MARTIN: An End-to-end Microservice Architecture for Predictive Maintenance in Industry 4.0

Abdessalam Elhabbash*, Kamil Rogoda*, Yehia Elkhatib†
* School of Computing and Communications, Lancaster University, United Kingdom
† School of Computing Science, University of Glasgow, United Kingdom

***Abstract*—The amount of data generated in Industry 4.0 and the introduction of advanced data analytics support establishing "smart factories" and one of its crucial characteristics – predictive maintenance. Current solutions primarily focus on offline predictions and do not provide end-to-end scalable solutions. Furthermore, there is a lack of support for incremental machine learning in predictive maintenance. This paper addresses these limitations by proposing `MARTIN`, a scalable microservice architecture for predictive maintenance that can collect, store, and analyse data, and make decisions based on the machine state. The architecture uses incremental learning as the basis for predictions. The designed system was implemented and its performance was evaluated experimentally. The results show that the solution can provide high prediction accuracy in terms of practical processing time.**

***Index Terms*—Predictive maintenance, Machine Learning, Microservice, Architecture**

I. Introduction

The industry is continuously developing and adopting technological improvements to adapt and improve its functioning, which has led to what is known as the fourth industrial revolution, *Industry 4.0* [1], [2]. One of the basic principles of Industry 4.0 is the reduction of human intervention in decision making processes by automating and optimising them using artificial intelligence [3], which is stimulated by the need to enhance the productivity, efficiency, and flexibility of production processes. However, equipment maintenance is a crucial factor that impacts long-term success of manufacturers, representing 15–60% of their total operational costs [4]. Unexpected downtime costs the industrial sector circa $50 billion a year in the US alone, with 42% of this being attributed to equipment failure [5].

Predictive maintenance is an effective approach that utilises data to prepare for equipment failure before it happens, which reduces downtime and increases profits. Such data are collected through monitoring the operational environment. For example, a smart factory is a cyber-physical system that contains a large number of networked devices and machines that are monitored by sensors. The monitoring data includes machine component states, *e.g.,* heat, pressure, motor rotation, circuit interruption, transmitter fault, etc. The collected data are leveraged to reason about issuing maintenance operations.

In this context, data are crucial in generating knowledge that is key in making automated predictive decisions. Identifying trends and behaviour patterns – using machine learning models – is needed in order to predict component failures. This enables proactive decision making, and avoids system failures and downtime, which, in turn, increases productivity and profit [6].

Predictive maintenance has attained considerable importance in the literature. There are existing approaches that incorporate several technologies such as Machine Learning, Big Data and the Internet of Things [3]. However, most of these attempts rely on collecting large amounts of data to perform batch learning (offline by experts) to predict failures. There is limited support for online learning capabilities that would incrementally provide insights about the running performance of system components, which enables dynamic adaptation to new patterns in data. More generally, we find a lack of proposals that address predictive maintenance in a holistic manner; *i.e.,* integrating mechanisms for data collection, analysis, and decision making in order to enable adaptive and timely predictive maintenance.

This work develops a Microservice architecture for pRedicTive Industry mainteNance (`MARTIN`) to support incremental learning in Industry 4.0. `MARTIN` facilitates smart factory characteristics such as real-time decision making and increased visibility, and provides predictive maintenance mechanisms. `MARTIN` uses the microservice architectural pattern to provide reliability and scalability, as is now the de facto standard practice in developing and deploying applications [7], [8]. Furthermore, the architecture supports the application of incremental machine learning for predictive maintenance tasks [9]. `MARTIN` can serve as a basis for a data-agnostic mechanism to support multiple smart factories within the same system. In summary, the main contributions of this paper are as follows:

- Design an architecture to facilitate data collection, data storage, real-time decision making, alert system, and predictive maintenance mechanisms.
- Evaluate the solution by executing use cases that involve multiple different time-series datasets.
- Evaluate the performance of the incremental machine learning method for predictive maintenance.

The rest of the paper is organised as follows. Section II discusses related works. Sections III and IV present the design and implementation, respectively, of the proposed architecture, `MARTIN`. Section V presents scenarios to demonstrate interaction with `MARTIN`. Sections VI and VII detail the setup and the results, respectively, of our performance evaluation. Section VIII provides a discussion of the main findings and the threats of validity. Section IX draws conclusions and outlines future work.

II. Related Work

Many publications studied different approaches to create an effective predictive maintenance mechanism. Sipos et al. [10] designed a system that uses logs generated by machines to create a model that is evaluated by data scientists then used to predict failures. Paolanti et al. [11] created a data analysis mechanism with a Random Forest approach trained using Azure Machine Learning Studio and reached 95% accuracy. Kaiser and Gebraeel [12] used a combination of component-specific real-time degradation signals together with historical data about reliability. These papers focused only on *predictive maintenance* aspects, omitting challenges related to the scalability of the predictive maintenance solution. It is hard to make the predictive maintenance mechanism optimal in real-life scenarios without a proper architecture to support it. Hasselbring et al. [13] developed a platform for integrating production environments with Industrial DevOps, and applied predictive maintenance as an application scenario of the platform. In contrast to the above contributions, this paper describes an end-to-end solution to facilitate scalable predictive maintenance.

Complex Event Processing (CEP) [14] is a technique that processes data from different streams to infer complex events in real-time. CEP usually works by having a set of predefined rules and matching them with the data to identify specific events. Etzion and Niblett [14] used an adaptive approach to predict future data and apply CEP rules. The implementation resulted in high accuracy, however, it is limited to only one data domain. In contrast, `MARTIN` supports various data domains simultaneously and along with customisable CEP rule sets. Akbar et al. [15] implement a generic architecture to combine predictive maintenance with CEP. However, they use offline learning which required a substantial amount of historical data for training. Eichler et al. [16] provide a generic architecture for event- and agent-based smart systems. The architecture is designed specifically to help with system comprehension and debugging. Other related contributions (*e.g.,* [17], [18]) offer purely theoretical models with no actual implementation of a prototype.

The work of Wang et al. [19] uses CEP together with dynamic Bayesian Networks, and applies the results to road traffic data. Christ et al. [20] tried to apply conditional density estimations to change CEP from reactive to predictive. The AutoCEP framework [21] uses historical data to predict patterns and transform them into CEP rules which are later used to configure the CEP engine. The work described in [22] connects CEP, predictive maintenance, and the microservices architecture in a single solution to predict potentially dangerous situations and act accordingly using offline learning. In contrast, this paper leverages incremental machine learning which enables adaptive real-time decision making. Furthermore, `MARTIN` allows users to alternate event processing rules dynamically, without the need to redeploy the architecture, reducing potential downtime.

III. Design

This section presents the design of the architecture. It first discusses the requirements that guided the design of the architecture then describes its components.

A. Requirements

`MARTIN` was designed based on the following set of requirements that are based on our analysis of the related works and discussion with different stakeholders as described in . The architecture should:

- accept, analyse, and store the generated data in real-time using an industry-standard communication protocol. Using popular protocols will make the architecture available and reduce compatibility issues.
- be able to process various data domains, ideally originating from multiple organisations. This would allow the system to be used by multiple companies without implementation adjustments.
- be capable of matching data with defined complex event rules and make decisions based on the actions defined in event processing rules, which is necessary for predictive maintenance tasks.
- be able to predict failures within a specified time-frame in order to allow users to schedule appropriate actions and avoid sudden failures.
- have means to alert system users about potential threats with a notification; such as email.
- allow users to define custom rules for event processing and custom schema of the gathered data.
- store raw sensor data, predictions, and decisions for analytical and historical purposes. Such data can be used to assess the performance of predictive maintenance and adjust the settings accordingly.
- be divided into loosely coupled microservices; each responsible for one functionality, allowing easier maintainability, and enabling scalability, flexibility and reusability.

B. Architecture components

`MARTIN` is based on the microservice architectural pattern, wherein the system is made of a set of independent microservices. This architectural style has multiple benefits over the monolith alternative, such as loosely coupled development and independent deployment, testing and scaling. In turn, this improves scalability and availability, and affords reuse of components, which fulfils our design requirements.

The boundary between services is set appropriately to allow services to be operationally independent and keep related functionalities within a single component [23]. This approach increases scalability and the ability to maintain each service independently. Furthermore, the designed architecture aims to minimise the number of direct links between services, which should further reduce the coupling between services and increase performance and availability [23]. Figure 1 depicts

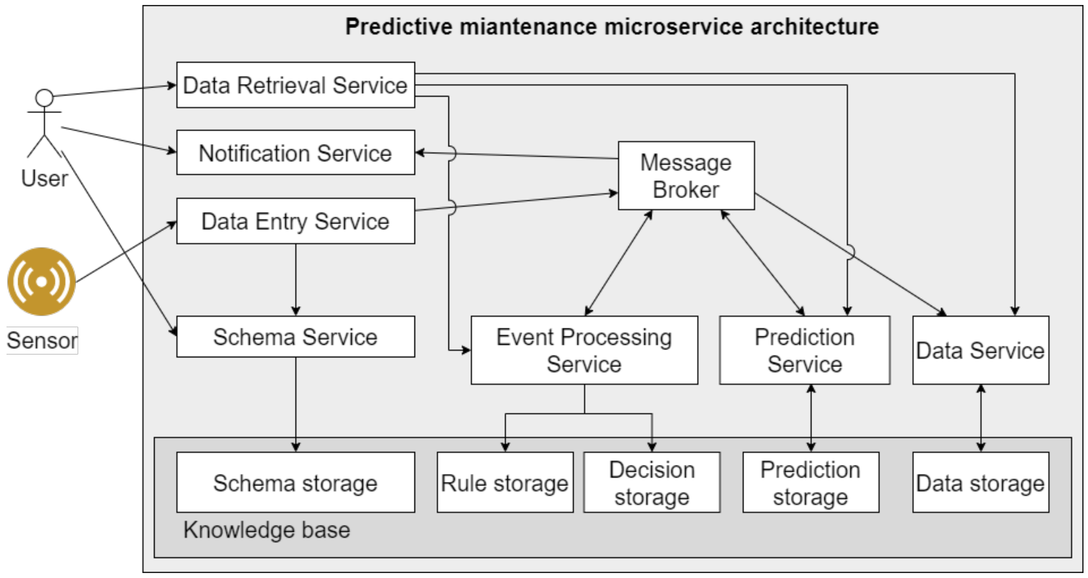


Figure 1: An overview of `MARTIN`, the predictive maintenance architecture.

the `MARTIN` architecture, which is composed of the following components:

1) Data Entry Service: This is an entry point of the architecture that accepts data from IoT sensors. This service supports REST as a popular and language-independent means of data exchange. Once a data point arrives, the service fetches the appropriate schema from the *Schema* service (explained below) and validates the entry. If the received data match the schema, the service forwards the entry to the message broker, which sends it to other services that use the information for their tasks. If the data do not match the defined schema, the service does not push the data further into the system, and is rejected.

2) Schema Service: This service is responsible for managing various data schemes that are supported by the system. It allows the user to define the data structure that the system accepts. Furthermore, the user can mark the required fields, which will lead to data rejection if one of these fields is missing. Apart from schema management, the service provides saved schemes if requested, and in the proposed architecture, the *Data Entry* service asks for various schemes when IoT data arrive.

3) Data Service: This service listens for the raw data points published by the message broker and stores them in the database. The raw data can be later retrieved from the database using the service and used for various tasks such as offline analytics or the training process of the prediction model if the architecture will be extended to support traditional machine learning. In `MARTIN`, *Data* communicates directly with the *Data Retrieval* service (explained below) and provides the raw data used to generate system reports.

4) Prediction Service: This service is responsible for two major tasks: incremental training of models and prediction of the possible failure of the asset based on incoming data points. The service listens to the data, fulfils the prediction tasks, and publishes the result to the message broker. In addition, each prediction is stored in the database. Furthermore, the service supports the learning process and stores trained models in a separate database. Although it appears that the service is broad and can be potentially split into two independent services – one for training and one for predictions – it was decided to keep those two functionalities within a single service. With two different services the system would have to maintain duplicated models in multiple databases, which could cause consistency issues.

5) Event Processing Service: This service handles event processing with a CEP engine that runs with a customizable set of event rules. This service allows users to create and apply new conditions to be matched against incoming data without redeploying the system. Furthermore, the component listens for the data published by the *Data Entry* service and the predictions from the *Prediction* service. The architecture utilises CEP to detect valuable behaviours that the user wants to catch. The acquired information is analysed and compared with the specified rules. If any of the rules are matched, the system makes a decision by looking at the action defined in it and creates the action object that is later published to the message broker. The service can also translate user-defined rules from an initial JSON structure to syntax understood by the CEP engine. All decisions made by the system and the rules defined by the user are stored in a dedicated database.

6) Notification Service: This service listens for actions that involve any type of user notification, reads the information, and notifies users of the notification topic and details. Currently, the service is designed to notify users via emails. This kind of notification represents an example of how actions can be consumed by the system. However, this service can be extended to perform other actions such as shutting down faulty assets, redirecting the manufacturing process somewhere else, or scheduling maintenance tasks that involve humans.

7) Data Retrieval Service: This service uses data stored by other services (such as *Prediction*, *Event Processing* and *Data* services) to generate reports about the system.

8) Message broker: The message broker implements asynchronous interactions between architectural services. It works as a publish/subscribe system with different services subscribing to and pushing messages to various topics. This component also increases the consistency and reliability of the architecture. The broker stores the generated messages until the subscriber can consume them. Asynchronous communication enables the independent evolution of the architecture, for example, when adding and removing services. In order to get access to the data streamed to the system, the services need to subscribe to a specific topic, and all new data entries will be pushed to it.

9) Knowledge base: This is a logical collection of databases. It encapsulates all the databases used within the architecture to store the historical data and information required by specific services. Each service has its own independent database, making the architecture less coupled. The databases are grouped in the diagram to increase clarity.

As shown in Figure 1, most of the components within the architecture communicate using an asynchronous message broker, except for the communication between the *Data Entry* and *Schema* services. Synchronous communication between these services is crucial for increasing reliability and providing better user experience. For example, synchronous communication makes it easier to indicate the status of a request to store data. If the request is rejected (which may indicate a fault in a component), the user must be promptly notified, thus the need for synchronicity. In the diagram, synchronous operations are depicted with arrows pointing directly from one service to another, and not through the Message broker.

C. Coordinating multiple organisations

One of the requirements of the proposed architecture is that it should support multiple data domains from various organisations; therefore, design decisions are taken to allow for such functionality. The system can be accessed by multiple companies, each of which is uniquely identified by *organisationId*. All requests accepted by the architecture must have *organisationId* to allow access to the appropriate resources within the system. Each service uses an identifier while handling the requests, and it is required to locate the information needed to fulfil the request. Furthermore, the system can dynamically adjust to different data structures owing to the user-defined schemes. Each schema can be identified using a *schemaId* unique to the organisation. Using the schema, users can decide which fields from the sensor readings should be expected and used by the system. The final parameter used by the architecture is *deviceId*, which uniquely identifies the sensor or machine within the system. These three values are used to support predictive maintenance for multiple companies or factories within a single system without the need to adjust the solution for each individual customer.

All sensor readings arriving into the architecture should contain `organisationId`, `schemaId`, and `deviceId`. The *Data Entry* service uses the `organisationId-schemaId` pair to execute the interface exposed by the *Schema* service and fetch the schema that the submitted reading should match. Furthermore, the *Schema* service uses the values to manage the database structure in which the schemes are stored. The *Event Processing* service utilises `organisationId` and `deviceId` to manage the storage and execution of event-matching rules. The rules can be defined within the organisation on two different levels: general conditions for the devices having the same type (using the device type information encoded in the `deviceId`) or conditions for the specific device. The *Data* service structures the raw data storage by dividing the database using `organisationId`, and the *Prediction* service utilises it together with `deviceId` to separate and locate prediction models responsible for different types of devices.

D. Prediction mechanism design

One of the main features of the architecture is the ability to analyse data and predict whether there is a possibility of failure occurrence in the factory. To achieve this, the prediction mechanism is designed considering things such as the presence of data from multiple companies or different types of machines exposing various behaviours that may indicate possible faults. The *Prediction* service uses real-time data to predict whether an accident will happen within the window of n following machine cycles. It is assumed that the physical assets connected to the system operate in equally spaced cycles over time. A cycle is the point in time when a machine submits the values of the sensors to the architecture for analysis. The cycle duration is used to determine the time frame during which the failure may occur and accordingly schedule maintenance.

The *Prediction* service is designed to support incremental machine learning techniques for predictive maintenance, as it enables adaptation to varying patterns of data that continuously arrive at runtime. The service maintains an expandable set of machine learning models that are organised so that there is one model responsible for predictions related to one type of device within the organisation. Such a structure supports the fact that different types of devices submit various data, and may have unique trends that indicate faults. Incremental machine learning has the ability to perform an ongoing learning process without the need to have the dataset upfront and train the model before deployment. Therefore, the service supports a constant learning process by consuming the training data once it is submitted to the system and adjusting the appropriate model. All models are serialised and stored in the database, and the service fetches the required one to effectively save memory resources. IoT sensor data are often unlabelled, which makes the process of gathering training data costly and time-consuming. Thanks to incremental learning, the model can be deployed with minimal training and further trained when new data are available without starting the learning process from scratch. This helps save resources and omit the requirement to load significant amounts of data into the service memory which can result in reduced availability.

Preprocessing is another crucial aspect of the machine learning process. Raw data often contain many details that do not contribute to the final predictions, or the range of the specific values is broader than that of other measurements. MARTIN supports simple data preprocessing, where every prediction model is trained together with a simple scalar that transforms the data for mean and unit variance equal to zero. Although it helps to increase the accuracy, it is known that preprocessing techniques must be adjusted according to the nature of the processed data, and it is difficult to generalise the methods used. Apart from consuming the training and actual, real-time data from the system, the *Prediction* service also generates the prediction objects that are to be used by other parts of the architecture. The trained models can classify submitted readings into two classes: NORMAL which indicates that the machine is functioning correctly, and ALARM which signalises possible faults within the following ten machine cycles. Each organisation has a general rule for ALARM prediction defined in the rule set loaded into the *Event Processing* service. Once the prediction object is published to the topic within the message broker, the *Event Processing* service attempts to match the defined condition against it and executes the action related to the rule (*e.g.,* sending an email to the specified user). When the prediction object is consumed, the entire process of the predictive mechanism is complete. Furthermore, the generated predictions are stored in an appropriate database and can be queried using the *Data Retrieval* service for analysis or statistics.

IV. Implementation

This section describes the implementation of the MARTIN architecture.

A. Technology stack

The services of the architecture are implemented using Kotlin, a statically typed programming language designed to be concise, expressive, and safe. MongoDB, a document-oriented NoSQL database, was selected to be used in the architecture. The use of a NoSQL database that does not require any constraints makes it simple to store unstructured data and to operate with multiple data domains.

B. Implementation details

1) Data Entry Service: This Service encompasses three elements: *SensorDataController*, *TrainDataController*, and *SchemaManager*. *SchemaManager* encapsulates a reactive WebClient used to make a GET request to the *Schema* service and fetch the appropriate schema. *TrainDataController* exposes the REST endpoint used to stream training data to the system, and *SensorDataController* handles a stream of real-time data submitted by machines for the analysis. The service exposes two REST endpoints accessible on the following routes: POST *organisationId*/readings - for real-time data, and POST *organisationId*/train - for the training data.

```
{
    "_id":{
        "\$oid":"62684a4e74fa892fbff4b2ec"
        // MongoDB id
    },
    "\$id":"a4944a5c-2a5f-4608-9674-f77a7cc8f2d6",
    // schemaId used by the system

    "title":"Example schema",
    "description":"Example description",
    "properties":{
        "valueOne":{
            "type":"float"
        },
        "valueTwo":{
            "type":"float"
        }
    },
    "required":[ "valueOne" ]
}
```

Listing 1: An example schema

2) Schema Service: This service contains two main classes: *SchemaController* which handles the exposed REST endpoints, and *SchemaRepository* - the component responsible for connection with the MongoDB instance. The schemes are fetched from the database using *SchemaRepository*. The database is organised such that each organisation has its own collection with *organisationId* as its name, and every schema is a new document inside this collection. Listing 1 shows an example schema stored in the database.

3) Data Service: This service consumes the data published by the message broker. It has three main elements: *DataController*, *SensorReadingListener*, and *DataRepository*. *DataController* allows external entities to access data stored in the database, *DataRepository* handles the communication with MongoDB, and *SensorReadingListener* consumes the data points published in the message broker.

SensorReadingListener intercepts messages published to the readings topic and calls *DataRepository* to store them in the database. Later, the endpoint managed by the *DataController* can be called to get the raw data stored for specified *organisationId*, *schemaId*, and *deviceId*. The database structure looks as follows: there is a separate document collection for each combination of *organisationId* and *schemaId* values, and every document in such collection contains the *deviceId* field. The presented implementation is simple; however, it can be extended with functionalities such as filtering by specific value or timeframe.

4) Event Processing Service: This service contains a set of crucial components. *RulesController* manages the endpoints responsible for adding and fetching user-defined conditions, *ReadingListener* that consumes the real-time data, *DroolsRuleTranslator* responsible for translating conditions from JSON to format understandable by the CEP engine. It contains also two repositories: *DecisionRepository* that manages the connection with decisions storage, and *RuleRepository* which handles communication with the rule storage.

The service uses Drools, an open-source rule engine that supports conditions written using Drools Rule Language

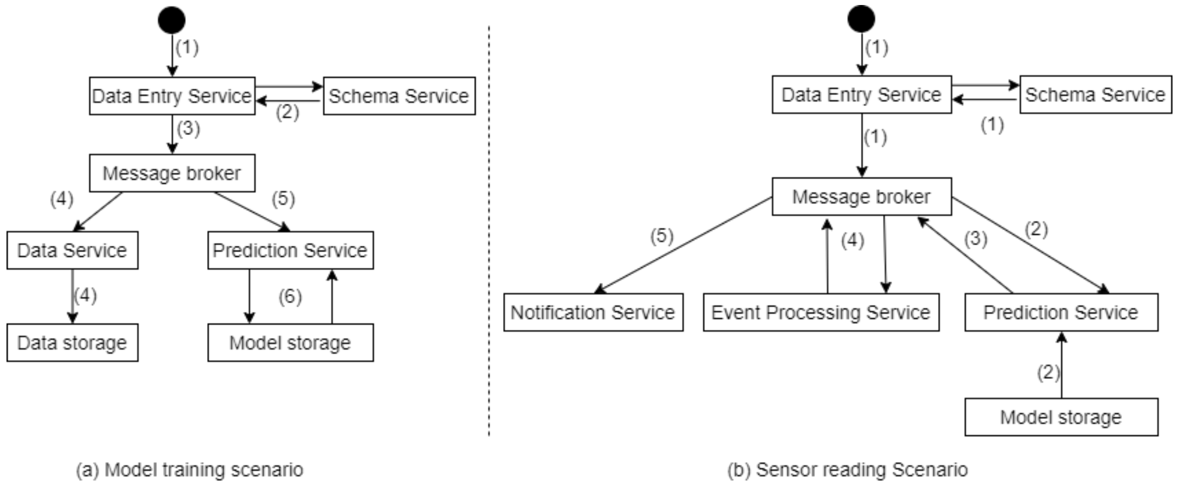


Figure 2: Predictive model training flow

(DRL). All user-defined rules are retrieved and submitted in JSON format; therefore, they need to be translated into the DRL. *DroolsRuleTranslator* can transform simple JSON rules into the **.drl* file loaded later to the engine. Rules are organised so that each organisation has a single *.drl* file with all conditions defined. Additionally, there is a default rule to match the prediction objects published by the Prediction Service. The service fires all loaded files over the incoming data, and if any is matched, the appropriate decision object is produced. It is later published to the decisions topic and stored in the decision storage for future usage.

5) Prediction Service: This is the most crucial component of the system, implemented with Python. There are two Kafka consumers: the first one consumes the training data, and the second consumes the real-time data and makes predictions. Furthermore, there is a producer publishing the predictions to the message broker. The current implementation uses four machine learning models, namely, Logistic Regression (LR), k-Nearest Neighbours Classifier (kNN), Passive-Aggressive Classifier (PA), and Hoeffding Adaptive Tree Classifier (HAT), which are implemented using the API provided by the River ML package. The models provide `learn_one(x, y)` and `predict_one(x)` APIs. The first one updates the model with a single set of features and the target, and the second one is used to make the prediction. When training data arrives, the service loads a serialised model from the database, uses the data point to train it and saves the updated model in the database. In case of the real-time data arriving in the system, the required model is loaded from the database and used to predict whether the failure is possible. The model uses two labels for the prediction - NORMAL and ALARM. If the predicted label is equal to ALARM, the service publishes the result with *organisationId* and *deviceId* to the predictions topic.

6) Notification Service: This service has only one main component - *DecisionListener*, a Kafka consumer that subscribes to the decisions topic. It maintains a fixed thread pool used to send emails if the decision contains the *send_email* action.

V. DEMONSTRATION

This section shows scenarios of the system processes that can be initiated by a client.

A. Predictive model training

A popular scenario that would be executed by the architecture is the learning process of the incremental machine learning model. Figure 2(a) shows the flow that is executed upon the arrival of a data point as a training sample. The process consists of the following steps:

1) Training data is streamed to the *Data Entry* Service.
2) *Data Entry* Service calls *Schema* Service for the appropriate schema.
3) The request is validated and published to the message broker.
4) *Data* Service fetches the data and stores them in the database.
5) *Prediction* Service pulls the message for the learning purposes.
6) Appropriate model is fetched, updated with the data and stored back in the model storage.

B. Sensor readings analysis

Another important scenario is initiated by the machines connected to the architecture, and involves the execution of the previously trained model. Figure 2(b) depicts the flow of actions from the submission of readings to the generation of an alarm and decision making by the system. The steps taken are as follows:

1) Data are streamed to the *Data Entry* Service, validated and published to the message broker similarly to the previous scenario.
2) *Prediction* Service pulls the message, fetches the appropriate model and generates the prediction.

3) For alarm predictions, an alarm message is generated and published to the message broker.
4) *Event Processing* Service fetches raw data and alarm messages, attempts to match them against the event rules loaded to the CEP engine, and generates the appropriate decision to the message broker.
5) *Notification* Service intercepts the decision and executes the action encapsulated inside.

VI. Evaluation

Our evaluation aimed to examine the feasibility of incremental machine learning for predictive maintenance and performance under different workload patterns.

A. Experimental setup

The architecture implementation was containerised and deployed using Docker, with each component packaged as a separate container. The experiments were conducted on a PC with an Intel Core i7-10750H CPU and 16GB RAM. Although all components are technically running on the same machine, they cannot access each other directly. They communicate using the internal network created within the Docker Engine instance through their respective RESTful interfaces. Note that deployment of `MARTIN` in a distributed fashion is a separate topic that we do not address here.

1) Datasets: Two different datasets describing machine operations were used in the experiment: the *turbofan engine degradation simulation dataset* (referred to as the NASA dataset) [24] and the *predictive maintenance modelling experiment* dataset (referred to as the Microsoft dataset) [25]. Both datasets contain time series data as required for the functionality of the architecture. The NASA dataset contains data that represent simulations of jet engines run-to-failure scenarios. Each entry contains a set of sensor measurements and is labelled as ALARM or NORMAL. Similarly, the Microsoft dataset contains maintenance measurements that include volt, pressure, and vibration, among others, as well as a metric that indicates the number of cycles before failure. Similar to the previous dataset, entries were labelled NORMAL or ALARM. Further details regarding the datasets can be found in the cited references.

2) Procedure: The experiment was divided into two parts: (1) measuring the performance of the incremental learning applied within the architecture, and (2) assessing the system performance under different workloads. Initially, the system did not store historical data or trained models. The only information present in the system is the appropriate schema to match the dataset and a set of basic event rules populated before the start of the experiment within the *Event Processing* Service.

To measure the machine learning metrics, the training and test parts were created for both datasets. Furthermore, the training dataset was divided into 10 smaller sets used for the learning process. Each small set was streamed to the architecture to train the data, and between the training sets architecture is evaluated with the test set, and multiple machine learning metrics were recorded. This procedure simulates a scenario in which the entire training dataset is not available at the start of the learning process, but becomes accessible incrementally over time. Measuring metrics after training with each part of the set helps monitor changes in the model's performance and the way of responding to the data. The experiment was repeated twice, each time with a different dataset, to assess the ability of adaptation to various data domains.

In the second part of the experiment, the architecture was tested against the workload. The data are constantly streamed to the system for 60 seconds and services record information related to the processed data points. The experiment was repeated five times, with 10, 100, and 1000 concurrent users submitting sensor readings to the system. In order to simulate different numbers of concurrent clients, Apache JMeter was used, which is an open-source solution for testing web services and measuring their performance.

B. Performance criteria

Throughout the experiment, a set of measures was collected and later processed to determine the feasibility of incremental machine learning for predictive maintenance and to assess behaviour under various workloads.

1) Machine learning metrics: The *Prediction* service was administered to gather the machine learning metrics during the first part of the experiment by storing them in a text file at the end of each evaluation with the test set. We report on the classification accuracy, which can be defined as a fraction of right predictions within the total number of predictions.

2) Performance measures: In order to gather the performance measures, a logging system was added to all services that were monitored during the evaluation. Logs are stored in a text file and contain important details that can be later used to process and draw conclusions. Each incoming request to the architecture is marked with a unique *requestId*, and services note it together with the time when the request enters the service and leaves. Because of this identifier, it is possible to track the request through the architecture and calculate the desired performance metrics. Performance was evaluated using the following metrics: *average time* required to process a single request, *number of requests processed per second*, *total number of requests processed*, and *total time required to finish processing*.

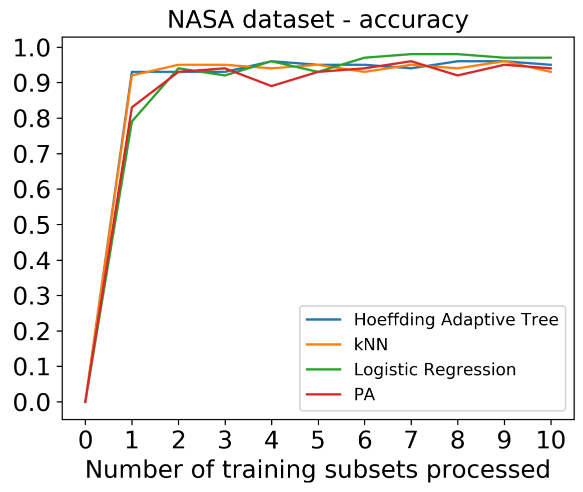
VII. Results

This section presents the results of the experiments.

A. Incremental machine learning

As previously explained, the implemented architecture is supposed to predict whether there will be a failure within the next ten machine cycles. Four different models were trained for two datasets to fulfil the task, and the architecture was able to produce models with the performance scores presented below.

Figure 3 shows the change in accuracy during training with the NASA (Figure 3a) and Microsoft (Figure 3b) datasets for



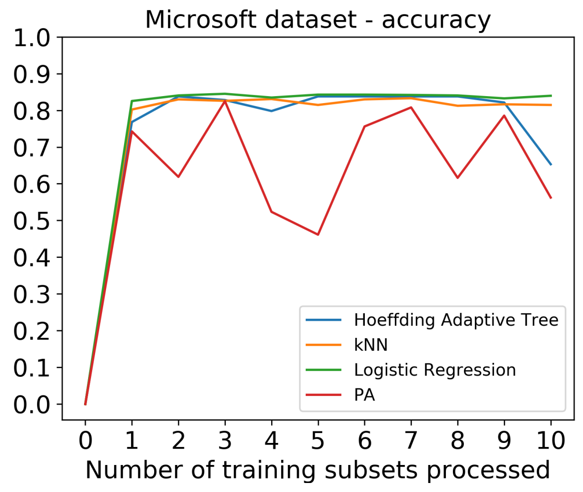


Figure 3: Accuracy measured during training dataset

the four classification algorithms. In the case of the NASA dataset, we observed that all models had high accuracy (≈ 0.9) after processing only one-tenth of the data. The results are somewhat different with the Microsoft dataset: model accuracy attains a relatively high value (> 0.8) after processing one-tenth of the data and is stabilised during the learning process, with only minor changes for the three algorithms; however, the PA algorithm struggles to achieve high accuracy and varies between 0.4 and 0.8 throughout the training.

B. *Scalability evaluation*

This section evaluates the scalability of the architecture with various workloads. The experiment stressed the system with different numbers of users submitting requests and measured the rate of processing requests and the average processing time.

Figure 4 shows the number of processed requests for different numbers of users. During 60 seconds of streaming data to the system, the system was able to process 67397, 82106, and 80663 requests when the number of users was 10, 100, and 1000, respectively. Furthermore, looking at the requests processed per second, one can see that the service starts to ramp up and, after about 30 seconds gets stabilised with the ability to process approximately 1500 requests per second.

With respect to the processing time, Figure 5 plots the request processing times for the three cases of the number of users. For the 10-users case, the architecture required an average of 25.08ms to process a single request, with a minimum of 8ms and maximum of 1466ms. The measured processing times changed when 100 concurrent users were working. Although there are 10 times more users, the architecture can process only 82,106 requests with an average of 98.73ms, a minimum of 22ms, and a maximum of 1891ms per request (roughly a 4-fold increase). Even though most of the requests are processed within 100ms, there is a growing number of those that take between 200 and 250ms as seen in the figure. In the case of 1000 concurrent users, the results were significantly different: the system needs 710ms on average to process a single request (minimum 255ms, maximum 4,079ms). The histogram shows that only a small number of requests are processed quicker than 500ms, and most require between 500 and 600ms. Furthermore, there are a significant number of calls that take 1 second or longer.

VIII. Discussion

We now comment on the observations from our evaluation.

A. *Incremental machine learning*

Although the models trained with both datasets scored relatively high accuracy, they exhibited different behaviours. The models trained with the NASA dataset performed better than those trained using the Microsoft dataset. For the NASA dataset, the performances of the four classification algorithms were somewhat similar. It can also be observed that the PA and Logistic Regression models had the best fit for the NASA dataset. With regard to the Microsoft dataset, it can be observed that the models have a lower accuracy than the NASA dataset. In addition, the PA algorithm struggles to stabilise its accuracy which may lead to unexpected performance drops in the future, making it an inappropriate algorithm for this dataset.

There may be several reasons for the differences between the two datasets. First, most of the Microsoft dataset is labelled as regular readings with only a little part being the ALARM data. Second, the obtained results show that pre-processing and classification algorithms must be carefully selected to match the specific data. Both models use the same pipeline with incremental standard scalar and various classification algorithms. Although some algorithms fit the NASA dataset, this may not be the case for the Microsoft dataset.

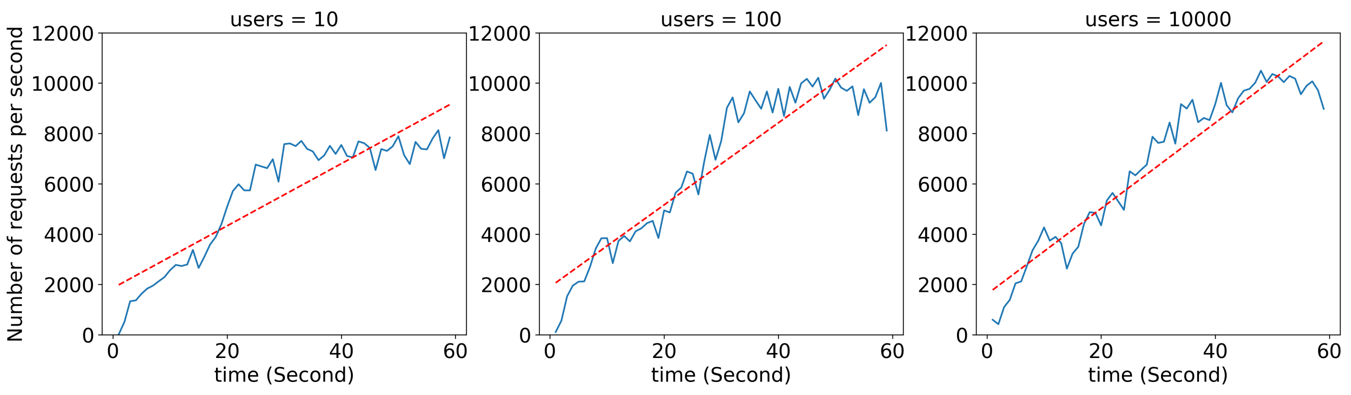


Figure 4: Number of requests processed per second

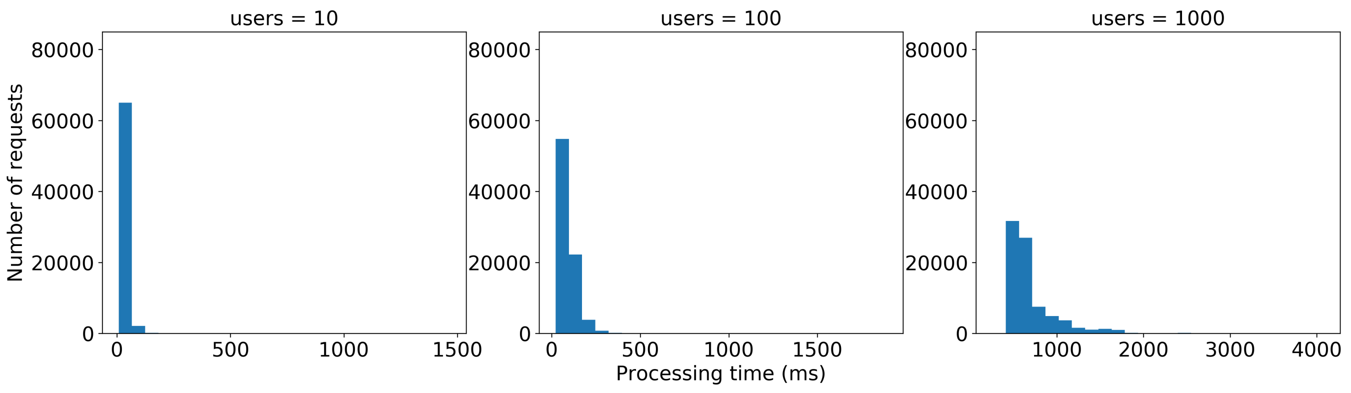


Figure 5: Histogram of number of requests vs processing time

B. *System performance*

Intuitively, one can observe that the system performance in the cases of 10 and 100 users is lower than that of 1000 users. One of the reasons for these results is the lack of horizontal scaling of the implemented system. With multiple instances of the *Data Entry* service running and appropriate load balancing, the overall performance can be increased.

The other value that must be interpreted is the maximum time required to process the request. In the scenario with 1000 users, the reason behind this can be the limited capabilities of the *Data Entry* service; however, with 10 and 100 users, the value is multiple times higher than the average. Even though it appears to be a performance issue, most of it is the time required to complete the TCP handshake between the *Data Entry* service and the *Schema* service, as those two communicate synchronously using the REST protocol over HTTP. After the initial connection is made, it is reused later and the exchange time between *Data Entry* service and *Schema* service is significantly lower. Owing to the stateless design and microservice pattern, the implementation can be deployed using tools such as Kubernetes to enable horizontal scaling.

C. *Generalizability*

The use of microservices for designing `MARTIN` allows for scalability and flexibility, making it adaptable to a wide range of environments and systems. Additionally, the use of incremental learning can enable the architecture to continuously improve its predictions over time, thereby increasing its accuracy and applicability to new scenarios. However, factors such as the quality and completeness of the data fed into the incremental predictive model and the selection of appropriate features can affect the generalisability of the architecture. These design considerations require appropriate attention.

D. *Threats to validity*

Although the evaluation process provided meaningful results, some aspects could be changed to improve the gathered data and provide more information about the solution. First, the experiment was performed using a local machine without any calls over the Internet; therefore, the obtained results do not include the latency and failures introduced by network operations. To extend the evaluation, the implemented architecture can be deployed in an actual cloud environment, where calls are issued over the Internet. Second, the incremental learning evaluation provided results sufficient to assess whether this technique is feasible for predictive maintenance mechanisms. The experiment showed that the designed architecture can be used as a maintenance tool; however, it requires further evaluation in terms of the reliability of predictions and learning pipelines implemented within the system.

IX. Conclusion

We propose an end-to-end architecture for predictive maintenance that supports online incremental learning, which enables timely proactive decision making in the context of Industry 4.0 applications. This architecture facilitates real-time predictions using incoming data to inform users about potential equipment failures. The architecture operates as a set of microservices that provide scalability and reliability for various and varying deployment sizes. Using two real-world datasets, we demonstrated the ability of the architecture to provide high accuracy at an acceptable processing overhead. In future work, we plan to expand the evaluation by applying the architecture to an operating real-world case study for a broader evaluation of the solution.

Acknowledgment

This work was supported in part by the UK EPSRC under grant number EP/R010889/2.

References

[1] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & Information Systems Engineering*, vol. 6, no. 4, pp. 239–242, Jun. 2014.

[2] T. Kalsoom, S. Ahmed, P. M. Rafi-ul Shan, M. Azmat, P. Akhtar, Z. Pervez, M. A. Imran, and M. Ur-Rehman, "Impact of IoT on manufacturing Industry 4.0: A new triangular systematic review," *Sustainability*, vol. 13, no. 22, 2021.

[3] T. Zonta, C. A. da Costa, R. da Rosa Righi, M. J. de Lima, E. S. da Trindade, and G. P. Li, "Predictive maintenance in the Industry 4.0: A systematic literature review," *Computers & Industrial Engineering*, vol. 150, p. 106889, Dec. 2020.

[4] M. Haarman, M. Mulders, and C. Vassiliadis, "Predictive maintenance 4.0: predict the unpredictable," *PwC and Mainnovation*, vol. 4, 2017.

[5] IndustryWeek and Emerson, "How manufacturers can achieve top quartile performance," https://partners.wsj.com/emerson/unlocking-performance/how-manufacturers-can-achieve-top-quartile-performance/, 2016.

[6] E. Sezer, D. Romero, F. Guedea, M. Macchi, and C. Emmanouilidis, "An Industry 4.0-enabled low cost predictive maintenance approach for SMEs," in *International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. IEEE, 2018.

[7] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, 2015.

[8] "DevOps Setups – a benchmarking study," https://humanitec.com/whitepapers/2021-devops-setups-benchmarking-report, Humantiec, Tech. Rep., 2021.

[9] X. Geng and K. Smith-Miles, *Incremental Learning*. Boston, MA: Springer US, 2009, pp. 731–735.

[10] R. Sipos, D. Fradkin, F. Moerchen, and Z. Wang, "Log-based predictive maintenance," in *SIGKDD International Conference on Knowledge Discovery and Data mining (KDD)*. ACM, Aug. 2014.

[11] M. Paolanti, L. Romeo, A. Felicetti, A. Mancini, E. Frontoni, and J. Loncarski, "Machine learning approach for predictive maintenance in Industry 4.0," in *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, 2018.

[12] K. Kaiser and N. Gebraeel, "Predictive Maintenance Management Using Sensor-Based Degradation Models," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 39, no. 4, pp. 840–849, Jul. 2009.

[13] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak, "Industrial DevOps," in *International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2019, pp. 123–126.

[14] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications, 2011.

[15] A. Akbar, A. Khan, F. Carrez, and K. Moessner, "Predictive Analytics for Complex IoT Data Streams," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1571–1582, Oct. 2017.

[16] T. Eichler, S. Draheim, K. von Luck, C. Grecos, and Q. Wang, "Integration of complex event processing into multi-agent systems: Two use cases for distributed software development support," in *International Tools and Methods of Competitive Engineering Symposium (TMCE)*, 2020.

[17] J. Krumeich, D. Werth, and P. Loos, "Prescriptive Control of Business Processes," *Business & Information Systems Engineering*, vol. 58, no. 4, pp. 261–280, Dec. 2015.

[18] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, and M. Mock, "Issues in complex event processing: Status and prospects in the Big Data era," *Journal of Systems and Software*, vol. 127, pp. 217–236, May 2017.

[19] Y. Wang, H. Gao, and G. Chen, "Predictive complex event processing based on evolving Bayesian networks," *Pattern Recognition Letters*, vol. 105, pp. 207–216, Apr. 2018.

[20] M. Christ, J. Krumeich, and A. W. Kempa-Liehr, "Integrating predictive analytics into complex event processing by using conditional density estimations," in *International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE, 2016.

[21] R. Mousheimish, Y. Taher, and K. Zeitouni, *autoCEP: Automatic Learning of Predictive Rules for Complex Event Processing*. Cham: Springer International Publishing, 2016, pp. 586–593.

[22] G. Ortiz, J. A. Caravaca, A. Garcia-de Prado, F. Chavez de la O, and J. Boubeta-Puig, "Real-Time Context-Aware Microservice Architecture for Predictive Analytics and Smart Decision-Making," *IEEE Access*, vol. 7, pp. 183 177–183 194, 2019.

[23] K. S. Kiangala and Z. Wang, "Initiating predictive maintenance for a conveyor motor in a bottling plant using Industry 4.0 concepts," *The International Journal of Advanced Manufacturing Technology*, vol. 97, no. 9-12, pp. 3251–3271, May 2018.

[24] A. Saxena, K. Goebel, D. Simon, and N. Eklund, "Damage propagation modeling for aircraft engine run-to-failure simulation," in *International Conference on Prognostics and Health Management*. IEEE, Oct. 2008.

[25] F. B. Uz, "Predictive maintenance modelling guide experiment," [Online; accessed 2022-05-25]. [Online]. Available: https://gallery.azure.ai/Experiment/Predictive-Maintenance-Modelling-Guide-Experiment-1