COALA

Cognitive Assisted agile manufacturing
for a LAbor force supported
by trustworthy Artificial Intelligence

Grant:	957296
Call:	H2020-ICT-2020-1
Topic:	ICT-38-2020
Type:	RIA
Duration:	01.10.2020 – 30.09.2023

Deliverable 2.6

Prescriptive quality analytics service – version 2

Lead Beneficiary:	ICCS
Type of Deliverable:	Other
Dissemination Level:	Public
Submission Date:	31.03.2022
Version:	1.0

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 957296.

Versioning and contribution history

Version	Description	Contributions
0.1	Table of Contents	ICCS
0.2	Outline	ICCS
0.5	First draft for internal review	ICCS
0.9	Addressing the internal reviewers’ comments	ICCS
1.0	Final version	ICCS

Reviewers

Name	Organisation
Mina Foosherian	BIBA

Disclaimer

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
AMA	Augmented Manufacturing Analytics
API	Application Programming Interface
AutoML	Automated Machine Learning
AutoRL	Automated Reinforcement Learning
DIA	Digital Intelligent Assistant
DRL	Deep Reinforcement Learning
DT	Decision Tree
EDA	Exploratory Data Analysis
IRL	Interactive Reinforcement Learning
kNN	k-Nearest-Neighbors
ML	Machine Learning
MORL	Multi-Objective Reinforcement Learning
NAS	Neural Architecture Search
NN	Neural Network
RL	Reinforcement Learning
UI	User Interface

Executive Summary

The Deliverable D2.6 “Prescriptive quality analytics service – version 2” extends the work performed in the Deliverable D2.3 in the context of Task 2.1 “Prescriptive analytics for process and product quality” within the time period between M10-M18. Task 2.1 develops the prescriptive quality analytics component, which is called PREscriptiVE aNalyTIcs for quality optimizatiON (PREVENTION) and performs advanced data analytics on the basis of quality-related data in order to detect and prioritize quality problems as well as to support decision making ahead of time. Among others, the component is able to predict future quality issues, and prescribe mitigating actions that improve the quality of products and processes. The prescriptive quality analytics service is able to support several quality related scenarios employing state-of-the-art Machine Learning algorithms and Automated Learning techniques and allow other services to use its functions through an Application Programming Interface (API).

This Deliverable reports the development and implementation of the second version of the PREVENTION component, focusing on the supported algorithms and the technical implementation, that realize the high-level view of the steps that constitute the augmented data analytics lifecycle and extend the first version of the prescriptive quality analytics component, presented in the Deliverable D2.3.

Specifically, we describe the Machine Learning algorithms that are incorporated into the current implementation of the PREVENTION component and can address classification, regression, time series forecasting and recommendation quality-related problems along with the implemented Automated Learning techniques that facilitate the machine learning models selection and lead to optimized models with minimum human effort. Additionally, in this Deliverable we document the current implementation by elaborating its technical architecture and describing the implemented interfaces, enabling the interaction with the component, and the advanced filtering capabilities that can provide the analytics results in a dynamic manner according to each case’s needs.

Finally, we present some preliminary results of quality analytics using PREVENTION in the context of the Whirlpool use case, with the goal to demonstrate the application of the implemented software services on several analytics scenarios utilizing the quality data derived from the Whirlpool data sources, as well as present the interfacing of the PREVENTION component that can expose the analytics outcomes to the user either with visualization graphs through the COALA Dashboard or by voice through the DIA.

In the upcoming period, we will focus on extending and adapting the analytics capabilities of PREVENTION and further test the component’s efficiency as soon as relevant data become available by the use case partners. Based on these tests, the current implementation will be extended to support additional, advanced analytics methods and machine learning algorithms and its application to additional sophisticated scenarios will be examined.

1 Introduction

1.1 Purpose and Objectives

The Deliverable D2.6 “Prescriptive quality analytics service – version 2” extends the work performed in the Deliverable D2.3 in the context of Task 2.1 “Prescriptive analytics for process and product quality” within the time period between M10-M18. Task 2.1 develops the prescriptive quality analytics component, which is called PREscriptiVE aNalyTIcs for quality optimizatiON (PREVENTION) and performs advanced data analytics on the basis of quality-related data in order to detect and prioritize quality problems as well as to support decision making ahead of time. Among others, the component is able to predict future quality issues, and to prescribe mitigating actions that improve quality of products and processes. The prescriptive quality analytics service allows other services to use its functions through an Application Programming Interface (API).

1.2 Approach

The Deliverable describes the second version of the prescriptive quality analytics service of COALA. Based on the business requirements related to the PREVENTION component and the work performed by the Deliverable D2.3, the system and technical specifications have been finalized, and the PREVENTION component has been implemented. Therefore, the focus of the current Deliverable is on the implementation of the PREVENTION architecture, the algorithms providing the descriptive, predictive, and prescriptive analytics capabilities and the interfaces enabling the interaction and communication with the COALA components.

1.3 Relation to other WPs and Tasks

Task 2.1 “Prescriptive analytics for process and product quality” is based on the broad requirements and needs defined in WP1 “Requirements and manufacturing use cases”, in order to address the use cases’ specific scenarios and challenges. The task is also informed by WP4 “Continuous integration and software quality” so that it becomes compliant with the overall COALA architecture design, system specification, and defined interfaces.

The outcomes of PREVENTION are exposed to the user in two ways: (i) through the COALA Dashboard (to be developed in the context of WP4), or (ii) through the DIA (Task 2.2 “Digital assistant for manufacturing demonstrator” and “Task 2.3: Features for augmented manufacturing analytics”).

Since the trained data analytics models will be communicated to the Why Engine, this Task is also related to Task 2.4 “WHY engine” for explainable digital assistant responses”. Finally, the Task is related to Task 5.2 “Evaluation of the use case implementations”, in the context of which the PREVENTION component will be evaluated as part of the COALA solution.

1.4 Structure of Deliverable

The Deliverable is structured as follows:

Section 2 extends the components definition presented in the Deliverable D2.3 “Prescriptive quality analytics service – version 1”, providing a high-level description of the component and the algorithms supported by the second version of the PREVENTION component.

Section 3 documents the technical implementation of the component focusing on the PREVENTION architecture and interfaces.

Section 4 demonstrates the preliminary results of the prescriptive quality analytics service application for the Exploratory Data Analysis, the Risk Assessment, the Predictive Quality and the Checklist Recommendation scenarios, as well as the interfacing mechanism implemented for communicating its results, in the context of the Whirlpool use case.

Section 5 presents the conclusions and the outlook for the future work.

2 Component Description

The PREVENTION (PREscriptiVE aNalyTIcs for quality optimizatiON) component addresses the prescriptive quality analytics needs of COALA. PREVENTION aims at building predictive quality models and prescribing mitigating actions to optimize quality-related manufacturing performance indicators. To achieve its aims, PREVENTION utilises 3 types of analytics: descriptive, predictive, and prescriptive analytics. PREVENTION aims at covering the whole lifecycle of data analytics in order to derive insights on quality, while enabling the development of several independent analytics process with different scopes (descriptive, predictive, prescriptive) from multiple data sources and algorithms.

2.1 Overview

This section provides a high-level description of the PREVENTION component focusing on its architecture and the interaction with other components of COALA. Based on the COALA architecture, PREVENTION employs quality-related data that are acquired and managed by the Data Collection and Management Layer of COALA, aiming at combining the power of data with the power of knowledge for generating appropriate recommendations exposed though the COALA Mobile or Dashboard. In addition, the analytics models are communicated to the WHY Engine for providing explanations.

2.1.1 Architecture

Figure 1 depicts the PREVENTION architecture, where the core layer consists of two sub-systems, the Analytics Designer, in which the data analyst designs and configures the analysis, and the Analytics Runtime Engine, which receives the designed analysis and carries out its building and deployment.

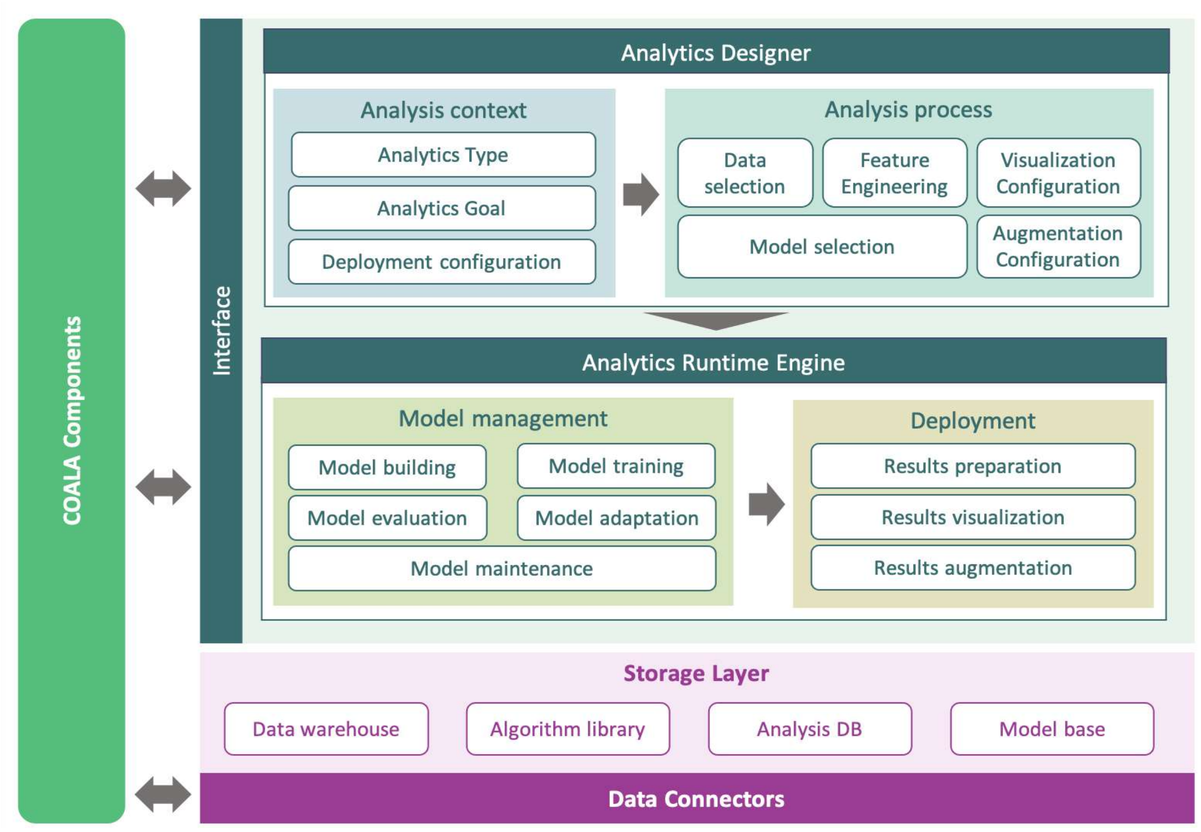


Figure 1: PREVENTION Conceptual Architecture

The Analytics Designer sub-system consists of two modules that enable the definition of the Analysis Context and the configuration of the Analysis Process.

The Analytics Context consists of three parameters: the **Analytics type** that can be either descriptive, predictive, or prescriptive, the **Analytics Goal** which is the analytics problem that the analysis solves and the **Deployment configuration** that represents the process of executing and making available the analysis results to the end users and the other COALA components.

The Analysis process is responsible for helping the data analyst in defining the required parameters and data for each step of the analytics process that will be executed by the Analytics Runtime Engine. The analysis process has five steps that are configured by the data analyst: **Data selection**, **Feature Engineering**, **Model selection**, **Visualization Configuration** and **Augmentation Configuration**.

The second sub-system of the PREVENTION component is the Analytics Runtime Engine. The Analytics Runtime Engine is responsible for the data processing, building and deploying the analysis and consists of two modules: Model Management and Deployment.

The first module, **Model Management**, deals with the iterative process of managing the analytics model lifecycle. The model management process is considered as the sequence of functionalities: Model Building, Model Training, Model Evaluation, Model Adaptation and Model Maintenance that may be executed iteratively multiple times, before and after many deployments of the analysis, individually or as a whole. The second module, **Deployment**, is responsible for deploying and executing the designed analysis to the selected time and according to the selected workflow defined by the corresponding Deployment configuration. It consists of the Results preparation, the Results visualization, and the Results augmentation.

The PREVENTION component that supports various types of advanced analytics processes indicates the need to support various data types, store and organize the input as well as the generated data in a sophisticated way. To support the data storage needs, the Storage layer of the architecture consists of four distinct but interrelated storage modules: Data warehouse, Algorithm library, Analysis DB and Model Base.

The **Data Warehouse** holds the processed input data for the designed analyses, that are collected and pre-processed from diverse data sources and are available by other COALA components. The **Algorithm Library** stores the algorithmic prototypes/templates that implement different algorithms and can be reused for defining multiple analyses. The **Analysis DB** holds all the parameters and configuration details for the designed analyses along with the generated outcomes. Finally, the **Model Base** is the main storage used by the Model Management module to store, retrieve and maintain the trained analytics models that have been trained or adapted during the execution of the designed analyses, in order to make them easily and immediately available.

2.1.2 Communication Interfaces

The PREVENTION implements advanced interfaces to communicate with other COALA components. Firstly, a communication between PREVENTION and the Data Collection and Management Layers is considered, to provide all the necessary data and information. Further the results produced are exposed to the end user while interacting with the Digital Assistant and the COALA Dashboard, and the trained models and Algorithms can be exposed to the WHY Engine.

2.2 Algorithms

PREVENTION's main goal is to address different advanced quality analytics scenarios of various levels of difficulty and value, while employing several algorithms. The component's architecture enables the development of the corresponding analyses, based on different Machine Learning (ML) algorithms, utilizing the available quality datasets. Following the defined architecture, the PREVENTION component is easily extendable, in terms of data and algorithms. New algorithms can be incorporated into the implementation to address additional analytics scenarios.

This section provides a brief overview of the algorithms that can satisfy the analytics needs of the predictive quality scenarios and are currently integrated into PREVENTION. The component provides algorithms that support two of the main ML scenarios, supervised learning and Reinforcement Learning (RL), and can support or provide an answer to exploratory data analysis, classification, time series forecasting and recommendations problems, representing different descriptive, predictive and prescriptive quality analytics cases. For this, it employs not only traditional ML algorithms but also automated learning methods that can help in building suitable models that solve these problems efficiently.

2.2.1 Exploratory Data Analysis

For the Exploratory Data Analysis (EDA), PREVENTION implements descriptive statistics to fulfil the needs of COALA for valuable information extracted from the raw available data. The EDA leverages several statistical methods to analyse the raw data, perform an investigation on the available data and extract important characteristics and hidden information, while restructuring the data in a form that can be consumed by the COALA components and is more comprehensible to the end user.

The supported EDA functionalities consist of several data processing actions that can filter, group and aggregate the selected features from the data and restructure them in a desired manner. Furthermore, PREVENTION enables the feature engineering (i.e. feature extraction, feature reduction and feature generation) functionalities included in the data pre-processing step of the lifecycle, that can result in new datasets, containing valuable information, and may also be provided as input to advanced analytics methods.

Additionally, the EDA is a case-specific process, as it depends on the data available for analysis and for this reason PREVENTION implements the EDA in a way that can be customizable and adjustable to provide suitable results. The outcomes of the EDA are provided in a format suitable for visualization in terms of graphs, charts, reports and dashboards, while they can also be consumed by the Digital Intelligent Assistant (DIA) to answer the user's questions. The EDA results generated by the PREVENTION component can be further enhanced using the filters available by the Analytics Results Query Engine.

2.2.2 Classification

One aspect of the predictive analytics produced by the PREVENTION consists of Classification predictions made by supervised ML algorithms. The application of supervised ML algorithms can be divided into a two-step procedure (Kotsiantis, et al., 2006). The first step consists of the data preparation and pre-processing actions that need to be carried out on the raw data available, in order to provide the algorithms with data that are the most informative and maximize their performance. The preparation and pre-processing actions are executed based on the available data resolving issues as: data cleaning, missing values, feature selection and feature extraction. The second step is the selection of the algorithm to implement. The selection is based on the algorithm's performance, which can be measured with different techniques and depends on the available data. ML algorithms try to learn patterns and correlations between the data and their classes from past historical data; they later use them to predict the class or category of future given data. Even though many supervised ML algorithms seek to make the same predictions, the difference between them lies on how the problem is formulated. In the context of the PREVENTION component, classification problems are supported mainly by the implementation of Decision Trees (DT) and k-Nearest-Neighbors (kNN) classifiers.

The DT classifier approaches the prediction problem by breaking up complex decisions into several simpler ones by dividing the data into subgroups (Safavian, et al., 1991), forming a decision tree. Each node in a decision tree represents a decision rule based on which the feature space of the training set is partitioned. The algorithm recursively constructs nodes with decision rules until each partition contains data from only one class. In that case the node is referred to as leaf or terminal node. With the above in mind while constructing a decision tree, we seek to find the optimal selection of decision rules for the nodes, determine the terminal nodes and assign a class label to each terminal node (Safavian, et al., 1991). One of the biggest advantages of the decision trees is their ability to perform well with a large dataset without the need of extensive data preprocessing, as they can handle unnormalized data and missing values. Despite their advantages, decision trees tend to overfit on the training data and be computational expensive if they don't get constructed optimally. For these reasons various methods and techniques are used to ensure optimal structures for decisions trees based on the available data.

Another approach, which is one of the simpler classification methods yet an effective one in many cases, is the kNN classifier. For the simple kNN method the given training samples are mapped to a N-dimensional space, where N is the number of features of each sample. To determine the class of a given sample **t** the **k** nearest neighbors are selected and **t** is assigned to the class that has the majority among the neighbors. In other words, the classifier seeks to find the most similar samples from the training data and classify the new sample into the most represented class (Hand, et al., 2007). Besides the quality of the data, the accuracy of the classification is directly impacted by the parameter **k** and the metric with which the distances are calculated. Regarding the parameter **k** neither a small or a big number must be selected as it might reduce the classifier's performance. Selecting a small number for **k** will make the classifier unstable with high variance, as it will be data sensitive and will not get enough neighbors for proper classification. The simpler strategy for selecting the optimal number for the parameter **k** is to run the algorithm many times with different values and chose the value with the best performance (Guo, et al., 2003). On the other hand, selecting a big number for **k** will increase the bias and selected neighbors may not be necessarily close to the sample we seek to classify. Moreover, considering the metric for calculating the distance between the samples many metrics can be used, and the selection of that metric is heavily affected by the given data (Chomboon, et al., 2015). Although many different metrics are available, the Euclidean metric is typically used (Hand, et al., 2007). Despite its simplicity, the algorithm of the kNN can be computationally expensive especially when the volume of the training data is big. The reasons behind this is that the classifier needs to store the entire training set and each time a classification is made all the distance computations must be executed. Because of the above drawbacks, many variations of the simple kNN method have been developed with techniques to alleviate them (Guo, et al., 2003).

2.2.3 Time Series Forecasting

In addition to the classification algorithms, PREVENTION supports the development of regression models and time series forecasting to make predictions about upcoming events. Regression refers to the machine learning models that rely on the regression analysis to make predictions. Regression analysis is a technique that is used for extracting main features and hidden relationships between them from a set of available data. This technique tries to approximate the true relationship functions between the features by using simple mathematical functions (e.g., polynomial) containing the appropriate variables. With a good approximation function, the models can learn about the underlying relationships of the data and make valuable predictions for variables, given some others (Draper, et al., 1998).

There are different categories of regression-based methods e.g., linear regression, logistic regression and polynomial regression. The main difference between these categories is related to the functions that are used for representing the relationship between the variables. PREVENTION component mainly utilizes linear regression methods to address the regression problems. The key property of linear regression is that it uses a relatively simple linear function to represent the relationships, that is linear for both the features of the data and the function's parameters (Bishop, et al., 2006). This function will make an accurate prediction about the feature or features we seek to predict, given the known features of a sample and good approximation for the function's variables.

Apart from ML models, other time series forecasting methods can also be used for predicting future events. Even though most known statistical methods such as ARIMA models use regression for forecasting, their approach is different, with the main difference being that only previous values of the feature are considered. As a result, those features form a sequence of historical measurements based on which the prediction, or in this case the forecast is made. In addition, instead of learning the relationships between features, in this case, we aim to decompose the variation of the time series into components due to seasonal variation, trend, cyclic variations and irregular fluctuations. Thus, the time series forecasting methods work well when the available data have a regular linear trend or seasonality and quite poorly if they don't (Chatfield, et al., 2000; Bontempi, et al., 2012).

2.2.4 Recommendations

PREVENTION implements prescriptive analytics approaches where the main goal is the utilization of the predictive analytics outcomes, the appropriate datasets and the machine learning algorithms for data-driven decision-making and recommendations. The implemented prescriptive quality analytics approach is able to: (i) recommend (prescribe) the appropriate actions, that can mitigate an undesired predicted future event; and (ii) model the decision making process under uncertainty instead of the physical quality process, thus making it applicable to various industries and processes. Following the conceptual architecture described in Section 2.1.1, this approach consists of three steps: prescriptive model building, prescriptive model solving, and prescriptive model adapting and is mainly based on RL algorithms.

RL is the third machine learning paradigm, alongside supervised learning and unsupervised learning (Sutton, et al., 2018). RL can be considered as a tool for discovering optimal policies in manufacturing problems (e.g., Dornheim, et al., 2019, Rocchetta, et al., 2019). RL formulates the problem as an environment consisting of states and actions and learning agents that have a defined goal state. The agents transition across different states by implementing the appropriate actions, each of which yields a different reward. The agents aim at reaching the goal state while maximizing the rewards by selecting actions and moving to different states. Agents' learning is not supervised or guided; on the contrary, agents learn the actions they can perform in order to reach the desired state through trial-and-error (Sutton, et al., 2018).

In RL, a model is a representation of the available environment knowledge, that in most cases is realized by the tuple $< S, A, T, R, \gamma >$ of the environment's Markov Decision Process, in which S represents a set of states and A represents an action set. T is the transition probability function $T : S \times A \times S \rightarrow [0,1]$, R is the reward function $R : S \times A \times S \rightarrow R$, both of which are usually unknown, and $\gamma \in [0,1]$ is the discount factor, that determines the current value of rewards received in the future. The agent's learned behaviour is represented by a policy $\pi : S \times A$, where $\pi(s, a) = P(a_t = a | s_t = s)$ is the probability of selecting a possible action $a \in A$ in a state s. The main RL interactions between the environment and the agent are depicted in Figure 2.

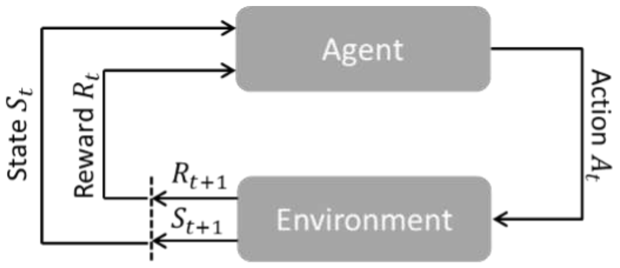


Figure 2: Reinforcement Learning

Various taxonomies of the RL methods have been proposed in literature. One of the most fundamental taxonomies available classifies the RL methods to model-based and model-free methods (Zhang, & Yu, 2020). The model-based category refers to the methods that the transition function *T* and the reward function *R* are learnt by the agent through an iterative process, in order to complete the available environment knowledge. Having the complete model representation available, allows the agent to follow the model and solve the respective problem, normally by applying planning algorithms.

On the other hand, model-free methods do not rely on strict modelling of the environment, but rather let the agent learn the optimal policy by interacting with the environment, searching for the policy that yields the maximum reward. Research on model-free methods has been highly active, further classifying the methods of the category into two types, the policy-based methods and the value-based methods, according to the proposed optimization strategy. Specifically, the goal of the policy-based methods is to optimize the policy learnt by the agent, while the value-based methods the agent utilizes a value-state function $V^{\pi}(s)$ or a value-action function $Q^{\pi}(s, a)$ instead, in order to optimize the corresponding policy. Figure 3 depicts the RL methods classification proposed by (Zhang, & Yu, 2020), that summarizes the most popular method categories of the literature. PREVENTION component implements several RL methods, mainly focusing on model-free methods, where the model is unknown or partially available.

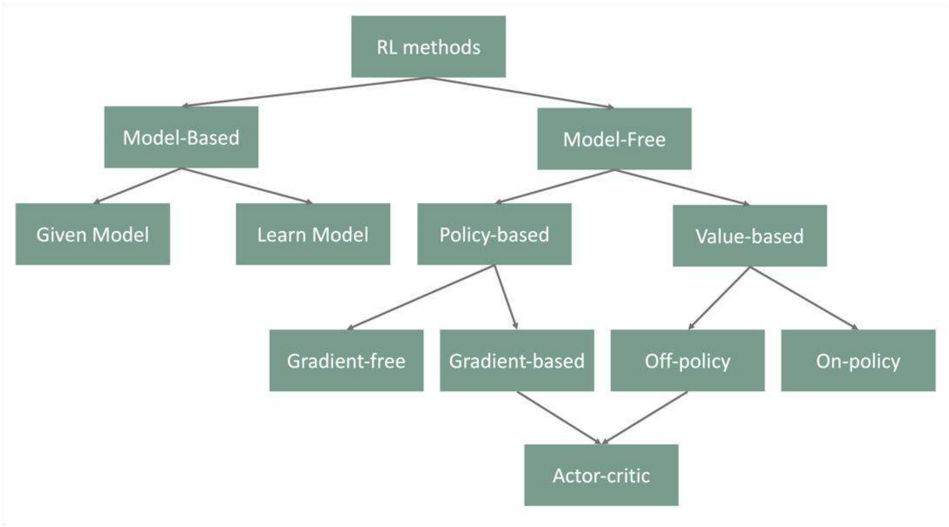


Figure 3: RL methods taxonomy

The incorporated RL algorithms can be further extended to solve recommendation problems where the generated recommendation must be the optimal with respect to several objectives and the expert's feedback can be utilized to guide the recommendation mechanism, by supporting Interactive Reinforcement Learning (IRL) and Multi-objective Reinforcement

Learning (MORL) approaches. IRL adds the capability of incorporating evaluative feedback provided by a human observer so that the RL agent learns from both human feedback and environmental reward (Li, et al., 2019). Additionally, MORL algorithms can address sequential decision-making problems with multiple objectives. MORL requires a learning agent to obtain action policies that can optimize multiple objectives at the same time (Chunming, et al., 2015). Furthermore, the implementation of Deep Reinforcement Learning (DRL) algorithms and Automated Reinforcement Learning (AutoRL) methods can be examined for supporting more sophisticated prescriptive quality analytics scenarios.

2.2.5 Automated Learning

Acquiring a good performing model for model-based methods and algorithms can be challenging as the model's performance is heavily impacted by the available data used, their preprocessing and the parameters used for the algorithms. As a solution to the problem of finding a good performing machine learning model, PREVENTION takes advantage of the Automated Machine Learning (AutoML) process, which is the intersection of automation and machine learning. As a definition, AutoML attempts to take the place of human intervention while constructing and configuring machine learning computer programs within limited computational budgets (Yao, et al., 2018). The main goals of AutoML are good performance of the proposed machine learning model, little human intervention and high computational efficiency. These three main goals can be further divided into more sub-goals that various efforts and techniques have been used to meet them.

The traditional process of obtaining a good machine learning model for a specific problem was mainly defined in terms of trial and error. The human expert would try to set a configuration using personal experience or intuition and then get feedback on its performance. Based on the feedback, adjustments to the configuration would be made getting new performance feedback and so on. The process usually ends once a desired performance is achieved, or the computational budgets runs out (Yao, et al., 2018). AutoML eliminates the above trial-and-error process as it tries to find the best performing ML model. The ML model and algorithms examined from the AutoML perspective in literature can be divided into two categories: the traditional models (SVM, DT, KNN) and Neural Networks (NN). In the case of NN a Neural Architecture Search (NAS) is executed to find the best performing NN, using different techniques and algorithms.

The AutoML pipeline, as shown in Figure 4, is divided into 4 main processes: data preparation, feature engineering, model generation, and model evaluation (He, et al., 2021). Each process can be independently configured based on the desired outcome and the available resources.

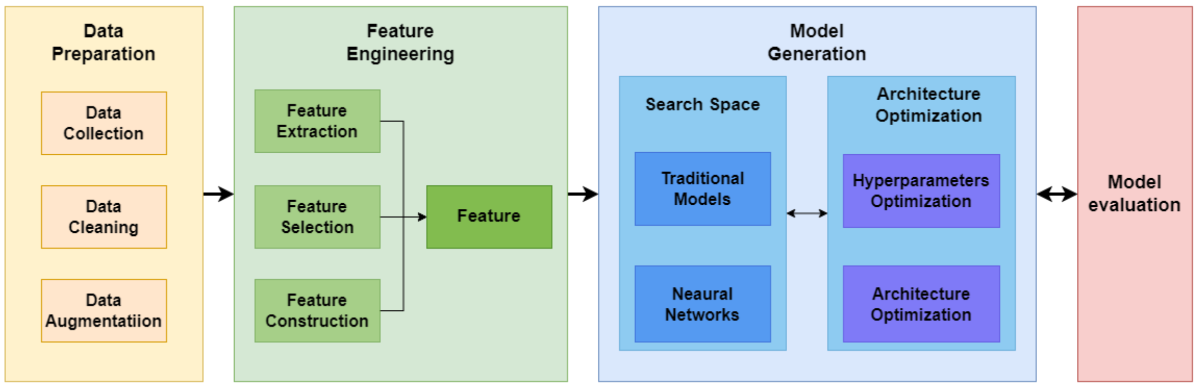


Figure 4: Automated Learning pipeline

The first step in the AutoML process is the preparation of the available data that will be used by the algorithms. The Data preparation is crucial and plays a major role in the overall performance as all of the following steps depend on the data used. This step helps to get the most value out of the available data by optimizing them based on the problem's needs. In this process, the aim is to obtain high quality data from credible sources (Data Collection), clean noisy data (Data Cleaning) and produce new data based on existing data (Data Augmentation). The next process in AutoML is Feature Engineering, with the purpose of extracting the maximum number of valuable features from the raw data. To achieve this, irrelevant or redundant features get dropped, new features are produced based on existing features and the dimensionality of the data is reduced. After preparing the data and extracting the necessary features, the next step in the process is the generation of ML models.

During the Model Generation phase based on some techniques and methods several ML models are constructed and optimized. Thus, Model Generation can be further divided into two steps: the Search Space and the Architecture Optimization. The search space defines the model structures used for building the model and can be divided into two categories of models: Traditional ML models (SVM, KNN) and the Neural Network architectures. The Architecture optimization defines the techniques and methods used for determining the performance of the architecture in order to find the best performing parameters. The optimizations can be one of two categories: Hyperparameter optimization and Architecture optimization (AO). The last process in the pipeline, Model evaluation, evaluates the performance of the proposed model. The simpler action related to the Model evaluation is to train the model until convergence using all the available training data is reached and then proceed with the evaluation of its performance. The issue with this approach is that this training process may result in extensive computational complexity in both time and computing resources. To decrease the computational complexity many methods can be used like transferring knowledge and weights from other networks and stopping the training when the performance of the network is not increasing significantly. In the end, when the AutoML process is finished, the optimal model is proposed, which has the best performance among all other models used.

2.3 Relation to business requirements

The prescriptive quality analytics service addresses the business requirements shown in Table 1, that are derived from the Deliverable D1.1 "Joint COALA requirements" and are strongly based upon data analytics capabilities of the COALA solution.

Table 1: Related business requirements and their current level of fulfilment

Req. ID	Requirement description	Business Case	Fulfilment until M18 (%)	Note
REQ-0320	COALA shall be able to run a root cause analysis based on historical quality data	WHR	100%	
REQ-0520	COALA shall be able to update a SKU risk profile data	WHR	100%	
REQ-0500	COALA shall be able to perform a risk assessment	WHR	100%	

	on a SKU or family of product based on the historical quality data			
REQ-0510	COALA shall be able to perform a risk assessment on a specific procedure based on the historical quality data	WHR	80%	Will be finalized in the context of integration activities and WHR deployment
REQ-0530	COALA shall identify procedure evolution suggestions based on the historical quality data, the past suggestions and the measured deviations	WHR	80%	Will be finalized in the context of integration activities and WHR deployment
REQ-0290	COALA shall be able to receive a structured hypothesis from the user to be tested	WHR	90%	
REQ-0310	COALA shall be able to perform a deeper root cause analysis to provide a second level explanation if requested by the worker (e.g. best practice identified using analytics on historical data)	WHR	100%	
REQ-0320	COALA shall be able to run a root cause analysis based on historical quality data	WHR	100%	
REQ-0330	COALA shall be able to provide the results of its analysis and presents all potential causes for the tested hypothesis	WHR	90%	

3 Technical Specifications

Prescriptive quality analytics is addressed by the PREVENTION (PREscriptiVE aNalyTIcs for quality optimizatiON) component. PREVENTION aims at building predictive quality models and prescribing mitigating actions to optimize quality-related manufacturing performance indicators. In this sense, it consists of 3 types of analytics: descriptive, predictive, and prescriptive analytics. PREVENTION aims at covering the whole lifecycle of data analytics in order to derive insights on quality.

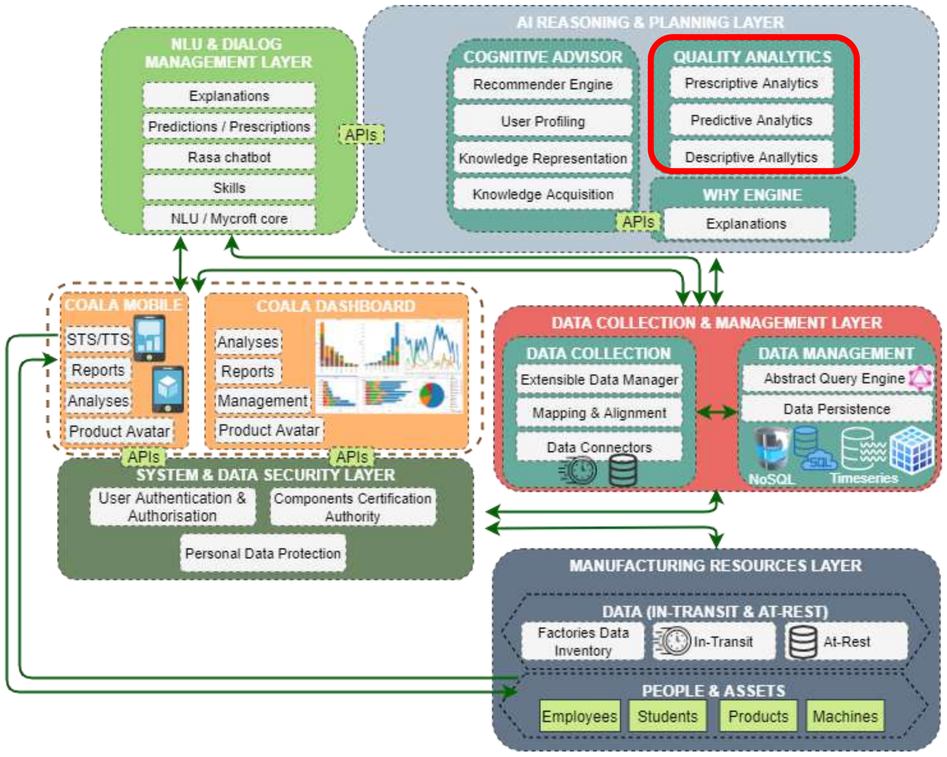


Figure 5: PREVENTION in the context of the COALA Architecture

3.1 Infrastructure

The implementation of PREVENTION is based on Python (version 3.6.8)[1], which supports multiple programming paradigms, including structured (particularly, procedural), object-oriented, and functional programming, and has an extensive collection of libraries for data analytics. In addition, to instrument the functionalities of the analytics component into a web-based service, the implementation is based on the Flask micro web framework (version 2.0.2)[2]. Flask does not require the usage of tools or libraries and supports extensions that can add features on demand.

[1] https://www.python.org/downloads/release/python-368/
[2] https://pypi.org/project/Flask/

The PREVENTION component is containerized with Docker (version 20.10.6)[3] and docker-compose (version 2)[4] to include the required services that facilitate the component deployment. To further decouple the development deployment from the production deployment we have considered two Docker deployment configurations. In order to enable the production deployment of the Flask application, the production deployment configuration relies on the Gunicorn "Green Unicorn",[5] a Python Web Server Gateway Interface (WSGI) HTTP server, that provides a simple, lightweight, and fast implementation supporting multiple workers. The Docker compose configuration consists of two services, the prevention component service, and a mongoDB service, both of which are configurable (i.e. database credentials and connection details) using environmental variables. The PREVENTION component can be deployed with the Docker compose command:

```
docker-compose -f ./docker/prod-env.yml --env-file ./.env up
```

Upon successfully deploying the services, the prescriptive quality analytics service will be accessible through its GraphQL interface at:

http://localhost:8081/graphql

The component’s source code is available at:

https://gitlab.ubitech.eu/coala/components/prevention.git

3.2 Technical Architecture

Figure 6 depicts the technical architecture corresponding to the implementation of the Prescriptive Quality Analytics Service. In this figure the conceptual architecture modules defined in the Deliverable D2.3 “Prescriptive quality analytics service – version 1”, are presented from a technical perspective. The correlation between the technical modules of the technical architecture with the conceptual modules is depicted following the same colour coding. The core of the implementation consists of two main components, the Analytics Manager and the Analytics Runtime Engine.

[3] https://docs.docker.com/engine/release-notes/
[4] https://docs.docker.com/compose/compose-file/compose-file-v2/
[5] https://gunicorn.org/

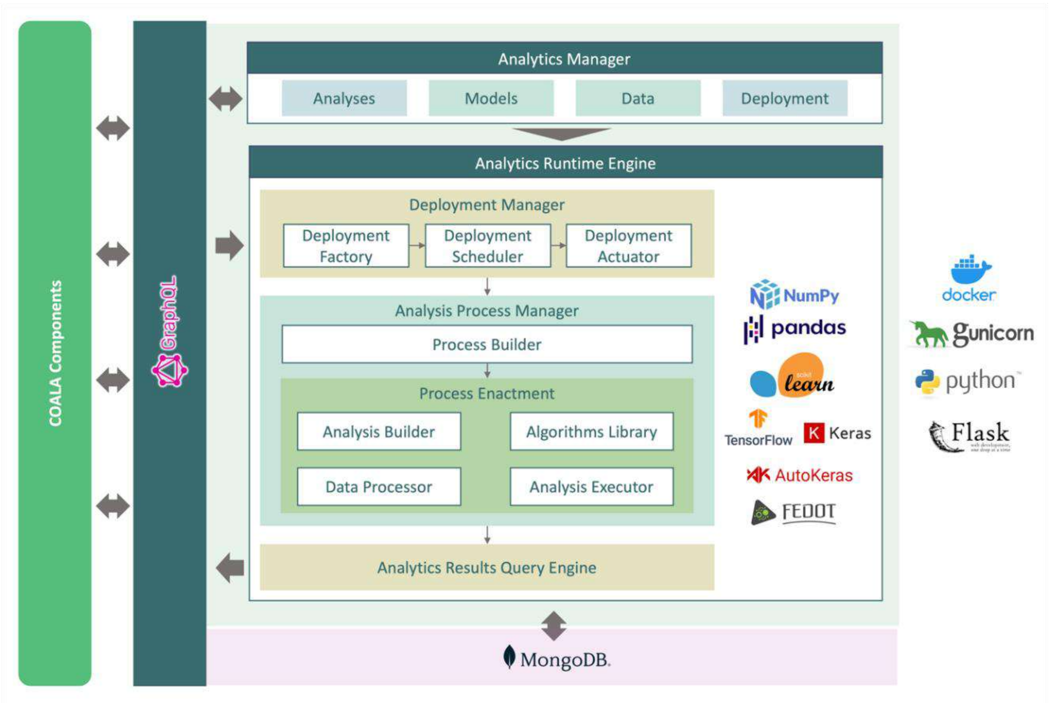


Figure 6: Technical Architecture

The *Analytics Manager* component enables the configuration and management of the available entities, including the defined analyses, models, the datasets, and the analyses deployments. Through the Analytics Manager component, the user can define a new analysis, add new models, provide new datasets and configure the analysis deployment, as well as manage and monitor all the existing analyses.

The *Analytics Runtime Engine* component consists of three sub-components, the Deployment Manager, the Analysis Process Manager and the Analytics Results Query Engine.

The *Deployment Manager* sub-component is responsible for creating, scheduling, and managing the analyses deployments according to the deployment configuration provided by the user. The Deployment Manager sub-component is implemented by three modules, Deployment Factory, Deployment Scheduler and Deployment Actuator. The *Deployment Factory* module constructs the deployment object according to the corresponding analyses deployment configuration provided by the user through the Analytics Manager. *Deployment Scheduler* is responsible for monitoring the running deployments and scheduling new deployments according to the deployment configurations. When a new deployment is scheduled, the Deployment Scheduler passes the deployment configuration and object to the Deployment Actuator that is responsible for triggering the new analysis deployment. The Deployment Actuator triggers the Analysis Process Manager.

The *Analysis Process Manager* sub-component has been implemented by two modules, Process Builder and Process Enactment. The Deployment Actuator triggers the *Process Builder* module that is responsible for creating the analysis process indicated by the analysis context configuration and specifically the defined analytics type, analytics goal, and the selected models. The process definition is provided to the Process Enactment module that is responsible for implementing the steps that constitute the analysis process. The Process enactment module consists of the Analysis Builder, the algorithms Library, the Data Processor and the Analysis Executor modules.

The Analysis Builder is responsible for implementing the first step of most of the defined analysis processes. The main functionality of the Analysis Builder module is initializing the required objects and bootstrapping all the resources, required before running the analysis. Among others, this step includes any data pre-processing and model training required for the specific analysis. When a new analysis is built, it becomes available to the *Analysis Executor*, that is responsible for running the defined analysis according to the deployment configuration.

Data processing is required during several steps of the analysis lifecycle, in different steps for different analyses. The *Data Processor* module is responsible for implementing all the data processing steps across the analysis lifecycle. For the data preparation, manipulation and processing this component utilizes related Python libraries, such as: pandas (version 1.1.5)[6] for data manipulation; numpy (version 1.19.5)[7] for the definition of large, multi-dimensional arrays and matrices and high-level mathematical functions to operate on these arrays.

The *Algorithms Library* holds the implementations of the available models and algorithms that realize the descriptive, predictive and prescriptive methods described in Section 2.2. The algorithms used by the PREVENTION have been implemented as Python classes. Additionally, each class is followed by “helper” functions to manage the created models from the algorithms. The helper functions are responsible for the data pre-processing necessary to create the input for the algorithms, the creation of the models, the predictions and the evaluation of the models.

The Analysis Builder uses the corresponding algorithms based on the Analysis definition. First, the Analysis Builder initializes the appropriate class needed to execute the desirable algorithm. Secondly, after the initialization and according to the model’s type and definition, the initialized model is trained using the Analysis dataset. When training is completed, the model is evaluated and stored. When an Analysis is triggered, the corresponding class is instantiated from storage and using the appropriate helper functions the (trained) model is called. For the predictive algorithms, in order to make a prediction, some input data are expected to be given from the user through the graphQL interface. The algorithms implementation manages efficiently the created and trained models along with their parameters, in order to have available timely trained models, without the need to retrain over the initial datasets. As a result, the PREVENTION component can make predictions in real time.

The predictive classification algorithms have been implemented with the scikit-learn[8] (or sklearn) python library. Among others, the scikit-learn library provides algorithms for classification, regression and clustering such as support-vector machines, decision trees, k-nearest neighbours implementations. The current implementation supports the DT classifiers, that belong to the supervised learning methods with the goal to predict a value of a target variable or feature. In order to make a prediction the algorithm learns simple decision rules from the data features. These classifiers have been implemented with an additional subclass (MultiColumnLabelEncoder) which supports the pre-processing of the data. The subclass is responsible for encoding the given data using labels. This class also implements the encoding of the data at the training stage and stores the encoders in order to be available for decoding the predicted values generated by the classifiers.

The current implementation focuses on the use of automated learning for the regression and time series forecasting methods. These methods have been implemented as AutoML

[6] https://pandas.pydata.org/pandas-docs/version/1.1.5/user_guide/index.html
[7] https://numpy.org/devdocs/release/1.19.5-notes.html
[8] https://scikit-learn.org/stable/

algorithms that are able to find the best network architecture that fits the available data and provides the most accurate solution for the problem at hand. For the AutoML algorithms implementation two libraries have been used, the AutoKeras (Jin, et al., 2019) and Fedot (Nikitin, et al., 2021). The AutoKeras library is an AutoML system based on the library Keras that provides the algorithms performing AutoML for deep learning models, mainly used for classification and regression. Given the input data, the AutoKeras library searches for the best network architecture and hyperparameters to train the regression models, by using a tree-structured acquisition function and the Bayesian optimization to generate several NNs. The Fedot library provides a framework for automated modelling and ML, with the goal to develop the complex composite models that allow obtaining the efficient solution of various applied problems. It uses a large variety of models to perform classification, regression and time series forecasting. Fedot creates composite models as a chain of actions, where each action can represent data pre-processing or a ML model. Given some input data, Fedot trains and evaluates several composite models while making changes to them and tuning their hyperparameters. After enough trials or when there is no significant improvement of the performance the best performing composite model is proposed. In the current implementation, Fedot has been used for selecting composite models that solve time series forecasting problems, where based on the available data the pipeline proposes a composite model for predicting the next n values of a time series.

The prescriptive analytics methods that can generate recommendations have been developed based on RL. The RL methods have been implemented using two RL libraries, the OpenAI[9] gym and Keras[10]. The Open AI gym formulates the recommendation problem as an RL environment, provides great flexibility when constructing the custom environment, a simple interface for the agent to interact and learn and the capability to solve the problem with multiple RL algorithms. The Agent that solves the recommendation problem has been implemented using Keras.

The *Analytics Results Query Engine* provides advanced filtering capabilities over the generated analysis results and is described in detail in Section 3.3.3.

The Data layer of the PREVENTION component is implemented using the MongoDB[11] as the data storage and the Mongoengine library (version 0.23.1)[12] and pymongo[13] (version 3.12.0) that enable the interaction of PREVENTION with the MongoDB service for accessing the input data as well as storing the analytics configuration data and the analytics results. The data layer of the component consists of two different data models, the Core Platform Data Model and the Quality Control Data Model. The Core Platform Data Model realizes the analytics functionalities of the architecture, providing the capability to dynamically build and deploy several different analyses. In this sense, it consists of six main entities (Dataset, Analysis, Deployment, Deployment Conf, Model, Analysis Results) and seven enumerations (Analytics Type, Analytics Goal, Deployment Mode, Model Type, Model Status, Results Type, Analytics Filters). The Quality Control Data Model aims at modelling the quality related information that is present in the available datasets and is critical for the successful implementation of the advanced

[9] https://openai.com
[10] https://keras.io
[11] https://www.mongodb.com/
[12] https://pypi.org/project/mongoengine/
[13] https://pypi.org/project/pymongo/

quality analytics cases addressed by the PREVENTION component. Figure 7 presents an overview of the data storage layer of the component.

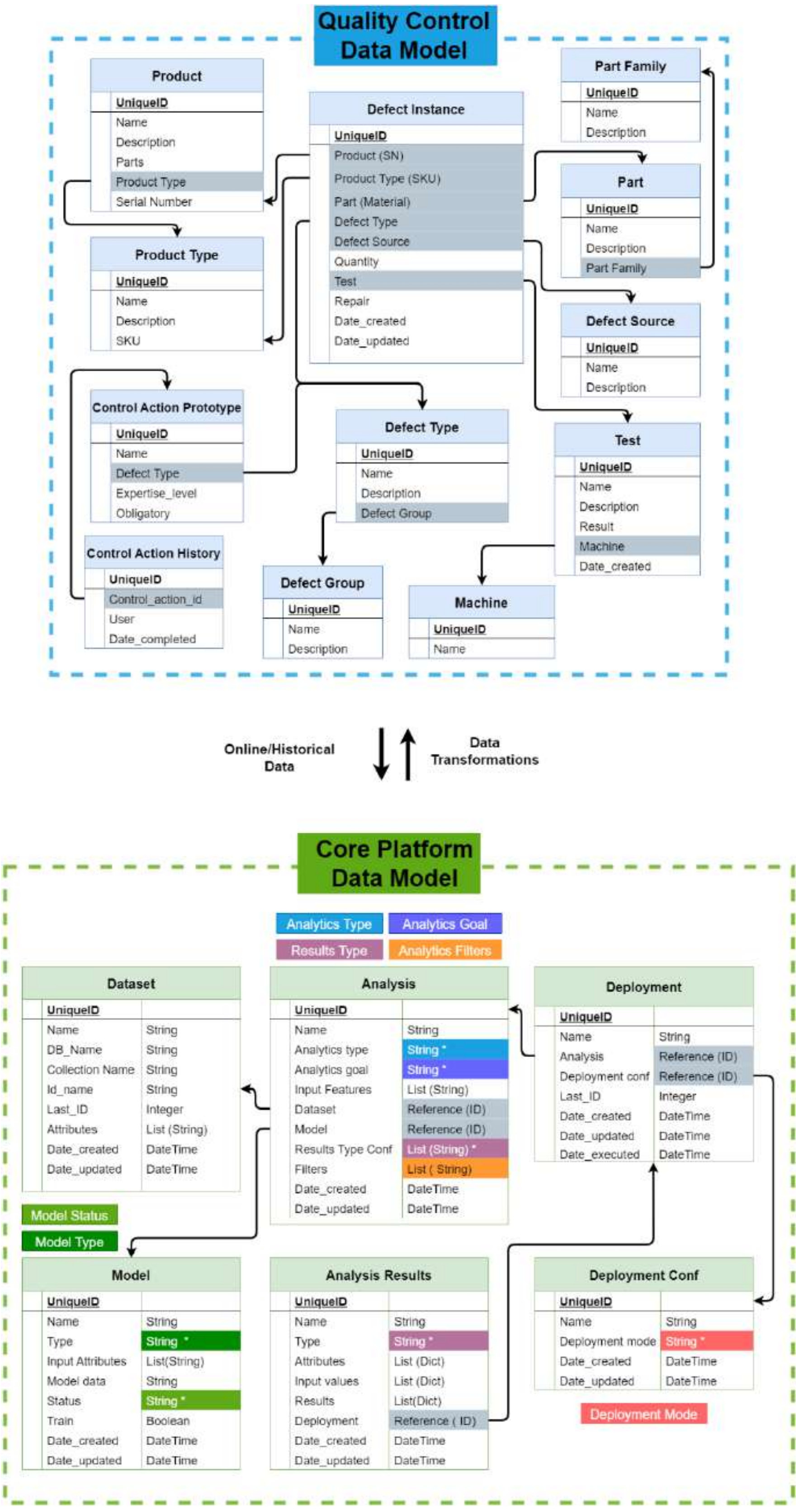


Figure 7: PREVENTION Data model

3.3 Application Interfaces

The main interface of the PREVENTION component is a GraphQL API that is responsible for communicating the analytics data and results to the COALA platform. GraphQL[14] is an open-source data query and manipulation language used for APIs. It provides a type system, query language and execution semantics, static validation, and type introspection, it enables reading, writing (mutating), and subscribing to changes to data and can support several languages, including Python. The API development has been implemented using the Graphene library (version 2.1.8)[15] along with the flask related library Flask-GraphQL (version 2.0.1)[16].

The requests exposed from the API are divided into two groups according to their functionalities:

- Analytics Management Queries: consists of CRUD queries, providing the ability to create, read, update and delete the various entities in the database
- Analytics Process Queries: responsible for realizing the requests of the steps of the analytics process described in detail in the Deliverable D2.3 "Prescriptive quality analytics service – version 1".

Even though by using the first group of requests the user can get the available raw Analysis Results data, using the second category of queries, lets the user query the Analysis Results filtered by the available filters of the Analytics Results Query Engine described in Section 3.3.3.

GraphQL has been widely used as an alternative to REST, as it provides more flexibility and customization on the constructed requests. In addition, it gives the ability to fetch exactly the data that the client requested from multiple data sources with only a single API call. With the GraphQL API the PREVENTION component is able to construct custom requests, with specific inputs fields needed for each request and provides the client the capability to retrieve specific fields of the query answer.

3.3.1 Analytics Management Queries

The Analytics Management Queries consist of CRUD queries, providing the ability to create, read, update and delete the various entities in the database. These queries can be further categorized into two groups, Mutations and Queries. Mutations are used when changes to the database are needed, specifically when creating, updating, and deleting, while the Queries are used for data retrieval.

3.3.1.1 Mutations

Mutations queries provide the capability to create and update the main entities of the PREVENTION component, i.e. the defined Analyses, Models, and Deployments. For the mutations, we will provide a total of three examples, two create mutations and one update mutation.

The first mutation example shown in Table 2 creates a new Analysis entity. As described in detail in the Deliverable D2.3 "Prescriptive quality analytics service – version 1", an analysis is

[14] https://graphql.org/
[15] https://pypi.org/project/graphene/
[16] https://pypi.org/project/Flask-GraphQL/

defined by an analyticsType to indicate whether it refers to a descriptive, predictive or prescriptive analysis and an analyticsGoal that refers to the specific problem that the analysis solves. These data define the analytics process that will be followed for the specific analysis. Additionally, an analysis is correlated with a given dataset, and a model that along with the analytics process steps (i.e. pre-processing, training, validating) implements the analysis goal. The input fields of the analysisData may not be mandatory when running this mutation; however, they contain critical information for the building and running the Analysis at later steps of the process. In the analysis attribute of the query, the user can specify which fields of the analysis will be returned in the answer. In this example, we request only the analysis id and name to be included in the answer.

Table 2: Example of creating a new analysis

Query	Answer
`mutation mCreateAnalysis{`	`{`
`  createAnalysis(analysisData:{`	`  "data": {`
`    name: "Analysis42"`	`    "createAnalysis": {`
`    analyticsType: "DESCRIPTIVE"`	`      "analysis": {`

`    analyticsGoal: "GROUP_DEFECTS_BY_SKU"`	`        "id":`
	`"62028d49a1b8d0b13f512bc4",`

`    inputFeatures: ["Input1","Input2"]`	`        "name": "Analysis42"`
`    dataset: "617fd12fdf1f3aa59e1b4130"`	`      }`
`    model: "617fd41819024bf13eca51a1"`	`    }`
`    resultsTypeConf: ["DA"]`	`  }`
`    dateCreated: "2021-10-27T10:48:11.023000"`	`}`
`    dateUpdated: "2021-10-27T10:48:11.023000"`	
`  })`	
`  {`	
`    analysis{`	
`      id`	
`      name`	
`    }`	
`  }`	
`}`	

Similarly, Table 3 presents an example of the mutation creating a new Model entity by providing again the relative fields. Among others, a model is defined by a model type, i.e. the type of algorithm that represents the model, a status indicating the current state of the model (whether it is created, trained, etc.), a train flag indicating whether the specific model requires training and the inputAttributes name list that defines the features of the dataset that are employed by the specific model. Again, for the answer we only specify three fields to be returned.

Table 3: Example of creating a new model

Query	Answer
`mutation mCreateModel{`	`{`
`  createModel(modelData:{`	`  "data": {`
`    name: "Model42"`	`    "createModel": {`
`    type: "GROUPBY"`	`      "model": {`

`    modelData: "None"`	`        "id":`
	`"62028c93a1b8d0b13f512bc3",`

`    status: "CREATED"`	`        "name": "Model42",`
`    train: false`	`        "type": "GROUPBY"`
`    inputAttributes: ["Input1","Input2"]`	

dateCreated: "2021-10-27T10:48:11.023000"	}
dateUpdated: "2021-10-27T10:48:11.023000"	}
})	}
{	}
model{	
id	
name	
type	
}}}	

The next example in Table 4 represents an update mutation executed on a pre-existing entity of an Analysis. In the case of the update mutation, the id field needs to be filled with the id of the entity we seek to update, while the rest of the fields can be filled with new values. In the example, the provided id is the returned id received in the previous example when the analysis was created, while the fields name, inputFeatures, dataset and model have been updated. The answer is requested to include only 3 of those fields. The answer depicted in this table shows that the analysis update was successful.

Table 4: Example of updating an existing analysis

Query	Answer
mutation mUpdateAnalysis{	{
updateAnalysis(analysisData:{	"data": {
id: "62028d49a1b8d0b13f512bc4"	"updateAnalysis": {
name: "Analysis4_New"	"analysis": {
analyticsType: "DESCRIPTIVE"	"id": "62028d49a1b8d0b13f512bc4",
analyticsGoal: "GROUP_DEFECTS_BY_SKU"	
inputFeatures: ["Input3","Input4"]	"name": "Analysis4_New",
dataset: "62028b0da1b8d0b13f512b6d"	"inputFeatures": [
model: "62028b0fa1b8d0b13f512b74"	"Input3",
resultsTypeConf: ["DA"]	"Input4"
dateCreated: "2021-10-27T10:48:11.023000"	]
dateUpdated: "2021-10-27T10:48:11.023000"	}
}){	}
analysis{	}
id	}
name	
inputFeatures	
}	
}	
}	

3.3.1.2 Queries

The requests of the Queries category can retrieve information from the database. These queries have been implemented by two types of queries for each entity type of the database, one for requesting all the available entities in the database and one for requesting a specific entity by providing its unique identifier.

In Table 5, we provide an example of a query that retrieves all the Analysis entities stored in the database. The answer includes all the Analysis in a list, where each record contains all the fields requested by the query. Due to the length of the specific answer, in this example we provide only the first Analysis of the list included in the answer.

Table 5: Retrieve all defined analyses

Query

```
query qAllAnalysis{
  allAnalysis{
              id
    name
    analyticsType
    analyticsGoal
    inputFeatures
    preprocessActions
    dataset{
      id
      name
    }
    model{
      id
      name
    }
    resultsTypeConf
  }
}
```

Answer

```
{
  "data": {
    "allAnalysis": [
      {
        "id":
"62028b0da1b8d0b13f512b70",
        "name": "Group defects by SKU",
        "analyticsType": "DESCRIPTIVE",
        "analyticsGoal":
"GROUP_DEFECTS_BY_SKU",
        "inputFeatures": [
          "Id",
          "InsertDate",
          "FGNum",
          "DefectID",
          "FGDes"
        ],
        "preprocessActions": [
          "{\"in_attr\":  \"InsertDate\",
\"out_attr\":              \"Date\",
\"preprocess_action\":
\"DATETIME_TO_DATE\"}"
        ],
        "dataset": {
          "id":
"62028b0da1b8d0b13f512b6d",
          "name": "FOR"
        },
        "model": {
          "id":
"62028b0da1b8d0b13f512b6f",
          "name": "Group By SKU"
        },
        "resultsTypeConf": [
          "U",
          "I"
        ]
      },
      …………
```

Table 6, demonstrates a query that can retrieve a specific entity of the Analysis model by providing the unique identifier of the Analysis as input. The query returns a record containing all the requested fields from the query.

Table 6: Example of retrieving a specific analysis

Query

```
query qAnalysis{
  analysis(analysisId:
"62028b0fa1b8d0b13f512b75"){
    id
    name
    analyticsType
    analyticsGoal
    inputFeatures
    preprocessActions
    dataset{
      id
      name
    }
    model{
      id
      name
    }
    resultsTypeConf
  }
}
```

Answer

```
{
  "data": {
    "analysis": {
      "id": "62028b0fa1b8d0b13f512b75",
      "name":  "Group  defects  by  Family
ID",
      "analyticsType": "DESCRIPTIVE",
      "analyticsGoal":
"GROUP_DEFECTS_BY_FAMILY_ID",
      "inputFeatures": [
        "Id",
        "InsertDate",
        "FamilyID",
        "FamilyDes"
      ],
      "preprocessActions": [
        "{\"in_attr\":    \"InsertDate\",
\"out_attr\":                  \"Date\",
\"preprocess_action\":
\"DATETIME_TO_DATE\"}"
      ],
      "dataset": {
        "id":
"62028b0da1b8d0b13f512b6d",
        "name": "FOR"
      },
      "model": {
        "id":
"62028b0fa1b8d0b13f512b74",
        "name": "Group By FamilyId"
      },
      "resultsTypeConf": [
        "U",
        "I"
      ]
 }}}
```

Using the answer from the example query of the Table 6, we can use the unique identifiers of the other entities of the database in order to retrieve additional information about them. Based on the previous example we could use the dataset and model unique identifiers to retrieve additional information related to the dataset and the algorithm used for the specific analysis.

In the example presented in Table 7, the model's unique identifier retrieved from an analysis request is used for accessing information about the model related to the analysis. The query is constructed with a similar way as the previous one, using the model request and providing the model's unique identifier, and the fields that are requested to be included in the answer. The answer contains all the requested fields that provide the information about the model. Note that in the case of Descriptive Analysis the modelData field will be empty and the train field set to False, while in the rest of the Analysis the modelData will include information about the algorithms used and the MODEL attribute of the modelData field will contain the algorithms implementation in binary form. Additionally, the inputAttributes field contains all the features from the raw data used by the model during training.

Table 7: Example of retrieving model details

Query	Answer
query qModel{	{
model(modelId: "62179759cc3c51210dc7eddc"){	"data": {
id	"model": {
name	"id": "62179759cc3c51210dc7eddc",
type	"name": "Predict Defect Group",
modelData	"type": "DT",
status	"modelData":"{'PREDICT_COLUMN'
train	['DefectGrp'], 'MODEL': […],
inputAttributes	"status": "TRAINED",
dateCreated	"train": true,
dateUpdated	"inputAttributes": [
}	"LinSeq",
}	"FGNum",
	"DefectSrcID",
	"StatID",
	"FamilyID"
	],
	"dateCreated": "2022-02-24T16:34:01",
	"dateUpdated": "2022-02-24T16:34:01"
	}
	}
	}

3.3.2 Analytics Process Queries

The Analytics Process Queries consist of three different requests, where each one has a distinct functionality, the resultReqest query, the triggerRequest query, and the buildAnalysis query.

Retrieve Results

With the resultRequest the user can request Analysis Results existing to the database from an already executed Analysis. In this query, the user can use the filters provided by the Analytics Results Query Engine in order to filter the raw results data and extract the appropriate information needed. The Analytics Results Query Engine is explained in depth in the following section.

Table 8 shows an example of a resultRequest query implemented by the component. The request can be split into the input of the request with the respective input fields and the requested format of the answer of the request. Within the body of the resultRequest request the user must provide the values for the necessary fields for the request, in this case the fields “request” and “requestFrom”, and if needed add optional fields like the filters “requestFilters”, “dateFilters”, “sumFilters”. The request field represents the goal of the analysis that the user expects to retrieve using this query, while the requestFrom field indicates the client submitting the request. Finally, the user can request which fields of the requests answer needs to receive in this case the “results” fields. The answer format is defined by the field “requestFrom”. This field has been introduced to provide the ability to specify the format of the results that will be answered by the query. In the current implementation, the user can choose between the Digital Intelligent Assistant (“DA”) and User Interface (“UI”).

Table 8: Example of retrieving analytics results

Query

```
query q1{
resultRequest(request:{
request: "GROUP_DEFECTS_BY_DEFECT_ID"
requestFrom: "DA"
requestFilters: {
          name: "DefectID"
             action: "greater"
             values: "3005"
            }
dateFilters:{
             name: “Date”
             action: "between"
             start: "2020-04-23"
             end: "2021-04-23"
             unit: "YEAR"
           }
sumFilters: {
             name: "count"
             action: "topn"
             numberOfValues: "3"
           }
 }){
       results}}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "Year": 2020,
            "DefectID": 3035,
            "count": 1879
          },
          {
            "Year": 2021,
            "DefectID": 3030,
            "count": 932
          },
          {
            "Year": 2020,
            "DefectID": 3017,
            "count": 494
          }
        ]
      }
    ]
  }
}
```

Run Analysis

The next query presented in Table 9, the triggerRequest is used when an analysis results, (i.e. a prediction or a recommendation) not available to the database needs to be executed and generate the requested outcomes. For this request, the implemented Algorithms are used to execute the analysis and return its results to the user.

Depending on the analysis and the corresponding model, the triggerRequest query may need some input data from the user in order to execute the analysis. In the specific example, with the “`request`” field the user can specify the analysis goal of the desired analysis, representing the kind of the prediction that needs to be executed while with the “`defectGroupPrediction`” the user can fill the necessary input fields for the specific analysis. The answer returns the predicted value generated by running the model of the analysis, that has been trained over the corresponding dataset.

Table 9: Example of running an analysis

Query	Answer
`query q1{`	`{`
`        triggerRequest(request:{`	`  "data": {`
`        request: "PREDICT_DEFECT_GROUP"`	`    "triggerRequest": [`
`        defectGroupPrediction: {`	`      {`
`        lineSequence: "10620"`	`        "results": [`
`        sku: "858749701900"`	`          {`
`        defectSourceId: "6"`	`            "DefectGrp": 1201`
`        statId: "501"`	`          }`
`        familyId: "0419"`	`        ]`
`            } }){`	`      }`
`            results`	`    ]`
`          }}`	`  }`
	`}`

Build Analysis

The last query, buildAnalysis is used for bootstrapping an Analysis, including all the steps required prior to its execution, i.e. preprocessing the initial dataset to build the model's input features and training the model. By providing the unique identifier of the Analysis and the type of results that are requested to be generated in a suitable format, the user can run for the first time or rerun an existing Analysis from the database. The query automatically calls the Deployment Manager in order to initialize the new deployment, if needed, and execute the analysis. The query returns the generated Analysis Results. Table 10 shows an example of the buildAnalysis query.

Table 10: Example of building an analysis

Query	Answer
`query {buildAnalysis(`	`{`
	`  "data": {`
`analysisId:"61fd054628fc1945a0fcc04b",`	`    "buildAnalysis": {`
`  resultsType: "DA"){`	`      "result": {`
`    result`	`        "_id": {`
`  }}`	`          "$oid": "61fd527228fc1945a0fcc0a6"`
	`        },`
	`        "name": "Results",`
	`        "type": "Defects",`
	`        "attributes": [`
	`          {`
	`            "name": "Date",`
	`            "type": "string"`
	`          },`
	`          {`
	`            "name": "FGNum",`
	`            "type": "integer"`
	`          },`
	`          {`

```
            "name": "count",
            "type": "integer"
          }
        ],
        "input_values": null,
        "results": [
          {
            "Date": "2020-04-23",
            "FGNum": 858743901900,
            "count": 1
          },…],
        "deployment": {
          "$oid": "61fd527228fc1945a0fcc0a5"
        },
        "date_created": "2022-02-04-18:21:06",
        "date_updated": "2022-02-04-18:21:06"
      }
    }
  }
}
```

3.3.3 Analytics Results Query Engine

The GraphQL flexibility in the queries makes it possible to add optional fields to the available queries in order to implement an advanced filtering mechanism over the Analytics Results based on the users' inputs. These filters have been implemented as optional fields of the resultRequest query. The filtering mechanism of the Analysis Results has been implemented to support three distinct filters, with the purpose to further enhance the information provided by the Analysis Results. All three filters are available for all the defined Analysis entities. In addition, the filters are independent to each other.

In the following examples we assign colours to the fields of the queries and the answers to help the reader. The red color represents the request name, the blue color represents the required fields, the purple color represents the optional fields and the green the answer fields. In the generated answers, the attribute on which the filters are applied is marked with purple.

3.3.3.1 Data filtering

The first filter, requestFilters, provides the ability to filter the results based on the values of the results' features. The user provides as input the features and the corresponding values that are desired to be used as filters and selects a filtering action, relative to the type of the features. If the feature is of type string, the available actions that can be applied are the "contains" and "equals" actions, where the "contains" action filters the results in order to isolate the results subset containing the requested feature values and the "equals" action returns the results that contain the exact values given as input. Accordingly, when the feature is of numeric type, the actions that can be performed are logical operations (greater, less, equal etc).

Table 11 shows an example of the data filtering mechanism of the `resultRequest` query, where the results of the analysis with goal `"GROUP_DEFECTS_BY_DEFECT_ID"` are filtered to include only the defect id records with a DefectID value greater than 3005. The answer is expected to follow the format suitable for the requested client (i.e. the User Interface or the

Digital Intelligent Assistant), indicated by the `requestFrom` attribute. The data filtering field, `requestFilters`, is optional and consists of three required fields (`name`, `action`, `values`). Based on those fields the filter will be executed on the `results` attribute as instructed by the field `name`, and perform the appropriate `action` based on the `values` field. In the Answer we will receive results that their “`DefectID`” attribute has a value greater than 3005 which was the given value in the filter.

Table 11: Example of filtering the analysis results with respect to the available features

Query	Answer
`query q1{`	`{`
`  resultRequest(request:{`	`  "data": {`
`    request: "GROUP_DEFECTS_BY_DEFECT_ID"`	`    "resultRequest": [ {`
`    requestFrom: "DA"`	`        "results": [{`
`    requestFilters: {`	`            "Date": "2020-04-24",`
`      name: "DefectID"`	`            "DefectID": 3017,`
`      action: "greater"`	`            "count": 2`
`      values: "3005"`	`          }, {`
`     }`	`            "Date": "2020-04-24",`
`  }){`	`            "DefectID": 3035,`
`    results`	`            "count": 6`
`  }`	`          }, {`
`}`	`            "Date": "2020-04-27",`
	`            "DefectID": 3035,`
	`            "count": 2`
	`          },…`

3.3.3.2 *Date filtering*

The second filtering mechanism, dateFilters, performs filtering using the features representing date and time data of the Analysis Results. The dateFilters has a dual functionality, the first is filtering results based on the timestamps and the second is grouping the results according to the selected time units. To filter based on the time stamps the filter performs logical operations (equal, greater, less, between) using the input values, to provide the results within the proper time intervals. To group the results an optional attribute in the filter is expected, which indicated the time unit (day, week, year), based on which grouping will be applied.

Table 12 depicts an example of the `dateFilters` on the `resultRequest` query, that filters the results of the analysis with goal `"GROUP_DEFECTS_BY_DEFECT_ID"` using the Date feature representing the manifestation date of the recorded defect. Similarly, to the data filtering fields, in order to receive results within the given dates range grouped by the appropriate unit of time the user must provide the required fields of the filter `(action, start, end, unit)`. The filtered results returned in the answer of the query contain the number of defects grouped by the defected id and the month, occurred between “2020-04-23” and “2021-04-23”.

Table 12: Example of filtering results with respect to time

Query	Answer
`query q1{`	`{`
`  resultRequest(request:{`	`  "data": {`
`    request: "GROUP_DEFECTS_BY_DEFECT_ID"`	`    "resultRequest": [ {`
`    requestFrom: "DA"`	`        "results": [{`
`    dateFilters:{`	`           "Year": 2020,`
`      name: "Date"`	`            "Month": 4,`
`      action: "between"`	`            "DefectID": 1001,`
`      start: "2020-04-23"`	`            "count": 1`
`      end: "2021-04-23"`	`          },{`
`      unit: "MONTH"`	`            "Year": 2020,`
`    }`	`            "Month": 4,`
`}){`	`            "DefectID": 1007,`
`    results`	`            "count": 3`
`  }`	`          },{`
`}`	`            "Year": 2020,`
	`            "Month": 4,`
	`            "DefectID": 1019,`
	`            "count": 3`
	`          },…`

3.3.3.3 *Aggregation Filters*

The third and last filter, sumFilters, provides aggregation capabilities based on a numeric value of the results. The filter supports three types of aggregation that can filter the maximum or minimum result of the initial results data or return the top n results. The sumFilters implementation is considered as an optional field of `resultRequest` named `sumFilters`. The aggregation filter expects two main values to be provided as input by the user, the `name` field that represents the name of the feature that is requested to be aggregated and the `action` field that indicated the type of aggregation. The action field supports three values “min”, “max” and “topn” that implement the corresponding aggregation. When a topn aggregation is requested, the user is expected to provide a third value `numberOfValues` indicating the top n number of records that the user expects to receive.

Table 13 depicts an example query using the `sumFilters` mechanism over the `resultRequest` query. In this case, the results of the analysis with goal `"GROUP_DEFECTS_BY_DEFECT_ID"` are aggregated with the “max” action, over the “count” feature, that represents the number of defects occurred per DefectID. The query returns exactly one result, that represents the defectID with the maximum number of occurrences, across the dataset.

Table 13: Example of aggregating results

Query	Answer
`query q1{`	`{`
`  resultRequest(request:{`	`  "data": {`
`    request: "GROUP_DEFECTS_BY_DEFECT_ID"`	`    "resultRequest": [`
`    requestFrom: "DA"`	`      {`
`    sumFilters: {`	`        "results": [`

`      name: "count"`
`      action: "max"`
`    }`
`})\{`
`    results`
`  }`
`}`
	`          {`
	`            "Date": "2020-07-20",`
	`            "DefectID": "3035",`
	`            "count": "59"`
	`          }`
	`]}]}}`

The three types of filters are optional and can be used together or independently to each other. Table 14 presents an example query with all three filters in the request. This example shows one of the available requests provided by the PREVENTION component.

Table 14: Example of combined filters

Query	Answer

`query q1{`
`  resultRequest(request:{`
`    request: "GROUP_DEFECTS_BY_DEFECT_ID"`
`    requestFrom: "DA"`
`    requestFilters: {`
`      name: "DefectID"`
`      action: "greater"`
`      values: "300"`
`     }`
`    dateFilters:{`
`      name: "Date"`
`      action: "greater"`
`      start: "2020-04-23"`
`      unit: "YEAR"`
`    }`
`    sumFilters: {`
`      name: "count"`
`      action: "topn"`
`      numberOfValues: "3"`
`    }}){`
`    results`
`  }`
`}`
	`{`
	`  "data": {`
	`    "resultRequest": [`
	`      {`
	`        "results": [`
	`          {`
	`            "Year": 2020,`
	`            "DefectID": 3035,`
	`            "count": 1879`
	`          }, {`
	`            "Year": 2021,`
	`            "DefectID": 3030,`
	`            "count": 932`
	`          },{`
	`            "Year": 2020,`
	`            "DefectID": 1371,`
	`            "count": 912`
	`          }`
	`        ]`
	` }]}}`

4 Implementation Status

This section describes the PREVENTION component current implementation status, the next development steps and the implementation history.

4.1 Current Implementation

The current implementation realizes the core platform of the prescriptive quality analytics service, the algorithms described in Section 2.2, focusing on ML and AutoML algorithms, that are capable of addressing classification, regression, time series forecasting and recommendation quality related problems as well as the interfaces described in Section 3.3, that facilitate the interaction and communication with the DIA and the other COALA components. Additionally, the current version of the component implements the prescriptive quality analytics demonstration scenarios of the WHR business case that are described in detail in Section 5.

4.2 Next Developments

The next development steps for the PREVENTION component will focus on two directions: (i) optimize and extend the current implementation and (ii) deploy the PREVENTION component on live data where advanced data needs are required, in order to efficiently support the prescriptive quality analytics requirements of the COALA business cases and facilitate the integration with the other components. In this context, we will investigate possible extensions, adaptations and optimizations of the implemented algorithms and further enhancement of the component's algorithms library with additional algorithms. We will also examine the big data processing needs of the business cases, revisit and validate the lifecycle process implementation to efficiently manage new data and extend the core implementation accordingly.

Furthermore, we will deploy the PREVENTION component with additional data to validate the implemented algorithms and analyses for the WHR business case and examine whether additional risk assessment scenarios can be implemented.

4.3 History

Table 15 summarizes the implemented versions and the respective features of the PREVENTION component.

Table 15: Implementation History

Version	Release Date	New Features
1	30/06/2021	Core platform Analytics Management Queries Descriptive Statistics Algorithms First version of the analyses realizing the EDA demonstration scenario
2	31/03/2022	Analytics Process Queries

		Analytics Results Query Engine Classification and Regression Algorithms Automated Learning Algorithms Quality Control Data Model Second version of the analyses realizing the EDA demonstration scenario First version of the analyses related to the Predictive Quality, Risk Assessment and Checklist Recommendations scenarios

5 Preliminary Results in a context of business cases

In this Chapter, we present some preliminary results of quality analytics using PREVENTION in the context of the Whirlpool use case. The main aim is to demonstrate and document the application of the implemented software services that were described in Section 3 on several analytics scenarios utilizing the quality data derived from the Whirlpool data sources, as well as present the interfacing of the PREVENTION component that can expose the analytics outcomes to the user either with visualization graphs through the COALA Dashboard or by voice through the DIA.

The Whirlpool use case and data sources have been described in the Deliverable D1.3 "White goods production use case and scenarios". The scenarios described in the following sections utilize the Whirlpool data collected from the available sources, that contain information modelled in accordance with the Quality Control Data Model presented in Section 3.2.

5.1 Demonstration scenarios

5.1.1 Exploratory Data Analysis

The first scenario examined in this chapter refers to the Exploratory Data Analysis of the data available by the Whirlpool data sources. The Whirlpool use case, the preliminary exploratory data analysis and the corresponding data sources have been described in the Deliverables D2.3 "Prescriptive quality analytics service – version 1" and D1.3 "White goods production use case and scenarios". The scenario described in this section, further extends the preliminary exploratory data analysis described in the Deliverable D2.3 and presents the instantiation of the scenario using the current implementation of the PREVENTION component, described in detail in Section 3. In this demonstrator, the exploratory data analysis scenario has been realized by a set of descriptive analyses answering several smaller scenarios of two types. The first category of scenarios described in the following section includes the scenarios aiming at providing the descriptive analytics results to the DIA, in order to equip the assistant with the required data that are capable of answering the user's questions. The second category of descriptive analytics scenarios presented in this section refers to the cases where the analytics results generated by the PREVENTION component are provided to the COALA Dashboard (UI), where the visualization of the analytics is available to the user. All of these scenarios are implemented by defining the corresponding analyses entities, using basic statistics and descriptive algorithms and are available to the other COALA components by querying the GraphQL interface of the component, as described in the following sections.

5.1.1.1 Digital Intelligent Assistant Queries

The examples of the queries addressing the Digital Intelligent Assistant scenarios, are presented in a tabular format, where the top part of the table represents the scenario under examination and the bottom part the corresponding queries providing the analytics results of the implemented analysis that is able to answer the user's question.

In Table 16, the user asks to get information about the products affected by the defect with defect ID 3035, and the digital intelligent assistant is expected to return the list of the SKUs of the affected products. This scenario has been implemented by the analysis with analytics goal `"GROUP_DEFECTS_BY_DEFECT_ID_AND_SKU"`, using the historical data of the recorded defect occurrences. In this case, a `resultRequest` query with the appropriate request and filters will return the list of affected products, where each record is described by three fields, the DefectID the FGNum (or SKU) and the count. All the records have the requested Defect ID, a unique FGNum and the number of recorded defects for the product (count). Based on the answer the user can understand that multiple products are affected with the given Defect ID with varying recorded occurrences.

Table 16: DIA query example requesting the products affected by a specific defect

USER	DIA
COALA, please verify which products are affected by the defect with defect ID 3035?	The SKUs of the products affected by Defect ID 3035 are the following: 858732101900, 858742301900 ...
Query	**Answer**

Query:

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_DEFECT_ID_AND_SKU"
      requestFrom: "DA"
      requestFilters: [{
             name: "DefectID"
             action: "equals"
             values: "3035"
      }]
      dateFilters: {
          name: "Date"
        action: "greater"
        start: "2019-01-01"
        end: "2021-12-31"
        unit: "ALL"
      }
      }]
  ){
    results
  }
}
```

Answer:

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
           "DefectID": 3035,
           "FGNum":
858732101900,
           "count":2
          },
          {
            "DefectID": 3035,

"FGNum":858742301900,
            "count":68
          },
          {
           "DefectID": 3035,
           "FGNum":
858742401900,
           "count":11
          },
          {
           "DefectID": 3035,
           "FGNum":
858743501900,
           "count": 6
          },
          .........
        ]
      }
    ]
  }
```

In the example of Table 17, the user requests the number of defects occurred for a specific date and a specific product SKU number. The question is addressed by the analysis with analytics goal `"GROUP_DEFECTS_BY_SKU"` and its results are available using the `resultRequest` query with the appropriate request field and filters. In this example, the `requestFilters` to filter the results with the given SKU and the `dateFilters` to filter based on the given date, have been deployed. The answer consists of the list with one value containing three fields, the date and the FGNum with the same values of the user's request and the count (the number of recorded defects). A single record in the answer is expected, as the user requests a specific FGNum for a single date grouped by day.

Table 17: DIA query example requesting the number of defects occurrences for a specific product on a specific date

USER	DIA
COALA, how many defects have occurred on “2021-01-08” for product with SKU 858743901900?	On “2021-01-08" a total of 3 defects were record for the product with SKU 858743901900.

Query	Answer

Query

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_SKU"
      requestFrom: "DA"
      requestFilters: [{
             name: "FGNum"
             action: "equals"
             values: "858743901900"

       }]
        dateFilters: {
        name: "Date"
        action: "equals"
        start: "2021-01-08"
        end: "2021-01-08"
        unit: "DAY"
      }
      }]
  ){
    results
  }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "Date": "2021-01-08",
            "FGNum": 858743901900,
            "count": 3
          }
        ]
      }
    ]
  }
}
```

The next example in Table 18, is similar to the previous one with the difference that the user is interested in getting similar information for each Month. In this case the question is answered by the analysis with analytics goal `"GROUP_DEFECTS_BY_SKU"` and the main difference is that the unit field takes the value “MONTH” in the `dateFilters`, in order to return the results grouped by month within the date range provided. The answer contains different fields than the previous one. The field Date from the previous example is split up to two fields specifying the year and month that the results were recorded, while the FGNum contains the given FGNum and the count the number of recorded defects each month.

Table 18: DIA query example requesting the defects per months for a specific product

USER	DIA
COALA, how many defects have occurred each month for product with SKU 858743901900?	For the product with SKU 858743901900 the following number of defects were recorded each month: 2020/04 - 11, 2020/05 - 16 ,…….

Query	Answer

Query

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_SKU"
      requestFrom: "DA"
      requestFilters: [{
             name: "FGNum"
             action: "equals"
             values: "858743901900"
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "Year": 2020,
            "Month": 4,
            "FGNum": 858743901900,
      }]
       dateFilters: {
        name: "Date"
        action: "greater"
        start: "2018-01-01"
        end: "2022-12-31"
        unit: "MONTH"
      }
      }]
  ){
    results
  }
}
```

```
          "count": 11
        },
        {
          "Year": 2020,
          "Month": 5,
          "FGNum": 858743901900,
          "count": 16
        },
        {
          "Year": 2020,
          "Month": 6,
          "FGNum": 858743901900,
          "count": 16
        },
        {
          "Year": 2020,
          "Month": 7,
          "FGNum": 858743901900,
          "count": 42
        },
        {
          "Year": 2020,
          "Month": 8,
          "FGNum": 858743901900,
          "count": 62
        },
        ...............
      ]
    }
  ]
 }
}
```

In Table 19 the user requests the most frequent defect for a specific product. This scenario is addressed by the analysis with the analytics goal `"GROUP_DEFECTS_BY_DEFECT_ID_AND_SKU"` and the filters depicted in the example query of the table. In this example, the `sumFilters` mechanism has been deployed to return the most frequent Defect ID by performing the action "max". As it is expected, the answer in this case contains only one record with three fields, the most frequent Defect ID the given FGNum and the number of recorder defects (count).

Table 19: DIA query example requesting the most frequent defect for a specific product

USER	DIA
COALA, which defect is the most frequent for the product with SKU 858743901900?	For the product with SKU 858743901900 the most common defect is 1371 with a total of 72 recorded defects.

Query

```
query q1{
  resultRequest(request:{
    request: "GROUP_DEFECTS_BY_DEFECT_ID_AND_SKU"
    requestFrom: "DA"
    requestFilters: {
      name: "FGNum"
      action: "equals"
      values: "858743901900"
     }
    dateFilters:{
      name: "Date"
      action: "greater"
      start: "2020-02-23"
      unit: "ALL"
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "DefectID": "1371",
            "FGNum":
"858743901900",
            "count": "72"
          }
        ]
      }
    ]
```

```
    }
    sumFilters: {
      name: "count"
      action: "max"
      }
  }){
    results
  }
}
```
	```
	  }
	}
	```

5.1.1.2 User Interface Queries

This section presents the different descriptive analyses and the available queries that have been implemented to support the exploratory data analysis and the visualization of the analytics results through the COALA User Interface.

In Table 20, a simple query providing the number of defects each day is demonstrated. In this example, no filters are used because the answer should be the raw result data of the appropriate analysis with analytics goal `"GROUP_DEFECTS_BY_DATE"`, while the `requestFrom` field indicates that the results should be provided in a form suitable for the User Interface client. The expected answer contains multiple records in a list, and each record contains two fields the date and the number of defects recorded.

Table 20: UI query example requesting the defect occurrences grouped by date

Defects per day	
Query	**Answer**

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_DATE"
      requestFrom: "UI"
      }]
  ){
    results
  }
}
```
	```
	{
	  "data": {
	    "resultRequest": [
	      {
	        "results": [
	          {
	            "Date": "2020-04-23",
	            "count": 12
	          },
	          {
	            "Date": "2020-04-24",
	            "count": 17
	          },
	          {
	            "Date": "2020-04-27",
	            "count": 60
	          },
	          {
	            "Date": "2020-04-28",
	            "count": 53
	          },
	          {
	            "Date": "2020-04-29",
	            "count": 44
	          },
	….
	        ]
	      }
	    ]
	  }
	}
	```

Figure 8 depicts the corresponding visualization of the Defects per Day query results, in the form of a line chart. From the chart the volatility of the number of defects each day is easily observed by the end user and the capability of monitoring the trend and spikes occurrences is provided.

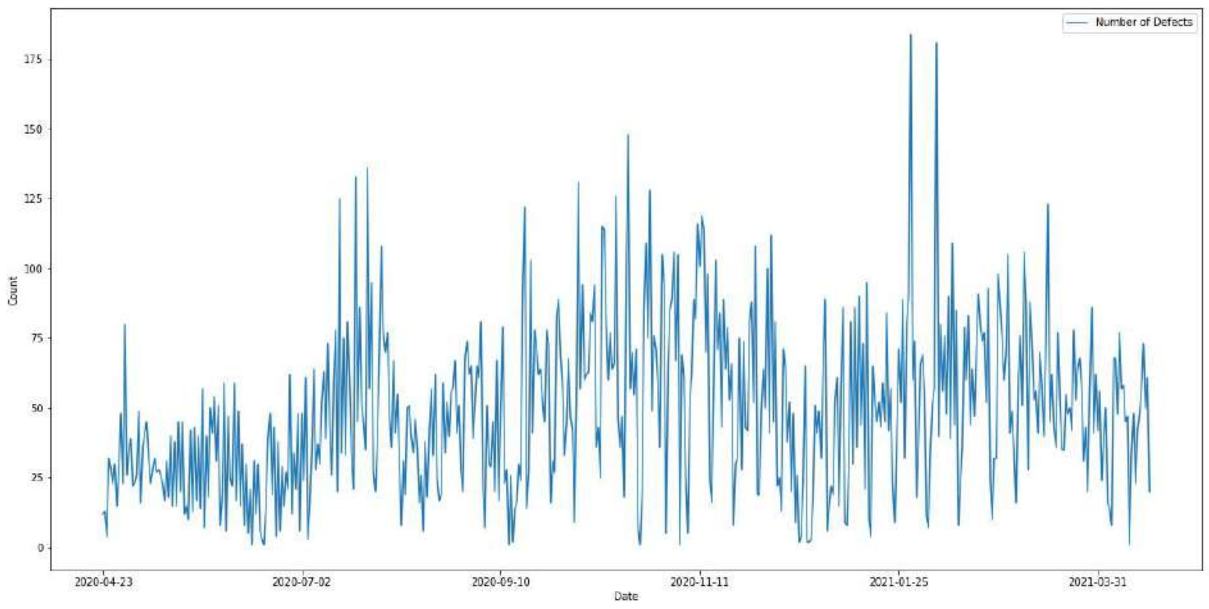


Figure 8: Number of Defects per day visualization example

Table 21 presents the query related to the analysis with goal type `"DEFECT_RATE"`, in which the defected products rate is calculated after a specific date, grouped by year. In this case the `dateFilters` have been employed to get the appropriate date range and group the values per year. The excepted answer may have multiple records, however, the specific dataset used for the presented results, included only data for the year 2021. Each record of the list consists of four fields, the year, the number of defect occurrences, the number of total entries, and the defect rate.

Table 21: UI query example requesting the defect rate for a specific year

Defect Rate

Query

```
query q1 {
  resultRequest(request:[
    {
      request:"DEFECT_RATE"
      requestFrom: "UI"
      dateFilters: {
        name: "Date"
        action: "greater"
        start: "2020-05-23"
        unit: "YEAR"
      }
      }]
  ){
    results
  }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "Year": 2021,
            "count": 20364,
            "total": 22486,
            "rate": 0.9056301698834831
          }
        ]
      }
    ]
  }
}
```

The example presented in Table 22, corresponds to the analysis with the goal `"GROUP_DEFECTS_BY_DEFECT_SOURCE"` that groups the defect occurrences by the source that generated the defect. The answer contains multiple records that consist of the fields: Defect Source ID, the Defect Source Description and the number of defects generated by the specific source. The available dataset included six different defect sources, all of which are present in the output result.

Table 22: UI query example requesting the defect occurrences grouped by the defect source

Defects per source	
Query	**Answer**

Query

```
query q1 {
  resultRequest(request:[
    {

request:"GROUP_DEFECTS_BY_DEFECT_SOURCE"
      requestFrom: "UI"
      dateFilters: {
         name: "Date"
        action: "greater"
        start: "2019-01-01"
        end: "2021-12-31"
        unit: "ALL"
      }
      }]
  ){
    results
  }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "DefectSrcID": 1,
            "DefectSrcDes": "Sconosciuta",
            "count": 8568
          },
          {
            "DefectSrcID": 2,
            "DefectSrcDes": "Reso CQA",
            "count": 3940
          },
          {
            "DefectSrcID": 3,
            "DefectSrcDes": "Primari",
            "count": 3466
          },
          {
            "DefectSrcID": 4,
            "DefectSrcDes": "Premontaggio",
            "count": 4069
          },
          {
            "DefectSrcID": 5,
            "DefectSrcDes": "Montaggio",
            "count": 4146
          },
          {
            "DefectSrcID": 6,
            "DefectSrcDes": "Cooking",
            "count": 1455
          }
        ]
      }
    ]
  }
}
```

A candidate visualization for the analysis results available by this example is depicted in Figure 9. We provide the results from the Defect Source query as a pie chart graph. Using a similar visualization, the user is able to identify the origin of defects and observe differences between the defect sources.

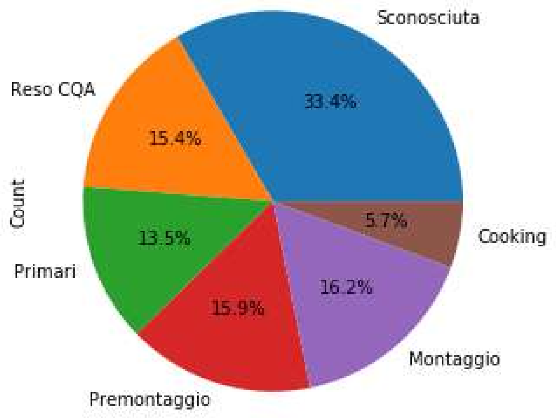


Figure 9: Defect occurrences grouped by defect source visualization example

Table 23 depicts an example of the analysis results implementing the grouping of the defect occurrences by the repair status. In this example, the answer consists of two records, where the first record contains the number of repairs and the second the number of replacements. The repair rate for the selected dates can be calculated by dividing the number of repairs with the number of repairs and replacements.

Table 23: UI query example requesting the defect occurrences grouped by their repair status

Defects by repair status

Query

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_REPAIR"
      requestFrom: "UI"
      dateFilters: {
        name: "Date"
        action: "greater"
        start: "2019-01-01"
        end: "2021-12-31"
        unit: "ALL"
      }
      }]
  ){
    results
  }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "RepairID": 1,
            "RepairDes": "Riparato",
            "count": 11748
          },
          {
            "RepairID": 2,
            "RepairDes": "Sostituito",
            "count": 13896
          }
        ]
      }
    ]
  }
}
```

Similarly to the previous examples, Figure 10 presents an example visualization of the output of the example query. The same graph with values from different dates can also be used to monitor their changes in respect to the time.

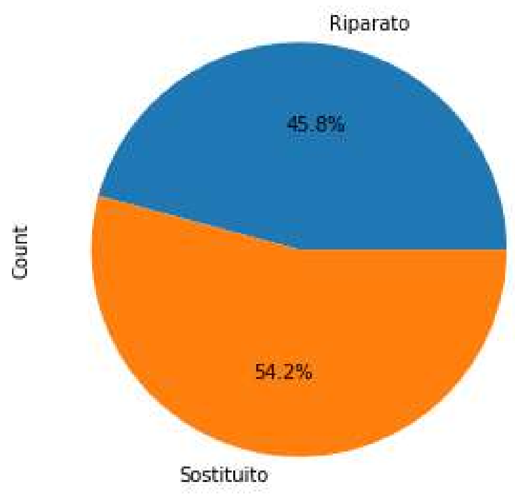


Figure 10: Defect occurrences grouped by repair status example

Following the same approach, Table 24 the query requesting the same analysis results, with different `dateFilters`. In this case, the results are grouped by date and repair status.

Table 24: UI query example requesting the defect occurrences grouped by their repair status per day

Repairs per day

Query	Answer

Query:

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_REPAIR"
      requestFrom: "UI"
      dateFilters: {
        name: "Date"
        action: "greater"
        start: "2019-01-01"
        end: "2021-12-31"
        unit: "DAY"
      }
      }]
  ){
    results
  }
}
```

Answer:

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "Date": "2020-04-23",
            "RepairID": 2,
            "RepairDes": "Sostituito",
            "count": 12
          },
          {
            "Date": "2020-04-24",
            "RepairID": 1,
            "RepairDes": "Riparato",
            "count": 13
          },
          {
            "Date": "2020-04-24",
            "RepairID": 2,
            "RepairDes": "Sostituito",
            "count": 4
          },
          {
            "Date": "2020-04-27",
            "RepairID": 1,
            "RepairDes": "Riparato",
            "count": 32
          },
          {
            "Date": "2020-04-27",
            "RepairID": 2,
            "RepairDes": "Sostituito",
            "count": 28
          },
……
        ]
      }
    ]
  }
}
```

The example of the Table 25 returns the analysis results of the analysis with goal “`GROUP_DEFECTS_BY_DEFECT_ID`” that groups the defect occurrences by the corresponding defect id and the years included in the requested date interval. The answer consists of a list of records with three fields, the year, the defect ID and the number of occurrences.

Table 25: UI query example requesting the defect occurrences grouped by the defect id and year

Type of defects

Query	Answer

Query:

```
query q1 {
  resultRequest(request:[
    {

request:"GROUP_DEFECTS_BY_DEFECT_ID"
      requestFrom: "UI"
```

Answer:

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
```

dateFilters: {
name: "Date"
action: "greater"
start: "2019-01-01"
end: "2021-12-31"
unit: "YEAR"
}
}]
){
results
}
}
	"Year": 2020,
	"DefectID": 1001,
	"count": 26
	},
	{
	"Year": 2020,
	"DefectID": 1003,
	"count": 1
	},
	{
	"Year": 2020,
	"DefectID": 1005,
	"count": 3
	},
	……
	]
	}
	]
	}
	}

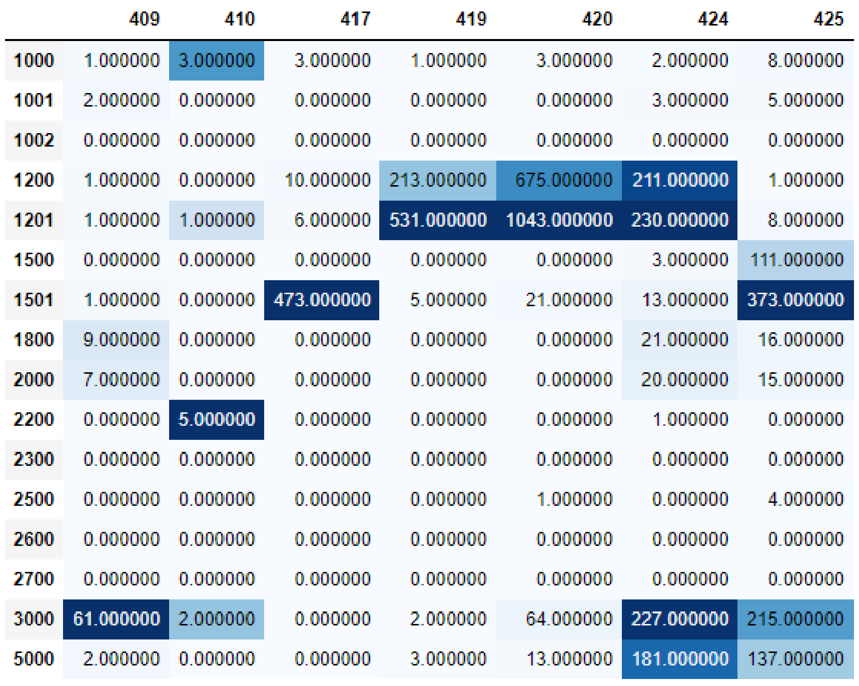
The purpose of the next example depicted in Table 26, is to provide infromation about the correlation between the Defect Groups and the Part family numbers and provide the data in a format that enables the creation of the corresponding heatmap visualization. The current example returns the results of the analysis with goal `"GROUP_DEFECTS_BY_DEFECT_ID_AND_FAMILY_ID"`, that is responsible for grouping the defect occurrences of the requested date interval by their defect id and family id. The answer contains a list of records, each of which holds the total number of times a defect with the indicated Defect Group and Family ID occurred.

Table 26: UI query example requesting the defect occurrences grouped by the defect id and family id

Defect groups – part family heatmap	
Query	**Answer**

query q1 {
resultRequest(request:[
{
request:
"GROUP_DEFECTS_BY_DEFECT_ID_AND_FAMILY_ID"
requestFrom: "UI"
dateFilters: {
name: "Date"
action: "greater"
start: "2019-01-01"
end: "2021-12-31"
unit: "ALL"
}
}]
){
results
}
}
	{
	"data": {
	"resultRequest": [
	{
	"results": [
	{
	"DefectGrp": 1000,
	"FamilyID": 0,
	"count": 1
	},
	{
	"DefectGrp": 1000,
	"FamilyID": 409,
	"count": 1
	},
	{
	"DefectGrp": 1000,
	"FamilyID": 410,
	"count": 3
	},
	……
	]
	}
	]
	}
	}

Figure 11 shows an example heatmap or confusion matrix between the Defect Group and Family ID based on some of the results from the above query. From this visualization the user is able to observe how many defects had the same combination of Defect Group and Family ID and get a better understanding of their relation.



	409	410	417	419	420	424	425
1000	1.000000	3.000000	3.000000	1.000000	3.000000	2.000000	8.000000
1001	2.000000	0.000000	0.000000	0.000000	0.000000	3.000000	5.000000
1002	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1200	1.000000	0.000000	10.000000	213.000000	675.000000	211.000000	1.000000
1201	1.000000	1.000000	6.000000	531.000000	1043.000000	230.000000	8.000000
1500	0.000000	0.000000	0.000000	0.000000	0.000000	3.000000	111.000000
1501	1.000000	0.000000	473.000000	5.000000	21.000000	13.000000	373.000000
1800	9.000000	0.000000	0.000000	0.000000	0.000000	21.000000	16.000000
2000	7.000000	0.000000	0.000000	0.000000	0.000000	20.000000	15.000000
2200	0.000000	5.000000	0.000000	0.000000	0.000000	1.000000	0.000000
2300	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2500	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	4.000000
2600	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2700	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3000	61.000000	2.000000	0.000000	2.000000	64.000000	227.000000	215.000000
5000	2.000000	0.000000	0.000000	3.000000	13.000000	181.000000	137.000000

Figure 11: Example visualization of the defect id and family id correlation data

The example of the Table 27, returns the total number of defects occurred for each SKU, for the requested date interval. The answer contains a list of records where each one has two fields, the FGNum (or SKU) and the number of occurrences.

Table 27: UI query example requesting the defect occurrences grouped by product SKU

Number of defects per SKU

Query

```
query q1 {
  resultRequest(request:[
    {
      request:"GROUP_DEFECTS_BY_SKU"
      requestFrom: "UI"
      dateFilters: {
        name: "Date"
        action: "greater"
        start: "2019-01-01"
        end: "2021-12-31"
        unit: "ALL"
      }
      }]
  ){
    results
  }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "FGNum": 858732101900,
            "count": 110
          },
          {
            "FGNum": 858742229900,
            "count": 2
          },
          {
            "FGNum": 858742301900,
            "count": 869
          },
          {
            "FGNum": 858742401900,
            "count": 1164
```

	`            },`
	`......`
	`          ]`
	`        }`
	`      ]`
	`    }`
	`}`

An example visualization of the output of this query is presented in Figure 12. Even though there are several SKUs, by visualizing the results we can clearly observe which SKUs are more or less affected by the defects.

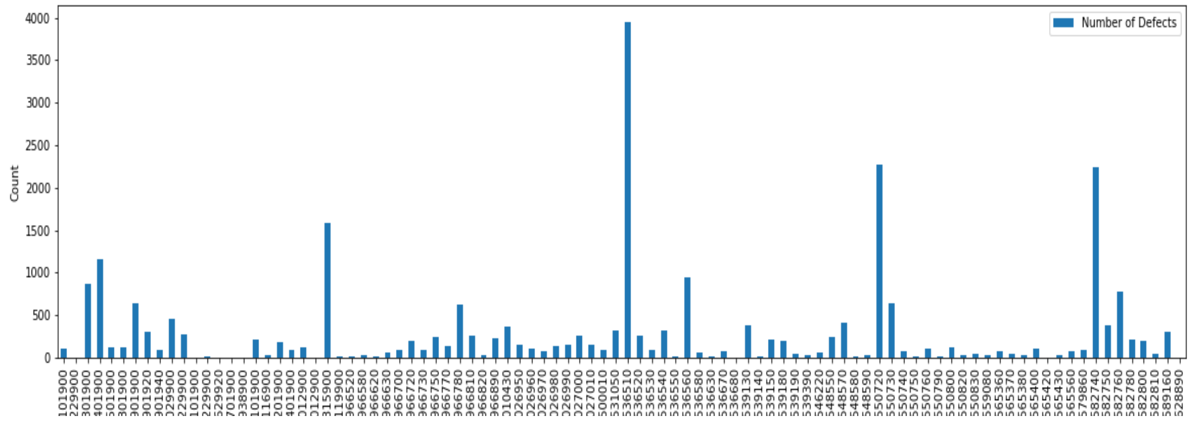


Figure 12: Number of defects grouped by SKU visualization example

In the last example of the descriptive statistics shown in Table 28, the goal is to group the defect occurrences by the Material Number feature, corresponding to the specific part numbers consisting the finished good. These results are generated by the analysis with the goal `"GROUP_DEFECTS_BY_MATERIAL_NUMBER"` while the answer is similar to the previous example with the difference that there is an extra field specifying the description of the Material Number.

Table 28: UI query example requesting the defect occurrences grouped by material number

Number of defects per Material Number	
Query	**Answer**

`query q1 {`
`  resultRequest(request:[`
`    {`
`      request:`
`"GROUP_DEFECTS_BY_MATERIAL_NUMBER"`
`      requestFrom: "UI"`
`      dateFilters: {`
`        name: "Date"`
`        action: "greater"`
`        start: "2019-01-01"`
`        end: "2021-12-31"`
`        unit: "ALL"`
`      }`
`      }]`
`  ){`
`    results`
`  }`
`}`
	`{`
	`  "data": {`
	`    "resultRequest": [`
	`      {`
	`        "results": [`
	`          {`
	`            "MatNum": 121911600006,`
	`            "MatDes": "TAPE PH304I 75MMX1000M`
	`STOCKVI",`
	`            "count": 22`
	`          },`
	`          {`
	`            "MatNum": 400010367349,`
	`            "MatDes": "CHASSIE  MIBI UPDATE",`
	`            "count": 4`
	`          },`
	`          {`
	`            "MatNum": 400010367350,`

	"MatDes": "CAVITY FRONT MIBI ZN",
	"count": 34
	},
	{
	"MatNum": 400010367356,
	"MatDes": "CAVITY WRAPPING ZN PRO-FIT MIBI UPDATE",
	"count": 406
	},
	……
	]
	}
	]
	}
	}

5.1.2 Predictive Quality

The second category of scenarios addresses the predictive quality requirements, based on the available data and the current implementation. For the implementation of the analyses corresponding to these scenarios, various ML algorithms are used in order to make predictions for specific features based on the available data. The main query capable of running and providing the requested predictions is the triggerRequest query of the analytics process queries described in Section 3.3.2. In the following paragraphs two main predictive quality scenarios are presented. These scenarios perform two different predictions, the Defect Group of a product and the Number of Orders with Defect in the following days.

5.1.2.1 Defect Group Prediction

To predict the Defect group of the defects that is most probable to manifest for a specific product, the user triggers the `"PREDICT_DEFECT_GROUP"` analysis providing the required input information about the product in the `defectGroupPrediction` field. The mandatory input fields are `lineSequence`, `sku`, `defectSourceId`, `statId`, and `familyId` of the product. The PREVENTION component uses the input values in order to make a prediction fitting the DT classifier model that has been created and correlated to the specific analysis and has been trained using the available historical data. The generated prediction consists of a single field holding the predicted Defect Group unique identifier.

Regarding the Whirlpool case, the defect ids are categorized into different defect groups. Even though predicting the most probable Defect Group for a specific product gives valuable information about the expected defects, the importance of the prediction is significantly higher considering that it can be used as the basis to make further predictions and recommendations. Thus, by having a reliable initial prediction the advanced analysis employing this prediction will result in more reliable predictions and recommendations.

The following example in Table 29, shows the example dialog between the user and the DIA, along with the query triggering the implemented analysis that is able to generate a defect group prediction about a specific product by providing the required input attributes.

Table 29: Example Defect Group Prediction

USER	DIA
What is the most probable group of defects to manifest for the product with SKU 858732101900, Line Sequence 10620, Defect Source Id 6, Stat Id 501 and Family ID 0419?	The most probable group of defects is defect group with id 1201.
Query	**Answer**

query q1{
  triggerRequest(request:{
    request: "PREDICT_DEFECT_GROUP"
    defectGroupPrediction: {
      lineSequence: "10620"
      sku: "858732101900"
      defectSourceId: "6"
      statId: "501"
      familyId: "0419"
    }
  }){
    results
  }
}
	{
	  "data": {
	    "triggerRequest": [
	      {
	        "results": [
	          {
	            "DefectGrp":
	1201
	          }
	        ]
	      }
	    ]
	  }
	}

5.1.2.2 Number of Orders with Defect in the following days Prediction

To predict the number of orders that will manifest some defect in the following days, the user triggers the implemented analysis with the goal "PREDICT_NUMBER_OF_DEFECTS". In order to trigger the analysis execution, the users need to provide the model parameter in the numberOfDefectsPrediction field. This analysis has been implemented using two different AutoML libraries, described in Section 3.2. The user selects the model to be used for generating the prediction, by providing the corresponding value in the model field. This field supports two values (“AUTOKERAS” and “FEDOT”), that specify which AutoML model will be used to make the prediction. In the current implementation, the AUTOKERAS (created and trained) model predicts the number of defects for the next day, while the FEDOT model generates predictions for the next 10 days.

The prediction of the number of orders with defects in the following days can be used from the workers to get a glimpse of the work ahead and can be used as an alarm mechanism for a failure in the production line causing a greater number of defects than expected.

The following two examples in Table 30 and Table 31, present the implemented queries that are capable ok making a prediction about the Number of Orders with Defects in the upcoming days. In both cases the user isn’t required to give additional input values except of the model selection.

Table 30: Example Next day Number of Orders with defect prediction - Autokeras

USER	DIA
How many orders will show defects tomorrow?	271 orders are predicted to have a defect tomorrow.
Query	**Answer**

query q1{
  triggerRequest(request:{
    request:
"PREDICT_NUMBER_OF_DEFECTS"
      numberOfDefectsPrediction: {
      model: "AUTOKERAS"
	{
	  "data": {
	    "triggerRequest": [
	      {
	        "results": [
	          {

}
}){
results
}
}
	"Defects Per Day": [
	271
	]
	}
	]
	}
	]
	}
	}

Table 31: Example Next day Number of Orders with defect prediction - FEDOT

USER	DIA
How many orders will show defects in the upcoming days?	The predicted number of orders with defects are: 12/03/2021 - 372, 13/03/2021 - 394, ….
Query	**Answer**

query q1{
triggerRequest(request:{
request: "PREDICT_NUMBER_OF_DEFECTS"
numberOfDefectsPrediction: {
model: "FEDOT"
}
}){
results
}
}
	{
	"data": {
	"triggerRequest": [
	{
	"results": [
	{
	"Defects Per Day": [
	372,
	394,
	376,
	413,
	350,
	392,
	465,
	404,
	384,
	328
	]
	}
	]
	}
	]
	}
	}

5.1.3 Risk Assessment

The PREVENTION component supports the development of multiple risk assessment analyses. Risk assessment is crucial for effectively identifying and managing the risks and undesired events that affect business continuity and performance. These scenarios employ the data available from the Defect Collection System, consisting of the manifested defects, to assess the risk criticality for each defect and provide an answer to questions similar to "how critical is a defect". Following the business requirements, the risk is evaluated by calculating the priority index metric, that is defined based on the frequency, the severity, the source that produced the defect, and the brand related to the defect under examination. Priority index with a high value indicates a risk critical defect, while a lower priority index indicates a less critical defect.

The implementation of the first scenario, upon the arrival of new data, calculates the priority index of all the identified defects. The risk assessment results are then available to the COALA components with the resultRequest query of the analytics process. Table 32 presents an example scenario where the user requests the priority index for a specific product and the DIA provides the answer based on the query resultRequest depicted in the table. The query returns the results of the analysis with analytics goal `"RISK_ASSESSMENT"` filtered by the requested defects and provides the priority index for the requested defects. The risk assessment results can be further classified following a risk taxonomy based on the priority index.

Table 32: Example of risk assessment results for a specific defect

USER	DIA
What is the priority index for defect with defect id 1240?	The priority index of the defect 1240 is 28
Query	**Answer**

Query:

```
query q1 {
  resultRequest(request: [
    {
      request: "RISK_ASSESSMENT"
      requestFrom: "UI"
      requestFilters: [{
      name: "DefectID"
      action: "equals"
      values: "1240"
     }]
  }]
 ){
   results
 }
}
```

Answer:

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "DefectID": 1240,
            "DefectSrcID": "[1 3 -1 5 4
6 2]",
            "MatNum": "[461968977761
400010845983 400010624553 400010697948
400010544646…]",
            "FamilyID": "[1047 1040
1016 1019…]",
            "FGNum": "[859991550720
858743901920 858744029900 859991550730
859991539190…]",
            "SeverityWeight": 4,
            "FrequencyPercentage":
0.002369668246445498,
            "FrequencyPoints": 1,
            "DetectionPoints": 5,
            "SourcePoints": 7,
            "PriorityIndex": 28
          }
        ]
      }
    ]
  }
}
```

Additionally, Table 33 presents an example scenario of retrieving the risk assessment results for a specific SKU and a specific defect group. These types of scenarios are answered by the analysis with the analytics goal `"RISK_ASSESSMENT_PER_SKU"`. Table 34 presents an example of retrieving the most critical defect for a specific SKU, while Table 35 presents an example of filtering the risk assessment results to retrieve the most critical defect for a specific SKU and a specific defect group.

Table 33: Example of risk assessment results for a specific SKU and defect group

USER	DIA
What is the priority index for the product with SKU 858732101900 and the defects of the defect group with defect group id 1201?	The product with SKU 858732101900 shows two defects of the defect group with id 1201, the defect with id 1285 and the defect with id 1297. The priority index of the defect 1285 is 60 and the priority index of the defect 1297 is 40.
Query	**Answer**

Query:

```
query q1 {
  resultRequest(request: [
```

Answer:

```
{
  "data": {
```

USER	DIA
What is the priority index for the product with SKU 858732101900 and the defects of the defect group with defect group id 1201?	The product with SKU 858732101900 shows two defects of the defect group with id 1201, the defect with id 1285 and the defect with id 1297. The priority index of the defect 1285 is 60 and the priority index of the defect 1297 is 40.

Query

```
    {
      request:
"RISK_ASSESSMENT_PER_SKU"
      requestFrom: "UI"
      requestFilters: [{
      name: "FGNum"
      action: "equals"
      values: "858732101900"
      }, {
      name: "DefectGrp"
      action: "equals"
      values: "1201"
     }]
  }]
 ){
   results
 }
}
```

Answer

```
    "resultRequest": [
      {
        "results": [
         {
            "FGNum": 858732101900,
            "FamilyID": 419,
            "DefectGrp": 1201,
            "DefectID": 1285,
            "Id": 8,
            "SourcePoints": 3,
            "totalDefectsPerGroup": 14,
            "FrequencyPercentage":
0.5714285714285714,
            "SeverityWeight": 4,
            "DetectionPoints": 5,
            "FrequencyPoints": 5,
            "PriorityIndex": 60
          },
          {
            "FGNum": 858732101900,
            "FamilyID": 419,
            "DefectGrp": 1201,
            "DefectID": 1297,
            "Id": 6,
            "SourcePoints": 2,
            "totalDefectsPerGroup": 14,
            "FrequencyPercentage":
0.42857142857142855,
            "SeverityWeight": 4,
            "DetectionPoints": 5,
            "FrequencyPoints": 5,
            "PriorityIndex": 40
          },
…
]
      }
    ]
  }
}
```

Table 34: Example of risk assessment results for the most critical defect for a specific SKU

User	DIA
What is the most critical defect for the product with SKU 858732101900?	The most critical defect for this product is the defect with id 1285 and priority index 60.

Query

```
query q1 {
  resultRequest(request: [
    {
      request:
"RISK_ASSESSMENT_PER_SKU"
      requestFrom: "UI"
      requestFilters: [{
      name: "FGNum"
      action: "equals"
      values: "858732101900"
      }]
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "FGNum": 858732101900,
            "FamilyID": 419,
            "DefectGrp": 1201,
            "DefectID": 1285,
            "Id": 8,
```

User	DIA
What is the most critical defect for the product with SKU 858732101900?	The most critical defect for this product is the defect with id 1285 and priority index 60.

Query

```
       sumFilters: {
         field: "PriorityIndex"
         action: "topn"
         numberOfValues: "1"
     }
   }]
  ){
    results
  }
}
```

Answer

```
             "SourcePoints": 3,
             "totalDefectsPerGroup": 14,
             "FrequencyPercentage":
0.5714285714285714,
             "SeverityWeight": 4,
             "DetectionPoints": 5,
             "FrequencyPoints": 5,
             "PriorityIndex": 60
           }
         ]
       }
     ]
   }
}
```

Table 35: Example of risk assessment results for the most critical defect for a specific SKU and defect group

User	DIA
What is the most critical defect of the defect group with id 1201 for the product with SKU 858732101900?	The most critical defect of the defect group 1201 for this product is the defect with id 1285 and priority index 60.

Query

```
query q1 {
  resultRequest(request: [
    {
      request:
"RISK_ASSESSMENT_PER_SKU"
      requestFrom: "UI"
      requestFilters: [{
      name: "FGNum"
      action: "equals"
      values: "858732101900"
      },
      {
      name: "DefectGrp"
      action: "equals"
      values: "1201"
      }]
      sumFilters: {
        name: "PriorityIndex"
        action: "topn"
        numberOfValues: "1"
    }
  }]
 ){
   results
 }
}
```

Answer

```
{
  "data": {
    "resultRequest": [
      {
        "results": [
          {
            "FGNum": 858732101900,
            "FamilyID": 419,
            "DefectGrp": 1201,
            "DefectID": 1285,
            "Id": 8,
            "SourcePoints": 3,
            "totalDefectsPerGroup": 14,
            "FrequencyPercentage":
0.5714285714285714,
            "SeverityWeight": 4,
            "DetectionPoints": 5,
            "FrequencyPoints": 5,
            "PriorityIndex": 60
          }
        ]
      }
    ]
  }
}
```

5.1.4 Checklist Recommendations

In this scenario, the available checklist definitions along with the predictive analytics outcomes and the risk assessment results of the previous scenarios are incorporated into the prescriptive analytics method of the PREVENTION component to generate the next action recommendations for a specific user level that will be able to address timely the predicted defects. In the present implementation, when a recommendation is required, the defined analysis that corresponds to this functionality is triggered, following the analytics process defined in the previous sections. Table 36 depicts the checklist recommendation scenario including the predictive quality and the risk assessment scenarios, in order to present the complete prescriptive analytics approach supported by combining the individual queries of the PREVENTION component.

Table 36: Checklist Recommendation scenario

Checklist Recommendation Scenario		
User	What is the most probable group of defects to manifest for the product with SKU 858732101900, Line Sequence 10620, Defect Source Id 6, Stat Id 501 and Family ID 0419?	Predict defect group scenario
DIA	The most probable group of defects is defect group with id 1201	
User	What is the most critical defect of the defect group with id 1201 for the product with SKU 858732101900?	Risk assessment scenario 2
DIA	The most critical defect of the defect group 1201 for this product is the defect with id 1285 and priority index 60.	
User	What is the priority index of the defect with id 1285 for all the available SKUs?	Risk assessment scenario 1
DIA	The priority index of the defect 1285 is 40	
User	What actions should I implement to mitigate the manifestation of the defect with id 1285 on the product with SKU 858732101900, given the fact that my expertise level is intermediate?	Checklist recommendation query
DIA	According to the product's checklist, you should implement the following actions sequentially: • check correspondence in COALA HMI (tablet) – REQUIRED • Use scanner to detect readability - OPTIONAL	

The implemented method receives as input the SKU of the selected finished good, the unique identifier of the defect, related to the finished good product, that has been predicted to occur and the expertise level of the user that is required to implement the checklist actions, Since the available checklist data indicate that users of different expertise level are required to implement different set of actions. The analysis results in the list of actions related to the predicted defect that the user should implement sequentially based on the mandatory status of the recommended actions. Table 37 presents an example query of this functionality and the generated answer.

Table 37: Example of checklist recommendation generation for a specific SKU, a predicted defect and user experience

Query	Answer

```
query q1{
  triggerRequest(request:{
    request:
"CHECKLIST_RECOMMENDATION"
   checklistRecommendation: {
    sku: "858732101900"
    defectId: "1285"
    expertLevel: "intermediate"
  }
  }){
    results
```
	```
	{
	  "data": {
	    "triggerRequest": [
	      {
	        "results": [
	          {
	            "Action (how)": "check
	correspondence in COALA HMI ( tablet)",
	            "Interm type": "always"
	          },
	          {
	```

```
  }
}
```

```
                "Action (how)": "Use scanner to
detect readability",
                "Interm type": "randomly"
              }
            ]
          }
        ]
      }
    }
```

6 Conclusion and Outlook

This Deliverable describes the development and implementation of the PREVENTION component, focusing on the supported algorithms and the technical implementations. The implementation presented in this deliverable highlights the descriptive, predictive and prescriptive quality analytics capabilities of the component and demonstrates indicative results using the quality data initially available by the Whirlpool use case.

The next steps related to Task 2.1 will focus on extending and adapting the analytics capabilities of the PREVENTION. Further capabilities of PREVENTION, such as time series forecasting and automated learning, will be further tested as soon as relevant data become available by use case partners. Based on these tests, the current implementation will be extended to support additional, advanced analytics methods and machine learning algorithms and its application to additional sophisticated scenarios will be examined.

7 References

Bishop, C. M., & Nasrabadi, N. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 738). New York: springer.

Bontempi, G., Ben Taieb, S., & Borgne, Y. A. L. (2012, July). Machine learning strategies for time series forecasting. In European business intelligence summer school (pp. 62-77). Springer, Berlin, Heidelberg.

Chatfield, C. (2000). Time-series forecasting. Chapman and Hall/CRC.

Chomboon, K., Chujai, P., Teerarassamee, P., Kerdprasop, K., & Kerdprasop, N. (2015, March). An empirical study of distance metrics for k-nearest neighbor algorithm. In Proceedings of the 3rd international conference on industrial application engineering (pp. 280-285).

Chunming Liu, Xin Xu, Dewen Hu: Multiobjective Reinforcement Learning: A Com-prehensive Overview. IEEE Transactions on Systems, Man, and Cybernetics: Sys-tems. 45, 385-398 (2015).

Dornheim, J., Link, N., Gumbsch, P.: Model-free Adaptive Optimal Control of Epi-sodic Fixed-horizon Manufacturing Processes Using Reinforcement Learning. Interna-tional Journal of Control, Automation and Systems. (2019)

Draper, N. R., & Smith, H. (1998). Applied regression analysis (Vol. 326). John Wiley & Sons.

Guo, G., Wang, H., Bell, D., Bi, Y., & Greer, K. (2003, November). KNN model-based approach in classification. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 986-996). Springer, Berlin, Heidelberg.

Hand, D. J. (2007). Principles of data mining. Drug safety, 30(7), 621-622.

He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. Knowledge-Based Systems, 212, 106622.

Jin, H., Song, Q., & Hu, X. (2019, July). Auto-keras: An efficient neural architecture search system. In Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining (pp. 1946-1956).

Kotsiantis, S. B., Zaharakis, I. D., & Pintelas, P. E. (2006). Machine learning: a review of classification and combining techniques. Artificial Intelligence Review, 26(3), 159-190.

Li, G., Gomez, R., Nakamura, K., He, B.: Human-Centered Reinforcement Learning: A Survey. IEEE Transactions on Human-Machine Systems. 49, 337-349 (2019)

Nikitin, N. O., Vychuzhanin, P., Sarafanov, M., Polonskaia, I. S., Revin, I., Barabanova, I. V., ... & Boukhanovsky, A. (2022). Automated evolutionary approach for the design of composite machine learning pipelines. Future Generation Computer Systems, 127, 109-125.

Rocchetta, R., Bellani, L., Compare, M., Zio, E., Patelli, E.: A reinforcement learning framework for optimal operation and maintenance of power grids. Applied Energy. 241, 291-301 (2019).

Safavian, S. R., & Landgrebe, D. (1991). A survey of decision tree classifier methodology. IEEE transactions on systems, man, and cybernetics, 21(3), 660-674.

Sutton, R., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, (2018)

Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y. F., Tu, W. W., ... & Yu, Y. (2018). Taking human out of learning applications: A survey on automated machine learning. arXiv preprint arXiv:1810.13306.

Zhang, H., & Yu, T. (2020). Taxonomy of reinforcement learning algorithms. In Deep Reinforcement Learning (pp. 125-133). Springer, Singapore.