Co-Funded by the Horizon 2020 programme of the European Union

Grant Agreement: 875358

Deliverable D2.3

FAITH Framework Conceptual Architecture

Work package:	WP2 – Stakeholders Identification, use cases definition, requirements specification, and architecture design
Prepared By/Enquiries To:	philip.obrien@waltoninstitute.ie – WIT
Reviewers:	Fernando Ferreira – Uninova
Status:	final
Date:	06/05/2021
Version:	1.0
Classification:	Public

Authorised by:

P O'Brien

Philip O'Brien, WIT

Authorised Date: 06/05/2021

FAITH Project Profile

Contract No H2020-ICT- 875358

Acronym	FAITH
Title	a Federated Artificial Intelligence solution for moniToring mental Health status after cancer treatment
URL	https://www.h2020-faith.eu/
Twitter	https://twitter.com/H2020_Faith
LinkedIn	https://www.linkedin.com/company/faith-project/
Facebook	https://www.facebook.com/H2020.FAITH
Start Date	01/01/2020
Duration	36 months

FAITH Partners

List of participants

Participant No	Participant organisation name	Short Name	Country
1 (Coordinator)	WATERFORD INSTITUTE OF TECHNOLOGY.	WIT	Ireland
2	UPMC Whitfield, Euro Care Healthcare Limited.	UPMC	Ireland
3	Universidad Politécnica de Madrid.	UPM	Spain
4	Servicio Madrileño de Salud.	SERMAS	Spain
5	UNINOVA, Instituto de Desenvolvimento de Novas Tecnologias.	UNINOVA	Portugal
6	Fundação D. Anna de Sommer Champalimaud e Dr. Carlos Montez Champalimaud.	CF	Portugal
7	Deep Blue.	DBL	Italy
8	Suite5 Data Intelligence Solutions Limited.	SUITE5	Cyprus
9	TFC Research and Innovation Limited.	TFC	Ireland

SC1-DTH-01-2019: Big data and Artificial Intelligence for monitoring health status and quality of life after the cancer treatment

H2020-SC1-DTH-2019

FAITH is co-funded by the European Commission - Agreement Number 875358 (H2020 Programme)

Document Control

This deliverable is the responsibility of the Work Package Leader. It is subject to internal review and formal authorisation procedures in line with ISO 9001 international quality management system procedures.

Version	Date	Author(s)	Change Details
0.1	18/01/21	Philip O'Brien (WIT)	Table of Contents defined.
0.2	12/02/21	Philip O'Brien (WIT)	ToC revised with partner's input.
0.3	15/02/21	Philip O'Brien (WIT)	Add text re features from proposal.
0.4	19/02/21	Philip O'Brien (WIT)	Continued additions.
0.5	23/03/21	Philip O'Brien (WIT)	Revised ToC to align with Software Requirements Specification (explained in doc).
0.6	25/03/21	Philip O'Brien (WIT)	Feature additions.
0.7	28/03/21	Philip O'Brien (WIT)	Feature additions.
0.8	04/04/21	Philip O'Brien (WIT)	Non-functional requirements additions.
0.9	08/04/21	Philip O'Brien (WIT)	Feature additions.
0.9.1	26/04/21	Fernando Ferreira (UNI)	Internal review.
0.9.2	27/04/21	Tom Flynn (TFC)	QA'd version.
1.0	06/05/21	Philip O'Brien (WIT)	Final release for submission to European Commission portal.

Executive Summary

Objectives:

This deliverable is a direct outcome of T2.5 and T2.6, documenting the technical requirements of the FAITH architecture and the FAITH reference architecture.

T2.5 will work on the identification and documentation of the technical requirements of the FAITH infrastructure. T2.5 will mainly be based on the outcomes of T2.3 and T2.4, documenting the functional and non-functional requirements and the demonstrator specific requirements respectively, which in turn will be translated into technical requirements, constraints and specifications. These specifications will guide the design of the reference architecture of the FAITH infrastructure. The requirements elicitation and analysis will follow the principles of agile software development, while a requirements backlog will be maintained during the project implementation to guide all development tasks. WIT will lead the task. All technical partners will be major contributors in the provision and analysis of the requirements.

T2.6 includes the design and documentation of the FAITH Reference Architecture, as well as the specification (i.e., functionalities and interfaces) of the core architectural components and models, which will then be implemented in the context of the implementation WPs. A preliminary conceptual architecture has already been designed and is proposed in Section 1.3.1. Within the context of this task and taking into consideration the thorough requirements analysis and specification of the usage scenarios, the consortium will proceed with the re-design and/or modification of the proposed conceptual architecture, highlighting the structural components of it, and the information exchange amongst them. WIT will lead the design of the architecture. All technical partners will contribute to this task and be supported by our healthcare organisations.

Results:

The primary results of this deliverable are a software requirements specification (SRS).

Table of Contents

1 Introduction 12

1.1 Purpose 12

1.2 Document Conventions 13

1.2.1 Labelling Requirements 13

1.3 Project scope 14

2 Abbreviations and Acronyms 16

3 System features 17

3.1 Data Check-In Service 18

3.1.1 Description 18

3.1.2 Functional Requirements 19

3.2 Federated Learning Layer 20

3.2.1 Description 20

3.2.2 Functional requirements 20

3.3 Model Compression Pipeline 22

3.3.1 Description 22

3.3.2 Functional requirements 22

3.4 Data Audit Service 22

3.4.1 Description 22

3.4.2 Functional requirements 23
3.5 Visualisation Service 23
3.5.1 Description 23
3.5.2 Functional requirements 23
3.6 Model Check-In Service 24
3.6.1 Description 24
3.6.2 Functional requirements 24
3.7 Feature Store 25
3.7.1 Description 25
3.7.2 Functional requirements 26
3.8 Anonymisation Pipeline 27
3.8.1 Description 27
3.8.2 Functional requirements 28
3.9 Logger 30
3.9.1 Description 30
3.9.2 Functional requirements 30
3.10 Authorisation and Authentication 30
3.10.1 Description 30
3.10.2 Functional requirements 31
3.11 Explainable AI 31
3.11.1 Description 31
3.11.2 Functional Requirements 31

4 Quality attributes 32
4.1 Usability 34
4.2 Performance 34
4.3 Security 35
4.3.1 Authentication 39
4.4 Safety 40
4.5 Availability 40
4.6 Integrity 40
4.7 Verifiability 40
4.8 Robustness 40
5 Internationalisation and localisation requirements 41
6 Conclusions 42
7 Bibliography 43

TABLE OF FIGURES

FIGURE 1 HIGH-LEVEL ARCHITECTURE FROM PROPOSAL 18
FIGURE 2 COMPONENTS OF A FEATURE STORE 26
FIGURE 3 TRIAL SITE COHORTS 29
FIGURE 4 PSEUDONYMISED COHORT 29

LIST OF TABLES

Table 1 – Functional Requirement - DCS 19
Table 2 - Functional Requirements – Federated Learning 21
Table 3 - Functional Requirements - MCP 22
Table 4 - Functional Requirements - Visualisation 23
Table 5 - Functional Requirements – MCP 2 25
Table 6 - Functional Requirements - FS 26
Table 7 - Functional Requirements - Anon 28
Table 8 - Functional Requirements - Logger 29
Table 9 - Functional Requirements - Auth 29
Table 10 - Functional Requirements - XAI 30

1 Introduction

1.1 Purpose

This document describes the initial architecture of the FAITH platform. The focus will be on the backend software components, since the mobile application has a dedicated set of deliverables under WP5. The structure of this document follows the template provided in (Wiegers and Beatty 2013) for a Software Requirements Specification (SRS) document. Some sections of that template e.g., ‘Data Requirements’, and ‘External Interface’ have been omitted from this document because they are treated in detail in other deliverables. This overlap of deliverables is also a consideration for some of the software components e.g., ‘Model Compression’ that have dedicated deliverables. In these instances, while the components are explored in detail in their deliverable, a high-level description and corresponding functional requirements will still be provided as part of this document, for the sake of completeness.

The motivating factor for aligning this document with the SRS template is to keep this as a living document in which requirements can be added/edited/removed as the development work of the project evolves.

We must strike a balance between traditional software engineering approaches, and the different approach of a Research and Innovation Action (RIA) project. That being said some tenets remain. Business requirements represent the top of the requirements chain. They define the vision of the solution and the scope of the project that will implement the solution. The user requirements and functional requirements must align with the context and objectives that the business requirements establish. Requirements that don’t help the project achieve its business objectives shouldn’t be implemented. A project without a clearly defined and well-communicated direction invites disaster. Project participants can unwittingly work at cross-purposes if they have different objectives and priorities. The stakeholders will never agree on the requirements if they lack a common understanding of the project’s business objectives. Without this understanding up front, project deadlines will likely be missed, and budgets will likely be overrun as the team struggles to deliver the right product.

1.2 Document Conventions

When a group of people begin discussing requirements, they often start with a terminology problem. Different observers might describe a single statement as being a user requirement, software requirement, business requirement, functional requirement, system requirement, product requirement, project requirement, user story, feature, or constraint. The names they use for various requirements deliverables also vary. A customer's definition of requirements might sound like a high-level product concept to the developer. The developer's notion of requirements might sound like a detailed user interface design to the user. This diversity of understanding leads to confusion and frustration. (Wiegers and Beatty 2013)

We adopt some definitions that we find useful to avoid confusion and frustration.

Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

1.2.1 Labelling Requirements

All functional and non-functional requirements for the FAITH platform will be captured in this document. Since they will be the basis of the development work, it is vital that every requirement has a unique and persistent identifier. We need to be able to refer to specific requirements in development work, change requests, cross-references etc. Uniquely identified requirements facilitate collaboration between team members when they're discussing requirements. There are several requirements labelling methods found in the wild.

The simplest approach gives every requirement a unique sequence number, such as UC-9 or FR-26. This approach is very commonly used but it is a poor choice for many reasons: it doesn't provide any logical or hierarchical grouping of related requirements, the number doesn't imply any kind of ordering, and the labels give no clue as to what each requirement is about.

An improvement to this is hierarchical numbering i.e., if the functional requirements appear in '3.2 Project Priorities' of your SRS, they will all have labels that being with '3.2'. Again, however, several shortcomings exist. The labels can grow to many digits in even a medium-sized SRS. Numeric labels tell you nothing about the intent of a requirement. If you are using a word processor, typically this scheme does not generate persistent labels.

The approach suggested in (Wiegers and Beatty 2013), and adopted in this document is hierarchical textual tags. Consider this requirement: "The system shall ask the user to confirm any request to print more than 10 copies". This requirement might be tagged Print.ConfirmCopies. This indicates that it is part of the print function and relates to the number of copies to print. Hierarchical textual tags are structured, meaningful, and unaffected by adding, deleting, or moving other requirements.

Any requirements added to this deliverable must adhere to this convention.

1.3 Project scope

Recapping the vision of this project, the aim is to provide an Artificial Intelligence application that remotely identifies depression markers, using Federated Learning, in people that have undergone cancer treatment. The aim of the observational study is to hopefully confirm what exactly these markers are. It is only then that we can develop the models that would take these markers as input. Crucially, in the context of a project architecture, this means that there are really two architectures that need to be developed, one for the trials, and one for post-trial. For example, the Federated Learning component obviously doesn't need to exist for the trial, but it is a vital component of the project.

For the sake of completeness, we will capture all the architecture descriptions (i.e., trial and post-trial) in this document, and as the work evolves the categorisation of each component in terms of being needed just for the trial or also for post-trial will be updated.

In very high-level terms during the trial, the platform will receive user data from their devices. Over time this data will be analysed to identify the markers of depression. If this is achieved, then the models can be developed. Post-trial, the user data will remain on their devices and the chief function of the platform will then be to receive model updates, aggregate these from many devices, create a new federated model that is then deployed to the devices. Naturally, there are many more things to consider, and these will be discussed in the following sections.

2 Abbreviations and Acronyms

Abbreviation	Description
API	Application Programming Interface.
AWS	Amazon Web Services.
CI	Continuous Integration.
CD	Continuous Deployment.
CRUD	Create Read Update Delete.
FL	Federated Learning.
HTTPS	Hypertext Transfer Protocol Secure.
ISO9001-2015:	International Quality Management Systems.
ML	Machine Learning.
SQL	Structured Query Language.
TLS	Transport Layer Security.
URL	Uniform Resource Locator.
WP	Work Package.
WPL	Work Package Leader.

3 System features

There are many ways to organise functional requirements e.g., by functional area, process flow, use case, mode of operation, user class, stimulus, and response. Hierarchical combinations of these elements are also possible, such as use cases within user classes. Here we have chosen to organise functional requirements by system feature. We believe that this makes it easy for the reader to get an overview of the various features, and crucially it makes it easier for the team to add new features or requirements.

Figure 3 shows the original high-level architecture diagram from the proposal. Naturally, our thinking has evolved considerably since then, and so this diagram is only included for context. When the architecture is finalised, in terms of technical requirements, then an updated set of diagrams will be produced, most likely following the C4 model for visualising software architecture[1]. Several of the components are described in detail in dedicated deliverables, and several new system features have been deemed necessary, and so are described in the following sections.

Also, some components viewed as running on the mobile side may run on the backend for the trial, but then be switched to a mobile deployment for the overall framework. Since the mobile application will be built using React Native, leveraging JavaScript for such features will make it much easier to port the feature to mobile. Since this deliverable is ongoing some of the requirements are laid out in detail, whereas others are acting as placeholders for current discussions, and will be filled out in a living version of this document.

[1] https://c4model.com/

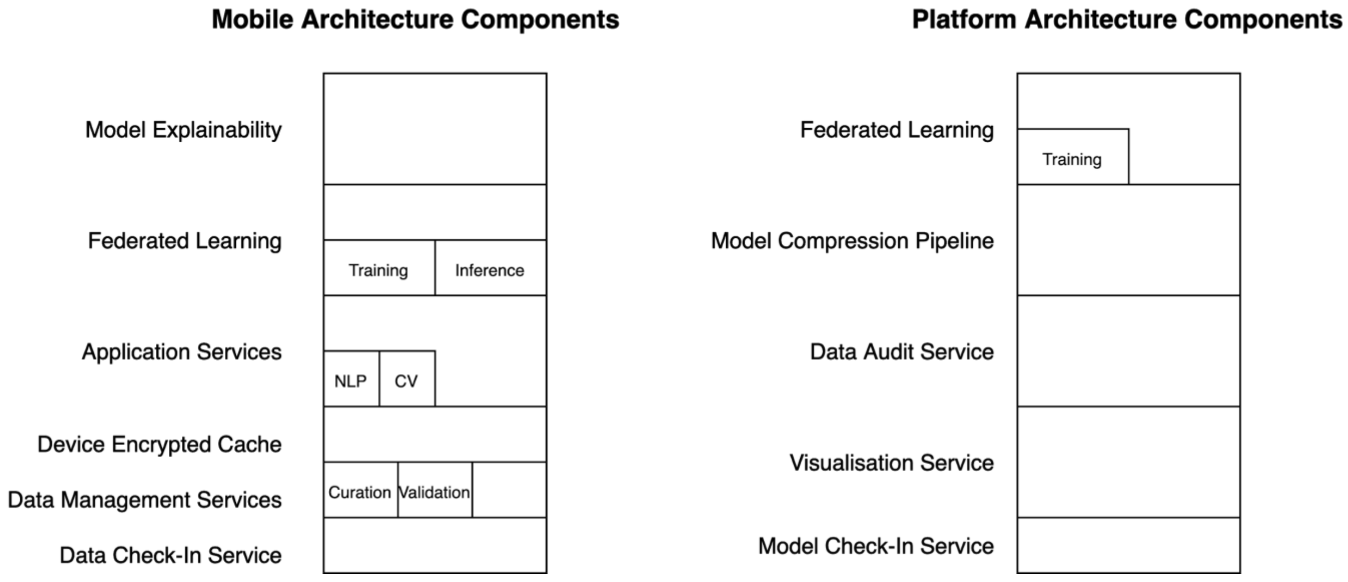


Figure 1 High-level architecture from proposal

3.1 Data Check-In Service

3.1.1 Description

This is one clear example of a feature that will ultimately need to run on the mobile device, but for the trial where the data is sent to the cloud, it makes more sense for this functionality to be implemented on the backend.

This service is comprised of three distinct steps: data ingestion/versioning, data validation, and data pre-processing.

Data Ingestion

As the name implies this is where data enters the platform and is usually processed into a format that the other components can digest. This is often a good time to version the data, using a tool such as Data Version Control, so that a data snapshot can be linked with an output model. Airbnb provide some great examples with their own key requirements[2]

[2] https://blog.anomalo.com/airbnb-quality-data-for-all-a4ca6b4c97e6

Data Validation

Before any new data can be used to create a new model, it must be validated. This usually comprises a set of checks to ensure that the statistics of the new data are as expected e.g., the range, number of categories, distribution of categories etc. Anomalies should also be checked for, and if detected an alert raised.

Data Pre-processing

Depending on the type of data there are many ways it might need to be processed before being used to train a model e.g., converting labels to one or multi-hot vectors, tokenizing text etc.

3.1.2 Functional Requirements

DCS.Validate:	**Validating new data**
.RowCount:	Are there any records from the most recent data? Is the row count within an expected range?
.Anomaly:	Are the metrics within expected ranges
.Standard:	Standard run-time quality checks e.g. are basic data constraints satisfied: unexpected NULL or blank values, violations of unique constraints, strings matching expected patterns, timestamps following defined order, etc.
DCS.Ingest:	**Ingesting new data**
.Receive:	Has new data been correctly accepted.
.Version:	If applicable, new data is versioned.
DCS.Preprocess:	**Preproccessing of new data**

Table 1 – Functional Requirement - DCS

3.2 Federated Learning Layer

3.2.1 Description

For many applications such as content recommendation, giving writing advice, or healthcare, a model's most important source of information is the data it has about the user. We can leverage this fact by generating user-specific features for a model, or we can decide that each user should have their own model. These models can all share the same architecture, but each user's model will have different parameter values that reflect their individual data. This idea is at the core of Federated Learning (FL). In FL, each client has their own model. Each model learns from their user's data and sends aggregated (and potentially anonymised) updates to the server. The server leverages all updates to improve its model and distils this new model back to individual clients. (Ameisen 2020)

As promising as this is, however, it does add an additional layer of complexity. Making sure that each individual model is performing well, and that the data transmitted back to the server is properly anonymised is more complicated than training a single model.

Care must be taken to avoid performance issues on devices that train an FL model. Training can quickly drain the battery on a mobile device or cause large data usage, leading to expense for the user. Even though the processing power of mobile phones is increasing rapidly, they are still only capable of training small models, so more complex models should be trained on a central server. (Nelson and Hapke 2020)

3.2.2 Functional requirements

FL.Preprocessing:	**Performs the required transformations on the raw data before it is fed to Model**
FL.Model:	**Encapsulates the AI Algorithm under a proper interface for better scalability and full customizability.**

FL.Train:	**Training new or existing federated model**
FL.Learning:	**Functional Library for developing Federated Learning algorithms.**
FL.Averaging:	**Performs federated averaging on client models via an Iterative Process**
FL.ServerState:	**Represents the state of the server carried between rounds to perform Federated Learning.**
FL.Compression:	**Zeroes out extremely large values for robustness to data corruption on clients, clips to moderately high norm for robustness to outlier**
FL.Simulation:	**Provides research datasets and other simulation-related capabilities for use in simulating federated learning experiments.**
FL.Computation:	**Helper functions that construct federated computations for training or evaluation, using existing models.**
FL.Deploy:	**Deploy the latest model to user devices**
FL.Aggregate:	**Aggregate learnings from different devices to produce new model**
FL.Evaluation:	**Builds the computation for federated evaluation of the given mode.**
FL.Test:	**Base class for Federated Learning tests, such as Secure and Unsecure Aggregation.**

Table 2 - Functional Requirements – Federated Learning

3.3 Model Compression Pipeline

3.3.1 Description

With the increasing size of machine learning models, model optimisation becomes more important to efficient deployments. Model quantisation allows you to reduce the computation complexity of a model by reducing the precision of the weight's representation. Model pruning allows you to implicitly remove unnecessary weights by zeroing them out of your model network. And model distillation will force a smaller neural network to learn the objectives of a larger neural network. (Nelson and Hapke 2020).

This component has a dedicated deliverable D4.4 and will be discussed in greater detail there.

3.3.2 Functional requirements

MCP.Receive:	**Receive trained model**
MCP.Compress:	**Compress model and return to FL layer for deployment**

Table 3 - Functional Requirements - MCP

3.4 Data Audit Service

3.4.1 Description

The audit service in FAITH is supported by a logging mechanism that ensures registry of all data handling activities. This is a critical aspect when preparing a framework to ensure data security and privacy. In that sense, the registry of user access and what data was visualized or processed, is essential to promote data security and privacy. The Distributed Ledger Technology (DLT) is used for this matter in ensuring that access data is logged and that logs are kept secure. In terms of architectural design, mode details are provided in D3.3 (FAITH Data Privacy & Protection), but essentially the process consist in providing that each time there is an access to data a log is created and uploaded to the DLT. The records are kept secure and immutable so that security verifications and audit can rely on the DLT records to verify and validate that all the accesses are permitted and is based on the assigned permissions.

3.4.2 Functional requirements

TBD

3.5 Visualisation Service

3.5.1 Description

This is one of the services that will take a different form depending on the stage of the project, i.e. the visualisations required during the trials will not be the same as those required for the delivered project.

For the trial, and immediately after the trial, the recorded data will need to be analysed by the domain experts to identify the depression markers. Visualisation will obviously play a key role in such analysis.

In the complete FAITH framework, however, a visualisation/reporting layer will be used to allow a healthcare professional to visualise historical monitored user data in the build-up to a negative forecast alert.

3.5.2 Functional requirements

Viz.History:	**Produce a historical visualisation of user data for review by clinician**
Viz.Explore:	**Provide a suite of visualisations to allow facilitate exploration of trial data.**

Table 4 - Functional Requirements - Visualisation

3.6 Model Check-In Service

3.6.1 Description

Similar to how the data check-in service performs, a set of checks on input data before it can be used, the model check-in service will need to carry out a set of checks on a trained model before it is deployed e.g., model analysis and model versioning.

Model Analysis

Accuracy and/or loss are the most commonly used metrics to determine the optimal set of model parameters. There are many other metrics, however, which can be used to provide a more in-depth analysis of a model's performance e.g., precision, recall, area under the curve (AUC), etc.

Model Versioning

Deploying model updates to production will inevitably affect the way models behave on new data, which presents a challenge - we need an approach for keeping production models up to date while still ensuring backward compatibility for existing model users (Munn, Robinson and Lakshmanan 2020). Model versioning is important to keep track of which model, set of hyperparameters, and datasets are linked. It should be straightforward to pinpoint all inputs relating to a released model.

3.6.2 Functional requirements

MCS.Analyse:	**Analyse trained models**
.Accuracy:	Get the model accuracy
.Loss:	Get the model loss
.Precision:	Get the model precision
.Recall:	Get the model recall
.AUC:	Get the model area-under-the-curve
MCS.Version:	**Version model**
.Number:	Number the model
.Tag:	Tag the model

.Date:	Date the model

Table 5 - Functional Requirements – MCS

3.7 Feature Store

3.7.1 Description

In Machine Learning (ML), a feature is data used as an input signal to a predictive model. When Uber released Michelangelo, their ML platform, they coined the term 'feature store'.[3] They have since emerged as a necessary component of the operational machine learning stack.[4] They are essentially a layer of data management that allows teams to share, discover, and use a highly curated set of features for their machine learning problems. When correctly implemented they should make it easy to:

- Productionise new features without extensive engineering support,
- Automate feature computation, backfills, and logging,
- Share and reuse feature pipelines across teams,
- Track feature versions, lineage, and metadata,
- Achieve consistency between training and serving data,
- Monitor the health of feature pipelines in production.

[3] https://eng.uber.com/michelangelo-machine-learning-platform/

[4] https://www.tecton.ai/blog/what-is-a-feature-store/

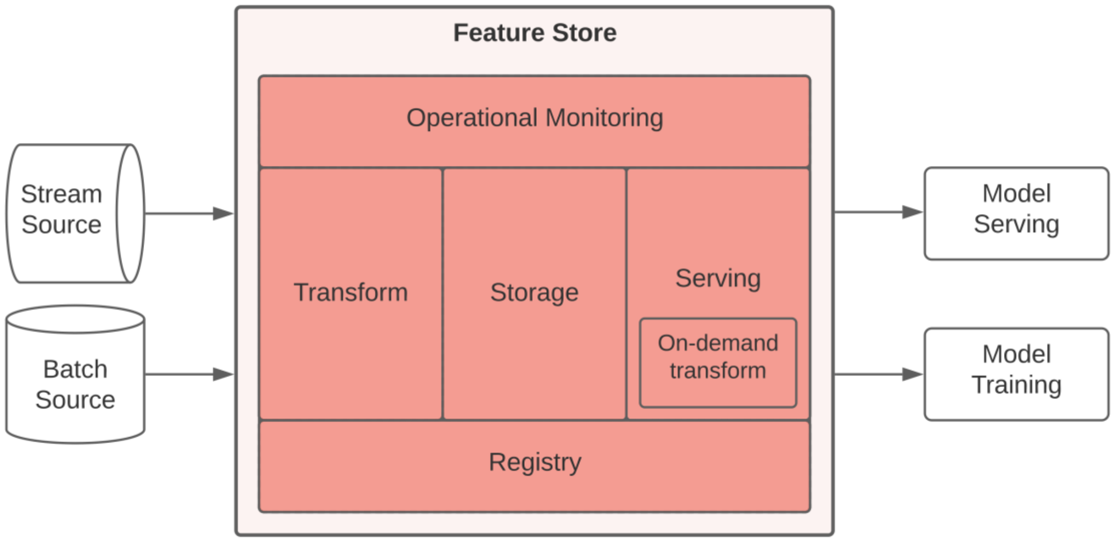


Figure 2 Components of a Feature Store

3.7.2 Functional requirements

FS.Serve:	**Serve data to models**
FS.Store:	**Persist feature data**
FS.Transform:	**Manage and orchestrate data transformations**
FS.Monitor:	**Monitor feature quality metrics**
FS.Register:	**Centralise feature definitions and metadata**

Table 6 - Functional Requirements - FS

3.8 Anonymisation Pipeline

3.8.1 Description

When we talk about anonymised data, it is helpful to think in terms of an identifiability spectrum (El Emam and Arbuckle 2020). When identifiability is viewed as a spectrum, with one end signifying identified data and the other end signifying anonymised data, we find ourselves with a range of options for sharing and using data responsibility.

There are two extremes:

- Identified data will include directly identifiable information such as the data subject's name or another field of information that is unique to them, such as a Social Security number.
- Anonymised data will not include directly identifying information, and indirectly identifying information will be sufficiently transformed to ensure the remaining information is sufficiently disassociated from personal identities.

Different types of variables can be used to reveal different types of information. Direct identifiers are data that can essentially be used *alone* to uniquely identify individuals or their households; indirect identifies are data that can be used in *combination* with one another to identify individuals.

Removing direct identifiers does not make data anonymised. Any directly identifying data must be properly masked (i.e., removed by field suppression, pseudonymisation, or random fake data). However, masked direct identifiers can play an important role in many anonymisation pipelines by providing a means to link across data sources.

There is another point to consider along this spectrum, beyond the basic pseudonymisation but before anonymisation. This is sometimes called *strongly* pseudonymised data. Singling out means that a data subject's information can be isolated as unique from the population in which the data about them was collected. There are two different classes of indirect identifiers: knowable to the public, and knowable to an acquaintance. An indirect identifier that is knowable to the public represents the greatest risk since, by definition, it’s more broadly available to be used in a reidentification. We can therefore define two classes of pseudonymisation:

- Basic pseudonymisation:
  - Direct identifiers are replaced or removed through masking, and any additional information required to re-identify is kept separate and is subject to technical and administrative controls.
- Strong pseudonymisation:
  - Direct identifiers are replaced or removed through masking, indirect identifiers that are knowable to the public are transformed to ensure that data subjects are not unique in the target population so that they can’t be singled out, and any additional information required to re-identify is destroyed if required, or kept separate and is subject to technical and administrative controls.

3.8.2 Functional requirements

Anon.Mask:	**Replace/remove direct identifiers**
Anon.Transform:	**If required, transform identifiers to remove uniqueness**

Table 7 - Functional Requirements - Anon

Crucially, a ‘Human API’ approach will be used whereby a designated person(s) at each trial site will have access to a lookup table to allow them to identify a specific patient from their own cohort e.g., find the name corresponding to the numeric ID, as illustrated in Figure 3. Neither the other trial sites, nor the broader consortium will have access to this lookup table. To them, it will look like one cohort with no identifiable participants, as in Figure 4.

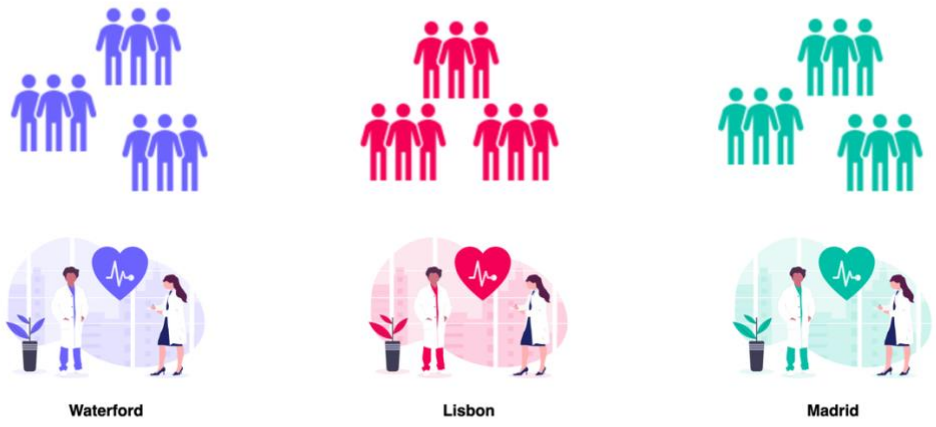


Figure 3 Trial Site Cohorts

Figure 4 Pseudonymised Cohort

3.9 Logger

3.9.1 Description

Although each feature should be responsible for its own logging, it helps to have a dedicated feature to collate the disparate logs to provide a more complete system-level view. All errors should be caught and handled gracefully. When errors happen, an application should never fail into an unknown state. It should always roll back any transaction it was performing, and "close" anything it may have opened (this is called "fail-closed").

3.9.2 Functional requirements

Logger.Collate:	**Bring together logs from different features**
Logger.Visualise:	**Visualise logs in human readable/searchable way**
Logger.Notify:	**If necessary, e.g., system roll-back scenario, notify developers**

Table 8 - Functional Requirements - Logger

3.10 Authorisation and Authentication

3.10.1 Description

When we talk about identity in a computer system or network, we mean the way the computer recognises who you are. When it verifies you are who you say you are, that's authentication (AuthN). When it uses your identity to figure out what you are or are not allowed to see and do, that is called authorization (AuthZ). The system of granting or denying functionality and information within your application or network is called *access control*. (Janca 2020)

3.10.2 Functional requirements

AuthZ.Verify:	**Verify user permissions**
AuthN.Verify:	**Verify user identity**
AuthN.Track:	**Track user identity**

Table 9 - Functional Requirements - Auth

3.11 Explainable AI

3.11.1 Description

D4.3 (Advanced Analytics Methodology) deals with this topic in much greater detail, but essentially it is the feature responsible for making sure that the reasons behind a model's output are understandable by a human.

3.11.2 Functional Requirements

XAI.Feature:	**Provide explanations of a model's features**
.Importance:	What features in the data did the model think are most important?
.Interaction:	What interactions between features have the biggest effects on a model's predictions?
XAI.Prediction:	**Provide explanations of a model's predictions**
.Explain:	For any single prediction, how did each feature in the data affect that particular prediction?

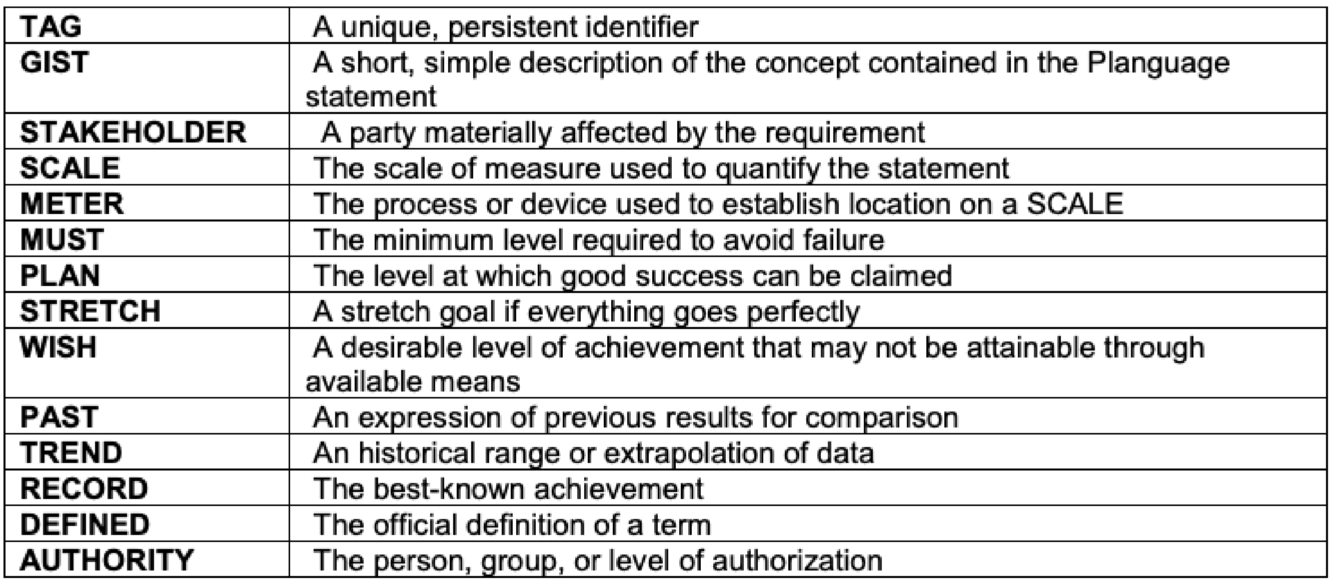
Table 10 - Functional Requirements - XAI

4 Quality attributes

This section specifies the non-functional requirements. Quality requirements should be specific, quantitative, and verifiable. Unfortunately, however, they are often vague and open to interpretation.

You can't evaluate a product to judge whether it satisfies vague quality requirements. Unverifiable quality requirements are no better than unverifiable functional requirements. Simplistic quality and performance goals can be unrealistic.

To address the problem of ambiguous and incomplete non-functional requirements, Tom Gilb developed *Planguage*, a language with a rich set of keywords that permits precise statements of quality attributes and other project goals. Each requirement receives a unique *tag*, or label, using the hierarchical naming convention described in Section 1.2. The *ambition* states the purpose or objective of the system that leads to this requirement. *Scale* defines the units of measurement and *meter* describes how to make the measurements.



TAG	A unique, persistent identifier
GIST	A short, simple description of the concept contained in the Planguage statement
STAKEHOLDER	A party materially affected by the requirement
SCALE	The scale of measure used to quantify the statement
METER	The process or device used to establish location on a SCALE
MUST	The minimum level required to avoid failure
PLAN	The level at which good success can be claimed
STRETCH	A stretch goal if everything goes perfectly
WISH	A desirable level of achievement that may not be attainable through available means
PAST	An expression of previous results for comparison
TREND	An historical range or extrapolation of data
RECORD	The best-known achievement
DEFINED	The official definition of a term
AUTHORITY	The person, group, or level of authorization

Figure 3 Planguage Keywords

Requirements often contain statements like the following[5]:

"The system must be easy to learn."

This is clearly not testable. What does "easy" mean? What does "learn" mean? An improvement on this in structured English would be:

"The system must be used successfully to place an order in under 10 minutes without assistance by at least 80% of test subjects with no previous system experience."

The Planguage version of this statement would be:

TAG:	Learnable.
GIST:	The ease of learning to use the system.
SCALE:	Time required for a Novice to complete a 1-item order using only the online help system for assistance.
METER:	Measurements obtained on 100 Novices during user interface testing.
MUST:	No more than 7 minutes 80% of the time.
PLAN:	No more than 5 minutes 80% of the time.
WISH:	No more than 3 minutes 100% of the time.
PAST [our old system]:	11 minutes <- recent site statistics.
Novice: DEFINED:	A person with less than 6 months experience with Web applications and no prior exposure to our website.

5 http://www.understandingrequirements.com/resources/2.23%20%20Quantifying%20Quality%20Requirements.pdf

Now the statement provides a great deal of information in a compact format. Additionally, it is testable and far less ambiguous than the previous structured English statement.

The non-functional requirements discussions naturally follow the functional requirements, and so will be updated here as they are agreed upon. The areas we intend to address are listed below, and over time these sub-sections will be enhanced with the relevant requirements.

4.1 Usability

Usability requirements deal with ease of learning, ease of use, error avoidance and recovery, efficiency of interactions, and accessibility. The usability requirements specified here will help the user interface designer create the optimum user experience.

4.2 Performance

State specific performance requirements for various system operations. If different functional requirements or features have different performance requirements, it’s appropriate to specify those performance goals right with the corresponding functional requirements, rather than collecting them in this section.

4.3 Security

Specify any requirements regarding security or privacy issues that restrict access to or use of the product. These could refer to physical, data, or software security. Identify any security or privacy policies or regulations to which the product must conform. If these are documented elsewhere, they will just be referred to rather than duplicating here.

(Janca 2020) provides an excellent summary of important security requirements. Rather than reinventing the wheel, we have included the list here, which can be expanded on:

- Encryption: In order to ensure the confidentiality of your data, it should be encrypted in transit (on its way to and from the user, the database, an API, etc.) and at rest (while in the database).
- Never Trust System Input: Any input to your system could potentially be tampered with or otherwise cause your application to malfunction or fail. Your program must be able to handle every type of input *gracefully*, even bad input. Input to your application means literally anything and everything that is not a part of your application or that could have been manipulated outside of your application. Examples of input to your application:
    - User input on the screen
    - Information from a database (even your own DB)
    - Information from an API (even one you wrote)
    - Information from another application that your application integrates with or otherwise accepts input from (this includes serverless apps and scripts)
    - Values in the URL parameters, cookie values, configuration files
    - Data or commands from cloud workflows
    - Images that you've included from other sites (with or without permission)
    - Values used from online storage
- Encoding and Escaping

  - The most commonly known security vulnerability in web applications is *cross-site scripting* (XSS); it's estimated to be present in over two-thirds of web applications. There are multiple mitigations for this risk: the Content Security Policy Header (CSP), input validation, and output encoding. The output encoding requirement only exists in order to prevent XSS. Encode (and escape if need be) all output.
- Third-party components
  - Scan third party components for known vulnerabilities. Use available tools and even consider baking into CI.
- Passwords
  - A secret store is a software vault that encrypts your secrets, and then allows your application to access them programmatically (via the code in your app or CI/CD pipeline build process). You can store certificates, credentials (username and password), connection strings, hashes, and anything else that you consider a "secret" that your application will use.
    - When an application logs in to a database, the connection string it used is a secret. When a web server has a certificate and key pair that it uses to enable HTTPS for the web app, the private key of the key pair used for that cert is a secret. Private keys, API keys, hashes, passwords, and anything else that is confidential that is used by the application, is considered a secret.
  - User passwords
    - They should be stored in the database, in a *salted* and *hashed* format. Hashing means a one-way cryptographic process, it cannot be undone. The process of salting is adding a unique, long value to a value before you hash it, in order to increase entropy and make it even more difficult for a potential attacker to crack or guess a password.

- Your salt should be at least 28 characters (but preferably much longer), be generated by a secure random number generator, and it should be unique for each user of your application. Store the salt value in the database along with the hashed value of the salt+password, for each user. The salt is not a secret.
- Ensure passwords for your application's users are long, but not necessarily complex; having an uppercase letter, a lowercase letter, a number, and a special character is good. Allow users to enter passwords of up to 64 characters. Verify that new users' passwords have not previously been in a breach by comparing sha1 hashes in a range, using a service such as HaveIBeenPwned. (https://haveibeenpwned.com/API/v3#SearchingPwnedPasswordsByRange)
- Never verify if it was the username of password that was incorrect when alerting the user that they have failed when logging in. Giving this information away allows potential attackers to harvest usernames (verifying that a user does or does not use your system).
- *Forgotten password features* should verify the user's identity before sending a password reset link via email or SMS, either by using another form of authentication or security questions. If the user fails authentication, never reveal which part failed. Never allow the user to reset the password directly on your system; always send it using an "out of band" form of communication (via an email or SMS link) as a second form of communication. The reset link should be one-use only, and expire within 1 hour if not used. Always log that the password has been reset (or an attempt, if the user was not successful), and send an email or SMS listed on that account to confirm that the password has been reset. If a user attempts to reset their password 10 times unsuccessfully in 24 hours, lock their account on the IP address that the requests are coming from, log the situation, and trigger an alert.

  - Rules for passwords
    - Hash and salt all user passwords. Make the salt at least 28 characters.
    - All application secrets must be stored in a secret store.
    - All accounts used within the application must be service accounts (not a human being's account). They must be unique for each application and should follow the concept of least privilege.
    - See more here: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
  - HTTPS everywhere
    - Only allow your site to be accessible via HTTPS. Redirect from HTTP to HTTPS. If someone attempts to downgrade a connection, redirect them. This can be done via security headers in your code, or settings on your server.
  - TLS
    - Ensure you are using the latest version of TLS for encryption.
  - Backup and rollback
    - All the data that your application uses, creates, or stores must be backed up regularly, with a second, geographically differing location for weekly backups. The system must be capable of data rollback, in the event of malware, security incident, or ransomware. Your backup and restore procedures must be tested and practiced.
- Parameterised Queries
  - When we use parameterised queries (in SQL they are called stored procedures), we send parameters and the name of the query we want to run to the database, rather than creating a line of code by concatenating user input to create a string

and then sending it to the database as a command. The difference is that with parameterised queries if an attacker attempts to add their own code via user input, the application will send it in one of the parameters, and it will not work. This is because the parameters are interpreted by the database as data, never as code, which makes injection attacks nearly impossible.

- URL parameters
  - Users have access to the address bar in browsers. So do attackers. Don't put any variables in URL parameters that matter to your application. Only pass values of zero importance, such as which language the user wants to view the page in.
- Least privilege
  - Ensure your application enforces the principle of least privilege, especially in regard to accessing the database or APIs. Your application should also only use a service account to call APIs, parameterised queries, or any other call that requires an account, e.g.
    - The service account that calls your database from your application should only have CRUD permissions, never database owner.
    - Ideally one service account will have read-only, for "select" calls, while another will have CRUD access when insert, update, or delete is required. Use the read-only account whenever possible.

4.3.1 Authentication

Authentication is proving that you are indeed the real, *authentic*, you, to a computer. A "factor" of authentication is a method of proving who you are to a computer. Currently, there are only three different factors: something you *have*, something you *are*, and something you *know*. (Janca 2020)

4.4 Safety

Specify requirements that are concerned with possible loss, damage, or harm that could result from use of the product. Define any safeguards or actions that must be taken, as well as potentially dangerous actions that must be prevented. Identify any safety certifications, policies, or regulations to which the product must conform.

4.5 Availability

The extent to which the system's services are available when and where they are needed.

4.6 Integrity

The extent to which the system protects against data inaccuracy and loss.

4.7 Verifiability

How readily developers and testers can confirm that the software was implemented correctly.

4.8 Robustness

How well the system responds to unexpected operating conditions. It is essential that our application not only has backups as part of the design, but quick rollback of data and restore of systems:

- Backups must be stored in a geographically differing location than your database and web server.
- Backups must be in a secure location (both physically and digitally) and protected just as you would protect your production database or web server.
- Schedule rollback practice. When there are major changes to production systems, ensure a rollback script has been made and tested.
- Your database, your configurations, and your application code must be backed up.

5 Internationalisation and localisation requirements

Internationalisation and localisation requirements ensure that the product will be suitable for use in nations, cultures and geographic locations other than those in which it was created. Such requirements might address differences in currency; formatting of dates, numbers, addresses, and telephone numbers; language, including national spelling conventions within the same language (such as American versus British English), symbols used, and character sets; given name and family name order; time zones; international regulations and laws; cultural and political issues; weights and measures; and many others.

Since the user trials will take place in Ireland, Spain and Portugal we know that English, Spanish, and Portuguese must be catered for. As the functional requirements advance then these requirements can be added here.

6 Conclusions

"Progressive refinement of detail" is a key operating phrase for requirements development, moving from initial concepts of what is needed toward further precision of understanding and expression (Wiegers and Beatty 2013). This deliverable captures the initial thinking around many of the key software features, both in terms of functional and non-functional requirements. We intend to use this as a 'living' document between deliverable iterations, capturing changes in our thinking and priority shifts in the project, hence some sections are created but the content is labelled 'TBD' or similar.

As the research and development work progresses, there are many useful tools that we will leverage e.g., context diagrams, ecosystem maps, etc. All of these will be added to this document for completeness by the next update, but most likely much sooner, as part of the working document.

One of the key decisions to be made is where to deploy the software for the trial i.e., self-hosted OpenStack servers or a public cloud such as AWS. The structure of this document, however, will remain the same, regardless of the deployment option so that we ensure the reusability of the research and code that goes into the deployment.

7 Bibliography

Ameisen, Emmanuel. 2020. *Building Machine Learning Powered Applications: Going from Idea to Product.* O'Reilly.

Aquasmart Project. 2015. *Redmine.* Πρόσβαση March 19th, 2015. https://www.aquasmartdata.eu/redmine.

Beck, Kent et al. 2001. *Manifesto for Agile Software Development.* Πρόσβαση March 18th, 2015. http://agilemanifesto.org/.

Brown, N. 1996. «Industrial-strength management strategies.» *IEEE Software* 94-103.

El Emam, Khaled, και Luk Arbuckle. 2020. *Building an Anonymization Pipeline: Creating Safe Data.* O'Reilly.

Git contributors. 2015. *git --distributed-is-the-new-centralized.* Πρόσβαση March 19th, 2015. http://git-scm.com/.

Janca, Tanya. 2020. *Alice and Bob Learn Application Security.* Wiley.

McLaughlin, John. 2015. *Creating User Stories.* TSSG. 19th March. Πρόσβαση March 19th, 2015. https://www.aquasmartdata.eu/redmine/projects/aquasmart/wiki/UserStory.

—. 2015. *How to Use Git.* Aquasmart Project. 19th March. Πρόσβαση March 19th, 2015. https://www.aquasmartdata.eu/redmine/projects/aquasmart/wiki/UsingGit.

Munn, Michael, Sara Robinson, και Valliappa Lakshmanan. 2020. *Machine Learning Design Patterns.* O'Reilly.

Nelson, Catherine, και Hannes Hapke. 2020. *Building Machine Learning Pipelines.* O'Reilly.

Robertson, Suzanne, και James Robertson. 2013. *Mastering the Requirements Process.* Addison-Wesley.

Wiegers, Karl, και Joy Beatty. 2013. *Software Requirements.* Microsoft Press.