Predictive Maintenance System for Industrial Equipment

Damiano Orlandi*, Clelia Porcelluzzi†, Martin Reinhard ‡
damiano.orlandi@studenti.unitn.it
clelia.porcelluzzi@studenti.unitn.it
martin.reinhard@studenti.unitn.it

The code can be found here: GitHub

***Abstract*—This project presents a Predictive Maintenance System designed to monitor, analyze, and predict the maintenance needs of industrial machinery. By leveraging IoT sensors for data collection, a NoSQL database for data storage, and a web-based dashboard for visualization, the system offers an efficient solution for preventing equipment failures. The final implementation demonstrates the system's ability to provide timely probabilities of failure, crucial for optimizing machine uptime and reducing operational costs.**

***Index Terms*—Predictive Maintenance, Industrial IoT, Machine Learning, Real-Time Monitoring**

I. Introduction

Industrial machinery is prone to wear and tear, leading to unexpected downtimes that can result in significant financial and productive losses. Predictive maintenance (PdM) aims to address this by forecasting maintenance needs before failures occur. This project aims to design and implement a big data system capable of real-time monitoring and prediction of equipment health by analyzing sensor data and historical maintenance records.

II. System Model

A. System Architecture

The Predictive Maintenance System is engineered for scalability, reliability, and efficiency in handling data for maintenance predictions. The architecture consists of several modular components, each fulfilling a distinct role in the data pipeline, integrated to work seamlessly together. The core components include the User Interface (UI), Data Consumer, Data Processor, and Predictive Model Integration , all of which communicate efficiently through an MQTT broker, facilitating smooth data flow and coordination between them.

The **User Interface (UI)**, developed with Flask, provides a web-based platform for interacting with the system. Users can start and stop the real-time data stream, simulating maintenance interventions by technicians on the machines. This action triggers the data processing pipeline by publishing the selected machine ID to the "machines/machine_ID" MQTT topic. The UI then listens for processed predictions on the "predictions/machine_ID" MQTT topic, displaying essential information such as the machine's health status and failure probabilities, ensuring a seamless and user-friendly experience.

Upon a user's selection, the **Data Consumer** is activated. This component, implemented as a continuously running process, collects real-time sensor data from IoT devices attached to the machines. The sensor data is created synthetically due to the lack of sufficient available resources and for versatility, allowing the project to be adapted to any type of machine with different sensors, each having its own unique characteristics. It subscribes to the "machines/machine_ID" MQTT topic and, once a machine ID is received, retrieves the corresponding sensor data and publishes it to the same MQTT topic.

The **Data Processor** collects data from MongoDB to retrain the predictive model. After retraining, it saves the updated model to a commonly accessible Docker volume. The processor then waits for a specified interval before retraining the model again with new data that has arrived in MongoDB. This cycle ensures that the model continuously improves and adapts, maintaining its effectiveness in making predictions.

The **Predictor** is another essential component of the system. The RandomForestClassifier continuously evaluates incoming sensor data in real-time, determining the likelihood of machine failures based on historical patterns. This approach streamlines the anomaly detection process, embedding it within the broader predictive framework.

At the heart of the system is the **MQTT Broker**, configured using Mosquitto. It acts as the central communication hub, efficiently managing message passing between the User Interface, Data Consumer, Data Processor, and Data Predictor. The MQTT broker ensures reliable and real-time data transmission across the entire system pipeline.

B. Technologies

The technology stack for this project was carefully selected to meet the requirements for real-time data processing, efficient data storage, and an intuitive user interface.

*1) **MQTT**:* was selected for its lightweight and efficient messaging protocol, which excels in environments requiring real-time data transmission and minimal bandwidth usage. It is designed for fast and reliable communication between system components, particularly in IoT and distributed systems like this. MQTT's publish-subscribe architecture ensures that data can be transmitted asynchronously, making it well-suited for handling the diverse and time-sensitive communication needs

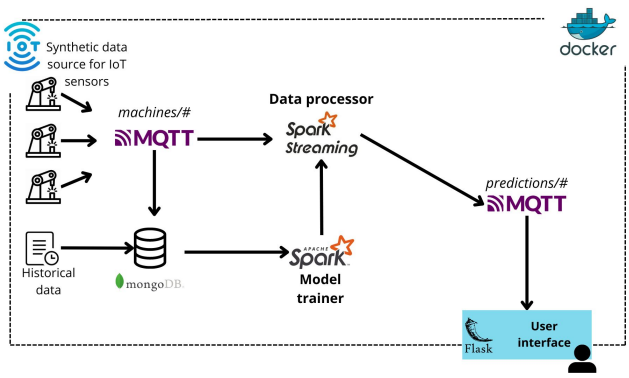

Fig. 1. System Architecture

of the system. In the Predictive Maintenance System, MQTT acts as the backbone for inter-component communication. The **User Interface (UI)** triggers the data pipeline by publishing the selected machine ID to a specific MQTT topic ("machines/machine_ID"). The **Data Consumer** subscribes to this topic and retrieves real-time sensor data from the machines, which is then stored in MongoDB. This raw data is then picked up by the **Data Processor**, which connects to MongoDB, collects and trains the model with the available data, and then saves the model.

*2) **MongoDB**:* MongoDB was chosen as the database solution for its flexibility and scalability, particularly when working with the diverse and complex data types associated with predictive maintenance. MongoDB's schema architecture allows for the storage of data in a JSON-like document format, which is well-suited for handling the various sensor readings, machine statuses, and historical performance metrics generated by the system. It serves as the primary data repository, because once the **Data Processor** completes its task of training and saving the model in a shared volume, the new raw data is stored in MongoDB. This includes not only the current machine health status and maintenance predictions but also historical data used for trend analysis and model training. MongoDB's ability to scale horizontally allows the system to handle large volumes of data, ensuring efficient storage and quick retrieval for real-time analysis and long-term forecasting.

*3) **Flask**:* Flask was selected to power the User Interface (UI) due to its simplicity, flexibility, and ease of integration with Python-based components. As a lightweight web framework, Flask enables rapid development while providing enough flexibility to build complex, data-driven applications. It integrates well with Python's ecosystem of libraries and tools, making it ideal for creating a dynamic and responsive UI. Flask drives the web-based UI, allowing users to interact with the system seamlessly.

*4) **Apache Spark**:* Apache Spark was chosen for its powerful data processing and real-time analytics capabilities, which are essential for managing the large volumes of data and complex machine learning tasks involved in predictive maintenance. Spark's ability to process both batch and streaming data makes it well-suited for handling the continuous flow of sensor data in the system while performing advanced analytics and model training on historical datasets. Within the Predictive Maintenance System, Apache Spark plays a critical role in the **Data Processor** component. It is responsible for performing real-time computations on incoming sensor data. Spark's distributed computing architecture allows the system to scale effectively, ensuring that large-scale data processing tasks are handled efficiently and without bottlenecks, even as the number of machines and sensors increases.

*5) **Docker**:* Docker was employed to ensure consistent, reliable deployment across different environments by containerizing each component of the system. Docker encapsulates the **User Interface**, **Data Consumer**, **Data Processor**, **Data Predictor**,**Mongodb**, **Spark** and **Spark Streaming** into isolated containers, each with its dependencies and configurations. This containerization guarantees that the system behaves identically across development, testing, and production environments, eliminating issues related to differences in system configurations. Docker simplifies deployment and scaling by allowing each component to run independently in its container. This modularity enables easier updates, scaling, and maintenance. For instance, if the **Data Processor** requires more resources due to increased sensor data, additional containers can be deployed without affecting other components. Docker also streamlines the process of updating the system, allowing individual components to be updated or replaced without disrupting the overall functionality, making it an essential tool for ensuring system reliability and scalability.

III. Implementation

The implementation of the Predictive Maintenance System is structured to ensure the efficient collection, processing, and storage of sensor data. The project is hosted on GitHub, where all the necessary files and Docker images are stored, enabling easy deployment and replication of the system across different environments. To illustrate our approach, we installed 20 machines, each equipped with 3 sensors. This configuration was chosen as an example to demonstrate our goal of developing a versatile project applicable across various industries. Our aim was to create an IT infrastructure that is easily adaptable to diverse requirements. Potentially, each machine could be considered as just a component of a productive line, thus the prediction of failure could be tailored on that specific part. In this way it could possible to predict when that component is likely to fail and when is better to operate maintenance on it, allowing for a flexible and scalable solution that can be tailored to different operational needs.

- **Data consumer**: The *consumer.py* script subscribes to the "machines/machine_ID" MQTT topic. Upon receiving a machine ID, it gathers real-time sensor data from IoT devices connected to the selected machine. It retrieves data related to critical parameters such as temperature, vibration, and pressure, and publishes this data to MongoDB. The Consumer is a critical component that ensures

the system has access to the latest machine data for accurate prediction.

- **Data Processor**: The*process.py* script is the heart of the data processing pipeline. It subscribes to MongoDB and processes the raw machine sensor data by extracting key features and applying the machine learning model. One the model is trained and stored, few seconds will be waited before new raw data will be used to train the model again.
- **Data predictor:** The *predictor.py* script handles the random forest classifier used for predictive maintenance. It employs a file locking mechanism to ensure exclusive access to the machine learning model during updates. This lock system prevents concurrent modifications or reads during critical operations, safeguarding data integrity. By managing file access with timeouts and lock checks, the system enhances fault tolerance, ensuring robustness against data corruption and operational errors. It receives raw data, applies a pre-trained random forest. and then publishes them on the "predictions/machine_ID" MQTT topic .The results are then sent back to the user through the UI, ensuring that users are always informed about potential maintenance issues before they occur.
- **Historical data:** The *generation.py* script is responsible for generating and managing historical data, which is used for training the machine learning models that predict maintenance needs. This directory ensures that a rich dataset is available for improving the accuracy and reliability of the predictive models.
- **User Interface:** The *app.py* and the *index.html* files within the UI directory collectively facilitate the Flask-based web application, serving as both the backend and frontend of the system. This integrated setup allows users to actively engage with the system by starting and stopping the real-time data stream, and by viewing real-time updates on machine health along with failure probabilities for each machine. When a user initiates the streaming process, the *app.py* file dynamically generates a continuous flow of data. This flow is highly customizable, accommodating specific configurations for varying numbers of machines, sensors, and data distribution patterns. It then publishes each machine ID to the "machines/machine_ID" MQTT topic. In response, Flask continuously listens for the processed predictions from the "predictions/machine_ID" MQTT topic and updates the user interface with the latest data on machine health and predictive maintenance insights. This seamless integration ensures that the system is not only interactive but also readily accessible to all stakeholders, enhancing user engagement and operational efficiency.
- **Docker compose** The *docker-compose.yml* file orchestrates the system, defining how each container interacts with others and how the services (UI, MongoDB, MQTT broker, etc.) are brought up and linked together.

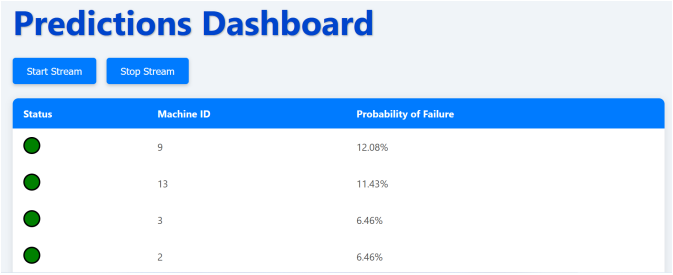

Fig. 2. User Interface

IV. Results

The system successfully integrates real-time sensor data and historical maintenance records, providing accurate maintenance predictions and real-time alerts. The system's core functionalities are operational, demonstrating its ability to predict equipment failures and provide actionable insights to prevent unexpected downtimes.

V. Conclusion

Some of the improvements that could ensure a smoother and more reliable system for real-life data can be:

Temporal Forecasting Models: Incorporating advanced machine learning models such as Recurrent Neural Networks (RNNs), ARIMA, SARIMA, and survival functions could significantly enhance the system's predictive capabilities. These models are adept at handling temporal data and can provide more nuanced predictions by considering time-dependent patterns. Implementing these models would allow for more precise forecasting, enabling the system to weigh predictions based on historical trends and seasonal variations, leading to improved accuracy in maintenance scheduling.

Apache Kafka Integration: Transitioning from MQTT to Apache Kafka for data streaming and processing could offer a more robust and scalable framework. Kafka's high-throughput, low-latency messaging capabilities are well-suited for handling large volumes of real-time data. It would provide improved reliability and fault tolerance, ensuring continuous and efficient data flow between components, especially as the scale of the system increases.

Enhanced User Interface: Several enhancements to the user interface (UI) could greatly improve user experience and operational efficiency:

- **Blocking Data Production**: Implement functionality that allows users to block or pause data production from specific machines. This would be useful for scenarios where maintenance is scheduled or for troubleshooting.
- **Failure Probability Averages**: Introduce features to calculate and display average probabilities of failure for machines over specified periods, aiding in trend analysis and maintenance planning.
- **Maintenance Calendar**: Develop a calendar view where users can schedule and view upcoming maintenance activities for different machines. This feature would help

in planning maintenance operations and ensuring that all machines are serviced in a timely manner.

Real Data Integration and Machine Differentiation: Using real-world data for model training and validation is crucial for enhancing the accuracy and relevance of predictions. Additionally, incorporating greater differentiation between machines based on their specific tasks and operating parameters will allow the system to adapt to the unique characteristics of each machine, improving the precision of maintenance forecasts.

External Factors Consideration: Expanding the model to include external factors such as the age of the machine and intensity of the workload could provide more comprehensive insights. These factors can influence machine performance and failure rates, and their inclusion would allow for more accurate predictions and better-informed maintenance decisions.

In conclusion, the system developed in this project demonstrates a promising approach to renovate the filed of predictive maintenance. Future work could focus on incorporating advanced forecasting models, provide a system with better scalability with Apache Kafka, refining the user interface, and integrating real-world data in order to achieve more accurate predictions. These proposed enhancements would further elevate the system's reliability and effectiveness in real-world industrial environments.

REFERENCES

[1] Apache Spark Documentation. [Online]. Available: https://spark.apache.org/docs/latest/

[2] Apache Spark Streaming Documentation. [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[3] MongoDB Documentation. [Online]. Available: https://docs.mongodb.com/

[4] Docker Documentation. [Online]. Available: https://docs.docker.com/

[5] Mosquitto Documentation. [Online]. Available: https://mosquitto.org/documentation/

[6] Flask Documentation. [Online]. Available: https://flask.palletsprojects.com/en/latest/