Machine Learning Techniques for Distributed IoT Platform

Joshoua Bigler
Author

Prof. Dr. Daniel Lenz
Advisor

Cedric Sutter
Contributor

July 8, 2025

Abstract

This project aims to develop an Internet of Things (IoT) platform offered as a Software as a Service (SaaS) solution, designed for small to medium-sized businesses (SMBs) in the mechanical engineering sector. The platform provides real-time monitoring and leverages machine learning techniques for anomaly detection and predictive Maintenance (PdM). The thesis focuses on seamlessly integrating these machine learning methods into the IoT platform to reduce unexpected machine downtime and ensure high system availability at minimal cost. To this end, it explores a range of data-driven approaches for PdM and time series anomaly detection (TSAD) and provides an overview of the existing methods.

Acknowledgements

I would like to thank Prof. Dr. Daniel Lenz for his supervision and constructive input throughout the course of this thesis. His feedback helped shape the direction and clarity of the work. I would also like to thank Cedric Sutter for his support in setting up the user interface.

Declaration of Independence

'I hereby certify that I have written this thesis independently and have not used any auxiliary materials other than those indicated. The passages of the work, which are taken in the wording or the sense after other works (to it also Internet sources count), were marked under indication of the source'.

Joshoua Bigler

July 8, 2025

Date

Contents

1 Introduction

The Internet of Things (IoT) offers significant benefits across industries through real-time data analysis and automation. However, small to medium-sized businesses (SMBs), particularly in mechanical engineering sector, often struggle to adopt IoT solutions due to limited expertise, budget, and infrastructure. This project develops a modular, configurable IoT platform as Software as a Service (SaaS) tailored to the needs of these SMBs, balancing a shared core architecture with the flexibility to support diverse customer requirements. The platform includes real-time monitoring capabilities, supports integration with legacy systems, and is built with scalability and security as core principles. Furthermore, it extends the previous work by Bigler (2025a) by incorporating a web application for data visualization, an analytics API for data analytics and online prediction using state of the art machine learning techniques for gear fault detection and classification.

A key focus of this thesis is the integration of modern machine learning methods to enable predictive Maintenance and anomaly detection. Various maintenance strategies are explored, including data-driven approaches for estimating Remaining Useful Life (RUL) and identifying abnormal system behavior. While the full landscape of predictive maintenance is beyond the scope of this thesis, an overview of relevant techniques and their applicability to the developed platform is provided.

2 Software Architecture

The following description of the software architecture is based on the requirements from prior work in Bigler (2025a, pp. 5–6).

2.1 Target Architecture

The target software architecture Figure 1 is split into three main components: the **IoT edge device** (on premise), the **backend** (cloud) and the **frontend** (cloud).

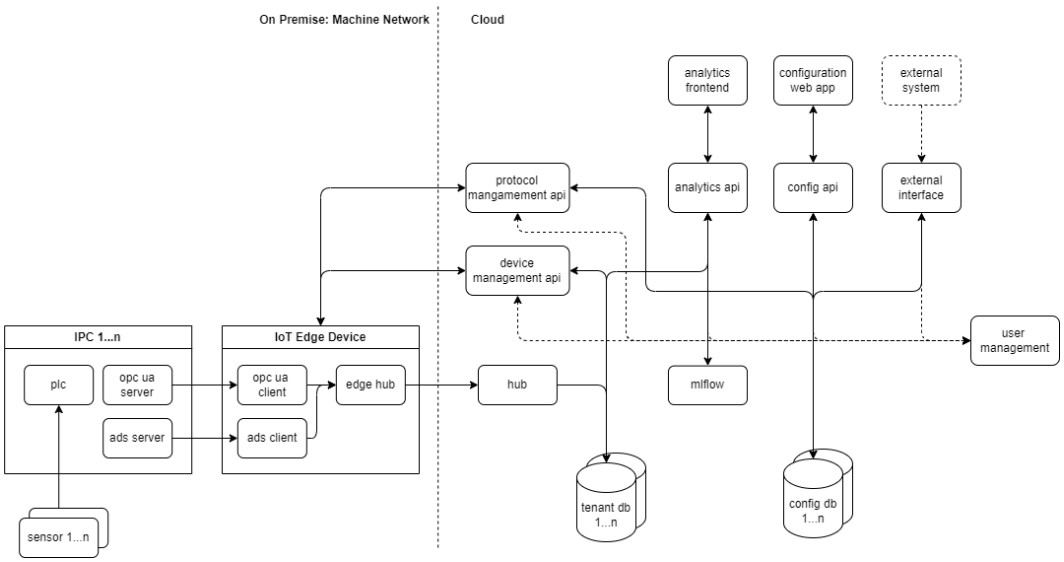


Figure 1: Target Architecture (Bigler 2025a, p. 5)

IoT Edge Device
The IoT edge device is responsible for collecting data from various sources (e.g. OPC UA or ADS server) from an industrial PC (IPC) provided by the customer. This data is then transmitted to the cloud hub for further processing. Each communication protocol operates within a dedicated, configurable container managed through the protocol management application programming interface (API). The collected data is first sent to the edge hub via gRPC and then forwarded to the cloud hub (also via gRPC) to ensure efficient and reliable communication.

Backend
The backend is responsible for processing and persisting the collected data from the edge devices. It provides the different API endpoints for configuration, data analytics using machine learning, device management and the user interface. Furthermore it handles authentication and authorization of users and devices.

Frontend
The frontend is the user interface (UI) of the platform. It allows the user to interact with the SaaS platform, such as configuring the protocol management or the device management API. Furthermore it visualizes the analysed data collected to gain insights into the underlying systems.

2.2 Phase 2 Architecture

In the minimum viable product (MVP) architecture Figure 2, the on-premise IoT edge device consists of a single application that simulates a device generating sensor data from various sources, such as temperature, vibration and humidity sensors. For this thesis, this data is simulated rather than originating from actual sensors in a 'real life' system. The simulated data is transmitted to the cloud hub via gRPC and stored in a PostgreSQL database (*PostgreSQL* 2024) with the Timescale extension (*Timescale* 2024) enabled to enhance the capabilities of handling time-series data. The device configuration, such as creating new temperature sensors or registering new devices, is managed through the device management API. The cloud hub handles receiving data from the devices over gRPC and storing it in the database.

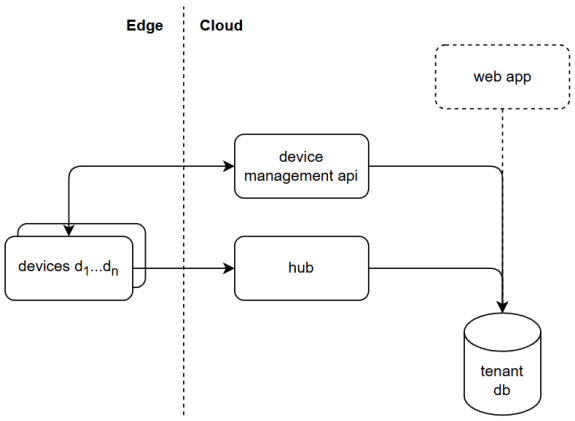


Figure 2: MVP Architecture (Bigler 2025a, p. 6)

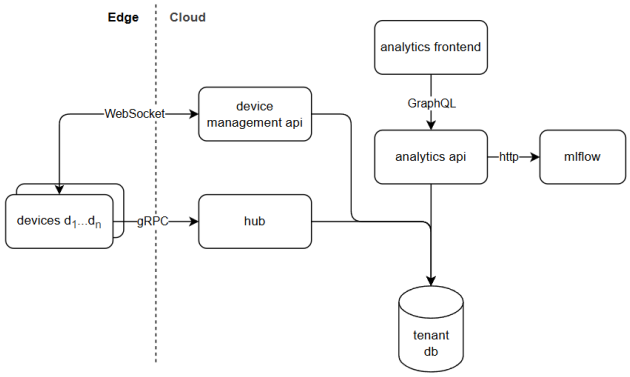


Figure 3: Phase 2 Architecture

The MVP architecture from phase one is extended by the phase two architecture as shown in Figure 3 with the analytics API and the analytics frontend application. The analytics frontend is a web application developed using TypeScript and the React library (*React* 2025), enabling users to visualize the collected data from edge devices. To provide the frontend with analyzed data, the analytics API exposes a GraphQL endpoint, offering a flexible and efficient interface for querying information from the database. Data processing and analytical computations are performed within the analytics API, which also uses pre-trained machine learning models retrieved from the MLflow model registry (*mlflow* 2025).

3 Maintenance

This chapter provides an overview of the maintenance techniques used in the machine engineering industry, and compares them based on their costs.

3.1 Purpose and Goals

As stated by Zhu et al. (2024, p. 1), 'In industry, any outages and unplanned downtime of machines or systems would degrade or interrupt a company's core business, potentially resulting in significant penalties and immeasurable reputation and economic loss. Existing traditional maintenance approaches, such as Reactive Maintenance (RM) and Preventive Maintenance (PM), suffer from high prevent and repair costs, inadequate or inaccurate mathematical degradation processes, and manual feature extraction'. Maintenance is a critical factor in the machine industry, significantly affecting both operational costs and system reliability. It also plays a key role in a company's competitiveness in terms of cost efficiency and performance. Unplanned machine downtime can disrupt operations, leading to economic losses and potentially causing damage to the company's reputation.

3.2 Maintenance Techniques

3.2.1 Reactive Maintenance

RM is a run-to-failure approach, where maintenance is performed only after a failure occurs. Zhu et al. (2024, p. 3) notes that, while reactive maintenance may initially seem appealing due to maximum equipment utilization, few companies adopt it fully because of the high long-term costs and operational risks. While this might save money upfront, the eventual repair or replacement costs could exceed the value gained from operating the equipment until failure. Moreover, as machines deteriorate, through issues like vibration or overheating, they can cause further damage, leading to even higher repair expenses. To manage this risk, companies must either keep a large inventory of spare parts for critical equipment or depend on suppliers who can quickly deliver replacements when needed.

3.2.2 Preventive Maintenance

PM also known as planned maintenance, involves performing maintenance on equipment at scheduled intervals to reduce the likelihood of failure. This maintenance is carried out even while the machine is still operating normally, helping to prevent unexpected breakdowns and downtime. Most PM systems are based on a time-based approach, where maintenance is performed based on elapsed time. PM typically involves two main steps: first, analyzing collected time-series data to understand the equipment's failure characteristics and second, determining the most cost-effective maintenance strategy that ensures high availability and safety at the lowest costs. While PM can help lower repair costs and avoid unexpected breakdowns, it may also lead to unnecessary maintenance or even serious equipment failures. This is because PM often relies on estimated failure timelines rather than real-time equipment data, which can result in servicing machines too early or too late, after damage has occurred. As a result, PM can increase planned downtime and complicate inventory planning. If equipment fails earlier than expected, reactive maintenance must be used, which studies show is typically three times more expensive than scheduled repairs as noted by R. Keith Mobley (2002), cited in Zhu et al. (2024, p. 4).

3.2.3 Predictive Maintenance

PdM aims to estimate the likely time of equipment failure and determine the most appropriate maintenance action to take, aiming to balance maintenance costs with how often maintenance is needed. PdM uses equipment real-time data to create predictive models which identifies the optimal time for maintenance, rather than running to failure or performing maintenance at fixed intervals. According to Zhu et al. (2024, p. 4), this approach can significantly reduce planned and unplanned downtime, high maintenance expenses, excess inventory, and unnecessary servicing of functional equipment.

3.2.4 Comparison of Maintenance Techniques

In Figure 4, the cost of maintenance is compared across the three maintenance techniques. On the x-axis the frequency of maintenance is shown, while the y-axis shows the cost of maintenance.

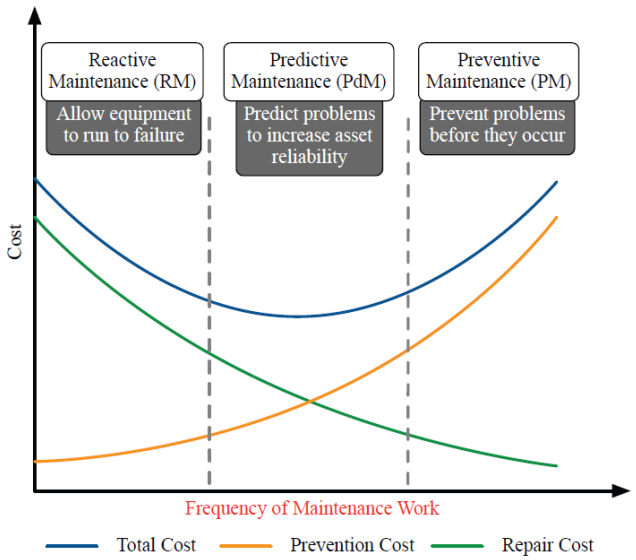


Figure 4: Maintenance Cost Comparison (Zhu et al. 2024, p. 5)

As shown, the predictive maintenance approach is the best trade-off between maintenance frequency and cost. It results in the lowest overall cost by effectively balancing preventive actions and repair expenses.

4 Overview of Predictive Maintenance

As stated in section 3, PdM aims to estimate the likely time of equipment failure, aiming to balance maintenance costs with how often maintenance is needed. PdM uses uses real-time equipment data to eliminate unexpected downtime, improve overall system availability and reduce operating costs. PdM is typically grouped into three main categories: *physical model-based*, *data-driven*, and *hybrid approaches*. Physical model-based methods rely on domain expertise to construct mathematical models that represent the degradation behavior of a system. While these models provide clear physical insight, they are often difficult to implement in complex systems. On the other hand, data-driven methods use historical data to monitor and predict the condition of systems. These approaches encompass statistical techniques, reliability analysis, and artificial intelligence algorithms. They are well-suited for complex systems, as they do not require detailed knowledge of system mechanics, though their predictions are often more difficult to interpret physically. The hybrid approach seeks to integrate the strengths of both physical and data-driven methods, combining expert knowledge with data-based insights (Serradilla, Zugasti, and Zurutuza 2020, pp. 1–2).

4.1 Challenges

Designing effective PdM systems is complex, as they are expected to meet a wide range of technical and operational criteria. As outlined by Serradilla, Zugasti, and Zurutuza (2020, p. 3), a robust PdM system should be capable of quick detection and diagnosis of faults, distinguishing between different types of failures (isolability) and remaining reliable under varying conditions (robustness). It should also be able to identify new or unknown failure patterns (novelty identifiability), estimate the likelihood of classification errors, adapt to changing system conditions, and offer interpretable results (explanation facility). Furthermore, it should require minimal prior modeling effort, operate in real time, manage storage efficiently, and support detection of multiple simultaneous faults.

Meeting all these criteria in real-world industrial environments is challenging. Two significant challenges in industrial applications are the variability in both system behavior and data. Such variability can arise even among machines operating under identical conditions due to factors like mechanical tolerances, installation differences, variations in environmental and operating conditions (EOC) and others. This diversity hinders the ability to reuse PdM models across different assets. Additional challenges include acquiring high-quality data, ensuring proper preprocessing, and conducting effective feature engineering to construct a dataset that accurately reflects the underlying issue. Moreover, since each data point is temporally dependent on previous ones, analysis must consider sequences of observations, thereby increasing the dimensionality and complexity of the modeling process. Collecting failure data is also difficult, as machinery is typically engineered and managed to avoid breakdowns, making such events rare.

4.2 Predictive Maintenance Stages

According to Serradilla, Zugasti, and Zurutuza (2020, p. 4), most data-driven methods follow an incremental process: anomaly detection, failure diagnosis, degradation prognosis, and mitigation as shown in Figure 5 based on the Open System Architecture for Condition Based Maintenance (OSA-CBM) standard.

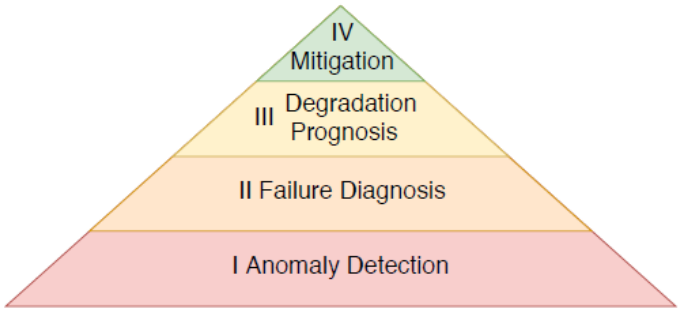


Figure 5: PdM stages (Serradilla, Zugasti, and Zurutuza 2020, p. 4)

Before the main PdM stages, data preprocessing and feature engineering are commonly applied to build a representative dataset and improve model performance. The PdM stages are described in more detail below as summarized by Serradilla, Zugasti, and Zurutuza (2020, pp. 4–8), based on prior work cited therein.

4.2.1 Preprocessing

Preprocessing prepares raw data for PdM models, which often have specific input requirements. It includes several techniques aimed at improving data quality and model effectiveness. Sensor data validation ensures that the input data is accurate and consistent. Data cleaning removes or interpolates missing values. Oversampling addresses class imbalance, particularly for rare failure events. Encoding transforms categorical or complex features into formats suitable for learning algorithms. Segmentation divides continuous data into smaller, manageable chunks for analysis. Feature scaling, such as normalization or standardization, brings features into a comparable range.

4.2.2 Feature Engineering

This stage involves selecting a subset of meaningful features to serve as input for subsequent modeling steps. Although not strictly necessary for deep learning approaches, since they can automatically learn relevant features from the data itself. Common techniques fall into several categories: statistical feature extraction in the time and frequency domains, which capture temporal or spectral patterns; projection-based methods such as Principal Component Analysis (PCA), which reduce dimensionality while preserving important information; concatenation and fusion methods, which create new features by combining existing ones; and feature selection, which eliminates low-variance, redundant, or irrelevant features to reduce complexity without sacrificing predictive power.

4.2.3 Anomaly Detection

This stage focuses on identifying deviations from normal operating behavior. It serves as a foundation for subsequent diagnosis and prognosis steps. Anomaly detection can be approached in various ways depending on the nature of the available data. More details are provided in section 6.

4.2.4 Diagnosis

Once an anomaly is detected, the diagnosis stage determines whether it corresponds to a genuine fault that may lead to failure, or if it is a false alarm. In the latter case, the anomaly detection model may require reassessment or retraining. Diagnosis typically involves Root Cause Analysis (RCA) to identify the underlying source of the issue. The choice of diagnosis method depends on the anomaly detection approach and the nature of the training data. Common strategies include multi-class classification (for multiple failure types) as shown in section 8, binary classification (for failure vs. non-failure), one-class classification (when only one class is available), and clustering (for unlabeled data). Additionally, Health Index (HI) methods are often used to quantify the severity of degradation by comparing current and historical data. These indices can provide either a percentage deviation from normal conditions or a numerical damage score, with values ranging from healthy to fully failed state.

4.2.5 Prognosis

After detecting and diagnosing an anomaly, prognosis aims to estimate how the asset's condition will evolve over time. This involves monitoring degradation based on current operational states and the key features identified during anomaly detection and diagnosis. When sufficient historical failure data is available, RUL models are used to predict the time or cycles remaining before failure subsection 7.1. In cases with limited degradation data, degradation can be approximated by tracking the evolution of a HI or the distance between current and healthy states (see section 7 for more details).

4.2.6 Mitigation

Once an anomaly is detected, diagnosed, and its remaining life estimated, maintenance actions can be planned to prevent failure and reduce downtime. Mitigation involves restoring the asset to a healthy state before failure occurs, minimizing costs and disruptions. Maintenance technicians develop and execute these plans, supported by data-driven PdM models that provide relevant statistics and actionable recommendations. Effective mitigation combines domain expertise with predictive insights into asset health and degradation.

5 Time Series Data

As described in Darban et al. (2024, p. 3), a time series represents a collection of data points arranged in chronological order. Typically, it consists of observations captured at successive time intervals. Time series data can be categorized as either univariate, involving a single variable, or multivariate, involving multiple variables. These categories are discussed in detail in the following subsections.

5.1 Univariate Time Series

A univariate time series (UTS) refers to a sequence of values that correspond to changes in a single variable over time. This type of time series with t time points can be expressed as an ordered list of observations (Darban et al. 2024, p. 3):

$$X = (x_1, x_2, \ldots, x_t) \tag{1}$$

where each x_i is the observed value at time step i, and $i \in T$ with $T = \{1, 2, \ldots, t\}$.

5.2 Multivariate Time Series

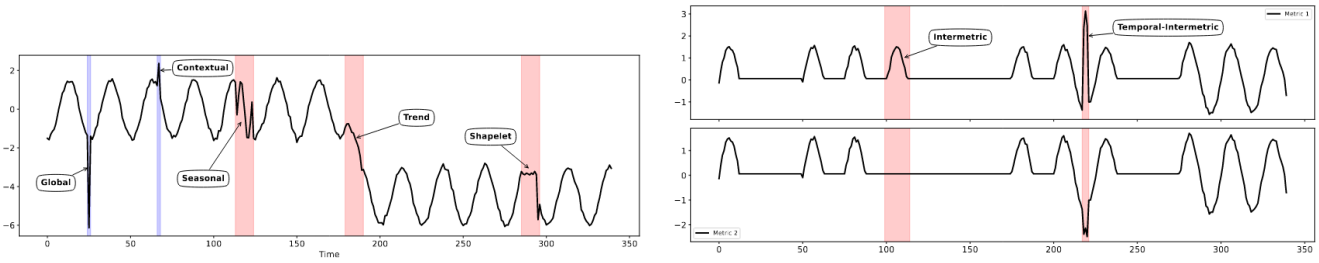
As stated in Darban et al. (2024, p. 3), 'a multivariate time series (MTS) represents multiple variables that are dependent on time, each of which is influenced by both past values (referred as 'temporal' dependency) and other variables (dimensions) based on their correlations. The correlations between different variables are referred to as spatial or intermetric dependencies in the literature, and they are used interchangeably'. For example, in addition to tracking humidity, one might also record temperature and air pressure at the same time intervals.

An MTS with d dimensions can be described as a series of vectors recorded at successive time steps. At each time point i, we observe a vector X_i that contains values for all d dimensions.

$$X = (X_1, X_2, \ldots, X_t) = \big((x_1^1, x_1^2, \ldots, x_1^d), (x_2^1, x_2^2, \ldots, x_2^d), \ldots, (x_t^1, x_t^2, \ldots, x_t^d)\big) \tag{2}$$

Here, each $X_i = (x_i^1, x_i^2, \ldots, x_i^d)$ denotes the vector of observations at time step i, where x_i^j is the value for the j-th dimension at that time. The index j ranges from 1 to d, the total number of dimensions (variables) in the series.

5.3 Time Series Decomposition



(a) Univariate time series

(b) Multivariate time series

Figure 6: (a) An overview of different anomalies in UTS; (b) Intermetric and temporal anomalies in MTS (Darban et al. 2024, p. 4)

5.4 Anomalies in Time Series

Anomalies are defined as deviations from the typical pattern in a dataset, either as single points or sequences. They generally make up a small portion of the data, while most observations follow a normal trend. Although real-world data often contains noise, only significant deviations from the norm are typically relevant. In time series data, anomaly detection and trend analysis are related but distinct tasks (Darban et al. 2024, p. 4).

5.5 Types of Anomalies

As stated in (Darban et al. 2024, p. 4), anomalies in time series data, both univariate and multivariate, can be categorized as temporal, intermetric, or a combination of both. Temporal anomalies are characterized by deviations in value either globally across the series or locally within a neighborhood. In multivariate settings, anomalies can impact multiple dimensions simultaneously. The most common forms of temporal anomalies include (Darban et al. 2024, pp. 4–6):

- **Global:** Isolated data points that significantly deviate from the overall pattern, such as a sudden spike. These are typically detected when the absolute difference between the true value x_t and the model output $\hat{x}_t$ exceeds a threshold:

$$|x_t - \hat{x}_t| > \text{threshold} \tag{3}$$

- **Contextual:** Values that are only anomalous within a specific context or local window, even if they appear normal globally. These are detected by comparing a point to its local neighborhood using a variance-based threshold:

$$\text{threshold} \approx \lambda \times \text{var}(X_{t-w:t}) \tag{4}$$

  where λ is a hyperparameter, and $\text{var}(X_{t-w:t})$ denotes the variance of the data within a window of size w up to time t.

- **Seasonal:** Deviations in the recurring seasonal patterns of a time series, despite otherwise normal behavior. These are identified by comparing the dissimilarity between actual S and expected seasonal subsequences $\hat{S}$:

$$\text{diss}_s(S, \hat{S}) > \text{threshold} \tag{5}$$

- **Trend:** Sudden or gradual changes in the overall direction or slope of the time series without necessarily affecting its cycle or seasonality. They are detected when the dissimilarity from the expected trend $\hat{S}$ to the actual trend T exceeds a threshold:

$$\text{diss}_t(T, \hat{T}) > \text{threshold} \tag{6}$$

- **Shapelet:** Distinct patterns or subsequences that differ in structure or cycle from the rest of the data. These represent local pattern shifts and are captured via shape-based dissimilarity:

$$\text{diss}_c(C, \hat{C}) > \text{threshold} \tag{7}$$

  where C is the candidate subsequence extracted from the time series and $\hat{C}$ is the expected shape.

6 Anomaly Detection

The following section provides an overview of time series anomaly detection methods, focusing on deep learning approaches. As stated in Serradilla, Zugasti, and Zurutuza (2020, p. 5), anomaly detection involves identifying instances where an asset's behavior deviates from its established normal operating conditions. Although deep learning models have evolved rapidly, anomaly detection still faces several challenges, as noted in Tuli, Casale, and Jennings (2022, p. 1). The task of identifying and diagnosing anomalies in time series is of great importance for modern industrial systems. Nevertheless, designing a framework that can rapidly and precisely detect irregular patterns remains a significant challenge. This difficulty stems from several factors, including the scarcity of labelled anomalies, high variability in the data, and the necessity for extremely low-latency inference in practical deployments. Although recent advancements in deep learning have contributed notable progress to anomaly detection, only a limited number of these methods effectively tackle all of these constraints simultaneously.

6.1 Traditional Methods

Traditional techniques for detecting anomalies in time series utilize diverse strategies. Statistical methods focus on modeling the typical behavior of time series data through probabilistic frameworks. Clustering-based techniques construct representations of normal behavior by grouping similar time series windows; anomalies are detected either by measuring the distance from cluster centroids or by identifying sparsely populated clusters. Distance-based methods assign anomaly scores based on the proximity of a time series segment to its nearest neighbors. Density-based techniques further extend this idea by evaluating local data density to highlight unusual patterns (Darban et al. 2024, p. 6).

6.2 Deep Learning

The following section presents a general overview of deep learning approaches for time series anomaly detection (Darban et al. 2024, pp. 7–10). These methods are particularly well-suited for modeling the complex structures and the temporal as well as spatial dependencies inherent in time series data. Neural networks play a central role in this context due to their strong capability to extract informative feature representations. As noted by Albuquerque Filho et al. (2022), 'One of the most explored hypotheses is that feature representations extracted by deep networks preserve discriminatory information that helps separate anomalies from normal instances'. Compared to conventional dimensionality reduction techniques commonly used in anomaly detection, such as PCA and random projection, deep networks have shown a significantly greater ability to capture both linear and nonlinear feature relationships. To support this discussion, Figure 7 illustrates the main components of a typical deep learning-based anomaly detection framework, highlighting how each part contributes to identifying anomalies over time.

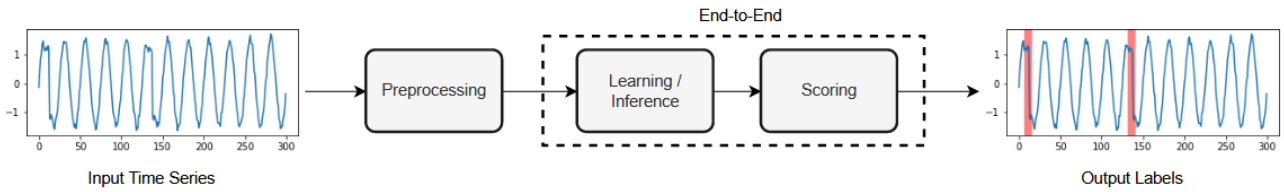


Figure 7: General components of deep anomaly detection models in time series (Darban et al. 2024, p. 7)

Deep anomaly detection models typically focus on learning normal patterns, as labelled anomalies are often scarce in real-world data. These learning methods can be categorized into four groups: *supervised*, which relies on full labels to distinguish anomalies; *unsupervised*, which detects anomalies without labels; *semi-supervised*, which learns from a small set of labelled data combined with a larger pool of unlabelled data; and *self-supervised*, which derives labels from the input data itself. Unsupervised, self-supervised and semi-supervised methods are especially useful in real-world scenarios where anomalies are rare and unlabeled.

Input and Preprocessing
Models in time series anomaly detection typically process either individual time points or sliding windows of sequential data. Sliding windows, which preserve temporal order, are commonly used to capture historical context. These windows are extracted during preprocessing, often following steps like imputation, resampling, and normalization. Instead of raw data, many models use learned representations of these windows to improve subsequence comparison (Darban et al. 2024, p. 8).

Temporal and Spatial Patterns
When using UTS as input, models primarily capture temporal patterns. In contrast, MTS enable learning of both temporal and spatial dependencies. Capturing spatial relationships in MTS allows models to identify intermetric anomalies arising from interactions between variables.

Point / Subsequence Anomaly
Anomalies in time series can appear as individual outlier points or as anomalous subsequences. Point anomalies refer to single time steps that deviate significantly from expected behavior, while subsequence anomalies involve a series of observations that collectively exhibit abnormal patterns, even if individual points seem normal.

Incremental Learning
Incremental learning allows models to continuously update their knowledge as new data becomes available, making it well-suited for streaming and evolving environments. Models can adopt either a *step-by-step* approach, where feature extraction and anomaly scoring are handled separately, or an *end-to-end* approach that directly produces anomaly scores or labels. End-to-end methods, such as DevNet (Pang, Shen, and Hengel 2019), are capable of learning effectively even with limited labelled anomalies. Anomaly scores are usually derived from reconstruction error or prediction loss and are converted into binary labels through thresholding. To identify anomalies, data points are ranked based on their anomaly scores (A_S), and classification is performed by applying a predefined threshold. A point is considered anomalous if the absolute value of its anomaly score exceeds this threshold (Darban et al. 2024, p. 9):

$$|A_S| > threshold \quad \text{(Darban et al. 2024, p. 9)} \tag{8}$$

Interpretability
Providing explanations for detected anomalies is essential, particularly when anomaly detection serves diagnostic or troubleshooting purposes. Interpretability allows users to understand the cause of anomalies, making the system more actionable. However, MTS pose a significant challenge due to the complexity of inter-variable relationships, and the stochastic nature of deep learning models further complicates interpretation. A common approach to improving interpretability involves identifying the dimensions that contribute most to an anomaly, typically by ranking variables based on their anomaly scores. This enables practitioners to focus on the most divergent dimensions when investigating entity-level anomalies.

6.3 Deep Learning Landscape and Methods

As one can see in Figure 8 the deep learning landscape for time series anomaly detection can be split into methods for univariate time series anomaly detection (UTSAD) and multivariate time series anomaly detection (MTSAD).

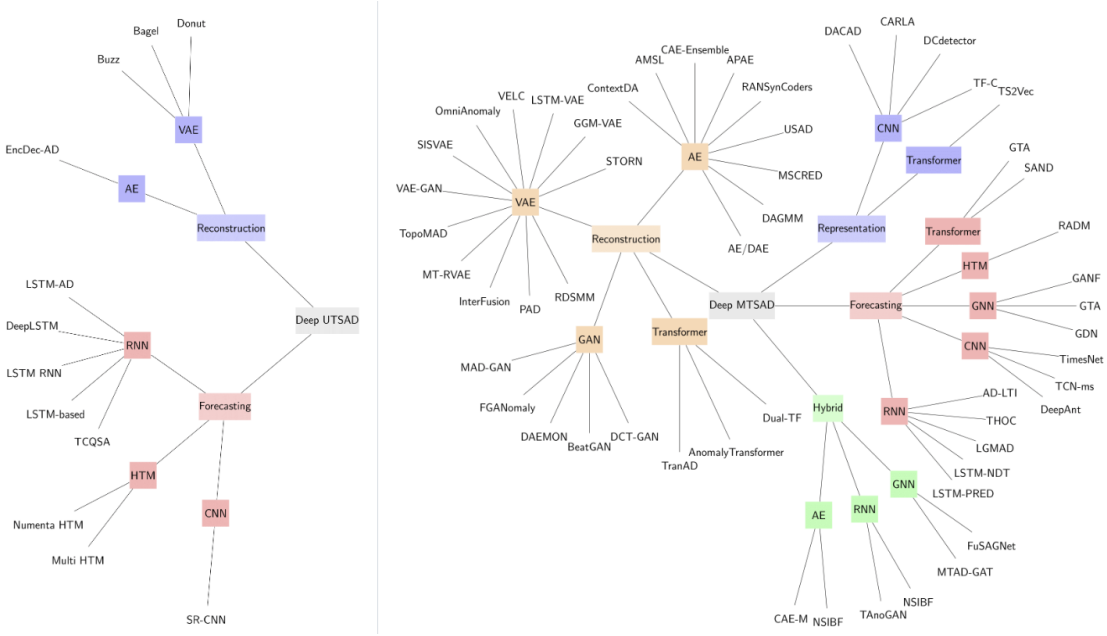


Figure 8: Left: UTSAD, Right: MTSAD (*Deep Learning for Time Series Anomaly Detection Models and Datasets* 2025)

The deep learning methods for TSAD can be further categorized into four categories: *Forecasting-based methods* which are trained to predict future values from recent observations, *reconstruction-based methods* which reconstruct input sequences and measure reconstruction errors, *representation-based methods* which extract meaningful latent features from time series data and *hybrid models* which combine forecasting, reconstruction, and representation learning techniques to improve time series anomaly detection.

6.3.1 Forecasting-Based Methods

Forecasting-based methods rely on models trained to predict future values from recent observations, typically using a sliding window. Anomalies are detected by measuring the deviation between predicted and actual values, with significant discrepancies indicating abnormal behavior. This approach is particularly effective in practical settings where normal data is plentiful, but anomalies are scarce (Darban et al. 2024, p. 9).

6.3.2 Reconstruction-Based Methods

Reconstruction-based TSAD models detect anomalies by reconstructing input sequences and measuring reconstruction errors. Unlike forecasting-based methods, they use current data to rebuild expected patterns, making them more robust in rapidly changing or unpredictable time series. Trained on sliding windows of normal data, these models assume anomalies will reconstruct poorly, resulting in higher errors or lower reconstruction probabilities. Although they may introduce slight detection delays, they offer improved accuracy where precision is critical (Darban et al. 2024, p. 17).

6.3.3 Representation-Based Models

Representation-based models focus on extracting meaningful latent features from time series data, which are then used for downstream tasks such as anomaly detection or classification. Instead of analyzing raw time series directly, these models operate in a learned feature space that better captures temporal dependencies, reduces noise, and handles non-stationarity and seasonality. They are especially effective in settings with limited labelled data, as many rely on unsupervised or self-supervised learning. Although representation learning has gained traction in the time series domain, its application to anomaly detection remains relatively underexplored. Darban et al. (2024, p. 23)

6.3.4 Hybrid Models

Hybrid models combine forecasting, reconstruction, and representation learning techniques to improve time series anomaly detection. Forecasting components predict future values, reconstruction components rebuild input sequences from latent representations, and representation-based modules extract informative features. These components are trained jointly using a unified objective function, allowing the model to leverage complementary strengths during optimization.

6.4 Model Selection Guidelines

This section aims to provide a model selection guideline for TSAD based on the characteristics of the dataset and the specific requirements of the anomaly detection task. As emphasized by Albuquerque Filho et al. (2022), anomaly detection lacks a universal solution: 'Anomaly detection is a difficult problem to solve in general and for that reason most of the techniques in the literature tend to solve a specific case of the general problem, based on the type of application, type of input data and model, availability of labels for the training and testing data and also the types of anomalies'. In this context, the guideline proposed by Darban et al. (2024, p. 17) offers a structured approach to selecting the most appropriate model depending on the scenario at hand.

- **Multivariate Data with Complex Dependendies:** Graph Neural Networks (GNNs) are well-suited for capturing both temporal and spatial correlations in multivariate time series. They are particularly effective in industrial system domains and where variables are strongly interconnected. Architectures such as Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs) are recommended.
- **Sequential Data with Long-Term Dependencies:** Models like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are ideal for capturing long-range temporal dynamics. LSTM is widely used in areas such as finance, healthcare, and maintenance prediction. GRU, being more lightweight, is preferred for faster training and efficient modelling of temporal relationships.
- **Scalable Solutions for Large Datasets:** Transformer-based models use self-attention mechanisms to capture long-range dependencies efficiently, making them suitable for large-scale tasks like network traffic monitoring. A prominent example like TranAD (Tuli, Casale, and Jennings 2022) is designed for detecting complex anomalies.

- **Noise-Robust Detection:** autoencoders (AEs) and Variatonal Autoencoders (VAEs) are effective in noisy environments. They are commonly applied to domains such as cyber-physical systems, sensor networks, and internet traffic where robust feature extraction is essential.
- **High-Frequency or Fine-Grained Patterns:** Convolutional Neural Networks (CNNs) are effective in identifying local temporal patterns and subtle deviations in high-frequency data streams, such as web usage or live system monitoring. Temporal Convolutional Networks (TCNs), which use dilated convolutions, extend CNNs to model both short-term and long-term dependencies simultaneously.
- **Evolving and Multi-Modal Time Series:** Hybrid architectures that combine components like GNNs, VAEs, and LSTMs are capable of managing complex and evolving datasets. These models are well-suited for tasks such as automated industrial processes, and climate pattern analysis.
- **Hierarchical and Multi-Scale Structure:** Hierarchical Temporal Memory (HTM) models are designed to identify patterns across multiple timescales. They are resistant to noise and capable of learning varied patterns simultaneously, making them effective in environments with complex and hierarchical temporal dynamics.

7 Predictive Maintenance Prognosis

As stated in Zhu et al. (2024, p. 9), 'The primary purposes of PdM are to eliminate unexpected downtime, improve overall availability/reliability of systems and reduce operating costs'. In order to achieve this, the following chapter will focus on the prognosis part of a PdM system as stated in subsubsection 4.2.5 which is the prediction of the Remaining Useful Life of a system or component.

7.1 Remaining Useful Life

As defined by Si et al. (2010, p. 1), 'The remaining useful life (RUL) of an asset or system is defined as the length from the current time to the end of the useful life'. Given the widespread application of RUL across various domains, interpretations of what constitutes the 'useful life' can vary. Remaining Useful Life serves as a pivotal metric in predictive maintenance, facilitating maintenance decisions based on the actual health status of equipment rather than adhering to predetermined schedules.

7.1.1 Definition of RUL

Let X_t denote a random variable representing the RUL at time t, which quantifies the expected time remaining until failure, given the current system age or accumulated usage. The variable Y_t denotes the sequence of operational conditions and condition monitoring (CM) data observed up to time t. The conditional probability density function (PDF) of X_t given Y_t is denoted as $f(x_t \mid Y_t)$ and provides an updated probabilistic estimate of the RUL based on the available information. When condition monitoring data is not available, the estimation reduces to the time-based formulation (Si et al. 2010, p. 2):

$$f(x_t \mid Y_t) = f(x_t) = \frac{f(t + x_t)}{R(t)}, \tag{9}$$

where $f(t + x_t)$ is the PDF of failure at time $t + x_t$, and $R(t)$ is the survival function at time t, representing the probability that the system has not failed up to time t. Incorporating Y_t typically leads to more accurate RUL predictions. Remaining Useful Life prediction methods can be categorized into model-based and data-driven approaches C. Cheng et al. (2022, pp. 1–2) to estimate RUL in a probabilistic manner, commonly represented as $f(x_t \mid Y_t)$ or $\mathbb{E}[X_t \mid Y_t]$. Unlike physics-based approaches, statistical data-driven techniques construct predictive models by fitting probabilistic frameworks directly to the observed data, without incorporating physical laws or engineering domain knowledge.

7.2 RUL Estimation Approaches Under Varying Data Availability

As stated in MathWorks (2025), the estimation of Remaining Useful Life depends on the kind of data available. A key concept in many RUL estimation methods is the Conditional Indicator (CI). Conditional Indicators are measurable quantities such as temperature, vibration, pressure, or derived metrics like a Health Index or Degradation Indicator, that reflect the current state of a component. Depending on the data type, CIs are used either explicitly or implicitly to assess degradation and predict remaining useful life.

7.2.1 Lifetime Data

If lifetime data, that is, historical records of the time until failure for components or systems, is available, RUL can be estimated using proportional hazard models and probability distributions derived from historical failure times. For example, battery discharge times can be predicted using previous discharge cycles and influencing factors such as ambient temperature and load conditions. A survival function, such as that in Figure 9, shows the probability of a component surviving over time. For instance, a battery operating for 75 cycles has only a 10% chance of remaining functional, indicating a 90% chance of failure by that point.

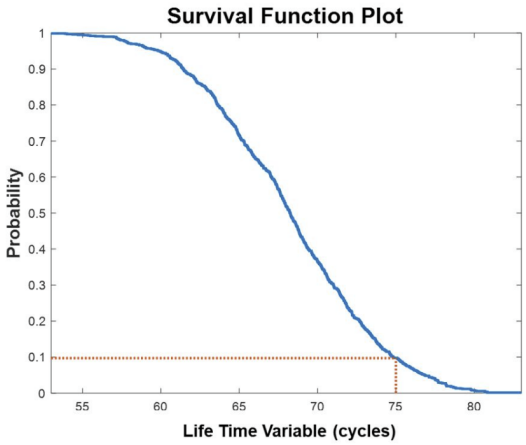


Figure 9: Survival function (MathWorks 2025)

7.2.2 Run-to-Failure Data

When detailed failure records, that is, historical data collected from components monitored until the point of failure, are available, similarity-based methods can be used to estimate RUL. These techniques compare current degradation patterns with historical run-to-failure profiles from similar components. The most similar historical cases help estimate the RUL of the component currently in use. As shown in Figure 10, historical degradation trajectories (blue) are compared to the ongoing profile (red). The endpoints of the most similar profiles provide an average RUL estimate in this example, approximately 65 cycles.

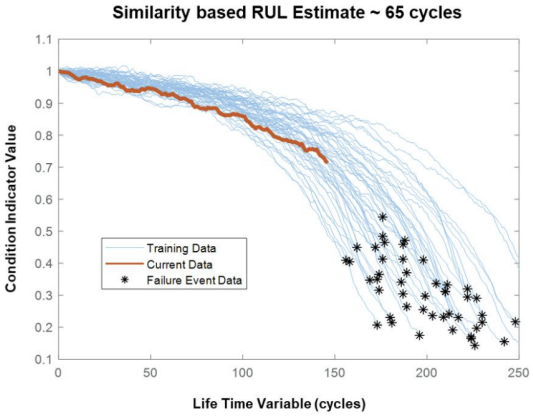


Figure 10: Run-to-failure degradation profiles (MathWorks 2025)

7.2.3 Threshold-Based Data

In environments where failures are rare, degradation data, that is, sensor measurements reflecting the gradual wear or deterioration of a component, may be available only up to predefined safety thresholds. These thresholds, such as maximum temperature or pressure limits, are used in conjunction with condition indicators to model degradation trends. By fitting models to sensor-derived indicators, it is possible to forecast when a threshold will be crossed. Figure 11 illustrates such an approach, where an exponential model predicts bearing failure within approximately 9.5 days, while fused condition indicators and techniques like PCA analysis can improve estimation accuracy.

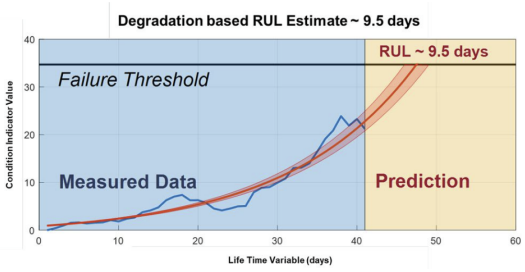


Figure 11: Exponential degradation model (MathWorks 2025)

7.3 RUL Prediction Process

The RUL prediction process is in general structured into four steps as stated in Ferreira and Gonçalves (2022, pp. 553–555):

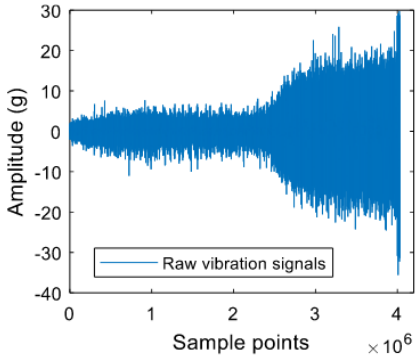
1. **Data Extraction**
   - Raw data extraction: Acquire sensor data (e.g. vibration, acoustic).
   - Data pre-processing: Exploiting the features associated with the degradation of the system.
2. **Feature Extraction and Classification**
   - Feature extraction: Transform raw signals into status-relevant features.
   - Feature classification: Identify sensitive features for degradation.
   - Health indicator construction: Generally involves combining multiple features derived from the time, frequency, and time-frequency domains into a single Health Index using dimensionality reduction techniques.
   - Fault detection: Determine fault location, type, and severity.
3. **Model Building and Training**
   - Model building: Select type of model or algorithm to predict RUL.
   - Model training: Involves establishing a connection between the true health state of the system and the monitored condition data using the selected features.
4. **RUL Prediction and Evaluation**
   - RUL prediction: Estimate time to failure on unseen data.
   - Evaluation and maintenance decision: Assess performance and determine maintenance strategy.

7.4 Examples

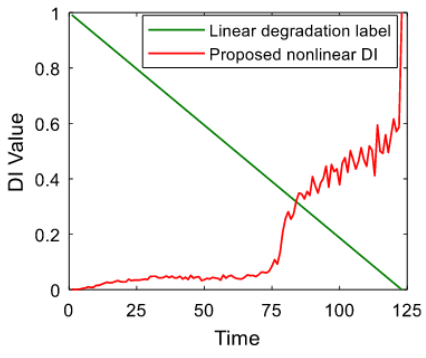
In order to show the process of PdM prognosis, the following subsection will focus on two examples of RUL prediction approaches. Whereby the second example subsubsection 7.4.2 will focus on a solution when failure data is sparse, that is, degradation data is only available up to a certain point, which is likely the most common case in real-world applications.

7.4.1 CNN-Based Health Indicator and Prognosis Using BiLSTM for Rolling Bearings

To address the challenge of accurate health prognosis for rolling bearings, Y. Cheng et al. (2021) proposes a deep learning framework that combines a CNN with a bidirectional long short-term memory (BiLSTM) model. The method begins by constructing a nonlinear DI, which is used as the target variable for model training. The DI is designed based on the observation that the intrinsic energy of vibration signals changes with bearing degradation, for an in depth implementation see in Y. Cheng et al. (2021, pp. 4–5). The raw vibration signal and the proposed nonlinear DI are shown in Figure 12.



(a) Raw vibration signals



(b) Linear DI vs. proposed nonlinear DI. The nonlinear DI reflects real degradation: stable at first, gradually increasing, and sharply rising near failure.

Figure 12: Vibration signals and proposed degradation indicator (Y. Cheng et al. 2021, p. 2)

As illustrated in Figure 13, a CNN is utilized to learn the mapping from raw vibration signals to a DI, effectively extracting high-level features that encapsulate the system's degradation state. The estimated DI serves as input to two separate BiLSTM models: one dedicated to forecasting the future trajectory of the DI by capturing temporal dependencies, and another focused on predicting the RUL based on the learned degradation patterns. During the training phase, these models are optimized using the training dataset to predict both the future DI and the RUL, thereby enabling effective health prognosis of the system.

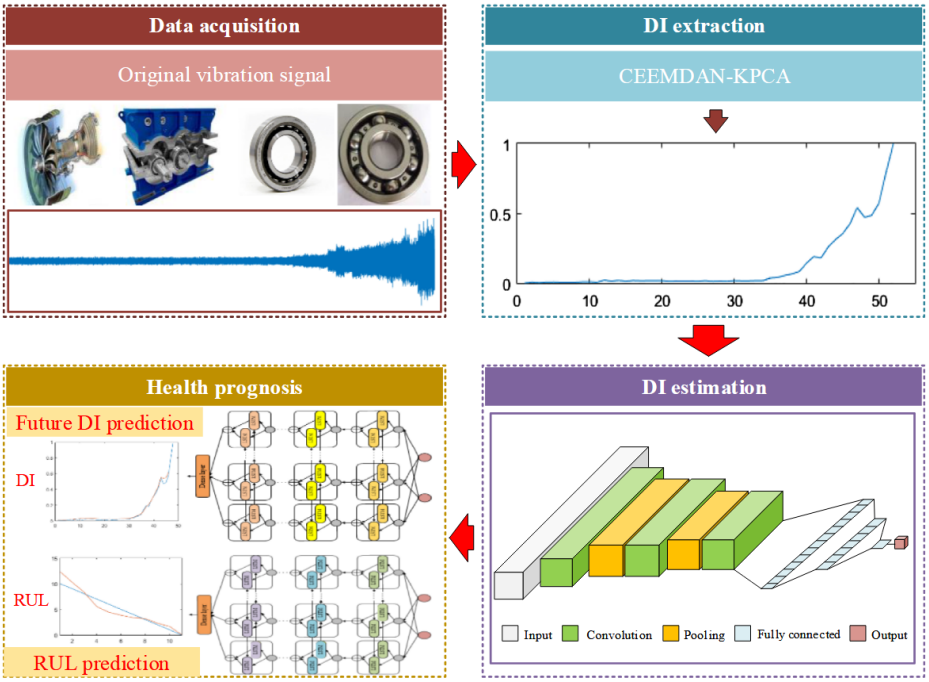


Figure 13: The procedure of DI construction and health prognosis method (Y. Cheng et al. 2021, p. 5)

As stated in Y. Cheng et al. (2021, p. 11), experimental results show that the error between the predicted and actual future DI is very small, demonstrating the proposed method's ability to effectively learn the degradation process of rolling bearings.

7.4.2 PCA-Based Degradation Modeling for RUL Prediction

In Çitil (2022) a threshold-based degradation modeling approach is employed for RUL prediction, utilizing statistical features extracted from bearing vibration data. The method is suitable for cases where limited run-to-failure data is available, and follows these main steps:

1. **Feature Extraction:** From raw vibration signals, time-domain statistical features are computed, including root mean square (RMS), entropy, skewness, kurtosis, peak-to-peak value, crest factor, and others. These features serve as indicators of the equipment's operating condition.
2. **Dimensionality Reduction via PCA:** PCA is applied to the extracted features to reduce dimensionality and isolate the most informative trend. The first principal component (PC1), which captures the maximum variance, is assumed to represent the dominant degradation pattern and is used as a one-dimensional health indicator.
3. **Health Indicator Smoothing:** To reduce noise, an exponential moving average (EMA) is applied to the health indicator.
4. **Exponential Degradation Model:** To model the degradation behavior of the system, the author employs an exponential function, which often accelerates over time in many natural phenomena, of the form:

$$h(t) = a \cdot e^{bt} \tag{10}$$

   where $h(t)$ represents the health indicator at time t, and a and b are parameters to be estimated. To accurately capture the current degradation trend the author fits the exponential model to the most recent n data points of the health indicator.
5. **Failure Threshold and RUL Estimation:** A failure threshold is predefined for the health indicator. The estimated failure time t_f is computed as:

$$t_f = \frac{\ln(\text{threshold}/a)}{b} \tag{11}$$

The Remaining Useful Life is then:

$$\text{RUL} = t_f - t_{\text{current}} \tag{12}$$

The threshold is typically selected based on expert knowledge or historical failure data (if available).

In Figure 14, the degradation model of the predicted and the true health indicator (PC1) and its assigned threshold is shown.

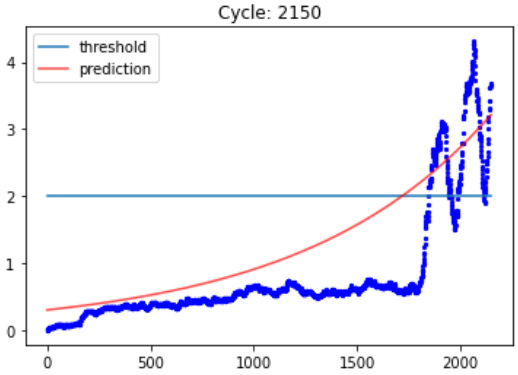


Figure 14: Exponential degradation model of the HI (PC1) (Çitil 2022)

Even if the exponential degradation model simplifies the degradation process and crosses the threshold before the true HI, it is available to estimate the underlying degradation trend.

This approach enables RUL prediction without full run-to-failure datasets by leveraging degradation trends in a compressed feature space. However, it assumes that PC1 correlates strongly with the system's health and that degradation follows an exponential pattern.

7.5 Degradation Models with Sparse Failure Data

In order to address the challenge of RUL prediction with limited failure data, some further methods are discussed in this subsection.

7.5.1 Deep Ordinal Regression Models

Tv et al. (2019) introduces a deep learning methodology for estimating the RUL of equipment using multi-sensor time series data. This approach is particularly effective in scenarios where failure data is limited and sensor measurements are noisy. The method reformulates RUL estimation as an ordinal regression problem, employing LSTM networks to learn the ordinal relationships. By incorporating both failed and censored operational instances into the training process, the model enhances its robustness and generalization capabilities. Empirical evaluations on the C-MAPSS turbofan engine benchmark datasets demonstrate that this ensemble approach outperforms traditional regression-based methods, especially in contexts with scarce failure data.

7.5.2 Feature-Based Transfer Learning

In Yang et al. (2019), a feature-based transfer neural network (ftnn) approach has been proposed. This method leverages abundant labeled data from laboratory-tested bearings (BLMs) to enhance fault diagnosis in bearings used in real-case machines (BRMs). A CNN is employed to extract transferable features from raw vibration data of both BLMs and BRMs. To mitigate the distribution discrepancy between these datasets, the model incorporates multi-layer domain adaptation and pseudo-label learning techniques. These regularization strategies constrain the CNN parameters, ensuring that the learned features are both domain-invariant and discriminative. Empirical evaluations demonstrate that this approach effectively bridges the gap between laboratory and real-world data, achieving higher diagnostic accuracy for BRMs compared to existing methods.

7.6 Challenges

As stated in Ferreira and Gonçalves (2022, pp. 555–556), the RUL prediction process using machine learning faces numerous difficulties. Sensor data often fails to capture meaningful time-varying behavior, and models generally assume that training and testing data are drawn from similar distributions, which may not hold in real scenarios. Data noise, uncertainty, sparsity, and censored measurements further complicate the process. Constructing effective health indicators is complex, especially under diverse operating conditions and degradation patterns. Feature

selection and interpretation require significant domain expertise. Moreover, models often ignore correlations between different sensor signals, and dealing with time-dependent data introduces additional complexity. Limited access to labeled failure data restricts model training and interpretability of machine learning outcomes remains a concern. Thus achieving high RUL prediction accuracy continues to be a major challenge.

7.6.1 Conclusion

As stated in Si et al. (2010, p. 3), 'Traditional methods based on failure time analysis for estimating the component lifetime rely only on the failure event data (Kalbfleisch and Prentice, 2002; Lawless, 2002). It is noted however that failure data is scarce or even nonexistent in practice. As an alternative roadmap, the RUL estimation models based on directly observed state processes have been developed to estimate the RUL'. So instead of relying on failure data, which is often sparse or nonexistent at all, these models learn patterns of degradation from CM data and predict the remaining time before a likely failure. To address the difficulties of sparse failure data, degradation, ordinal regression or transfer learning methods could be used. This section has only scratched the surface of the topic of PdM, which is both broad and still evolving field.

8 Gear Fault Detection

To provide an example of the potential of a machine learning based approach, a supervised data driven fault classification approach is presented, based on the mechanical gear vibration dataset (Dao, Biswa, and Drimus 2022).

8.1 Provenance

Two vibration sensors were mounted on the shaft of a test gear system. Sensor 1 measures vibration (in mm) along the x-axis, while Sensor 2 captures vibration along the y-axis. The gear system was tested under three rotational speeds: 8.33, 25, and 40 rev/s with two different load conditions: 0 and 80 Nm. Vibration data was sampled at a rate of 200 µs per record. The rotational speed was regulated by an AC motor connected through an inverter, and the load was applied using a brake clamped to the main shaft.

- Sensor 1 records vibration along the x-axis.
- Sensor 2 records vibration along the y-axis.
- Operating speeds: 8.33, 25, and 40 rev/s.
- Load conditions: 0 and 80 Nm.
- Sampling interval: 200 µs.

8.2 Goal

The objective is to classify the gear condition based on vibration data collected along both x- and y-axes. The classification includes identifying whether a fault is present, and if so, determining the specific type. The fault categories include: No fault, Surface Defect, Chipped Tooth, Missing Tooth, Root Crack, and Eccentricity.

8.3 Stages

In order to classify the gear faults, different machine learning and deep learning approaches were tested. The following steps were taken as stated in subsection 4.2:

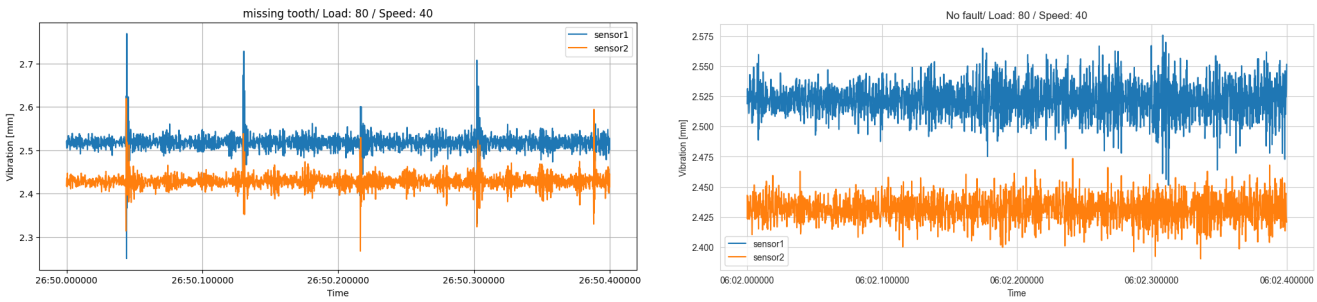
1. Preprocessing: The raw data was cleaned and prepared for analysis.
2. Feature Engineering: Relevant features were extracted from the data (only for machine learning approaches).
3. Fault Detection: A model was trained to detect if the gear has a fault and if yes which one.
4. Diagnosis: The model was evaluated to determine its effectiveness in identifying faults.

8.4 Feature Engineering

Features can either be automatically learned by deep learning systems or they have to be extracted manually as shown in the following subsections.

8.4.1 Sensor signals

In Figure 15 the time series vibration signals of missing tooth and no fault are shown. The time series data is sampled at a rate of 200 µs and has a length of 400 samples. The time series of all fault conditions can be found in Bigler (2025b).



(a) Gear vibration time series with missing tooth fault

(b) Gear vibration time series with no fault

Figure 15: Comparison of gear vibration signals under different fault conditions

Visually comparing the time series data, one can see a significant difference between the signals. The periodic high amplitude in Figure 15a might come from the missing tooth, which causes a periodic impact on the gear system.

8.4.2 Mutual Information

Before investigating further into the conditional indicator in the time and frequency domain, one can analyze the relevance of the signals on fault type using mutual information (MI). As stated in Pedregosa et al. (2011), MI between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency. The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances as described in (Ross 2014). The mutual information $MI(X, Y)$ between two discrete random variables X and Y is defined as:

$$MI(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right) \tag{13}$$

The mutual information between each input feature and the fault condition is summarized below:

- `sensor1`: 0.0789
- `sensor2`: 0.0796
- `speed`: 0.0113
- `load`: 0.0190

As shown, the mutual information between the sensor signals and the fault condition is significantly higher than the mutual information between the operating conditions (speed and load) and the fault condition. Thus the sensor signals are more relevant for the fault condition than the operating conditions.

8.4.3 Data distribution

Understanding the distribution of the raw time series data is a critical step in exploratory data analysis. Analyzing the distribution of sensor signals under the operating conditions (such as speed and load) provides insight into the variability, skewness, and presence of outliers in the dataset. This information helps to identify potential preprocessing requirements, such as normalization and guides the selection or design of meaningful features.

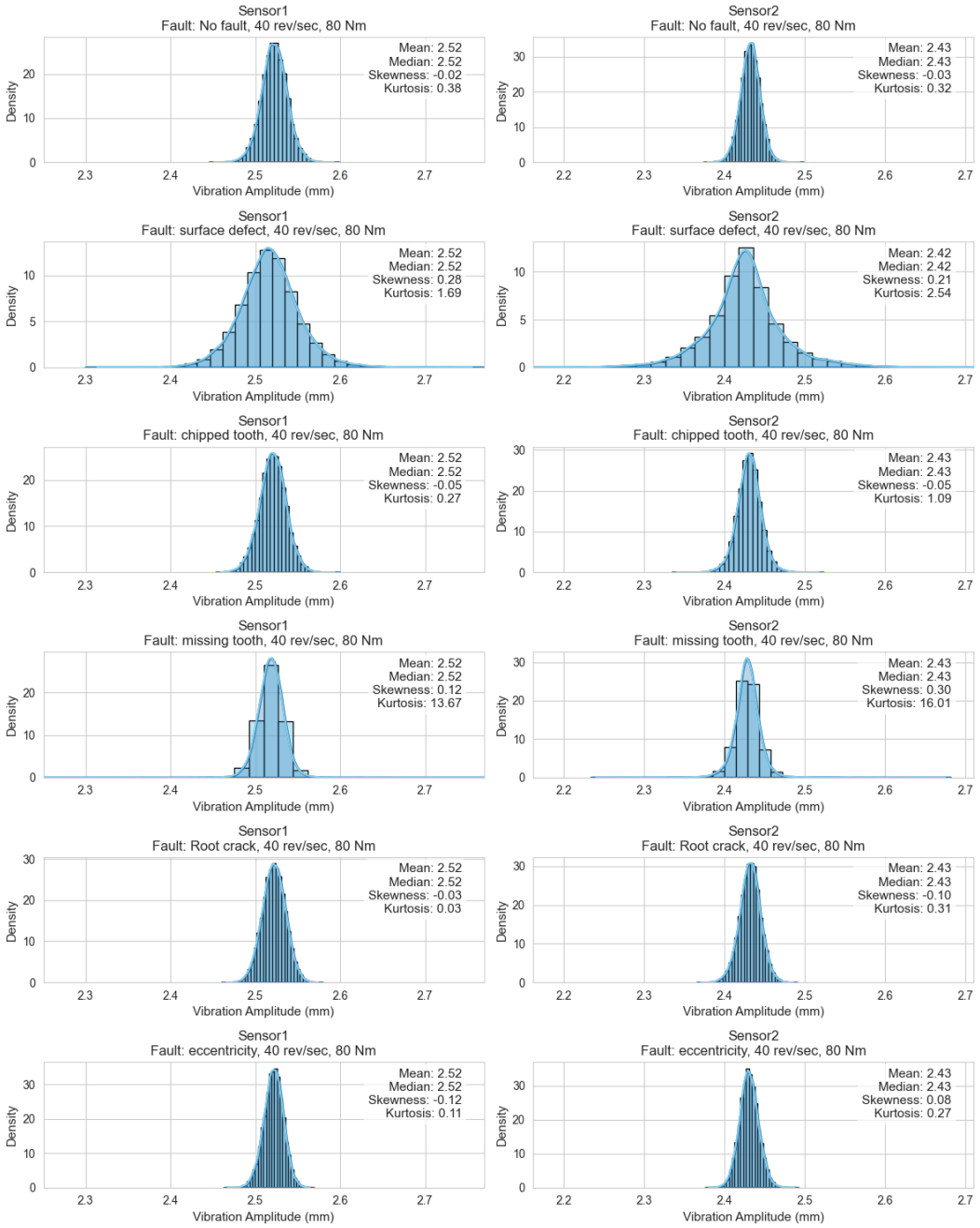


Figure 16: Distribution of time series data of each fault condition

As shown visually in Figure 16, the distribution of the time series data follows a bell-shaped, symmetric curve. This suggests that the data are approximately normally distributed, even without further statistical investigation. Interestingly, the mean and median values for all fault conditions are identical for each sensor signal: $sensor_1 = 2.52$ mm and $sensor_2 = 2.42$ mm. This suggests that the mean or median of this feature may be a weak predictor for the fault condition. On the other hand, the sometimes quite different kurtosis may be a good indicator for the fault condition.

8.4.4 Frequency Domain

In addition to time-domain analysis, frequency-domain analysis provides a complementary view of the sensor signals, capturing periodic behaviors and spectral characteristics that may not be evident in the raw signal. The sensor signals from $sensor_1$ and $sensor_2$ were transformed to the frequency domain using the Fast Fourier Transform (FFT), as implemented in Harris et al. (n.d.) (see Listing 8), which computes the Discrete Fourier Transform (DFT) efficiently. The DFT of a discrete signal x_n of length N is given by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i \frac{kn}{N}}, \quad k = 0, 1, \ldots, N-1 \tag{14}$$

where X_k represents the magnitude and phase of the frequency component at index k. This transformation reveals the dominant spectral content of the signal, which is often characteristic of specific fault types. Since the raw sensor signals were sampled at a fixed interval of $T_s = 200\,\mu s$, corresponding to a sampling frequency of $f_s = \frac{1}{T_s} = 5000\,\text{Hz}$, the frequency domain analysis is limited to the Nyquist range of [0, 2500] Hz. Any spectral content beyond this limit is subject to aliasing and is excluded from further interpretation.

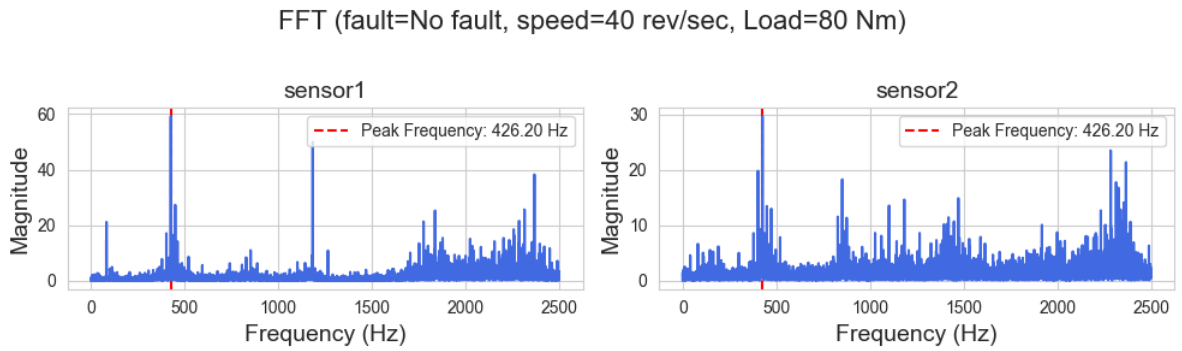


Figure 17: Frequency domain representation of the sensor signals under no fault condition

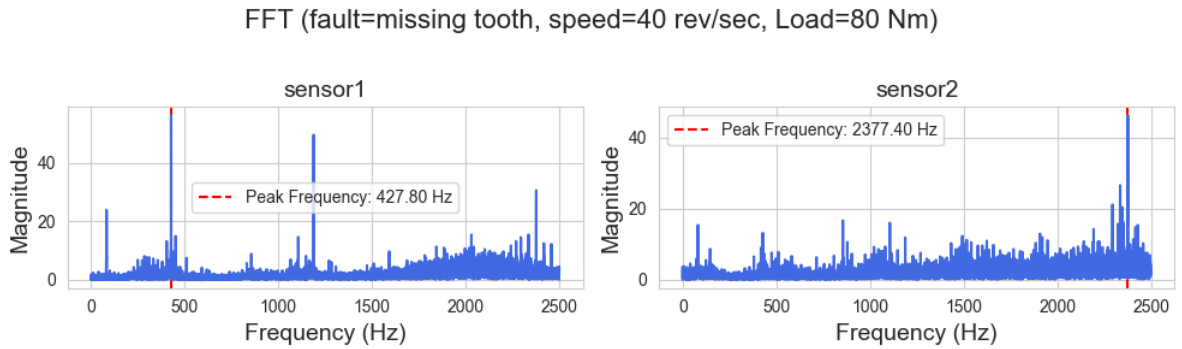


Figure 18: Frequency domain representation of the sensor signals under missing tooth fault condition

As shown in Figure 17 and Figure 18, the frequency domain representation of the sensor signals reveals several prominent peaks at characteristic frequencies. These peaks reflect the underlying vibration modes of the system, some of which may be influenced by the presence or absence of mechanical faults. One practical approach is to extract frequency-domain features, such as the dominant peak frequencies, their magnitudes, and the energy distribution within predefined frequency bands. However, it is important to note that the FFT assumes stationarity of the signal over the analysis window. This can limit its ability to capture transient behaviors. In such cases time-frequency methods like the wavelet transform may be more appropriate, as stated in Brunton and Kutz (2017, p. 84), 'In time-frequency analysis, there is a fundamental uncertainty principle that limits the ability to simultaneously attain high resolution in both the time and frequency domains. In the extreme limit, a time series is perfectly resolved in time, but provides no information about frequency content, and the Fourier transform perfectly resolves frequency content, but provides no information about when in time these frequencies occur. The spectrogram resolves both time and frequency information, but with lower resolution in each domain'. Having this in mind, one could use the time-frequency domain to extract some further features. But within the scope of this thesis, the time-frequency domain is not further investigated. Further more one can inspect the power spectral density (PSD) which is the normalized square magnitude of the FFT which indicates how much power the signal contains at each frequency (Brunton and Kutz 2017, p. 70).

$$\text{PSD}(f) = \frac{1}{N f_s} |X(f)|^2 \tag{15}$$

where:

$$\begin{aligned} X(f) &= \text{FFT of the signal } x[n] \\ N &= \text{number of samples} \\ f_s &= \text{sampling frequency (Hz)} \\ |X(f)|^2 &= \text{power at frequency } f \end{aligned}$$

The PSD as shown in Figure 19 was computed using welch's methods from Virtanen et al. (2020) as shown in Listing 9.

Figure 19: Power Spectral Density (PSD) of the sensor signals under different fault conditions

The PSD shows that the power is distributed over a wide range of frequencies, with a few peaks at specific frequencies.

8.4.5 Feature Extraction

After analyzing both the time and frequency domains, features were extracted from the time series data using a sliding window approach, as shown in Listing 1. The window size was set to 256 samples, and the step size to 128 samples.

```
df_window, df_labels = sliding_window_features(df=df, window_size=256, step_size=128)
y, fault_code = pd.factorize(true_labels['fault'])
```

Listing 1: Feature extraction over sliding window

For a detailed description of the feature extraction process, see Bigler (2025b). To evaluate the relevance of each feature, MI was computed between each feature and the fault condition, as shown in Listing 2.

```
mutual_info = mutual_info_classif(X, y, discrete_features=False)
feature_importance = pd.Series(mutual_info, index=X.columns)
```

Listing 2: Mutual Information (MI)

The resulting MI scores are summarized in Table 1, with the full list provided in Table 4.

Feature	**MI Score**
sensor1_mean	0.9236
sensor1_rms	0.9112
sensor2_rms	0.7640
sensor2_mean	0.7639
sensor2_ptp	0.6327
...	
speed	0.0056
load	0.0000

Table 1: Top 5 and bottom 2 features by mutual information with the target class

As shown in Table 1, the top five features with the highest relevance to the fault conditions include the mean, RMS, and peak-to-peak values of the sensor signals (see Listing 3).

```
sensor1_rms = (df['sensor1']**2).mean()**0.5
sensor1_ptp = df['sensor1'].max() - df['sensor1'].min()
sensor2_mean = df['sensor2'].mean()
```

Listing 3: Feature extraction of the top 5 features

The bottom two features, speed and load, have a very low MI score, indicating that they are not relevant for the fault condition prediction. Logically, this makes sense, as the speed and load are mostly constant inside a window, and thus do not provide much information about the fault condition. To reduce the dimensionality of the feature space, one could consider removing the five lowest-ranked MI features or excluding all features with an MI score below a defined threshold.

In order to lower the feature space, the correlation between the features was analyzed using the Pearson correlation coefficient, which is defined as:

$$\text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{16}$$

where:

$$\begin{aligned} \text{cov}(X, Y) &= \text{covariance of } X \text{ and } Y \\ \sigma_X &= \text{standard deviation of } X \\ \sigma_Y &= \text{standard deviation of } Y \end{aligned}$$

The correlation matrix was computed using the `corr()` method from team (2020), as shown in Listing 4.

```
correlation_matrix = X.corr()
correlated_features, correlated_pairs = check_correlation(correlation_matrix=correlation_matrix,
    threshold=0.8)
X = X.drop(correlated_features, axis=1)
```

Listing 4: Correlation matrix of extracted features

Features with a correlation coefficient above 0.8 were removed leading to the correlation matrix as shown in Figure 24.

If the feature space is still too large, one could apply dimensionality reduction techniques such as PCA to further reduce the feature space to n dimensions which can be used for visualization of the feature space. PCA performs a linear mapping of the data to a lower-dimensional space in such a way that the variance of the data in the low-dimensional representation is maximized. This is typically done by solving an eigenvalue decomposition of the covariance matrix:

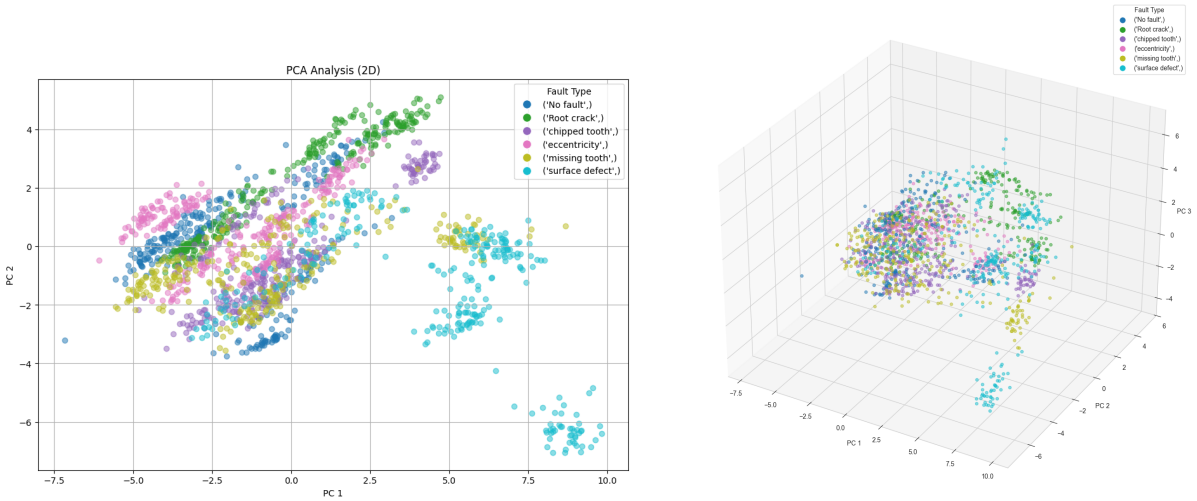
$$X^\top X v = \lambda v \tag{17}$$

where v is a principal component and λ is the corresponding eigenvalue. The PCA was computed using the `PCA` class from Pedregosa et al. (2011), as shown in Listing 5.

```
df_window, df_labels = sliding_window_features(df=df, window_size=256, step_size=128)
scaler = StandardScaler()
df_window_norm = scaler.fit_transform(df_window)
pca = PCA(n_components=2)
pca_components = pca.fit_transform(df_window_norm)
```

Listing 5: Principal Component Analysis (PCA)

In Figure 20 the principal components of the features are shown in a 2D and 3D representation.



(a) PCA 2D representation of the features

(b) PCA 3D representation of the features

Figure 20: Principal Component Analysis (PCA) of the features

Visually, it can be observed that the features are not completely separated in the 2D space, while the 3D representation makes it even harder to identify patterns. In order to inspect the principal components, one can examine the features contributions to each principal component, as shown in Listing 10 and the explained variance of each principal component, as shown in Table 2.

Principal Component	**Top 3 Features**	**Explained Variance (%)**
PC1	sensor1_spectral_centroid, sensor2_ptp, sensor1_std	34.8
PC2	sensor1_mean, sensor2_mean, sensor2_rms	13.4
PC3	sensor1_top_freq_1, sensor1_top_freq_2, sensor1_peak_power	9.7

Table 2: Top 3 contributing features per principal component and their explained variance ratios

Interestingly, when comparing the principal components to the features ranked by Mutual information (see Table 1), it becomes apparent that some of the top PCA features do not perfectly overlap with the top MI-ranked features. This is expected, as PCA is an unsupervised technique that identifies directions of maximum variance in the data, regardless of class labels, while MI measures how informative each feature is with respect to the target labels. As a result, PCA captures dominant structural patterns in the feature space, whereas MI highlights features that are most predictive of the fault condition.

8.5 Fault Detection

After the feature extraction, the next step is to train a model to detect the fault condition. The following classifiers were evaluated: Random Forest, Multinomial Logistic Regression, Support Vector Machine (SVM), neural network (NN), LSTM, and CNN as shown in Table 3.

Model	Ver.	Win / Step	Hyperparams.	Precision	Recall	Accuracy	F1-Score
Random Forest architecture: Listing 12 input: extracted features	v0.0.8	256 / 128	normalized = False n_estimators = 200 solver = 'lbfgs' penalty = 'l2' n_features = sqrt(n_features)	98.29 %	98.29 %	98.29 %	98.29 %
Multinomial Logistic Regression architecture: Listing 13 input: extracted features	v0.0.4	256 / 128	normalized = True solver = 'lbfgs' penalty = 'l2' C = 1.0 max_iter = 10000	77.37 %	76.92 %	76.95 %	76.92 %
SVM architecture: Listing 14 input: extracted features	v0.0.6	256 / 128	normalized = True kernel = 'rbf' C = 1.0 gamma = 'scale'	93.37 %	92.87 %	92.87 %	92.97 %
NN architecture: Listing 15 input: extracted features	v0.0.1	256 / 128	normalized = True lr = 0.001 weight_decay (L2) = 1e-3 batch_size = 128 optimizer = Adam loss = CrossEntropyLoss	97.73 %	97.72 %	97.72 %	97.72 %
LSTM architecture: Listing 16 input: raw signals	v0.0.5 v0.0.7	256 / 128 64 / 32	normalized = True lr = 0.001 weight_decay (L2) = 1e-3 batch_size = 128 optimizer = Adam loss = CrossEntropyLoss	42.16 % 82.55 %	43.01 % 82.53 %	43.01 % 82.53 %	37.37 % 82.53 %
CNN architecture: (Listing 11) input: raw signals	v0.0.1	256 / 128	normalized = True lr = 0.001 weight_decay (L2) = 1e-3 batch_size = 128 optimizer = Adam loss = CrossEntropyLoss	96.85 %	96.51 %	96.51 %	96.51 %

Table 3: Model comparison with hyperparameters and performance metrics, win / step indicates the number of samples per window and the step size

As shown in Table 3, the Random Forest model achieved the highest overall performance, with a precision, recall, accuracy, and F1-score of approximately 98.3%. This was followed by the NN (97.7%) and the CNN (96.5%). While the LSTM and CNN models were trained directly on the raw sensor signals ($sensor_1$ and $sensor_2$), the other models used the extracted features described in subsubsection 8.4.5. The initial LSTM model with a window size of 256 samples and a step size of 128 samples underperformed (43%), likely due to the limited amount of training data. When the window size was reduced to 64 samples, thereby increasing the number of training data, the performance improved significantly to 82.5%. This indicates that, given sufficient training data, the LSTM was able to learn relevant temporal patterns from the raw sensor signals. The CNN model, capable of learning spatial patterns directly from raw input, achieved a competitive performance (96.5%) despite not relying on handcrafted features. This demonstrates the CNN's ability to learn useful representations from time-series sensor data.

8.6 Conclusion

As shown in Table 3, the CNN and lstm methods (representation-based methods described in subsubsection 6.3.3) were able to extract meaningful features directly from the raw sensor signals. In contrast, manual feature extraction, as discussed in subsubsection 8.4.5, requires substantial domain knowledge in signal processing and can be complex and time-consuming. Thus, the ability to automatically extract meaningful features is highly beneficial. Since this is more of an academic example due to the fact that the dataset is labeled and many signals under faulty conditions were available, these simple models may not be applicable in real-world scenarios, where the data is most likely unlabeled and both anomalies and faulty conditions are rare. In such cases one might try some more advanced methods as introduced in section 6 or section 7.

9 System Design and Implementation

In this chapter, the software architecture of phase two Figure 3 and its software implementation (changes or extensions from Bigler (2025a)) are described based on the prior knowledge from Bigler (2025a).

9.1 Device to Hub Communication and Hierarchical Storage

To logically group sensors into a system, such as a conveyor belt with multiple sensors (vibration, temperature, current, etc.) the gRPC protobuf schema Listing 17 was extended by introducing a *path* variable. This variable is used to identify the associated system. Whereby the hub receives data sent by devices over gRPC and stores it in the corresponding database as shown in Figure 21.

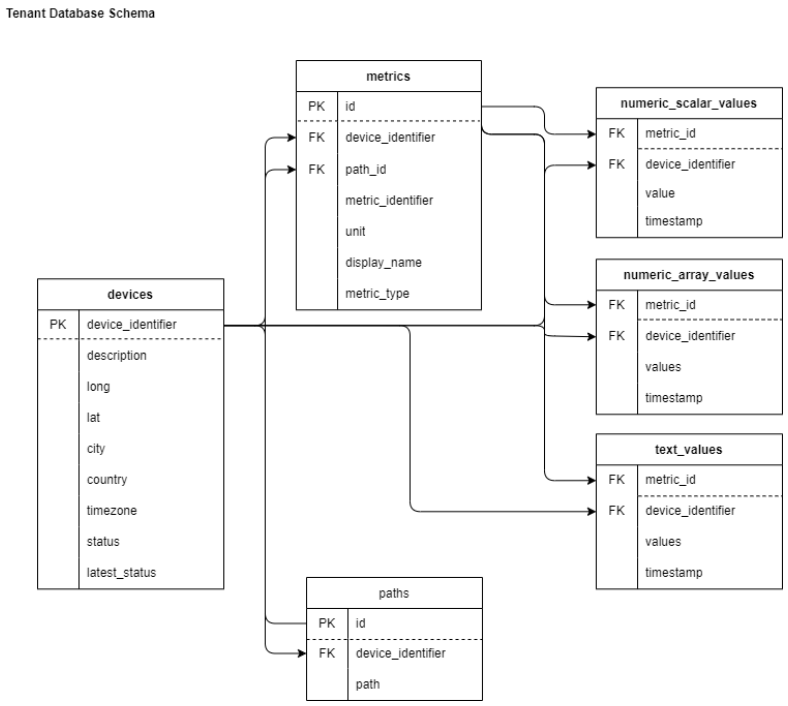


Figure 21: Database schema

The *path* variable is stored using the PostgreSQL *ltree* data type, which supports hierarchical, tree-like data structures. This enables querying based on hierarchical relationships. For example, a sensor with the path *conveyor.motor.temperature* is part of the *conveyor.motor* system. Using *ltree* operators, one can query all subcomponents of *conveyor.motor* as follows:

```
select * from numeric_scalar_values
where path <@ 'conveyor.motor';
```

Listing 6: Querying all sensors under a specific subsystem using *ltree*

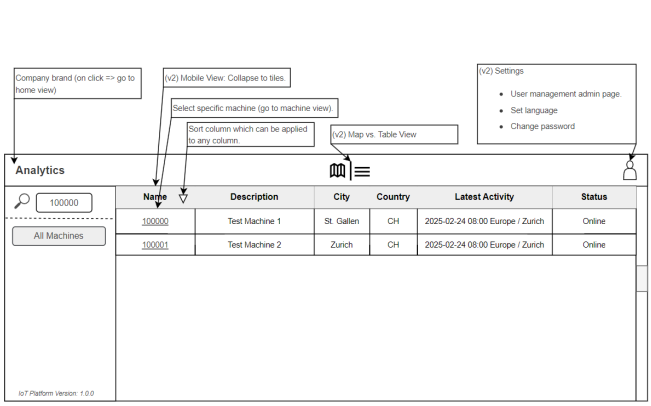
This query returns all sensors under the *conveyor.motor* hierarchy, e.g. temperature, vibration and so on.

9.2 Analytics Frontend and Backend

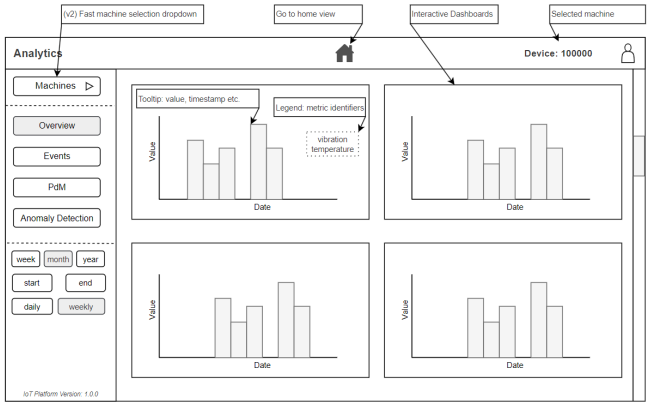
The distributed IoT platform is designed with a clear separation of concerns between data processing and data visualization. It is split into two major components: the *analytics API* (backend) and the *analytics frontend* UI. The **analytics API** is responsible for retrieving, processing, and analyzing IoT data. It exposes a GraphQL interface through which the frontend can request data. Centralizing this logic on the backend ensures consistent results across different clients and allows for more efficient computation near the data storage layer. In contrast, the **analytics frontend** focuses solely on data presentation and interaction. This architecture promotes several advantages such as separation of concerns, scalability, reusability and maintainability.

9.2.1 Design and Implementation

The analytics frontend is a web application written in Typescript using the React library (*React* 2025) and Material UI components (*Material UI: React components for faster and easier web development* 2024). It is designed to visualize the distributed IoT platform's analyzed data as shown in Figure 22. Whereby the design is extended to future capabilities (marked with *v2*) in order to give an outlook on future features.



(a) Home view with an overview of all devices



(b) Device specific view

Figure 22: Analytics UI

The home view Figure 22 provides an overview of all devices, their status, timezone and so on. The device table supports sorting via column headers, and a search bar enables filtering by device name. To access detailed information about a specific device, users can click on an entry in the table. This navigates to the device-specific view (Figure 22b), which presents in-depth metrics and model predictions e.g. anomaly detection or predictive maintenance, if available. Users will be able to select the data time range and aggregation level (planned for *v2*) using the interactive buttons in the lower left corner. Figures of the implemented UI can be found in the appendix (subsection D.1).

React, implemented as a single-page application uses the *react-router-dom* library for client-side routing and navigation between views. The frontend queries the analytics API via GraphQL, allowing to fetch only the necessary data for each component (described in more detail in subsection 9.3). Visualization is handled through modular and reusable chart components built on *D3* (2024). For example, time-series metrics such as vibration or temperature are displayed using the same D3 line charts, only with different data and configurations. These visualization components which are decoupled from data logic, enabling flexibility and reusability.

9.3 Analytics API

As stated in subsection 9.2, the analytics API is responsible for retrieving, processing, and analyzing IoT data. The API is implemented using *Uvicorn an ASGI web server for Python* (2025) as the Asynchronous Server Gateway Interface (ASGI) server, *FastAPI* (2025) as the application framework, and *Strawberry GraphQL library* (2025) for schema definition and query execution. Using GraphQL comes with several advantages, such as:

- **Client-specified data retrieval:** Clients can request only the specific fields they need, thereby reducing over-fetching.
- **Single endpoint for all data:** A unified GraphQL endpoint simplifies the API surface and removes the need for multiple endpoints.
- **Strongly typed schema:** The API schema is explicitly defined and type-safe, which improves tooling support, validation, and documentation.
- **Support for real-time capabilities:** GraphQL subscriptions support real-time data updates (not implemented yet).

However, GraphQL also introduces several challenges:

- **Increased server complexity:** Building and maintaining a GraphQL schema and resolvers requires more upfront design and implementation effort compared to basic REST endpoints.
- **Performance risks from complex queries:** Clients can construct deeply nested or costly queries that may impact backend performance unless they are properly restricted.

Despite these challenges, the benefits of GraphQL make it a suitable choice for the analytics API.

9.4 Schema Design

In GraphQL there is one main endpoint `/graphql`, which can be queried with different operations. The schema is defined with different queries e.g. `devices`, `numericScalar` and `numericScalarModel`. An example query for the `numericScalarModel` field is provided in Figure 23.

```
query MyQuery {
  numericScalarModel(
    body: {
      tenantIdentifier: "100000",
      deviceIdentifier: "000001",
      start: "2025-04-28",
      end: "2025-04-30",
      metricIdentifier: ["vibration.gear.x_axis",
                         "vibration.gear.y_axis"],
      model: {
        name: "gear_vibration_cnn_w_256",
        modelType: PYTORCH,
        windowSize: 128,
        version: "2"
      }
      # path: "",
      # aggregation: AGGREGATION,
      # grouping: GROUPING
    }
  ) {
    deviceIdentifier
    metricIdentifier
    unit
    model {
      name
      predicted
      probability
    }
    values {
      timestampLocal
      value
    }
  }
}

```

(a) GraphQL Request


```
{
  "data": {
    "numericScalarModel": [
      {
        "deviceIdentifier": "000001",
        "metricIdentifier": "vibration.gear1.x_axis"
    ,
        "unit": "",
        "model": {
          "name": "gear_vibration_cnn_w_256",
          "predicted": "No fault",
          "probability": 0.9855
        },
        "values": [
          {
            "timestampLocal": "2025-04-29T11:22:15.7
    ",
            "value": 2.5185
          }
        ]
      },
      {
        "deviceIdentifier": "000001",
        "metricIdentifier": "vibration.gear1.y_axis"
    ,
        "unit": "",
        "model": {
          "name": "gear_vibration_cnn_w_256",
          "predicted": "No fault",
          "probability": 0.9855
        },
        "values": [
          {
            "timestampLocal": "2025-04-29T11:22:15.7
    ",
            "value": 2.4254
          }
        ]
      }
    ]
  }
}

```


(b) GraphQL Response

Figure 23: GraphQL query and response for numeric scalar model prediction

The `numericScalarModel` query retrieves both model predictions and time series values for specified metrics. Instead of listing each metric individually, the optional *path* parameter can be used to retrieve all metrics associated with a particular subsystem, e.g. *path: 'conveyor.motor1'* retrieves all relevant metrics under that system namespace. The *model* field specifies which model to use for inference, referencing metadata such as the model name, type, version, and window size. Models are retrieved from a model registry managed by an `mlflow` server (*mlflow* 2025), hosted within a containerized environment. The system follows an online prediction paradigm (Model-as-a-Service), where the model is dynamically loaded at inference time, with local caching enabled. This design ensures that the latest data for the model prediction is used for each request, which offers up-to-date predictions at the cost of slightly increased response latency. The `numericScalar` query is similar to `numericScalarModel`, without the model prediction, but with the option of a *analysis* parameter (optional) to specify the type of analysis to be performed. Furthermore, the `devices` query retrieves a list of all devices and their status (see Listing 19).

9.5 Conclusion

Using GraphQL as the foundation of the analytics API enables flexible data retrieval, allowing clients to request exactly the information they need (no over or under fetching). However, this flexibility introduces additional complexity on the backend, particularly in managing a robust, future-proof schema. The GraphQL schema is a central component of the system architecture. Designing it to accommodate both current and evolving requirements is challenging but essential for long-term scalability. A clear separation between the analytics API and the analytics frontend has proven highly effective. This architectural decision improves scalability, encourages reusability, and ensures a clean separation of responsibilities between data processing and visualization layers. Looking ahead, an area for improvement could be the introduction of asynchronous batch prediction. This would allow the system to process larger inference workloads more efficiently, reducing latency and improving responsiveness in high-demand scenarios.

10 Conclusion and Outlook

This work lays a solid foundation for a scalable and modular IoT platform, establishing key architectural decisions necessary for future expansion. While the envisioned target architecture remains a long-term goal, the current implementation demonstrates clear progress toward a maintainable and extensible system.

A focus was placed on the encapsulation of data processing, analytics, and visualization, adhering to the principle of separation of concerns. This modularity promotes scalability, maintainability, and reusability across various components of the platform. The adoption of GraphQL as a flexible API for data access and manipulation enables efficient data retrieval, avoiding issues of overfetching or underfetching. However, this choice also introduces complexity in the initial setup and requires careful schema design and API governance.

Predictive Maintenance offers substantial benefits, such as reducing unexpected machine downtime, improving system availability, and lowering overall costs. Nevertheless, implementing PdM is nontrivial and demands sufficient data for meaningful insights. Data-driven approaches, particularly deep learning methods, show promise in modeling complex systems without requiring extensive domain or physical system knowledge. Predictive Maintenance in scenarios with sparse time-to-failure data and unlabeled datasets poses additional challenges. In such cases, unsupervised or self-supervised techniques, such as transfer learning, degradation modeling with learned thresholds, or ordinal regression, to name a few, are necessary to enable robust detection and forecasting. The anomaly detection and PdM landscape is broad, encompassing numerous techniques with varying assumptions and trade-offs. No universal solution exists, instead selecting the right method for a given use case and dataset is critical to achieving meaningful results. As an evolving field, PdM and anomaly detection continue to see advances driven by improvements in data availability, machine learning methods, and computational tools. Small to medium-sized businesses in the mechanical engineering industry could benefit greatly from general-purpose SaaS solutions that abstract away the underlying complexity of these methods.

Finally, this thesis represents only an initial exploration into the domain and could be further investigated in future work.

Abbreviations

AE autoencoder

API application programming interface

ASGI Asynchronous Server Gateway Interface

BiLSTM bidirectional long short-term memory

BLM laboratory-tested bearing

CI Conditional Indicator

CM condition monitoring

CNN Convolutional Neural Network

DFT Discrete Fourier Transform

DI Degradation Indicator

EOC environmental and operating conditions

FFT Fast Fourier Transform

ftnn feature-based transfer neural network

GAT Graph Attention Network

GCN Graph Convolutional Network

GNN Graph Neural Network

GRU Gated Recurrent Unit

HI Health Index

HTM Hierarchical Temporal Memory

IoT Internet of Things

IPC industrial PC

LSTM Long Short-Term Memory

MI mutual information

MTS multivariate time series

MTSAD multivariate time series anomaly detection

MVP minimum viable product

NN neural network

OSA-CBM Open System Architecture for Condition Based Maintenance

PCA Principal Component Analysis

PDF probability density function

PdM predictive Maintenance

PM Preventive Maintenance

PSD power spectral density

RCA Root Cause Analysis

RM Reactive Maintenance

RMS root mean square

RUL Remaining Useful Life

SaaS Software as a Service

SMBs small to medium-sized businesses

SVM Support Vector Machine

TCN Temporal Convolutional Network

TSAD time series anomaly detection

UI user interface

UTS univariate time series

UTSAD univariate time series anomaly detection

VAE Variatonal Autoencoder

Glossary

ADS Automation Device Specification is a communication protocol developed by Beckhoff for data exchange between devices and software in automation technology. 5

API An application programming interface is a set of rules and protocols that allows different software applications to communicate with each other. 4

edge device An edge device is a hardware component or device that serves as an interface between a local network and a cloud environment. This device is responsible for collecting, processing and sometimes storing data close to the source of data generation, typically at the "edge" of the network. 5

GraphQL GraphQL is a data query and manipulation language for APIs and a runtime for executing those queries, developed by Facebook in 2012. 6, 31

gRPC gRPC is a remote procedure call framework developed by Google. It uses HTTP/2 for transport, Protocol Buffers as the interface description language and provides features such as authentication, load balancing and more. 5, 31

OPC UA Open Platform Communications Unified Architecture is a machine-to-machine communication protocol for industrial automation developed by the OPC Foundation. 5

List of Figures

List of Tables

Listings

References

Albuquerque Filho, Jose Edson De et al. (2022). "A Review of Neural Networks for Anomaly Detection". In: DOI: 10.1109/ACCESS.2022.3216007. URL: https://ieeexplore.ieee.org/document/9925159.

Bigler, Joshoua (2025a). *Distributed IoT Platform v1.0.0*. Tech. rep. Ostschwiezer Fachhochschule. URL: https://gitlab.com/mse-jbi/project_work/iot/-/blob/main/documents/iot_v1.0.0.pdf?ref_type=heads.

— (2025b). *Mechanical Gear Vibration Implementatoin*. Accessed: 2025-05-24. URL: https://gitlab.com/mse-jbi/project_work/iot/-/tree/main/src_analytics?ref_type=heads.

Brunton, Steven L and J Nathan Kutz (2017). *Data Driven Science & Engineering*. Cambridge University Press.

Cheng, Cheng et al. (2022). "A deep learning-based remaining useful life prediction approach for bearings". In: URL: https://doi.org/10.48550/arXiv.1812.03315.

Cheng, Yiwei et al. (2021). "A convolutional neural network based degradation indicator construction and health prognosis using bidirectional long short-term memory network for rolling bearings". In: DOI: 10.1016/j.aei.2021.101247. URL: https://linkinghub.elsevier.com/retrieve/pii/S1474034621000021.

Çitil, Furkan (2022). *NASA Bearing Dataset RUL Prediction*. Accessed: 2025-05-24. URL: https://www.kaggle.com/code/furkancitil/nasa-bearing-dataset-rul-prediction/notebook.

D3 (2024). Accessed: 2025-05-29. URL: https://d3js.org.

Dao, Hieu, Agni Biswa, and Alin Drimus (2022). *Mechanical Gear Vibration Dataset*. Accessed: 2025-05-24. URL: https://www.kaggle.com/datasets/hieudaotrung/gear-vibration/data.

Darban, Zahra et al. (2024). "Deep Learning for Time Series Anomaly Detection, A Survey". In: URL: https://doi.org/10.1145/3691338.

Deep Learning for Time Series Anomaly Detection Models and Datasets (2025). Accessed: 2025-05-06. URL: https://github.com/zamanzadeh/ts-anomaly-benchmark?tab=readme-ov-file#deep-learning-for-time-series-anomaly-detection-models-and-datasets.

FastAPI (2025). Accessed: 2025-05-30. URL: https://fastapi.tiangolo.com.

Ferreira, Carlos and Gil Gonçalves (2022). "Remaining Useful Life prediction and challenges: A literature review on the use of Machine Learning Methods". In: DOI: 10.1016/j.jmsy.2022.05.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0278612522000796.

Harris, Charles R. et al. (n.d.). "Array programming with NumPy". In: (). DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

Material UI: React components for faster and easier web development (2024). Accessed: 2025-05-29. URL: https://mui.com.

MathWorks (2025). *Three Ways to Estimate Remaining Useful Life for Predictive Maintenance*. Accessed: 2025-05-13. URL: https://ch.mathworks.com/company/technical-articles/three-ways-to-estimate-remaining-useful-life-for-predictive-maintenance.html.

mlflow (2025). Accessed: 2025-05-06. URL: https://mlflow.org/.

Pang, Guansong, Chunhua Shen, and Anton van den Hengel (2019). "Deep Anomaly Detection with Deviation Networks". In: DOI: 10.48550/arXiv.1911.08623. URL: http://arxiv.org/abs/1911.08623.

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

PostgreSQL (2024). Accessed: 2024-12-23. URL: https://www.postgresql.org.

React (2025). Accessed: 2025-05-06. URL: https://react.dev/.

Ross, Brian C. (Feb. 2014). "Mutual Information between Discrete and Continuous Data Sets". In: ISSN: 1932-6203. DOI: 10.1371/journal.pone.0087357.

Serradilla, Oscar, Ekhi Zugasti, and Urko Zurutuza (2020). "Deep learning models for predictive maintenance a survey, comparison, challenges and prospect". In: URL: https://doi.org/10.48550/arXiv.2010.03207.

Si, Xiao-Sheng et al. (2010). "Remaining useful life estimation, A review on the statistical datadriven approaches". In: URL: https://doi.org/10.1016/j.ejor.2010.11.018.

Strawberry GraphQL library (2025). Accessed: 2025-05-30. URL: https://strawberry.rocks.

team, The pandas development (2020). *pandas-dev/pandas: Pandas*. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

Timescale (2024). Accessed: 2024-12-23. URL: https://timescale.com.

Tuli, Shreshth, Giuliano Casale, and Nicholas R. Jennings (2022). "TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data". In: DOI: 10.48550/arXiv.2201.07284. URL: https://arxiv.org/abs/2201.07284.

Tv, Vishnu et al. (2019). "Data-driven Prognostics with Predictive Uncertainty Estimation using Ensemble of Deep Ordinal Regression Models". In: DOI: 10.36001/ijphm.2019.v10i4.2612. URL: https://papers.phmsociety.org/index.php/ijphm/article/view/2612.

Uvicorn an ASGI web server for Python (2025). Accessed: 2025-05-30. URL: https://www.uvicorn.org.

Virtanen, Pauli et al. (2020). *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*. DOI: 10.1038/s41592-019-0686-2.

Yang, Bin et al. (2019). “An intelligent fault diagnosis approach based on transfer learning from laboratory bearings to locomotive bearings”. In: DOI: 10.1016/j.ymssp.2018.12.051. URL: https://linkinghub.elsevier.com/retrieve/pii/S0888327018308367.

Zhu, Tianwen et al. (2024). “A Survey of Predictive Maintenance: Systems, Purposes and Approaches”. In: URL: https://arxiv.org/abs/1912.07383.

A Feature Engineering

A.1 Data Distribution

```python
def distribution(data: pd.Series, ax: plt.axis, title: str = None) -> plt.axis:
    mean = data.mean()
    median = data.median()
    skewness = data.skew()
    kurtosis = data.kurtosis()
    sns.histplot(data, kde=True, stat='density', bins=30, color='skyblue', edgecolor='black', alpha=0.5, ax=ax)
    sns.kdeplot(data, fill=True, alpha=0.3, linewidth=2, ax=ax)
    if title:
        ax.set_title(title, fontsize=11)
    ax.set_xlabel('Value', fontsize=8)
    ax.set_ylabel('Density', fontsize=8)
    stats_text = (f'Mean: {mean:.2f}\n'
                  f'Median: {median:.2f}\n'
                  f'Skewness: {skewness:.2f}\n'
                  f'Kurtosis: {kurtosis:.2f}\n')
    ax.text(0.95,
            0.95,
            stats_text,
            transform=ax.transAxes,
            fontsize=8,
            verticalalignment='top',
            horizontalalignment='right',
            bbox=dict(boxstyle='round', facecolor='white', alpha=0.7))
    return ax


def plot_distributions(df: pd.DataFrame, speed: float, load: float) -> plt:
    faults = df['fault'].unique()
    sensors = ['sensor1', 'sensor2']
    num_faults = len(faults)
    _, axes = plt.subplots(nrows=num_faults, ncols=2, figsize=(12, 2.5 * num_faults))
    for row_index, fault in enumerate(faults):
        for col_idx, sensor in enumerate(sensors):
            data = df[(df['fault'] == fault) & (df['speed'] == speed) & (df['load'] == load)][sensor]
            title = f'{sensor.capitalize()}\nFault: {fault}, {speed} rev/sec, {load} Nm'
            distribution(data, ax=axes[row_index, col_idx], title=title)
    plt.tight_layout()
    return plt
```

Listing 7: Data distribution

A.2 Fast Fourier Transform (FFT)

```python
def plot_fft(df: pd.DataFrame, fault: str, speed: float, figsize=(12, 4)):
    df = df.sort_values(by='timestamp')
    ts = (df['timestamp'].iloc[1] - df['timestamp'].iloc[0]).total_seconds()
    fs = 1 / ts
    for load_value, group in df.groupby('load'):
        fig, axs = plt.subplots(1, 2, figsize=figsize)
        fig.suptitle(f'FFT (fault={fault}, speed={speed} rev/sec, Load={load_value} Nm)', fontsize=14)
        for ax, sensor in zip(axs, ['sensor1', 'sensor2']):
            signal = group[sensor] - np.mean(group[sensor])
            n = len(signal)
            fft_vals = np.fft.fft(signal)
            freqs = np.fft.fftfreq(n, d=1 / fs)
            mask = freqs >= 0
            freqs = freqs[mask]
            magnitude = np.abs(fft_vals[mask])
            max_freq = freqs[np.argmax(magnitude)]
            ax.axvline(max_freq, color='red', linestyle='--', label=f'Peak Frequency: {max_freq:.2f} Hz')
            ax.legend()
            ax.plot(freqs, magnitude, color='royalblue', lw=1.5)
            ax.set_xlabel('Frequency (Hz)', fontsize=11)
            ax.set_ylabel('Magnitude', fontsize=11)
            ax.set_title(sensor, fontsize=11)
            ax.grid(True)
        plt.tight_layout(rect=[0, 0, 1, 0.95])
```

```
    plt.show()
```

Listing 8: Fast Fourier Transform (FFT)

A.3 Power Spectral Density (PSD)

```
def plot_psd(df: pd.DataFrame, speed: float, figsize=(12, 4), sensors: list[str] = None):
  sensors = ['sensor1', 'sensor2'] if sensors is None else sensors
  df_filt = df[df['speed'] == speed]
  if df_filt.empty:
    print(f'No data found for the given speed={speed}')
    return
  loads = sorted(df_filt['load'].unique())
  n_loads = len(loads)
  _, axs = plt.subplots(n_loads, len(sensors), figsize=(figsize[0], figsize[1] * n_loads), sharex
    =True, sharey=True)
  if n_loads == 1:
    axs = axs.reshape(1, -1)
  for i, load in enumerate(loads):
    df_load = df_filt[df_filt['load'] == load]
    for j, sensor in enumerate(sensors):
      ax = axs[i, j]
      for fault, group in df_load.groupby('fault'):
        group = group.sort_values(by='timestamp')
        sensor_signal = group[sensor] - np.mean(group[sensor])
        ts = (group['timestamp'].iloc[1] - group['timestamp'].iloc[0]).total_seconds()
        fs = 1 / ts
        f_psd, psd_values = welch(sensor_signal, fs=fs, nperseg=1024, average='median')
        ax.semilogy(f_psd, psd_values, label=f'Fault: {fault}')
      ax.set_title(f'{sensor} (Load: {load} Nm, Speed: {speed} rev/sec)')
      ax.legend()
      ax.grid(True)
      if i == n_loads - 1:
        ax.set_xlabel('Frequency [Hz]')
      if j == 0:
        ax.set_ylabel('PSD [mm^2/Hz]')
  plt.tight_layout()
  plt.show()
```

Listing 9: Power Spectral Density (PSD)

A.4 Mutual Information (MI) of Features

Feature	MI Score	Feature	MI Score
sensor1_mean	0.9236	sensor1_top_freq_3	0.3092
sensor1_rms	0.9112	sensor1_top_freq_2	0.3056
sensor2_rms	0.7640	sensor2_top_freq_4	0.2908
sensor2_mean	0.7639	sensor1_kurt	0.2895
sensor2_ptp	0.6327	sensor1_top_freq_4	0.2856
sensor1_ptp	0.6152	sensor2_top_freq_5	0.2806
sensor2_peak_freq	0.4783	sensor1_top_freq_1	0.2708
sensor2_std	0.4514	sensor1_top_freq_5	0.2380
sensor2_energy	0.4473	sensor2_spectral_bandwidth	0.2198
sensor1_peak_freq	0.4440	sensor2_spectral_centroid	0.1895
sensor1_std	0.4316	sensor1_spectral_bandwidth	0.1633
sensor1_energy	0.4243	sensor2_skew	0.1370
sensor1_peak_power	0.3943	sensor1_spectral_centroid	0.1349
sensor2_peak_power	0.3789	sensor1_skew	0.1316
sensor2_kurt	0.3419	speed	0.0056
sensor2_top_freq_1	0.3367	load	0.0000
sensor2_top_freq_3	0.3291		
sensor2_top_freq_2	0.3136		

Table 4: Mutual Information (MI) scores between features and the target class

A.5 Feature Correlation Matrix

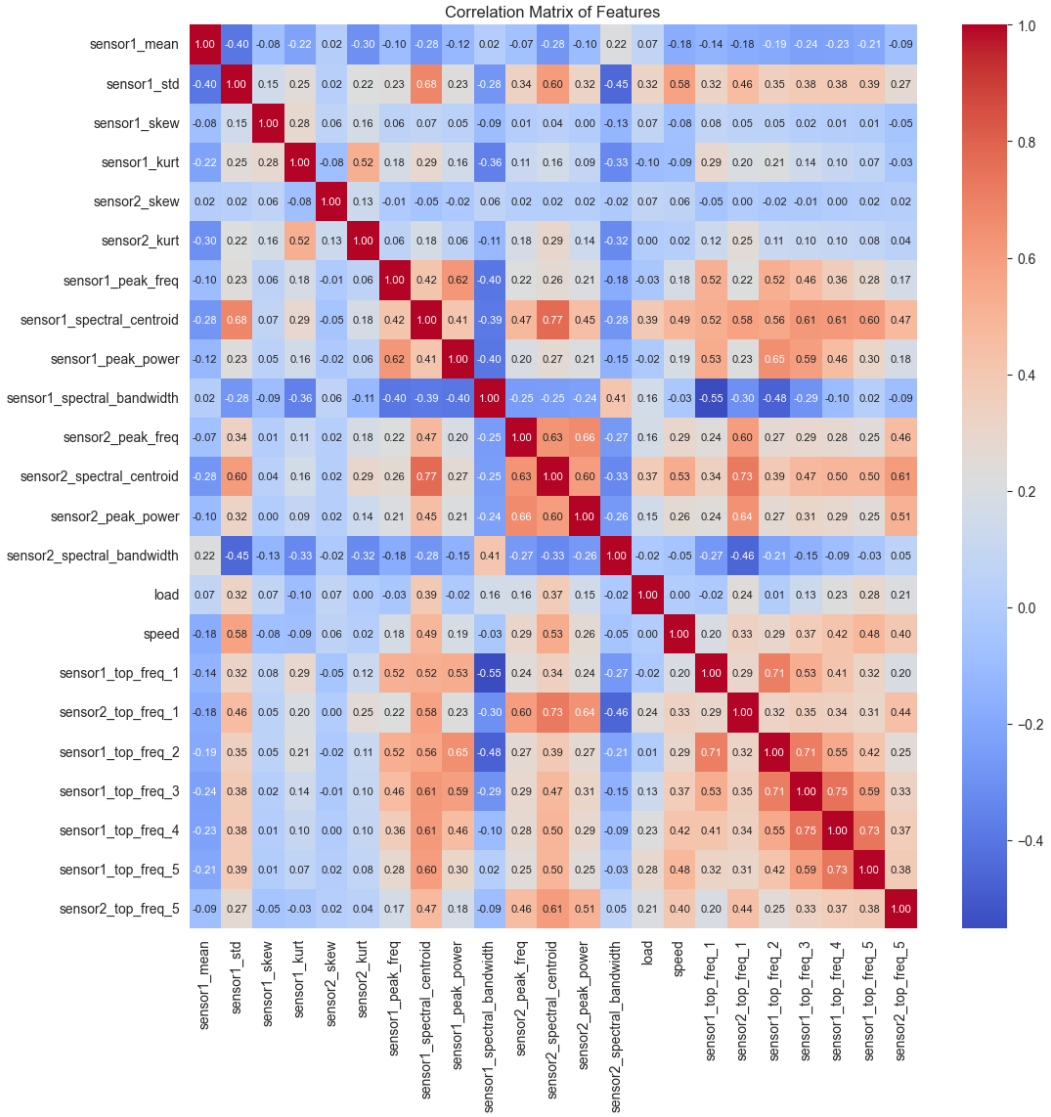


Figure 24: Feature correlation matrix of the extracted features

A.6 Principal components inspection

```
pca = PCA(n_components=3)
pca_components = pca.fit_transform(df_window_norm)
pca_components = pca.components_
feature_names = list(df_window_norm.columns)
pca_df = pd.DataFrame(pca_components, columns=feature_names, index=['PC1', 'PC2', 'PC3'])
top_features = {}
for i, pc in enumerate(pca_components):
  top_features[f'PC{i + 1}'] = np.argsort(np.abs(pc))[-3:]
  print(f'Top features for PC{i + 1}: {[feature_names[j] for j in top_features[f"PC{i + 1}"]]}')
print('Explained Variance Ratio:')
for i, ratio in enumerate(pca.explained_variance_ratio_):
  print(f'PC{i + 1}: {ratio*100:.1f}%')
```

Listing 10: Principal components inspection

B Gear Fault Detection Models

```
import torch
import torch.nn as nn

class CnnClassifier(nn.Module):
  ''' Convolutional Neural Network for time series classification.'''
```

```
  def __init__(self, in_channels: int, out_features: int, input_length: int):
    super().__init__()
    self.features = nn.Sequential(
        nn.Conv1d(in_channels, 64, kernel_size=5, padding=2),
        nn.BatchNorm1d(64),
        nn.ReLU(),
        nn.MaxPool1d(2),
        nn.Dropout(0.2),
        nn.Conv1d(64, 128, kernel_size=3, padding=1),
        nn.BatchNorm1d(128),
        nn.ReLU(),
        nn.MaxPool1d(2),
        nn.Dropout(0.2),
        nn.Conv1d(128, 256, kernel_size=3, padding=1),
        nn.BatchNorm1d(256),
        nn.ReLU(),
        nn.Dropout(0.2),
        nn.Conv1d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(),
        nn.AdaptiveAvgPool1d(1),
    )
    self.classifier = nn.Sequential(nn.Flatten(), nn.Dropout(0.3), nn.Linear(256, out_features))

  def forward(self, x):
    x = self.features(x)
    return self.classifier(x)
```

Listing 11: CNN architecture

```
from sklearn.ensemble import RandomForestClassifier
rfc_settings = {
    'oob_score': True,
    'max_features': np.sqrt(n_features) / n_features,
    'random_state': 1,
    'n_estimators': 200
}
rfc = RandomForestClassifier().set_params(**rfc_settings)
```

Listing 12: Random forest architecture

```
from sklearn.linear_model import LogisticRegression
mlr_model = LogisticRegression(max_iter=10000, random_state=42)
mlr_metadata = {
    'solver': mlr_model.solver,
    'max_iter': mlr_model.max_iter,
    'penalty': mlr_model.penalty,
    'C': mlr_model.C,
    'n_features': n_features,
    'n_classes': len(np.unique(y)),
    'n_samples': len(X_train)
}
```

Listing 13: Multinomial Logistic Regression architecture

```
from sklearn.svm import SVC
svm_classifier = SVC(kernel='rbf', random_state=42, C=1.0)
```

Listing 14: SVM architecture

```
class NnClassifier(nn.Module):
  ''' Fully connected neural network for time series classification.'''

  def __init__(self, input_dim: int, output_dim: int, dropout: float = 0.3):
    super().__init__()
    self.model = nn.Sequential(
        nn.Linear(input_dim, 128),
        nn.ReLU(),
        nn.Dropout(dropout),
        nn.Linear(128, 128),
        nn.ReLU(),
        nn.Dropout(dropout),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Dropout(dropout),
        nn.Linear(64, output_dim),
    )
```

```
18
19    def forward(self, x: torch.Tensor) -> torch.Tensor:
20      return self.model(x)
```

Listing 15: NN architecture

```
1  class LstmClassifier(nn.Module):
2    def __init__(self, input_size: int, hidden_size: int, num_layers: int, out_features: int):
3      super().__init__()
4      self.lstm = nn.LSTM(input_size=input_size,
5                          hidden_size=hidden_size,
6                          num_layers=num_layers,
7                          batch_first=True,
8                          bidirectional=True)
9      self.dropout = nn.Dropout(0.3)
10     self.fc = nn.Linear(hidden_size * 2, out_features)
11
12   def forward(self, x):
13     out, _ = self.lstm(x)
14     out = out[:, -1, :]
15     out = self.dropout(out)
16     return self.fc(out)
```

Listing 16: LSTM architecture

C Protobuf Schema

```
1  syntax = "proto3";
2
3  import "google/protobuf/timestamp.proto";
4  import "google/protobuf/empty.proto";
5
6  service Hub {
7    rpc SendNumericScalarValues(stream NumericScalarValues) returns (google.protobuf.Empty) {}
8    rpc SendDeviceStatus(DeviceStatus) returns (google.protobuf.Empty) {}
9  }
10
11 enum Status {
12   SUCCESS = 0;
13   FAILED = 1;
14   PENDING = 2;
15 }
16
17
18 message DeviceStatus{
19   string tenant_identifier = 1;
20   string device_identifier = 2;
21   int32  status = 3;
22   google.protobuf.Timestamp timestamp = 4;
23 }
24
25 message NumericScalarValues {
26   string tenant_identifier = 1;
27   string device_identifier = 2;
28   string metric_identifier = 3;
29   string path = 4;
30   double value = 5;
31   string unit = 6;
32   string display_name = 7;
33   google.protobuf.Timestamp timestamp = 8;
34 }
35
36 message Response {
37   Status status = 1;
38   string message = 2;
39 }
```

Listing 17: Hub Protobuf Schema

D System Design and Implementation

D.1 Analytics Frontend

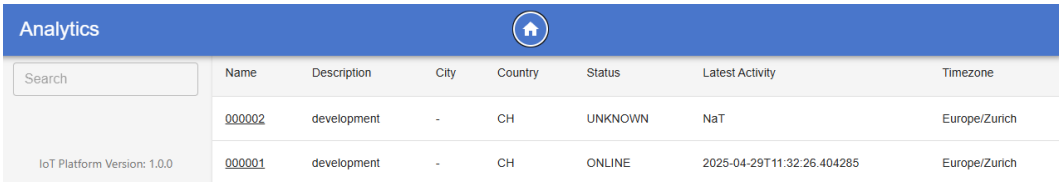


Figure 25: Frontend overview

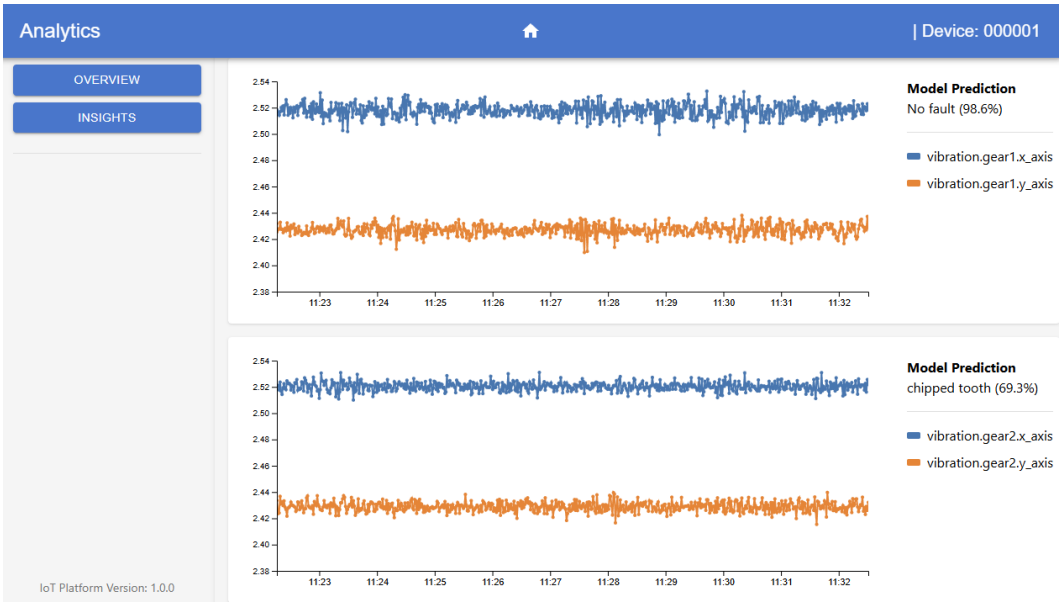


Figure 26: Device specific view

D.2 Analytics API

```
query MyQuery {
  devices(body: {tenantIdentifier: "100000"}) {
    country
    description
    lat
    deviceIdentifier
    latestAliveLocal
    long
    status
    timezone
  }
}
```

Listing 18: GraphQL devices request example

```
{
  "data": {
    "devices": [
      {
        "country": "CH",
        "description": "development",
        "lat": 47.415,
        "deviceIdentifier": "000002",
        "latestAliveLocal": "NaT",
        "long": 9.3469,
        "status": "UNKNOWN",
        "timezone": "Europe/Zurich"
      },
      {
        "country": "CH",
        "description": "development",
        "lat": 47.415,
        "deviceIdentifier": "000001",
        "latestAliveLocal": "2025-04-29T11:32:26.4",
        "long": 9.3469,
        "status": "ONLINE",
        "timezone": "Europe/Zurich"
      }
    ]
  }
}
```

Listing 19: GraphQL devices response example