TagItSmart

Smart tags driven service platform for enabling ecosystems of connected objects

Grant agreement 688061

Qualytics

D1.2 Qualytics Software and documentation

Deliverable ID:	**D1.2**
Deliverable Title:	**D1.2 Qualytics Software and documentation**
Project acronym	
Project title:	
Extension acronym:	**Qualytics**
Extension title:	**Continuous product quality improvement driven by smart tagging and data analytics**
Contractual date of delivery:	

Executive Summary 3
1. Introduction 4
1.1 Scope of the deliverable 4
1.2 Contributions to other WPs and deliverables 4
1.3 Structure of the deliverable 4
2. Business case deployment 5
2.1 Business System 5
2.1.1 New quality control approach 5
2.2 General scenarios and advantages 7
3. Smart tags 9
4. Software development 13
4.1 Technical architecture 13
4.2 Development View 14
4.3 Use Cases details 17
4.3.1 File Transfer 17
4.3.2 Training and Training Scheduling 18
4.3.3 Detection 20
4.4 Logical View 21
5. Guideline for using the software 22
5.1 Library 22
5.2 REST API 26
5.2.1 Description 26
5.2.2 Service explanation 33
6. Conclusion 38

Executive Summary

This document reports about the software development based on the user requirements defined in D1.1. The work is based on an intensive analysis of the problems in existing manufacturing process (assembling), esp. from the quality control point of view and available technologies, which can be used for improving that process.

The goal of the deliverable D1.1 is to demonstrate that the Qualytics approach can be applied in a broad range of use cases. This deliverable elaborates on two additional conceptual aspects:

1) The concrete scenario from the quality control, where Qualytics will be applied;

2) The concrete TagsIt technology, which will be applied in the selected scenario.

However, the main part of this work is related to the technical/software development of the Qualytics system (which outcome is software code). Qualytics extends our data analytics approach for the quality control in the home appliances domain. The focus is on the refrigerators, although the approach can be applied on an arbitrary scenario from this domain.

We modified initial scenario (QR based) slightly so that the thinfilm technology can be used. The main modification is related to the ranges of the temperature that will be considered in the use case. Thinfilm delivered us the tags, which are operating in the range 15-30 grades. It monitors the temperature and flags if the temperature has changed from that range.

We received several tags for the experimentation. Since the tags are for the demo purposes, they can be used many times, which will enable us to do the various types of validations, as planned in Task 2.1.

However, we have designed the solution in the way that enables that the type of the sensors used for monitoring can be changed, by assuming that the data format (structure) will be not changed.

This deliverable is accompanying documentation to the software provided in the task T1.3. We provided two methods for reusing the software (library and REST API) and created a detailed guidelines how the software can be used

1. Introduction

1.1 Scope of the deliverable

This deliverable is part of the WP1 (Concept and Development M1-M6), which goal is to understand the requirements for the envisioned service and develop the Qualytics Platform. In particular the work is related to the software development (T1.3).

This deliverable reports about the software development that is based on the user requirements defined in D1.1. Mainly, it provides the documentation related to the developed software.

This deliverable is accompanying documentation to the software provided in the task T1.3.

1.2 Contributions to other WPs and deliverables

This deliverable will serve as basis for the work in Task 2.1: Preparation/Piloting, month M2– M6 and Task2.2: Validation, month M2 – M7.

1.3 Structure of the deliverable

The deliverable is structured in the following way:

In the second section, we provide the information about the desired business system (as an end to end implementation).

In the third section we provide some information about the selected TagIt technology.

In the fourth section we provide some documentation related to the software development.

In the fifth section we provide detailed guideline how the software can be reused.

In the sixth section we provide concluding remarks.

2. Business case deployment

In this section we provide details of the relevant business scenario based on the requirements provided in D1.1.

2.1 Business System

In order to illustrate the approach, in this section we present the details from the industry use case, which has been performed for Whirlpool in the domain of the ZHQ (Zero Hour Quality), a quality management approach that aims in having a valid product (high quality) from the beginning of the usage (zero hour) of a product/device. In order to achieve this, there is a very strict process of functional testing after assembling process that consists of 10+ measurement stations where different parameters of the operation of the assembled devices are checked.

2.1.1 New quality control approach

The quality control process is done using a traditional threshold-based process, producing so called out-of-limits (OOL) alarms. It consists of defining an upper and lower threshold so that when a measurement goes above the upper limit or below the lower one, an alarm is triggered. Thereafter engineers will inspect the parameter that is out of limits and determine whether it is an anomaly or not and decide which action to take (for example, run a procedure).

The main issue is that the thresholds are defined by experts and might be suboptimal, i.e. they could be refined, but the data for the refinement is missing.

We briefly list the most important elements from the Whirlpool use case:

- For each refrigerator that Whirlpool produces, a functional test is performed by attaching sensors and measuring a number of parameters;
- Functional tests validate that a refrigerator works as it should and represent one of basic measures of quality control;
- For the dataset provided following 6 (behavioural) parameters are measured (see Figure 1 for an example):
  - Temp. Destra (Rossa), Temp. Sinistra (Nera), T EV Cap Fz., T EV Cap. Fr., Cos Fi, Potenza;
- There is a need to include contextual parameters into analysis as well. Following contextual parameters can be provided:
  - Time when the functional test was performed;
  - Temperature of the environment in which functional test was performed;
  - Location (part of the factory) where the functional test was performed;
- The dataset is too large to be processed on a single machine and that is why we need a scalable, distributed solution which is able to bear with Big data;
- The feedback from customers, (esp. through maintenance process) is missing;

- The dataset should be extended with the customer-driven information.

In the following figure the real architecture that integrates quality control approach, as described above, is shown. The most important part is Q-Box, which implements a simple detector and can be easily moved from one to another place.

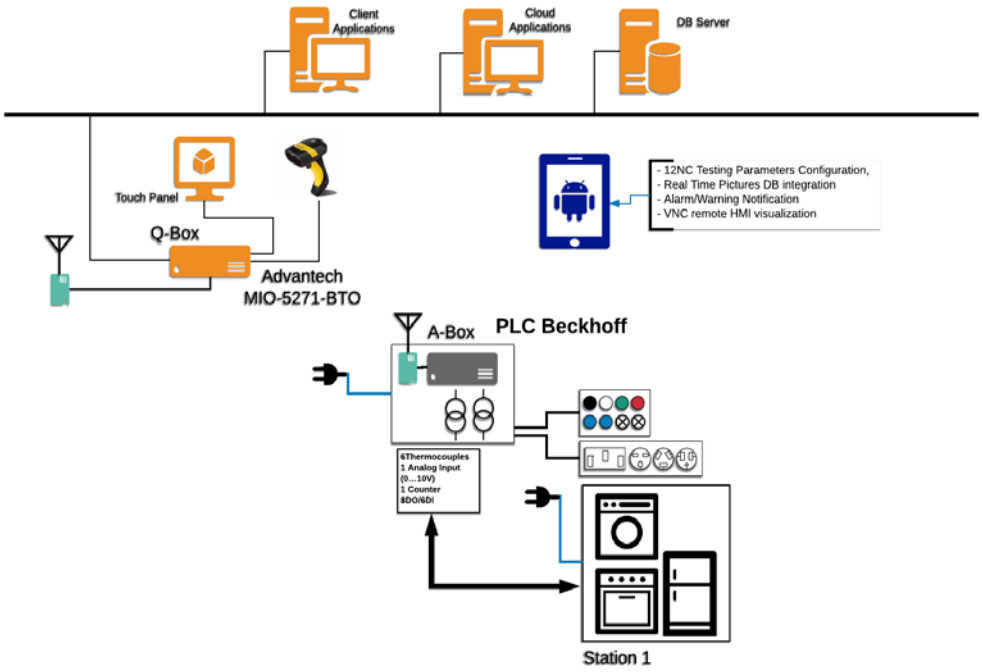

Figure 1. Improved quality control process using portable measurement device (Q-Box)

In the following figure we present a possible usage of the system in the quality control. Based on the past data, the system learns what normal behaviour is. This knowledge is used in the simple detector for anomaly detection.

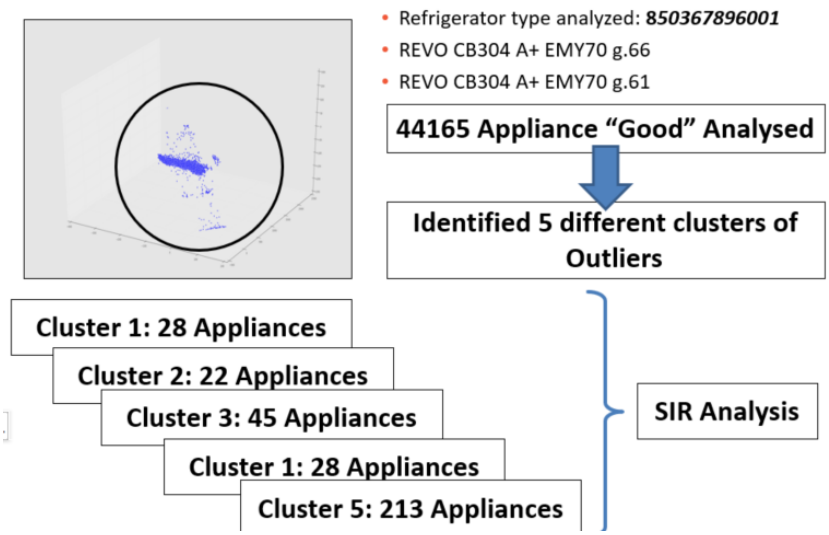

Figure 2. Example for the usage of simple detector

2.2 General scenarios and advantages

We argue that there are many business scenarios that can be supported by the presented approach (in the case that also other techniques for the smarttaging can be used).

We mention here two main applications in maintenance service:

1. To record that a device behaved (at home) in an anomalous way
    - Smarttags values indicate the existence of the anomalous temperatures;
    - This data is used for matching with the functional tests of that device in order to create a basis for analytics;

2. To check how a device will behave
    - To check the behaviour of the device, which has been repaired (maintenance service done on demand);
    - Such a device is equipped with several sensors/tags (like Thinfilm), which record the behaviour of the device in a limited time period (1-2 days);
    - The data from tags are analysed in order to check if some anomalous behaviour;
    - An approach similar to the „holter" for patient monitoring (eHealth).

The main advantages are:

- A very efficient way for improving:
  - Maintenance service
    - Better diagnostics (more data to be used)
    - Personal with less experience can perform the tasks
  - Functional test process
    - Data from the anomalies at home-usage will be used in the data analytics to find causes of some potential problems in the usage of devices
  - „Self-alarming“ service
    - A customer can check on his/her own if the device behaved anomalously (related to the values for the temperature)

We argue that the usage of different smartags (e.g. QR codes) would enable even more additional scenarios. The main advantage is the possibility to model more complex situations of interest. More information is provided in Final report.

3. Smart tags

Based on the specifications and the discussion with partners we decided to use the Thinfilm's Temperature Sensor. It monitors the temperature and flags if the temperature has:

- Been above the predefined High trip value
  and/or
- Been below the predefined Low trip value

The monitoring starts immediately after being activated. Activation starts by a short click (<1s) on the "Press to Start" push button and the "ON" indicator turns on. The excursion duration must last continuously at least during the predefined duration called "Time Delay" – see description later. Once activated, the Temperature Sensor cannot be Reset.

Following figures provide more information about the key elements and the activation behaviour of the Thinfilm Temperature sensors.

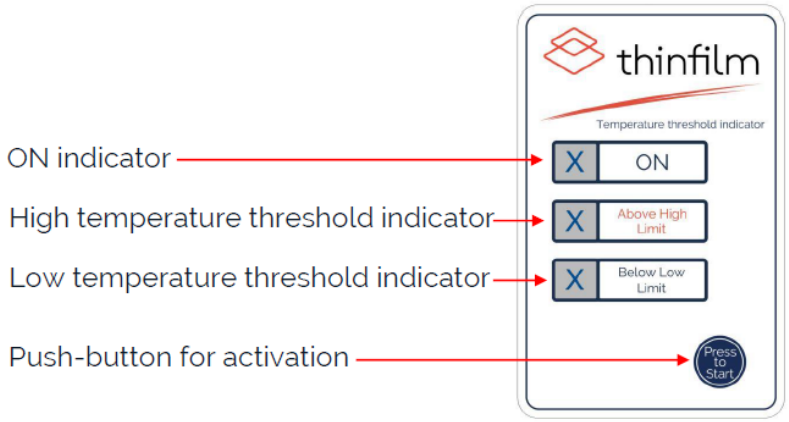

Figure 3. Key elements of a Thinfilm Temperature Sensor

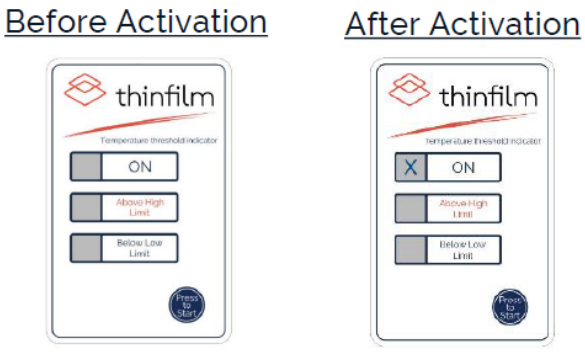

Figure 4. Thinfilm sensor before and after activation

We received several tags for the experimentation. Since the tags are for the demo purposes **they can be used many times** (the difference to ordinary sensors), which will enable us to do the various types of validations, as planned in Task 2.1.

The following figure shows the back side of the sensors we received for testing our approach.

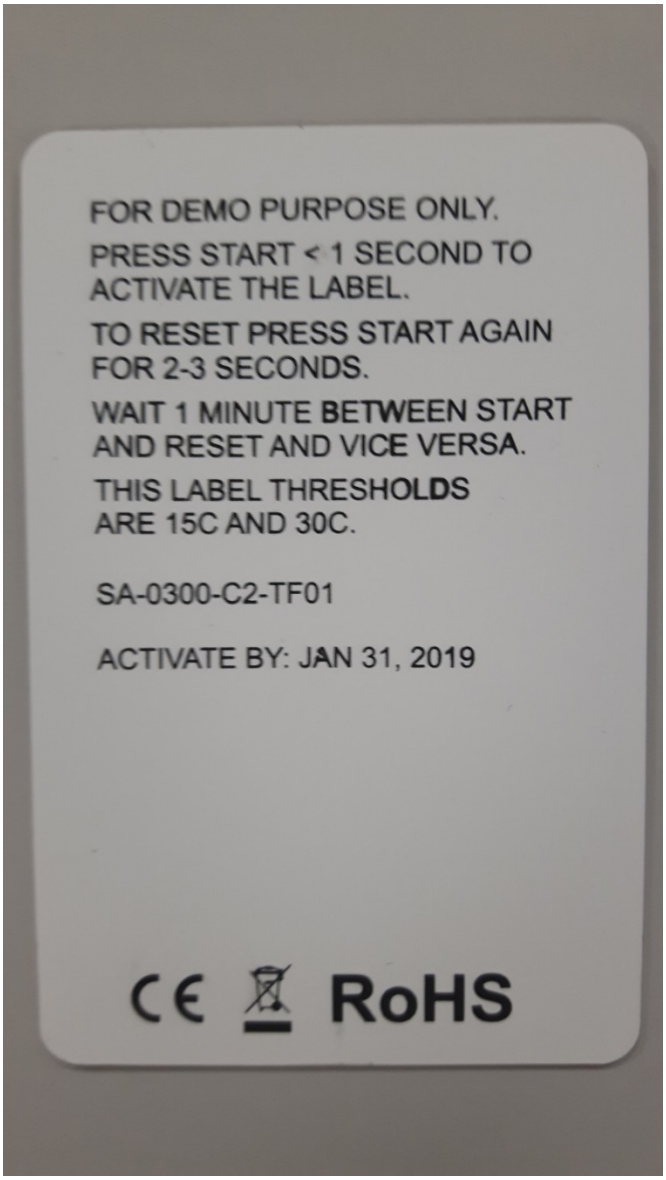

Figure 5. Demo Thinfilm sensor

The following figure illustrates the functionalities of the thinfilm sensor, which are matching the requirements from D1.1. It shows the temperature accuracy and good/faults regions.

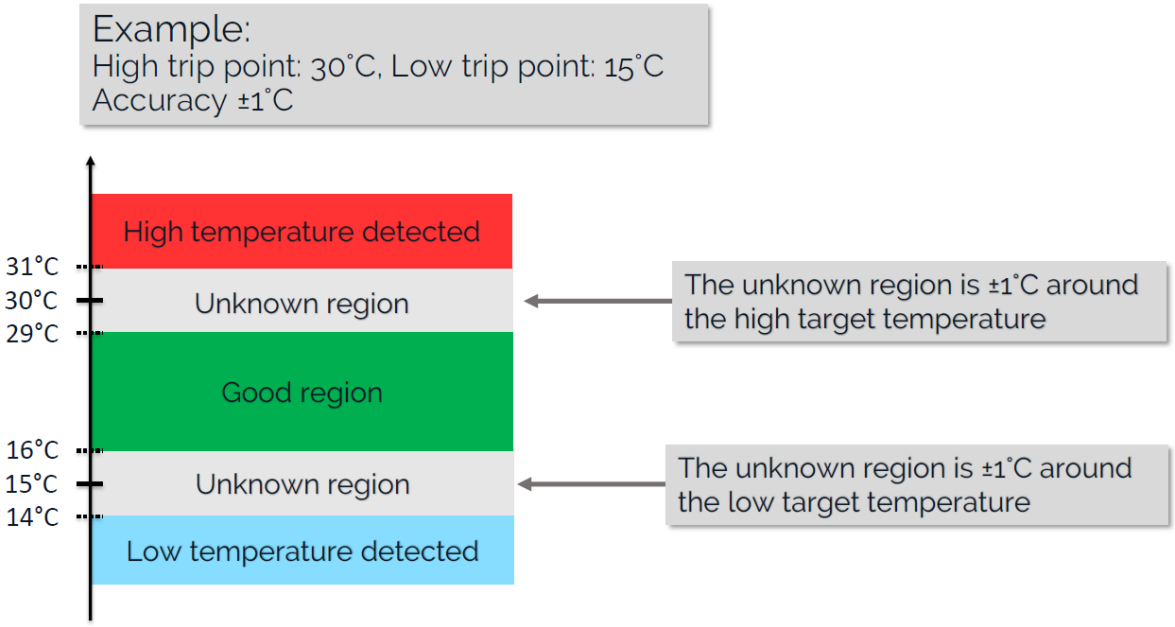

Figure 6. Thinfilm sensor logic of processing changes in the temperature

Regarding the temperature sensors basic functionality, the temperature monitoring starts immediately upon activation. The measured temperature must be above T_{high} (or below T_{low}) for a continuous time equal to TimeDelay to generate a valid trip. A valid trip is indicated by the corresponding 'X-icon' on the display. The timer is reset if the temperature goes above/below $T_{high/low}$ before reaching TimeDelay. This is shown in figure below.

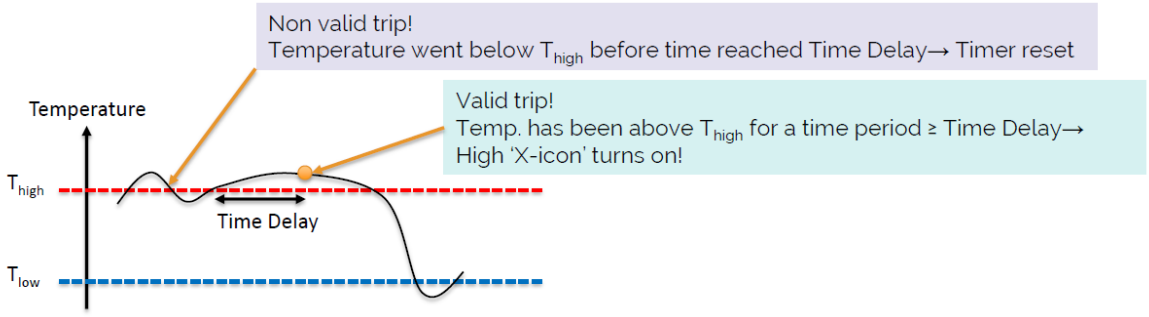

Figure 7. Thinfilm sensor illustration of processing changes in the temperature

Key attributes of the temperature sensors are:

- Simple to use

- Explicit START and Label ON indicators
- Easy to read when excursion occurred –'X' indicator
- Small form factor and ultra thin for space savings
- Customer-defined thresholds integrated on low-cost system
- Customizable graphics
- No pre-conditioning nor stringent storage requirement
- Printed technology for high-volume production and low unit cost

As explained in the received documentation, Thinfilm Smart Label Temperature Sensor Specifications are:

- Factory configurable to satisfy various customer requirements
- Temperature accuracy: ±1C or ±2C options
- Number of temperature trips: 2 (1 threshold capable)
- Temperature trips set range: -3oC to 30oC
- Allowed temperature excursion time "Time Delay" :
- 10 minutes to 15 hours delay in 30 minute increments
- Duration of the excursion (not cumulative time)
- 7-day operation
- 1-year shelf life
- Visual indication "X" excursions above/below presets
- Cannot be reset once activated
- Customizable graphics
- Requires no pre-conditioning, nor stringent storage temperature requirements
- Volume shipments starting in 4Q, 2017

4. Software development

In this section we describe the main parts of the software development related to the Platform (Section 2).

4.1 Technical architecture

Based on the requirements from the previous section, as well as the concrete requirements from the underlying use case (see next section) and by leveraging our past work, we have developed an efficient and scalable system for data-driven quality control.

The main innovation is related to the data-driven management of usual/unusual behaviour that enables us to treat unusualities as first class citizens and consequently to provide support for the whole life cycle of the usual/unusual (anomalous) behaviour.

The system will be based on the D2Lab Framework (TRL8) for developing data analytics solutions.

The framework consists of the following components:

1. Data storage layer for storing training data, meta-data, models and results of detection;
2. Processing layer provides different execution engines for scalable, distributed and parallel algorithm execution;
3. Data analytics layer implements different machine learning and pattern recognition algorithms on top of processing layer and technologies such as Hadoop, Spark and CEP;
4. User interaction layer is used for data ingestion (most often through REST services) and presentation of data and enables the user to interact with the system and experiment with different algorithms and parameters through the web portal;
5. Integration layer provides flexible and scalable integration of components in the system;
6. Security layer makes the interactions secure through different concepts such as OAuth protocol, two factor authentication and HTTPS supported data transfer.

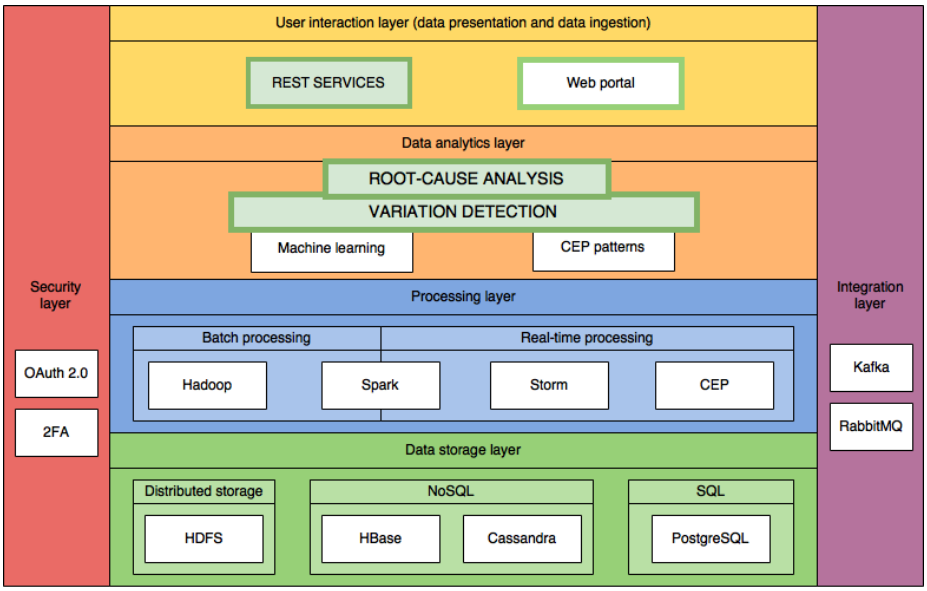

Figure 8. Technical Architecture

As shown in Figure 8, the main extensions in the architecture related to the Qualiytics Environment are:

1. Variation detection service that enables the detection of variations in various structures (e.g. production process) in an easy (unsupervised) way;
2. Root-cause analysis supports the analysis of the anomalies/variations discovered by the Variation detection service in order to find the most important causes (root-causes);
3. Rest services (additional) are responsible for the interaction with the Cloud where challenging data analytics tasks are executed.

4.2 Development View

The following UML diagrams provide the implementation details. It shows the most important classes used for orchestration and execution of training (learning) procedure. OrchestratorCore performs adaptation, preprocessing, training and postprocessing using implementations of corresponding interfaces. It uses DataApiProvider to communicate with the web service and get information about the training setup. Orchestrator, as initializer of a series of MapReduce jobs represents an entry point of training procedure.

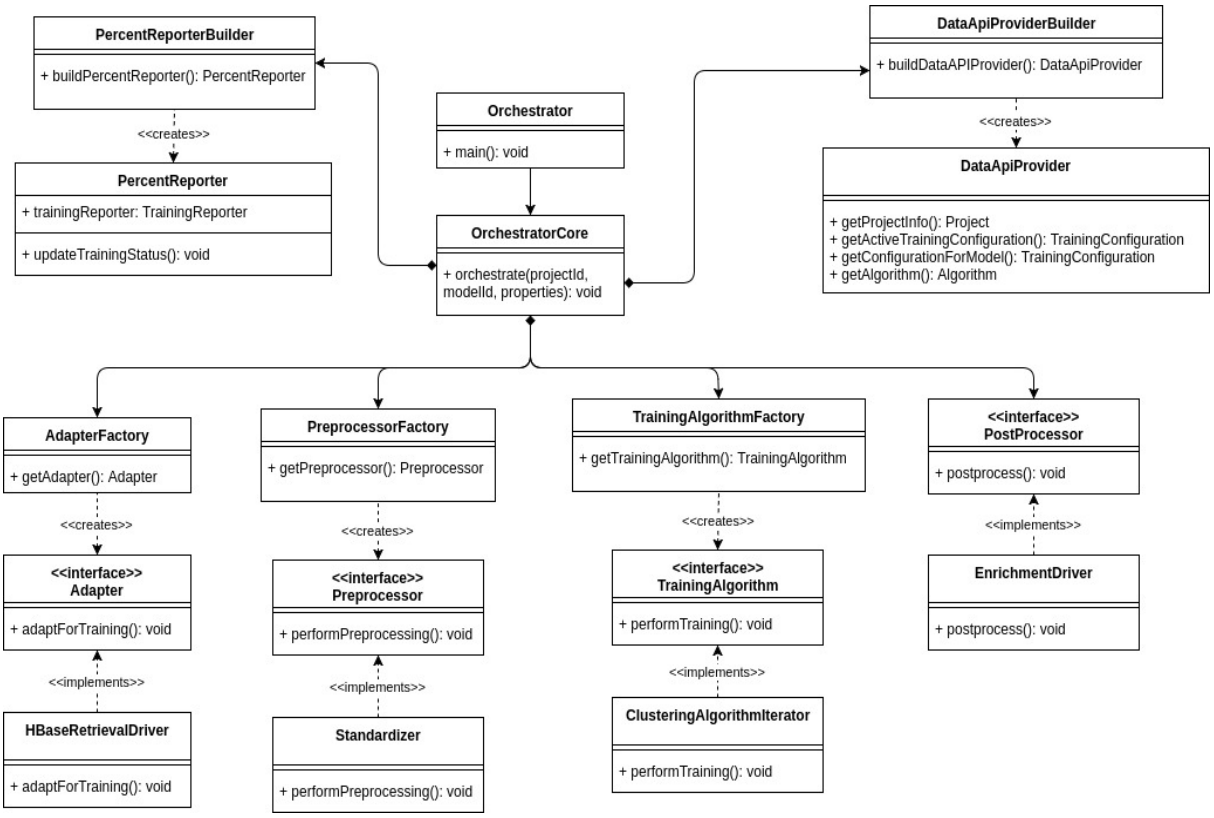

Figure 9. Overview of the most important classes

Figure 10 depicts classes related to execution of clustering based anomaly detection and concretely implementation of KMedoidsUsingFAMES training procedure. There are classes related to repetition, iteration and clustering itself. Classes for each step of training procedure are given, both during initialization and during the part of cluster assignment and cluster representative selection.

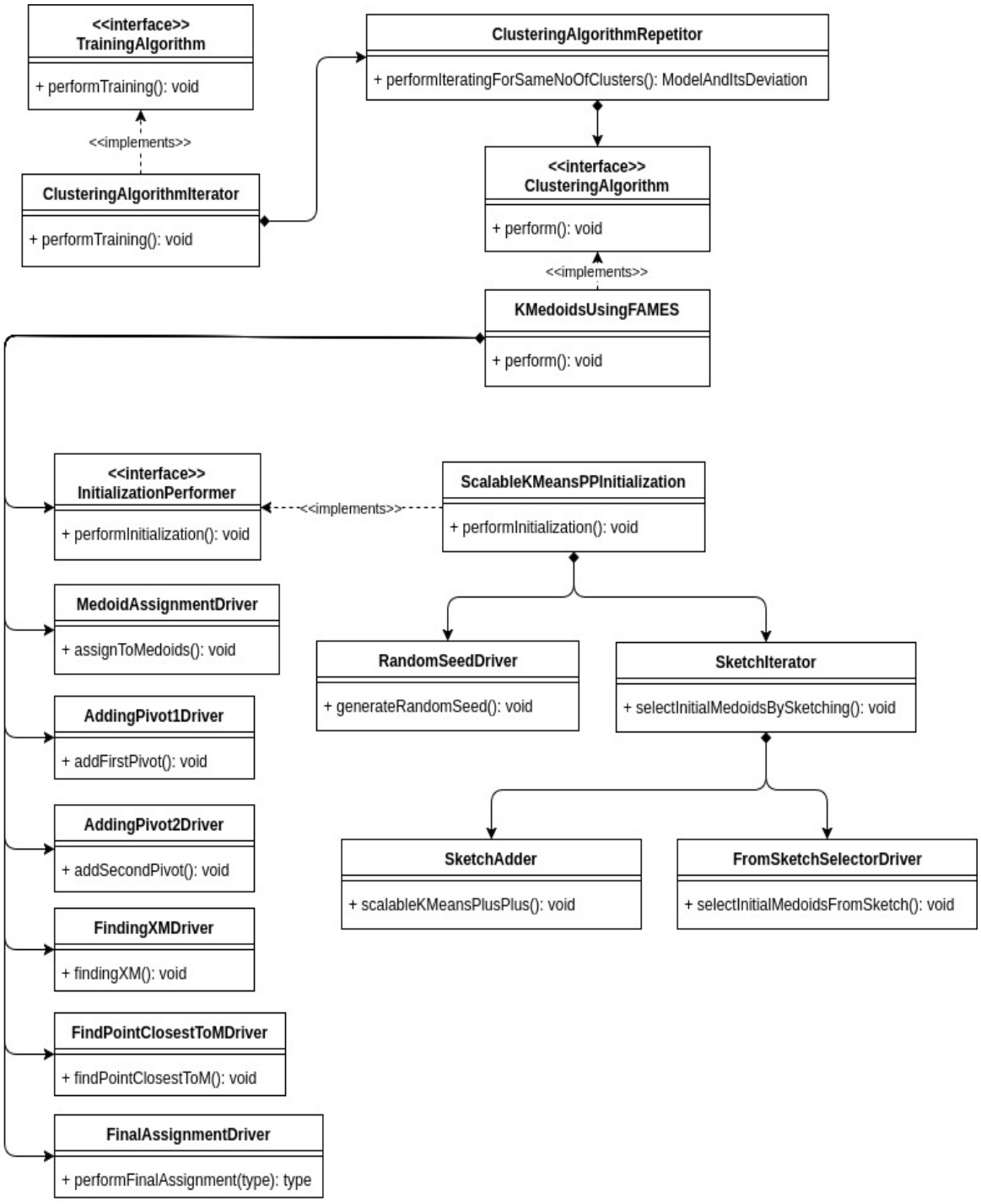

Figure 10. Classes related to execution of clustering-based anomaly detection

The most important classes related to job scheduling are shown in Figure 12. Additionally, Figure 11 shows classes related to upload of training files and real time anomaly detection. Scheduling is performed using Apache Oozie and hence appropriate client is used.

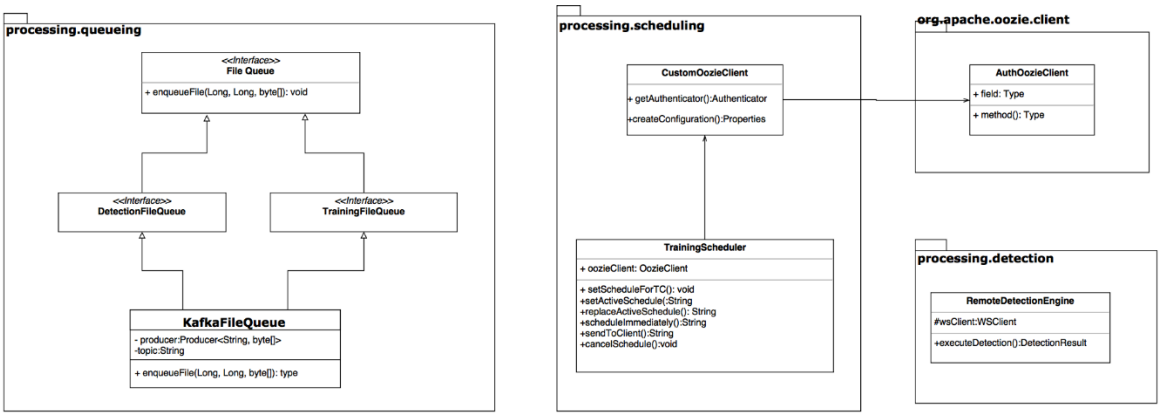

Figure 11. Classes related to job scheduling

4.3 Use Cases details

In this subsection we provide more explanation about the three above introduced use cases.

4.3.1 File Transfer

First and foremost the use should be able to upload a file to the system, either for training or testing. Presumably, that will be done using a REST API, and in a more or less automatized way on the client side. Since the system should be ready to accept a larger number of files (“big data”), an asynchronous method has been developed. It is shown in the figure below.

Each kind of upload (training and detection) is executed in the same way. The operator, in the broadest sense as an automated client or human input, uploads the file to the Data Web Service, using a REST API. As soon as the file is received from the service and stored in some temporal storage, the user gets an approval and an id which bill be used for later reference.

However, the file is not exactly stored now in the operational databases. Currently, the file is only in the temporal storage, which is relatively fast and simple. The web service puts the file into the Kafka queue for further processing, and also writes an entry into the relational database that a file has been received and it should be processed. That way we can track each file, and its state (has the processing failed, reason of the failure, failed for an internal reason, parsing error, corrupted file etc.). The data worker reads files from the queue, one by one or in smaller batches, and steadily loads those files into the relational database (where mostly metadata about the files is stored) and the raw data from the file into HBASE. The worker shall accordingly change the file status in the relational database.

Before actually writing the files, the worker will read the file schema from the relational database, for that project. The schema describes the format of the file, and imposes certain limitations, such as the minimal and maximal values that can be present for each parameter in the file. The worker will execute filtering and cleaning on the file according to that schema.

The worker can also publish a message to the broker as a trigger to any component interested that the file has been stored. This message is described in a later section.

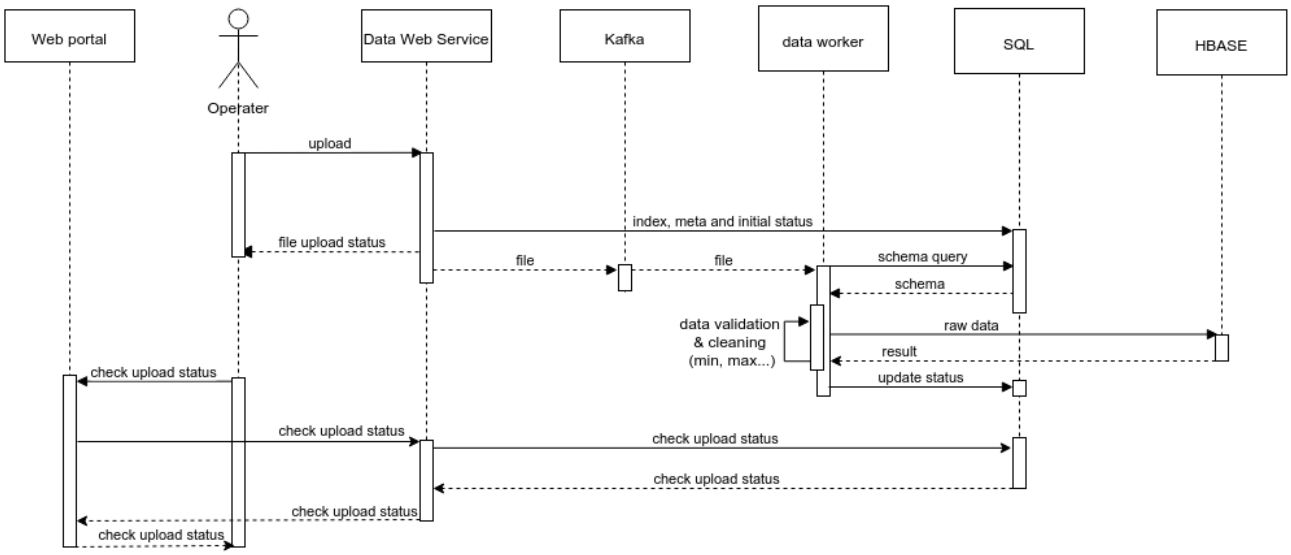

Figure 12. Upload Sequence

To confirm what has actually happened with the uploaded file, the user may and should (at a bit later time) go to the web portal and check the state of the upload. The web portal will call the data web service which will relatively simply read the state from the relational database.

4.3.2 Training and Training Scheduling

The user of the system will be able to schedule the training to start at a specified time or recurring or even start it one time only. Those functions will be setup from the portal, and the detail sequence is shown in the figure.

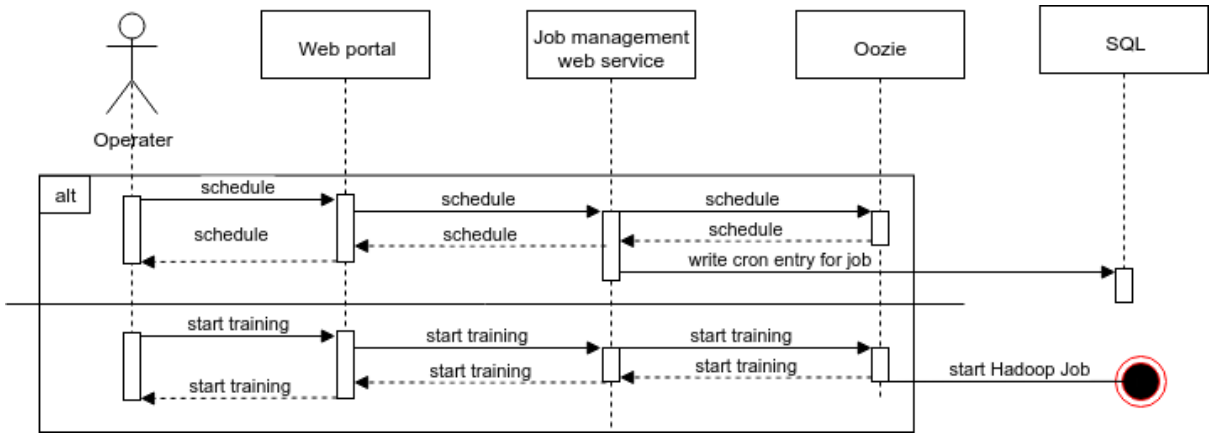

Figure 13. Training Schedule

A user can choose one out of two operations with the scheduling mechanism. They can schedule using the web portal, which will use the job management rest service backen to schedule with Oozie, and also write that information to SQL to the appropriate project configuration, as a cron[1] string.

Alternatively, we can also use Oozie to start the job immediately. Nevertheless, after the instruction sent to Oozie, the training process is the same, as shown in the figure.

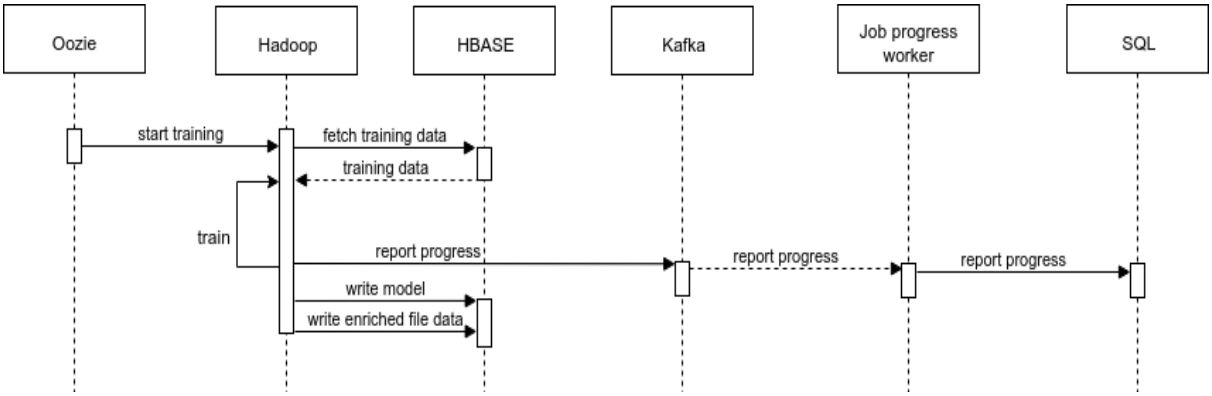

Figure 14. Training

When Oozie hits a specific time to start the training process, it will send an instruction to Hadoop with the job configuration. Each time the job progress changes, starts, stops, fails or is somewhere in between (for example on each 5% progress) it will publish a message on the broker with that job id and the current progress. That message can be consumed by any interested party, but most importantly by the job progress worker which will update that

[1] http://www.nncron.ru/help/EN/working/cron-format.htm

information to the job record in the relational database. That status should be observable through the web portal for the user to have a sense of progress.

4.3.3 Detection

Naturally, the system should provide the option to detect anomalies in the client's files. This is shown in figure below.

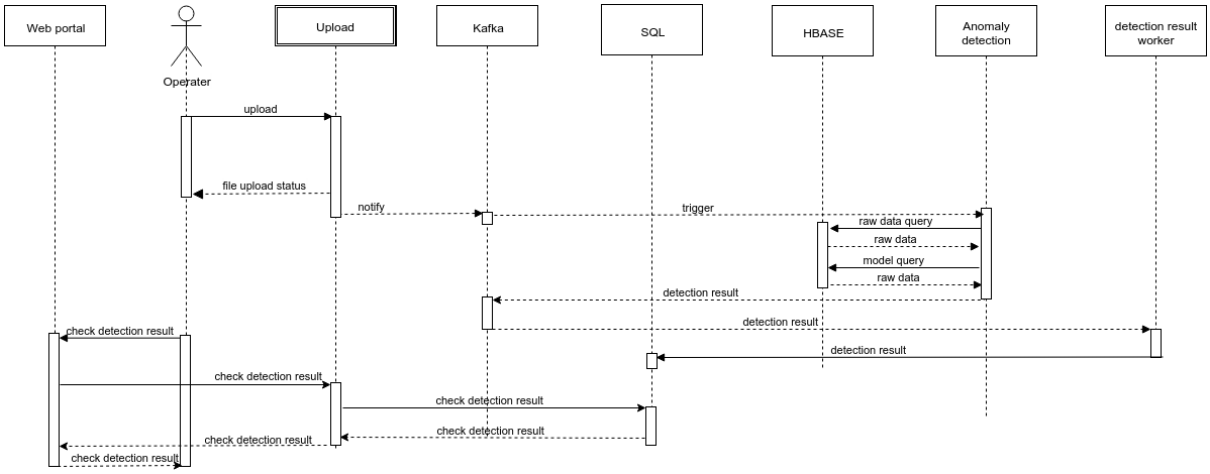

Figure 15. Testing Sequence

Please note that the *Upload* task is a complex one, and described on the *File transfer* subsection. After the upload sequence is finished, a trigger is dispatched using Kafka to the *Anomaly Detection* component that a file is ready for detection. The trigger contains elementary file information such as file id and project id. The anomaly detection component can read the project configuration (i.e. algorithm, distance) based on that reference, load the mode[2] from HBASE and apply the algorithm on the new file. The detection result is published to the broker for any interested component, but most importantly, the *detection result worker* which specifically listens to these messages and writes the detection result into the database.

The user may check the results of the anomaly detection progress from the portal at a later point when the detection is finished, which should in general be in a very short time since the anomaly detection does not take too long (matter of seconds).

[2] This model may actually be cached in the future, since the principle of time locality may be applied, meaning that if one file is tested using one model, another file may come in soon using the needing the same model.

4.4 Logical View

The following UML diagram describes the deployment process.

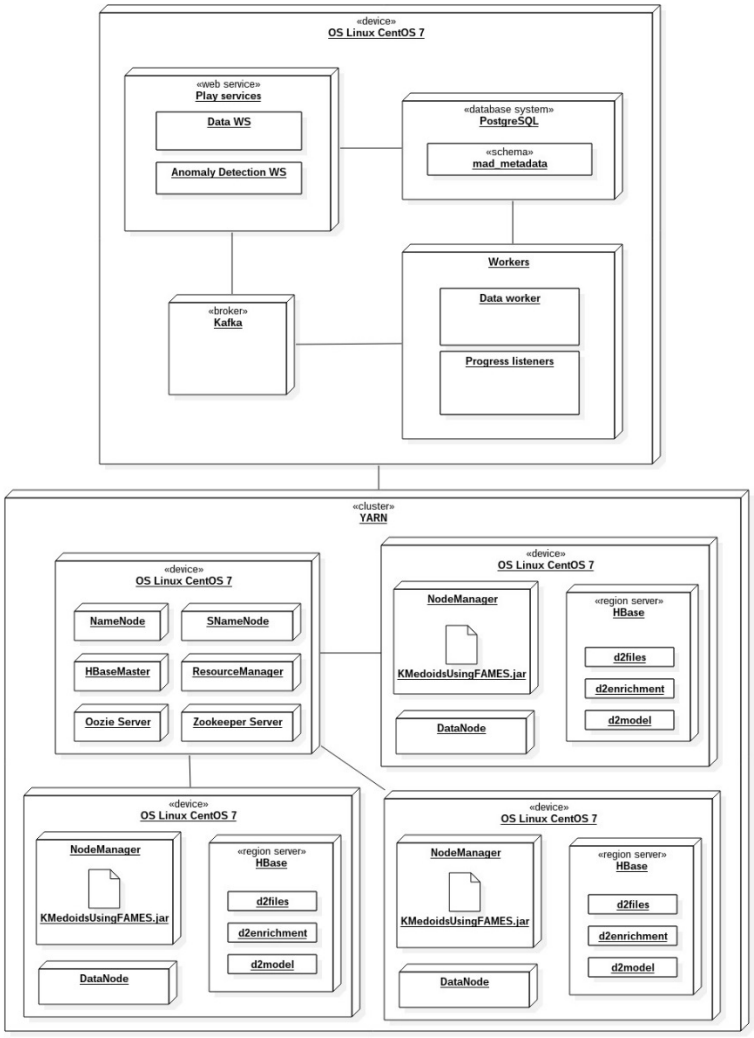

Figure 16. Data Management: Deployment

5. Guideline for using the software

In this section we provide a guideline that explains how the produced software can be used.

We developed two ways of reusing in order to enable:

- Providing the library: embedding our code in another code in order to enable better controlling of the programming process
- Providing the web service: enabling REST API (interface)

5.1 Library

Variation detector library

Variation detection is performed using anomaly-detection *Java 8* library, packed as a *.jar* file which can be found on Nissatech public *Maven* repository. Having the anomaly detection logic encapsulated in a library allows to use it in different contexts, in the cloud (e.g. wrapped in a web service), or locally, on the edge (e.g. Java application, Android application).

To use the anomaly detection library you need to add two things in the *.pom* file of your *Maven* project:

1) Specification of maven dependency

```
<dependency>
  <groupId>com.nissatech.d2lablite</groupId>
  <artifactId>anomaly-detection</artifactId>
  <version>1.0.0</version>
</dependency>
```

2) Specification of maven repository

```
<repositories>
    <repository>
        <id>archiva.public</id>
        <name>Nissatech Public Repository</name>
        <url>http://maven.nissatech.com/repository/public/</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
```

```
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

To be able to perform anomaly detection a model is needed. Instances on which the detection is performed are compared to the given model. It is up to the library client to acquire the model in some way (e.g. retrieve it from a remote web service).

Model has an id, time when the model was created, number of clusters, total number of instances and a list of centroids. Each centroid is again a list of double values. For each cluster number of instances, total intra-cluster distance, average intra-cluster distance and standard deviation of intra-cluster distances are specified. In addition, model contains average value and standard deviation of each standardized parameter, minimal, maximal, average value and standard deviation of each (non-standardized) parameter and same statistics for each of the clusters.

```
public class Model {
    private int pk; // unique id of a model
    private Date generationTime; // time when the model was generated
    private int clusterNum; // number of clusters in the model
    private int instanceNum; // total number of instances when performing clustering
    private List<List<Double>> centroids; // centroids for each of the clusters
    private List<Integer> instInClusterNum; // number of instances in each cluster
    private List<Double> totalDistances; // total distance from each instance to its
closest centroid
    private List<Double> avgDistances; // average distance from each instance to its
closest centroid
    private List<Double> deviations; // std dev of distance from each instance to its
closest centroid
    private List<Double> standardMeans; // mean of each (standardized) parameter
    private List<Double> standardDevs; // standard deviation of each (standardized)
parameter
    private List<Double> paramsMin; // minimal value of each parameter
    private List<Double> paramsMax; // maximal value of each parameter
    private List<Double> paramsMeans; // average value of each parameter
    private List<Double> paramsDevs; // standard deviation of each parameter
    private List<List<Double>> clusterParamMin; // minimal value of each parameter in each
cluster
    private List<List<Double>> clusterParamMax; // maximal value of each parameter in each
cluster
    private List<List<Double>> clusterParamMean; // average value of each parameter in
each cluster
```

```
        private List<List<Double>> clusterParamStd; // std deviation of each parameter in each
cluster
    }
```

It should be noted that this information is given informative and the user doesn't need to know the meaning of every parameter of the model if the already generated model is acquired somehow (e.g. from a web service periodically).

Library user should create an Instance out of data for which he wants anomaly detection to be performed. Each instance has a unique id, allowing to identify data for which the anomaly detection was performed in a unique way. Addition date can be specified, explaining when the instance was created. Instance consists of Map of parameters, where each parameter has a name (the key of the map) and a value, which is a list of Datapoints, where each datapoint represents a measurement with a timestamp and a value.

```
    public class Instance {
        private String uuid; // unique id of an instance
        private Date dateAdded; // timestamp when the instance was added
        private Map<String, List<Datapoint>> parameters; // list of instance parameters
        private Map<String, String> metadata; // metadata information about the instance
        private boolean isAnomaly; // flag indicating whether instance is considered to be
anomalous
    }

    public class Datapoint {
        private Date timestamp; // measurement timestamp
        private double value; // measurement value
    }
```

Anomaly detector receives a Model and an Instance, compares them and returns a Detection result.

Detection result contains id of the instance for which the anomaly detection was performed, status indicating whether the instance is an anomaly or not, time when the detection was performed, time when the model used for the detection was generated, id of the closest centroid and distance to the closest centroid (if the instance is not an anomaly).

```
    public class DetectionResult {
```

```
        private String instanceId; // unique id of an instance
        private boolean isAnomaly; // flag indicating whether instance is considered to be
anomalous
        private Date detectionTime; // time when detection was performed
        private Date modelGenerationTime; // time when the model was generated
        private int closestCentroidIndex; // index of the closest centroid from the model
        private double distanceToClosestCentroid; // distance to the closest centroid from the
model
    }
```

Here is an example of how the Anomaly Detector can be used:

```
    import detection.AnomalyDetector;
    import detection.ClusteringBasedDetector;
    import detection.DetectionException;
    import domain.DetectionResult;
    import domain.Instance;
    import domain.Model;

    public class Main {

        public static void main(String[] args) throws DetectionException {

            // Instantiate anomaly detector
            AnomalyDetector detector = new ClusteringBasedDetector();

            // Retrieve model and instance somehow, for example:
            ModelProvider modelProvider = new ModelProvider();
            InstanceProvider instanceProvider = new InstanceProvider();
            String instanceId = "br-11072009";

            Model latestModel = modelProvider.getLatestModel();
            Instance instance = instanceProvider.getInstanceWithUuid(instanceId);

            // Perform anomaly detection
            DetectionResult detectionResult = detector.detect(latestModel,
instance);
```

```
        // Use result of detection somehow...
    }
}
```

Notice that ModelProvider and InstanceProvider are not actually imported, but represent an example of retrieving the model and the instance somehow. It is up to the user to acquire the model and the instance in some manner.

5.2 REST API

5.2.1 Description

Variation Detection as a Service

This document explains the APIs supporting necessary phases for successful service usage:

1. **Uploading training files and training**
2. **Uploading files for detection**
3. **Real-time detection**

Training is complex, iterative operation and it's execution time can be in some cases long, but on the other hand it's not needed to be done frequently because model changes are minor when most of the data is same.

1. Uploading files for training and training

Training data represents a set of files that is uploaded in JSON format using HTTP POST request described below. One file is uploaded per request. A subset of training data is used for training based on time window specified in active training configuration. Only the data that belongs to that time window is used. Format of the file will be presented in the last section.

Request description for training file upload:

POST /projects/{projectID}/files/training

Parameters:

Name	Located in	Description	Required
projectID	path	We provide this after project is created	YES

file	formData	JSON formatted file	YES

Responses

Code	Description	Schema

202	File accepted for processing	**trainingFileCallback**{
		id: long
		Callback URI to the newly uploaded training file
		uri: string
		}

404	Project with given ID not found	

Response contains URI for checking file status and file ID number. Status can be *QUEUED, STORED, PROCESSING* and *ERROR*.

QUEUED *– file is enqueued and waiting to be processed*

PROCESSING *– file is being processed according to project setup*

STORED *– file is stored and ready to be used*

ERROR *– file is rejected due to corrupted parameter values*

2. Uploading files for detection

Detection data represents the data that is uploaded with the purpose of detecting abnormal behavior. Detection is performed based on the model that was generated during training phase. Detection data is uploaded in a similar way as training data and has **the same format**.

Request description for detection file upload:

POST /projects/{projectID}/files/detection

Parameters:

Name	Located in	Description	Required

projectID	path	We provide this after project is created	YES
file	formData	JSON formatted file	YES

Responses

Code	Description	Schema

202	File accepted for processing	**trainingFileCallback**{
		id: long
		Callback URI to the newly uploaded training file
		uri: string
		}

404	Project with given ID not found	

Response contains URI for checking file status. Refer to previous section for possible file statuses explanations.

3. Real-time anomaly detection

After training has been done, detection configured and detection files uploaded - have status STORED, these files can be used for real time detection. Detection is done by calling HTTP POST request described below:

POST /projects/{projectID}/files/detection/{fileID}/start

Parameters

Name	Located in	Description	Required
projectID	path	We provide this after project is created	YES
fileID	path	ID of file for detection, provided as upload response	YES

Responses

Code	Description	Schema

200	Detection successful	**detectionResult**{
		id : long
		isAnomaly : boolean
		closestMedoidId : string
		distanceToClosestMedoid : double
		reason : string
		}

400	Active detection configuration missing	string

404	File does not exist	**errorResponse**{
		errors: [string]
		}

404	File not stored yet	**errorResponse**{
		errors: [string]
		}

404	Project not found	**errorResponse**{
		errors: [string]
		}

In case of successful detection, indication whether this file is anomaly can be found in boolean **isAnomaly** parameter of the JSON response. Other detection result details are also part of response message.

Detection response contains only system generated file ID. In order to see the file metadata, specific HTTP GET request for file information retrieval must be issued:

GET /projects/{projectID}/files/detection/{fileID}

Parameters

Name	Located in	Description	Required
projectID	path	We provide this after project is created	YES
fileID	path	ID of file for detection, provided in detection response	YES

Responses

Code	Description	Schema

200	Detection successful	**MadFile**{
		id: long
		location: string
		who: string
		description: string
		status: string
		madFileType: string
		statusAttribute: string
		timestamp: long
		hBaseKey: string
		}

404	File not found	

4. Training and detection data format

Our system supports two types of data records. One is **time series** and the other is **contextual-behavioral.** This document covers time series data format.

Time series data format

As we mentioned earlier, training and detection data files have the same JSON format. This file is separated in two parts: **header** and **parameters**.

Header is JSON object which contains following key/value fields which are file metadata:

- ***timestamp***
- ***location***
- ***who***
- ***description*** (This parameter can be used arbitrarily. For example, you can set it as your internal test file ID (note that we generate new file ID for our system) to be able to identify test file after detection results are ready)

Parameters is array of objects. These objects contain parameter name and array of values. Time series file example can be found on the next page.

File example


```
{
  "header":{
            "timestamp": "1490565600000",
            "location": "Italy",
            "who": "Federico B.",
            "description": "105-6-367666"
          }
  "parameters": [{
            "name": "Cos Fi",
            "values": [638.0, 630.0, 627.0, 635.0, 635.0, 631.0, 631.0, 632.0, 631.0, 632.0]
            },
            {
            "name": "Temp. Sinistra (Nera)",
            "values": [631.0, 631.0, 629.0, 627.0, 627.0, 628.0, 631.0, 628.0, 630.0, 632.0]
            },
            {
            "name": "Temp. Destra (Rossa)"
            "values": [485.0, 485.0, 486.0, 486.0, 487.0, 487.0, 487.0, 488.0, 488.0, 489.0]
            },
            {
            "name": "T EV Cap. Fr."
            "values": [271.0, 272.0, 271.0, 272.0, 271.0, 272.0, 271.0, 272.0, 271.0, 272.0]
            },
            {
            "name": "T EV Cap. Fz."
            "values": [270.0, 270.0, 270.0, 270.0, 270.0, 270.0, 270.0, 270.0, 270.0, 270.0]
            },
            {
            "name": "Potenza"
            "values": [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0]
```

```
            }
        }]
    }
```

5.2.2 Service explanation

This document provides explanations about exposed services.

1. Training files upload

One of the ways to upload data for training is by using simple Apache HttpClient library. Details about this library can be found here: https://hc.apache.org/httpcomponents-client-ga/index.html. Here, we provide the example of function for sending the upload request using this library, more precisely using multipart request. For processing JSON format we use widely known Jackson library.

```
public D2UploadResponse sendRequest(Measurement payload) {

    CloseableHttpClient httpClient = HttpClients.createDefault();
    //we will provide you the URL for upload when we setup D2Lab for your use case
    HttpPost httpPost = new HttpPost(urlProvider.getUploadUrl());
    //you can set arbitrary file name, but note that it has .json extension
    String filename = payload.getHeader().getDescription().split("-")[1] + ".json";
    try {
        //we build file entity by serializing an object of class Measurement,
        //Note that file part name must be "file" while setting content type is optional
        HttpEntity entity =
               MultipartEntityBuilder.create().addPart("file", new ByteArrayBody(
       new ObjectMapper().writeValueAsString(Json.toJson(payload)).getBytes(),
filename)).build();

        httpPost.setEntity(entity);
        CloseableHttpResponse response = httpClient.execute(httpPost);
        //We get string serialized response
        String result = EntityUtils.toString(response.getEntity());
        //checking response status to see whether file was accepted by the service
        if (response.getStatusLine().getStatusCode() != 202) {
            //error occurred, details are in result variable so we can log that
            logger.error("Error uploading file to D2Lab. Details: " + response);
            return null; // or you can simply throw exception
        }
       D2UploadResponse uploadResponse =
                       Json.fromJson(objectMapper.readTree(result), D2UploadResponse.class);

    } catch (IOException e) {
        logger.error("Error in communication with D2Lab.", e);
        return null; // or you can simply throw exception
    } finally {
        try {
            //http client must be closed
            httpClient.close();
        } catch (IOException e) {
            logger.error("Error while closing http client");
        }
    }
```

```
}
```

You may notice that in previous function there are two classes that are not from HttpClient library but created for modeling file format (Measurement) and upload response (D2UploadResponse). Please note that you don't have to use Measurement class for sending the request, but we find it useful. If you have properly formatted file on disk, you can put it as file part of the multipart request. Implementation of these two classes is given here for the completeness of the document.

Measurement class implementation

```
public class Measurement {

    private Header header; //implementation of Header class follows
    private List<Parameter> parameters; //implementation of Parameter class follows
    public Header getHeader() {
        return header;
    }
    public void setHeader(Header header) {
        this.header = header;
    }
    public List<Parameter> getParameters() {
        return parameters;
    }
    public void setParameters(List<Parameter> parameters) {
        if(this.parameters == null)
            this.parameters = new ArrayList<>();
        this.parameters.addAll(parameters);
    }
}

//Header is used for specific metadata

public class Header {

    private String timestamp;
    private String location;
    private String who;
    private String description;
    public String getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
    public String getLocation() {
        return location;
    }
    public void setLocation(String location) {
        this.location = location;
    }
    public String getWho() {
        return who;
    }
    public void setWho(String who) {
        this.who = who;
    }
    public String getDescription() {
```

```
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}

//Parameter is used for the actual data

public class Parameter {

    private String name;
    private List<Double> values;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<Double> getValues() {
        return values;
    }
    public void setValues(List<Double> values) {
        if (this.values == null)
            this.values = new ArrayList<>();
        this.values.addAll(values);
    }
}
```

D2UploadResponse class implementation

```
public class D2UploadResponse {

    protected long id;
    protected String uri;

    public D2UploadResponse(@JsonProperty("id") long id, @JsonProperty("uri") String uri)
    {
        this.id = id;
        this.uri = uri;
    }
    public long getId() {
        return id;
    }
    public String getUrl() {
        return uri;
    }

}
```

2. Detection files upload

Files for detection are uploaded in the same way as training files. The only difference is another URL that we provide you to send this kind of request.

3. Starting real-time detection

After a file for detection is uploaded, you have to request anomaly detection result by sending HTTP POST request. If the requested file is not ready for the detection(note that it has to be validated, and this may take time depending on the load), the response status code will not be 200(OK). If this happens you can back-off for some period and eventually retry. Example of sending the detection request follows.

```
private D2DetectionResult detection(D2UploadResponse uploadResponse) {


    //Note that file id from uploadResponse as part of detection API URL is neede
    HttpPost httpPost =
                new
HttpPost(urlProvider.getDetectionUrl(String.valueOf(uploadResponse.getId())));
    CloseableHttpClient httpClient = clientProvider.getHttpClient();
    try {
        HttpResponse response = httpClient.execute(httpPost);
        if (response.getStatusLine().getStatusCode() == 200)
            return Json.fromJson(objectMapper.readTree(
                    EntityUtils.toString(response.getEntity())), D2DetectionResult.class);
    } catch (IOException e) {
        logger.error("Error while sending detection request to D2Lab.", e);
    } finally {
        try {
            httpClient.close();
        } catch (IOException e) {
            logger.error("Error while closing http client", e);
        }
    }
    return null; // or you can throw an exception
}
```

D2DetectionResult class implementation

```
public class D2DetectionResult {
    private Long id;
    private boolean isAnomaly;
    private String closestMedoidId;
    private double distanceToClosestMedoid;
    private String reason;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public boolean isAnomaly() {
        return isAnomaly;
    }
    public void setAnomaly(boolean anomaly) {
        isAnomaly = anomaly;
    }
    public String getClosestMedoidId() {
        return closestMedoidId;
    }
    public void setClosestMedoidId(String closestMedoidId) {
        this.closestMedoidId = closestMedoidId;
    }
    public double getDistanceToClosestMedoid() {
        return distanceToClosestMedoid;
```

```
}
public void setDistanceToClosestMedoid(double distanceToClosestMedoid) {
    this.distanceToClosestMedoid = distanceToClosestMedoid;
}
public String getReason() {
    return reason;
}
public void setReason(String reason) {
    this.reason = reason; }}
```

6. Conclusion

This document reports about the software development based on the user requirements defined in D1.1.

This deliverable elaborates on two additional conceptual aspects:

- The concrete scenario from the quality control where Qualytics will be applied
- The concrete TagsIt technology which will be applied in the selected scenario

The main part of this work is related to the technical/software development of the Qualytics system (which outcome is software code).

Qualytics extends our data analytics approach for the quality control in the home appliances domain. The focus is on the refrigerators, although the approach can be applied on an arbitrary scenario from this domain.

Moreover, we have designed the solution in the way that enables that the type of the sensors used for monitoring can be changed, by assuming that the data format (structure) will be not changed.

We will continue to work on the improving and extending the system