and also on the data availability (e.g., a sensor produces a value every minute and it does not produce any value for five minutes) evaluated for some period of time that depends on the frequency of the data that is produced. If the data is received and the values are according to the rules defined in the data schema, then the data source and the dataset are reliable. If data is not received, the data source is unreliable. The dataset unreliability, it is calculated based on the percentage of out-of-range values received during a period of time.

4.4 Data Harmonizer

In the scope of facilitating the aggregation of the distributed heterogeneous data coming from multiple health related datasets, the Data Harmonizer function will be utilized and integrated in the overall data processing pipeline of the iHelp framework. The Data Harmonizer function will be utilized as an integrated microservice, and its main objective is to support and harmonize data coming from divergent sources into a common format. To this end, its main aim is to provide annotated/correlated health data & harmonize them with the HHR model and format that will be defined in the scopes of the Task 3.1 "Data Modelling and Integrated Health Records". On top of this, it seeks to (i) enhance the interoperability of incoming data, and (ii) provide a structure mapping mechanism between data resources and HHR resources. The Data Harmonizer microservice will be integrated with the project's provided message bus mechanism, i.e. Kafka, and it will consume and produce corresponding messages to specific Kafka topics. The messages to be consumed should be cleaned data, which will be further harmonized and transformed, hence an annotated, transformed and in HHR compliant format message will be the output of this specific microservice. To this end, the Data Harmonizer microservice integrates closely with the Data Cleaner microservice presented in the previous subsection.The overall functionality, the specifications, and the implementation steps of this specific microservice, the Data Harmonizer, will be further explained and analysed in the context of D3.7 "Standardisation and Quality Assurance of Heterogenous Data I", which will be delivered also in M10.

4.5 Raw-to-HHR Converter

As the name of this ingestion function implies, its role is to convert items coming from raw data that has been captured into the common data model supported by the iHelp integrated solution, the HHR. It must be placed at the middle of the data ingestion pipeline, as all functions that will be deployed before this must be domain/schema agnostic, and all functions that will be deployed after the output of this function must be domain/schema specific. There might be two possibilities of how this function can be implemented: one option is to have a specific implementation per dataset that will need to be captured and ingested. As each dataset contains raw data that can be of any schema concerning different aspects of the overall HHR data model, dataset specific implementation must be provided in order for converter to be applied. The second option is for this function to rely on the outcomes of the data harmonizer, whose purpose is to annotate and integrate different datasets so that they can be converted to HHR. However, as the development of both the data harmonizer and the *raw-to-hhr converter* is being done in parallel, a first version of this function will be implemented with the first option: there will be specific implementations per dataset, in order to accelerate the work that needs to be done in the integrated data ingestion pipelines, so that the overall process will not be blocked under the first delivery of the harmonizer.

4.6 HHR Data Importer

The *HHR Data Importer* is a function that must be always placed at the end of the data ingestion pipeline. Its role is to accept data items converted in the common data model, the HHR, and take the responsibility to persistently store them into the big data platform of the iHelp integrated solution. As such, it is considered as a schema/domain specific functionality, due to the fact that it can only understand and interpret data items that can be deserialized into HHR objects. As all data ingestion functions, it listens to a specific Kafka topic to get the input data, however its difference is that it does not sends data to another

target Kafka topic, but instead, it opens a data connection with the big data platform and stores the datum there. Its role is to encapsulate all technical details regarding how to establish data connectivity with the datastore, manage these connections, handle database transactions and provide an efficient manner on storing the data. It relies on the provision of the big data platform for data ingestion in very high rates, and as a result, it benefits from its direct API that bypasses the database query engine and can access its storage engine directly, while ensuring database transactions on the same time. Regarding its business logic, it translates data received in the HHR conceptual model, and translates them to the compliant relational schema of the operational datastore. It must be highlighted that the HHR is a conceptual Entity-Relational schema, and the big data platform defines a compatible relational schema. Therefore, the role of the *HHR Data Importer* is twofold: firstly to translate the HHR data items to the relevant POJOs (Plain Old Java Objects), where each POJO is mapped to a specific data table of the relational schema, and secondly, to provide all data connectivity mechanisms in order to efficiently store those POJOs to the storage engine.

An important consideration that needs to be addressed in the forthcoming period is whether there will be the need to have various instances of the *HHR Data Importer* that need to be responsible for different aspects of the HHR common data model. The HHR contain numerous different entities (i.e. Persons, Measurements, etc) and it is not clear at this stage whether everything will be sent in a general HHR entity, or only the internal entities will be sent. This might be the case for instance that a data provider firstly uploads a dataset that contains anonymized information regarding the patients, and then uploads a new dataset that contains information about clinical exams, that will be translated to the HHR *measurement* entity. In that case, instead of having a single instance of the *HHR Data importer* that can handle an HHR as a whole, there might be the need of having various instances, with each one of those to be specialized to a specific entity of the common HHR. In this scenario, different entities will need to be passed through different Kafka topics, and different implementations of the *HHR Data Importer* will listen to their corresponding topics, translate the entities into POJOs, and store them to the relevant data tables. This decision must be taken once the outputs of the *Raw-To-HHR Converters* will be available, as these functions are currently under development. However, even if there might be different instances of the *HHR Data Importer,* they will be still domain/schema specific, and this does not break the overall design of the data ingestion pipelines.

5 Deployment of Data Ingestion Pipeline

In this section we will discuss the deployment and run-time phase of the data pipelines, which means the establishment and execution of the data ingestion flow, from the external sources to the big data platform.

The data pipelines consist of chains of functions or software components that read data from one and pass them to the other. These functions or software components can be packaged and started as a standalone process, can be encapsulated inside a microservice or can be provided as docker images that can be initialized as a virtualized container inside a Kubernetes pod. The only constraint is that they should be hosted inside the cluster or infrastructure that the integrated solution of iHelp will be deployed. As they can be either microservices that expose REST APIs for inter-communication, or they can speak directly via TPC and RPC calls, it is irrelevant, at least in theory, on the way they will interact among them, as long as they inter-exchange the predefined data elements and meta-information, as defined in the previous subsections. However, as are not designed to communicate by exchanging data communication messages, rather than raw data, it has been designed that a different medium should be used in order for the functions to pass data to each other. When ingesting vast amount of data, either from a big static file or by consuming a data stream where data items arrived in very high rates, the use of REST or RPC calls is not considered favourable. Due to this, the decision has been initially taken to rely on Kafka queues for the inter-exchange of raw data.

The use of Kafka queues allows for asynchronous invocations of the data ingestion functions that formulates the data pipeline, and loosely-coupled micro-services or functions. That way, the data provider can define various types of data pipelines and he or she is not obliged to use a predefined chain of functionalities. Moreover, according to the requirements for the functions described in the previous subsection, each function can be adapted dynamically during the deployment or runtime phase in order to support the specific needs for its own implemented algorithm. And as the invocation takes place asynchronously, this leads to the so called choreography of microservices design paradigm.

As it has been described in the two previous subsections, the design of all software components are involved in the data capture and ingestion pipelines is generic enough and gives the opportunity for a variety of different types of deployments. This is due to the fact that firstly both the data capture and data ingestion software components can be instantiated as single processes or in a Function-as-a-Service fashion, thus allowing for dynamically configure their initialization during the deployment or runtime phase. Even in the case of the domain/schema specific data ingestion functions, the schema is the HHR, which is the common data model of the iHelp. Moreover, the use of Kafka queues allows for loosely-coupled and asynchronous interaction of the involved parts of the data pipeline, which gives us flexibility on deciding about how the deployment will be. We can distinguish four different types of deployments that will be further analysed in the next subsections, and are characterized by whether or not they allow for dynamic or static deployments and how tightly or loosely coupled these deployments will be. Each of the identified deployment type has a different level of importance and difficulty and we prioritize and plan our next steps accordingly.

5.1 Fully automated and dynamic deployments

This type of deployment has difficulty of level 4 and prioritization of level 1.

This deployment does not demand to have anything pre-deployed and the establishment of the data capture and ingestion pipeline, the deployment of its internal components and their inter-connectivity happens upon the data provider request. This type of deployment requires the design and implementation of an additional technology building block that will be the orchestrator of such deployments, meaning that instead of having a microservice choreography design paradigm, we will have an hybrid, with this building block to serve as the orchestrator for the initial establishment of the data pipelines, and after their instantiation, the microservices will interact in a choreography manner. The deployment orchestrator will need to be provided by a configuration file that defines the data pipeline itself. This will have to include the types of ingestion functions that need to be involved in a specific pipeline, along with their runtime configuration parameters. Upon request, the orchestrator will have to create the corresponding pods from the underlying Kubernetes cluster, create the intermediate Kafka topics that the involved functions would need to use for the inter-exchange of the data items, and finally instantiate the functions themselves, by passing to them the user-defined configuration parameters, along with the system specific ones (i.e. name of the kafka topic for the function to listen to). This deployment type is feasible due to the fact that both the data capture software component and the data ingestion functions are required to be designed in a Function-as-a-Service manner, so that they can be dynamically instantiated and interconnected.

This deployment type allows for a fully custom implementation of both the orchestrator and the functions themselves, thus giving to the system architecture of the iHelp integrated solution the freedom to do any optimizations might be needed, while on the other hand, the ingestion functions can put constraints and requirements to the dynamic orchestrator, that can be designed accordingly.

The drawback of this deployment type is that it requires for the dynamic orchestrator to be designed and implemented, which is not included as an objective of the activities that need to be performed under the scope of task T3.2 (“Primary Data Capture and Ingestion”), therefore it is out of the scope of the project and it has the lowest priority.

5.2 Fully automated and dynamic deployments using a serverless platform

This type of deployment has difficulty of level 3 and prioritization of level 2.

This deployment type is similar to the previous one, with the main difference that instead of having to implement a custom iHelp dynamic orchestrator, we can rely on the serverless platform that is being currently implemented under the scope of the task T4.2 (“Model Library: Implementation and Recalibration of Adaptive Models”). The serverless platform will take the responsibility for the automated reservation of infrastructure resources from the Kubernetes cluster and deploy the data ingestion functions that formulate the data pipeline, while it will have to ensure the proper instantiation of the software components and their corresponding interconnectivities. The serverless platform requires the involved software components to be deployed in a Function-as-a-Service manner, and as our design for

both the data capture component and the data ingestion functions is compliant with this requirement, this deployment type is feasible by design.

However, the main drawback with this type of deployment is that the serverless platform is currently under development and is planned to be delivered to the project at a later phase. Moreover, the corresponding report of the task T4.2, whose main activity is the delivery of that platform, is planned to be delivered also at a later phase. This implies two important obstacles at this phase of the project: Firstly, the initial integration and validation process for the data capture and ingestion pipeline has to be blocked until the delivery of the first prototype of the serverless platform and secondly as the requirements for the functions are still not well defined, our design might need to be altered at a later phase. For instance, it is not known how the function specific parameters (either function or system related) can be passed to the newly created instances or which type of deployments can be supported (i.e. using docker images, jar files or other packaged distribution according to the supported programming languages). Therefore, we will examine if this deployment type is feasible after the first half of the project in M18 and we will evaluate our decision at that phase. As a result, this deployment type is considered as “nice to have”.

5.3 Configurable static deployments

This type of deployment has difficulty of level 2 and prioritization of level 3.

The difference between this type of deployment and the aforementioned two, is that in this case, we only have static deployments of the data capture and ingestion functions. Instead of deploying the requested data pipeline on demand of the request and shut it down after its completion, in this case, the deployment takes place ones and the data pipeline is online for the whole lifecycle of the integrated solution of the iHelp platform. There can be numerous of deployed data pipelines, as per dataset, giving the flexibility that for a specific dataset only a small list of the overall provided functions might be required. In this type of deployment, the data functions will connect to predefined topics and will need to be instantiated by predefined parameters that will be passed during the deployment phase, however these parameters will be given manually rather than having an external building block (i.e. the dynamic orchestrator or the serverless platform) to resolve them during the runtime phase of this block. The automation of the deployment can be achieved using Kubernetes manifest files and Kubernetes configmaps; in the configmaps, these parameters must be manually provided by the system administrator while the Kubernetes manifest must also be written manually. However, once this is defined, the deployment is automated and the data pipelines can be portable to each instance of the iHelp platform.

The main drawback of this type of deployment is that is resource greedy, as it needs to maintain deployed various data pipelines, one per dataset, which will be inactive during the majority of the lifecycle of the iHelp platform. However, having the Kubernetes manifest, it will be possible to shutdown the pipelines and restart them upon request of the data provider, thus releasing the unused resources, however this process must be done manually. This is the target deployment type as it has been identified at this phase of the project.

5.4 Static deployments

This type of deployment has difficulty of level 1 and prioritization of level 4.

This last scenario also involved static deployments of the data capture components and the data ingestion functions, as of the previous one. The difference between the configurable static deployment of the previous subsection is that in this case, there is only one data pipeline established that can be used by all the datasets. All data functions are chained together in the pipeline, with the evident drawback that during a data ingestion of a specific datasets, the corresponding raw data will have to be passed through all the provided data ingestion functions, even if they are not involved in this specific ingestion process. In this scenario, there is no need for any automation of deployment, as the same deployment of the data pipeline will be running during the whole lifecycle of the iHelp platform, without the possibility to shut it down in order to release resources. As the deployment will happen once, it is not of a high priority to automate it.

The evident drawback of this deployment type, apart from being resource expensive, is that it introduce a significant overhead, as the data ingestion process has to involve all provided ingestion functions, even if they will not be used. As a result, the overall latency will be increased considerably enough. This deployment is the minimum for the task to accomplish its target objectives, however it should be used only for testing purposes in order to validate the overall data capture and ingestion process.

5.5 Planning for deployment in the future steps

At the time when this report was written, the initial design of the data capture components and the requirements for the data ingestion functions had been defined, and the development and implementation of these software components was in progress. Moreover, different deployment options had been identified that were discussed in the previous subsection. Table 1 summarizes the identified deployment options, along with their level of difficulty and prioritization, ordered by the prioritization that drives the our implementation plan.

Table 1: Deployment Types.

Deployment Type	Difficulty	Prioritization
Static deployments	1	4
Configurable static deployments	2	3
Fully automated and dynamic deployments using a serverless platform	3	2
Fully automated and dynamic deployments	4	1

The plan is to initially focus on the first deployment type, which is the static deployment. This will be used for testing and validation purposes, and it will boost the development and integration process for the data capture and ingestion pipelines. This should be ready to be delivered at the end of the first phase of the project (M18). After having the main pillars ready and functional, in the second part of the project, the *configurable static deployments* scenario will be the main focus, as it will provide a more efficient way of deployments. At that phase, it will be further investigated the possibility to make use of the serverless platform, whose first prototype will have been delivered by then. After a feasibility study along with the experimentation of this platform, it will evaluate if it is worth to be used in the data capture and ingestion pipelines, and if yes, an additional effort will be put to benefit from the *fully automated and dynamic deployments using a serverless platform*. Last but not least, the *fully automated and dynamic deployments*

scenario is considered as out of the scope of the project, and it was included in this report as an option for future work after the end of this project.

6 Conclusions

This deliverable reports the work that has been carried out under the scope of task T3.2 ("Primary Data Capture and Ingestion") at this phase of the project (M10). The purpose of this task is to provide the data pipelines for capturing and ingesting raw data coming from external sources in order to be eventually stored and accessed via the big data platform of the integrated iHelp solution. Towards this outcome, the initial design of such data pipelines has been decided and reported in this document, identifying the involved building blocks and software components, how they will interact, what kind of requirements our design is putting to them and how they will be finally deployed.

Firstly, an overview of the data capture and ingestion pipelines has been given, focusing on their basic principles and the different design approaches that have been foreseen along with their benefits and drawbacks that should be taken into consideration. There are three important aspects that need to be treated separately: One part is the functionality related with the actual capture of the raw data themselves, coming from a variety of different and heterogeneous data sources, in different formats, either in a structured, semi-structured or totally un-structured manner, that might also require a batch ingestion process, or a scheduling of a periodic data capture. The second part concerns the functionalities that are involved in the overall data ingestion pipeline, after the data has been initially captured. The third part concerns deployment patterns that might include static or dynamic deployments of such data pipelines, while the second option also concerns whether or not such dynamic deployments could be done in an automated manner.

After the identification of these three aspects that concerns the data capture and ingestion pipelines, further details was given for all three of them. Regarding the data capture functionality, the *data gateway* has been decided to be the responsible software component. The overview of this technological building block has been provided, how it should be interact, what type of requirements must address, along with a an exhaustive list of basic principles that its implementation must respect. Then, a detailed description of its initial design was given, along with a class diagram, following by a discussion about its main software modules and different approaches that are still open at this phase of the project. Finally, the definition of its API has been included, which additionally contains examples of data that needs to be passed as input parameters or will be returned as the output of the invocation of the defined API.

Regarding the functions that concern the data ingestion pipeline, after data has been initially captured by the *data gateway,* these functions can be categorized as domain/schema agnostic or specific. Domain/schema agnostic functions can be applied in each type of dataset, with the requirement that they can be provided with the data schema of the latter during the deployment or runtime phase. On the other hand, domain/schema specific functions need to now a priori (i.e. during the development phase) the schema of the dataset that will be applied upon. However, as the iHelp integrated solution will rely on its common data model, the HHR, all such functions will consume HHR data. As the functionalities themselves and the implementation of these data ingestion functions is out of scope for this task, an overview of them was given in this report, and the focus was on the requirements that these functions must address in order for them to be part of the integrated data ingestion pipeline. Different design approaches have been also foreseen at this phase of the project, while the decision will be taken on a later phase, as the main focus at this time is on the implementation of the core functionalities. However,

in this report we included all different options, along with the requirements that each design will need to be taken care of.

Finally, regarding the deployment decisions, four different choices have been identified: static deployments, configurable static deployments, fully automated and dynamic deployments and fully automated and dynamic deployments using the serverless platform that is being currently implemented under the scope of task T4.2 (“Model Library: Implementation and Recalibration of Adaptive Models”). Different levels of difficulty and prioritization have been given to each one of those, while this report includes the benefits and drawbacks for each of the aforementioned option. Finally, there has been included a concrete planning regarding which type of deployments will be supported and when they should be available.

In order to conclude, the work that has been carried out under the scope of the task T3.2 at this phase of the project can be considered adequate and it is under the overall plan of the project. The main components that concerns the activities of T3.2 have been identified, designed while their implementation is on track. Moreover, as task T3.2 is responsible for the establishment of the integrated data capture and ingestion pipelines, design options regarding the overall integration have been also identified. A second version of this report will be delivered in the second phase of the project (M22), which will contain the decisions about the aforementioned design options, the final design of the involved software components along with a demonstrator regarding the deployment of such data pipelines.

List of Acronyms

API	Application Programming Interface
CSV	Comma Separated Values
FTP	File Transfer Protocol
HHR	Holistic Health Record
IoT	Internet of Things
JSON	JavaScript Object Notation
M	Month
POJO	Plain Old Java Object
REST	Representational State Transfer
T	Task
WP	Work Package