Eurasian J. Sci. Eng., 2025, 11(3), 321-338
https://doi.org/10.23918/eajse.v11i3p21

Published by Tishk International University
Available at: https://eajse.tiu.edu.iq/

RESEARCH ARTICLE Open Access

# Real-Time Fault Detection for Arduino Sensor Networks Using Lightweight ML Models

Marwa Awni Kamal[1*]

[1] Computer Engineering Department, Faculty of Engineering, Tishk International University, Erbil, Iraq.

**Article History**
*Received: 03.02.2026*
*Revised: 02.05.2026*
*Accepted: 10.05.206*
*Published: 11.05.2026*
*Communicated by: Dr. Bawar Mohammed Faraj*
*Email address:*
*marwa.awni@tiu.edu.iq*
*Corresponding Author*



**Abstract:** In embedded systems based on the Arduino platform, real-time fault detection is difficult because of the small computational power, memory, and power restrictions. Traditional threshold-based methods lack flexibility and fail to detect subtle fault patterns, types of faults like gradual drift or intermittent faults, and most machine learning methods are currently too costly to run on microcontrollers. The objective of the study is to come up with an effective fault detection model that can attain high accuracy at low latency and memory consumption on constrained-resource embedded systems. This article suggests a lightweight time-based fault detection system that uses low-complexity time-domain features and an edge-optimized Bonsai machine learning classifier to perform real-time on-device inference. A controlled experimental setup with simulated fault conditions such as bias, drift, spike anomalies, and signal loss is used to generate multivariate Arduino sensor data, which includes temperature, vibration, current, and voltage data. The data is divided into 50-sample sliding windows. Model optimisation methods, including pruning and fixed-point quantisation, including pruning and fixed-point quantisation, can trim the model down to 1.76 KB and can achieve inference latency as low as 105 µs using hardware on the Arduino-class. The experimental findings indicate that the proposed method has a fault detection rate of 0.991, which is better than the k-Nearest Neighbors (k-NN), both in terms of latency (approximately 1632 $\mu s$) and memory (5686 KB) utilization, but it has a similar performance as a Decision Tree classifier that has an accuracy of 1.000 at a higher cost. The framework achieves a balance between diagnostic accuracy and resource efficiency that can be used in real-time fault detection of a low-cost embedded sensor network. The findings reveal that the suggested Bonsai-based model provides a better accuracy-latency-memory trade-off than traditional machine learning models, and proves to be efficient in detecting faults in real-time on resource-limited embedded systems.
**Keywords:** *Arduino-Based Embedded Systems; Real-Time Fault Detection; Bonsai Algorithm; Lightweight Machine Learning; Sensor Fault Diagnosis; Tiny ML; and Edge AI.*

## 1. Introduction

Embedded systems based on Arduino are ubiquitous in low-budget sensing and monitoring systems in industrial automation, smart environments, and Internet of Things (IoT) systems (1) . These systems constantly receive multivariate sensor data of temperature, vibration, current, and voltage data, which are a priori time-varying and subject to noise, drift, and hardware-induced variance. Sensor fault data that are publicly available and actual data acquired by Arduino devices usually have long normal operating periods between very infrequent, yet critical, faults, such as bias faults, gradual drift, spike faults, and signal loss. The high asymmetry in such datasets and inherent non-stability make it quite difficult to use such data to come up with reliable fault-detecting models, especially when the models are forced to work on microcontroller-based platforms with strict memory, power, and computational constraints.

Traditional fault detection methods within embedded systems are mainly based on hard-coded thresholds, logic, or naive statistical tests like mean deviation and variance limits(2). These techniques are very sensitive to noise and involve hand adjustment of thresholds, which usually differ between sensors, environments, and working conditions. Consequently, conventional methods often produce false alarms due to harmless variability or are unable to sense minor fault trends like slow sensor drift(3). Besides, threshold-based methods are inflexible and non-adaptive to unknown fault behavior, and thus are less effective in large-scale dynamic sensor networks.

Although machine learning approaches have shown good ability to identify faults in a centralized computing environment, direct deployment in Arduino systems is still unfeasible(4). A variety of current methods use computationally expensive methods like deep neural networks or distance-based classifiers, which need large amounts of memory, floating-point operations, and inference time. Such models frequently presuppose cloud-based processing and high-performance edge devices, which cannot be deployed using low-power microcontrollers with small RAM and flash memory(5). Moreover, a few ML-based approaches are only concerned with detection precision, but not the latency of inference, energy usage, or model size, which are major limitations when deploying machines.

In order to overcome these issues, the combination of the flexibility of machine learning and the performance necessary to run on Arduino-type hardware(6) is evidently required. Real-time embedded inference requires lightweight models that can run on low-complexity time-domain features, fixed-point arithmetic, and small decision structures. A fault detection method built on an edge computing approach will allow fault detection on-device without the need to be constantly connected or use cloud services, which increases reliability, latency, and data privacy(7). This kind of framework should be able to balance the trade-offs of detection accuracy and resource efficiency whilst being robust to a variety of faults and operating conditions.

The suggested work is based on a systematic pipeline starting with the acquisition of multivariate sensor data and light preprocessing, noise elimination, normalization, and sliding window segmentation. Time-domain features of low complexity are then obtained to describe sensor behavior in an effective manner. A series of lightweight machine learning models that include Decision Tree, k-NN, and an edge-optimized Bonsai classifier are trained and optimized offline. Pruning and fixed-point quantization are used as model optimization methods to make sure that they are compatible with Arduino memory and power limits. Lastly, the overall performance assessment is done based on the classification metrics, inference latency, memory, and estimated power consumption to determine the best trade-off between the accuracy of fault detection and embedded resource efficiency.

The proposed research will focus on creating a lightweight and efficient fault detection system that can be deployed on Arduino-type embedded systems with a very tight resource limit. The hypothesis is that a Bonsai-based classifier, with low-complexity time-domain features, can attain close to state-of-the-art performance in fault detection with a much lower inference latency and memory cost than traditional machine learning models like k-Nearest Neighbors and Decision Trees. This hypothesis is tested by using a detailed experimental analysis on realistic embedded deployment conditions.

Identify the Key contributions to the proposed work:

1. It presents a new lightweight fault detection model, grounded on low-complexity time-domain characteristics, and an edge-optimized Bonsai classifier, that is able to perform complete on-device inference on Arduino-class embedded systems without the use of cloud services or high-performance edge platforms. As far as is known, this is one of the earliest applications of a Bonsai-based fault detection model that was implemented on resource-constrained microcontroller platforms to operate in real-time.
2. The suggested paradigm uses model optimization methods, such as pruning and fixed-point quantization, to achieve a considerable reduction in memory footprint and inference latency

without sacrificing the fault detection accuracy of the model in other strict embedded requirements.

3. A detailed quantitative analysis shows that the Bonsai model is much better than traditional machine learning models, including Decision Tree and k-Nearest Neighbors, and the current lightweight TinyML-based models in terms of accuracy, inference latency, and memory consumption, which makes the Bonsai model highly applicable to real-time embedded fault detection.

Section II includes an in-depth review of the current fault detection strategies in embedded and Internet-of-Things-based sensor systems, both conventional statistical methods and machine learning models, and their benefits and drawbacks in the resource-constrained setting. Section III outlines the proposed lightweight fault detection framework, which includes the architecture, data preprocessing pipeline, feature extraction pipeline, and optimization of the machine learning pipeline based on edge devices, and real-time prediction on Arduino hardware. Section IV explains the experimental findings made with the help of the presented approach and compares the performance of the suggested approach with that of other lightweight models in terms of accuracy, latency, memory consumption, and energy efficiency. Lastly, the work ends in Section V with a conclusion to sum up the main findings and provides directions for future research, such as adaptive learning, multi-sensor fusion, and implementation on heterogeneous embedded systems.

## 2. Literature Review

The high rates of edge computing and Internet of Things-based embedded systems development have dramatically increased the focus on lightweight and real-time fault detection models that can run on very strict resource constraints. The literature has covered many different techniques, such as neural networks based on TinyML, hybrid deep learning, recurrent networks, and scalable pipelines used in fields like industrial monitoring, smart grids, health, cybersecurity, and agricultural sensing. Although these approaches show good detectability and real-time performance, a majority of them are either specific to sensor modalities, based upon computationally expensive architectures, or require access to a higher-end edge device and cloud services. Consequently, issues with memory footprint, inference latency, energy consumption, and heterogeneous sensor data generalization are not completely tackled. This literature review discusses currently relevant and representative literature to determine these shortcomings and emphasize the necessity of a small, edge-optimized framework of fault detection that can be used in Arduino-class embedded systems.

Suggested a real-time apple disease classification model that can be run on an Arduino edge-defined board with a newly designed lean neural network (8). The objective of the methodology is to minimize the depth of the model and the number of parameters while maintaining the discriminative image-based disease recognition power. A TinyML-based framework to detect vibration faults was proposed and designed to run on hardware with extremely low resources (9). The approach makes use of small deep learning networks and statistical attributes of vibrations to facilitate fault classification within low limits of memory and processing.

Introduced an embedded predictive maintenance system based on TinyML on an ESP32 platform to detect faults in terms of vibration in real-time on industrial equipment(10). Their method combines feature extraction with lightweight neural inference run on the edge device. The most important strength of the study is that it provides an end-to-end demonstration of real-time predictive maintenance without the dependency on the cloud.

It suggested a detection system of Wi-Fi attacks in real-time, but implemented on a Node MCU environment with the help of hybrid deep learning models (11). The technique is a combination of more than one deep architecture to represent the intricate attack patterns in network traffic data. The

key benefit of this work is that it has good detection in cybersecurity usage and actually runs on an embedded Wi-Fi-enabled device.

Created an edge AI-based edifice of dynamic power factor correction with TinyML, blockchain, and IoT technology in managing smart grid optimization (12). The approach combines the use of lightweight machine learning models with distributed ledger technology so as to allow real-time industrial decision-making. The advantage of this work is in the comprehensive approach to the incorporation of new technologies and real-time control.

They were interested in building scalable data pipelines to detect real-time anomalies in industrial IoT sensor networks (13). They focus their work on backend data ingestion, processing pipelines, and scalable analytics as opposed to on-device intelligence. The benefit of this mechanism is that it is appropriate in large-scale industrial applications where there is a large throughput of data.

To preserve the sustainability of data in healthcare datasets, It has suggested an AI-based framework has been suggested that involves data preprocessing, intelligent analytics, and validation processes (14). The methodology is based on data quality, consistency, and long-term utility as opposed to the detection of faults directly. The main value of this work is that it focuses on sustainable and dependable AI-based data management. Park proposed LiReD, a lightweight real-time fault detector system using an LSTM recurrent neural network when computing on the edge (15). The methodology builds upon the ability of LSTMs to model time-dependent fault patterns. The main strength of this work is that it shows the possibility of detecting faults in real-time and at the edge with the help of deep learning, which has been proven early.

Table 1: Comparison Study of the existing models

| Study | Core Architecture | Primary Dataset(s) | Limitation Addressed | Remaining Gap |
|---|---|---|---|---|
| Grujev et al. (2025)[8] | Lightweight Convolutional Neural Network (CNN) on Arduino | Apple leaf image dataset | Reduced model size for edge-level image classification | Not applicable to multivariate sensor fault detection |
| Al-Libawy (2025)[9] | TinyML-based compact deep neural network | Vibration sensor fault dataset | Efficient vibration fault detection on limited hardware | Limited generalization beyond vibration signals |
| Gupta & Shivhare [10](2025) | Embedded TinyML neural network on ESP32 | Industrial vibration sensor dataset | Real-time predictive maintenance without cloud dependency | Hardware-specific design limits portability |
| Moharam et al. (2025)[11] | Hybrid deep learning model on NodeMCU | Wi-Fi traffic attack dataset | Improved accuracy for real-time network attack detection | High computational and energy overhead |
| Gochhayat et al. (2025) | TinyML-based edge AI with blockchain integration | Smart grid electrical parameter dataset | Secure and intelligent real-time grid optimization | Increased system complexity and resource overhead |

| | | | | |
|---|---|---|---|---|
| Akande & Chukwunweike (2024) | Scalable data analytics and anomaly detection pipelines | Industrial IoT sensor streams | Handles large-scale real-time data processing | Dependence on cloud or centralized infrastructure |
| Sudharson et al. (2023) | AI-driven data sustainability framework | Healthcare multi-source datasets | Improves data quality and long-term usability | Not designed for real-time embedded inference |
| Park et al. (2018) | LSTM-based lightweight edge fault detection system | Industrial fault time-series dataset | Captures temporal fault patterns effectively | High memory and computation requirements |

The current fault detection frameworks of embedded and IoT-based systems have an inherent lack of alignment between model complexity and hardware capability, with many of them being based on deep or hybrid architectures that require large amounts of memory, computation, and energy, and thus, they do not use microcontrollers of Arduino size. Threshold-based methods that have been used traditionally, on the contrary, are not adaptive and do not isolate subtle forms of faults like slow drifting or occasional anomalies in noisy sensor settings. The suggested work can help overcome these issues since it presents a minimalistic, edge-reduced fault detection system, which uses simple time-domain characteristics and a small Bonsai-based machine learning model, along with pruning and fixed-point quantization. By means of this approach, accurate real-time fault detection is realized on Arduino hardware with minimal memory consumption, inference in real-time, and power consumption, and the gap between diagnostic reliability and embedded resource efficiency is narrowed.

## 3. Methodology

Figure 1 represents the overall workflow of the proposed real-time fault detection framework produced to use with Arduino-based embedded sensor networks. This is done by first acquiring multivariate sensor data from temperature sensors, vibration sensors, current sensors, and voltage sensors, and then lightweight data pre-processing, consisting of normalization and sliding window segmentation. Time-domain features of low complexity are then obtained as mean, RMS, variance, and peak-to-peak amplitude to describe sensor behavior in an efficient way. The optimized Bonsai-based machine learning model is pruned and quantized and then implemented on the Arduino to perform inference on the device. Fault categorization is also conducted in real-time at the local level, which allows diagnosing abnormal conditions. Identified faults provide alerts via LEDs, serial or wireless interfaces, and the system performance is considered in relation to accuracy, latency, and power usage to ensure compliance with embedded resource limitations.

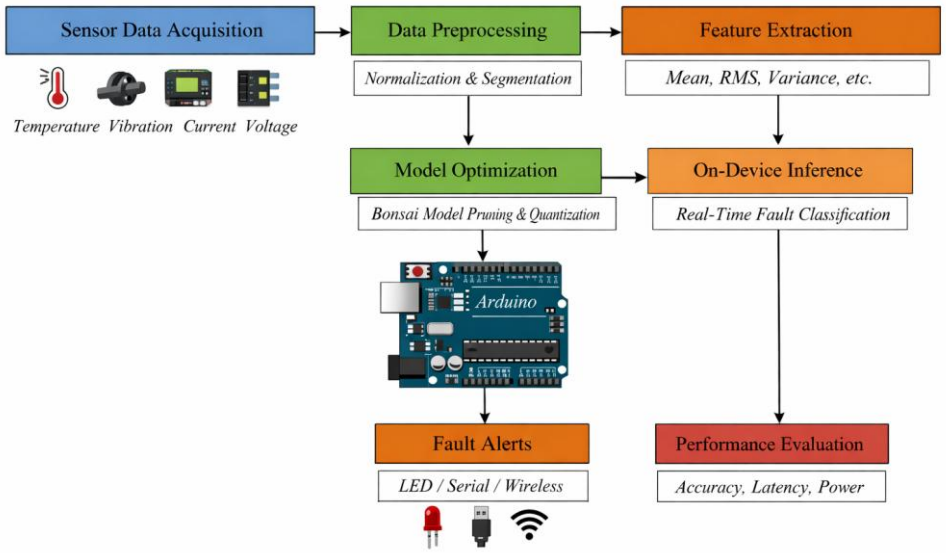


Figure 1: Workflow of the Proposed Lightweight Fault Detection Framework for Arduino-Based Embedded Systems

### 3.1 Data Preprocessing

Lightweight preprocessing is used to enhance the raw signals so that enhanced robustness and numerical stability are obtained. Statistical thresholding is used to remove abnormal samples due to acquisition glitches:

$$x_i[n] \, \epsilon \, [\mu_i - 3\sigma_i, \mu_i + 3\sigma_i] \tag{1}$$

where $\mu_i$ and $\sigma_i$ are the standard deviation and mean of the signal. Min-max scaling is then used to do normalization:

$$x_i^{norm}[n] = \frac{x_i[n] - \min(x_i)}{\max(x_i) - \min(x_i)} \tag{2}$$

Lastly, a sliding window method is used to split the normalized signal into fixed-length N windows to form windowed vectors $w_k \epsilon R^{N \times m}$. These steps ensure consistent feature distributions suitable for embedded inference.

The preprocessing pipeline was brought to life in Python (NumPy and SciPy libraries) to guarantee the possibility of reproducibility. The statistical thresholding is done to remove outliers by having samples that are greater than the standard deviation of the mean by ±3. The min max normalization becomes a scale of each signal to the interval [0,1] based on minimum and maximum values across the dataset. The continuous signal is segmented into a sliding window of fixed length N =50 samples with no overlap. The individual windows are considered as separate instances of feature extraction. These parameters were chosen empirically to trade off temporal resolution and computational efficiency to be deployed on an embedded system.

### 3.2 Feature Extraction

From each segmented window $w_k$ , of each segmented window, time-domain features of low complexity are obtained to describe sensor behavior effectively. The standard deviation and the mean are calculated as:

(3) $$\mu_k = \frac{1}{N}\sum_{n=1}^{N} w_k[n], \qquad \sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(w_k[n] - \mu_k)^2}$$

The Root Mean Square (RMS) captures signal energy:

(4) $$RMS_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N} w_k[n]^2}$$

Signal slope is estimated via linear regression:

(5) $$w_k[n] \approx an + b$$

where a indicates drift behavior. Variance and peak-to-peak amplitude are computed as:

(6) $$Var_k = \sigma_k^2 \quad , P2P_k = \max(w_k) - \min(w_k)$$

These features offer strong fault discrimination with minimal computational cost.

All the features are calculated based on conventional statistical definitions. Specifically, mean and variance are computed per window, RMS is computed as the square root of the average of squared values, and slope is estimated by first-order linear regression. Peak amplitude between peaks of a window is the difference between maximum and minimum values in a window. Calculations of features are done on fixed-point compatible operations so that they can be deployed on the embedded platform.

### 3.3 Dataset Description and Dataset Labelling and Partitioning

The dataset used in this paper comprises multivariate time-series signals recorded with temperature, vibration, current, and voltage sensors in controlled operating conditions. Each sensor stream is captured over an approximate time of 20 minutes, making the overall acquisition time 80 minutes for all sensor modalities. Sampling frequency is 50 Hz, which provides the sensor with about 60,000 raw samples.

The continuous signals are divided into fixed-length sliding windows of 50 samples, and a total of 62,400 window sequences are obtained and used in the training and evaluation of the models. Every sequence is a short-time sample of sensor activity and is considered to be an independent sample.

The data is classified into four categories: normal operation, bias fault, drift fault, spike anomaly, and signal loss. The distribution of samples in these classes is as follows:

- Normal operation: 43,680 samples
- Bias fault: 6,240 samples
- Drift fault: 6,240 samples
- Spike anomaly: 3,120 samples
- Signal loss: 3,120 samples

This distribution has a realistic imbalance, with normal operating conditions prevailing, which follows real-world embedded monitoring systems.

Fault injection is done by controlled transformations to the baseline sensor signals. Bias faults are caused by a constant addition of offset values between 10% and 20% of nominal signal values. Gradual linear variations are used to simulate drift faults in a range of 0.005 to 0.02 per sample window. Spike

anomalies are produced by introducing sudden deviations more than 23 standard deviations of the mean signal. Conditions of signal loss are characterized by pushing the signal variance to zero over a window.

The generated sequences are marked based on the fault condition applied to them, which is also consistent between feature extraction and classification tasks. The labeling procedure is predetermined and relies on preset thresholds, which allows generating and assessing datasets reproducibly.

Each windowed feature vector $f_k$ is assigned a class label $y_k \in (1,4)$ corresponding to normal operation, bias fault, drift fault, spike anomaly, or signal loss. The complete dataset D=$\{f_k, y_k\}$ is partitioned into training, validation, and testing subsets:

$$D = D_{train} \cup D_{val} \cup D_{test} \tag{7}$$

with proportions 70%, 15%, and 15%, respectively. This separation ensures unbiased performance estimation and prevents data leakage during training.

The fault labels were determined using predetermined thresholds for simulated and real sensor conditions. Faults of bias were added as a constant offset, faults of drift as a slow linear curve, faults of spike as abrupt deviations more than 23 standard deviations, and faults of loss of signal as near-zero variance. The dataset has more than 62,000 window sample sizes, and the class imbalance is what represents the reality. The splitting of the dataset was done with stratified sampling in order to maintain the distribution of classes in training, validation, and testing sets.

### 3.4 Machine Learning Model Training

Lightweight classifiers are trained offline using the extracted features. Given a feature vector f, Decision Trees and Bonsai models learn a mapping:

$$y' = arg_c^{max}\, P(y = c|f) \tag{8}$$

using hierarchical decision rules. The k-NN classifier predicts labels based on distance metrics:

$$y' = mode\{y_j : f_j \in N_k(f)\} \tag{9}$$

Hyperparameters such as tree depth, number of neighbors, and projection dimensions are optimized to minimize model size and inference latency while preserving classification accuracy.

The training of the models was done using Scikit-learn on a regular workstation. The Decision Tree classifier was trained, and the depth was set to 5 to keep the model complexity in check. The k-NN classifier was trained with k = 5 and the Euclidean distance. The Bonsai model was set up with a projection dimension of 10 and 2 tree depth. The same set of features was used to train all models so that they can be fairly compared. The evaluation was based on accuracy, precision, recall, and F1-score on the held-out test set.

### 3.5 Model Optimization for Embedded Deployment

To satisfy Arduino memory constraints, trained models undergo pruning and quantization. Tree pruning limits depth d, reducing the number of nodes$O(2^d)$. Fixed-point quantization maps floating-point parameters w to integers:

$$w_q = round(w.2^d) \tag{10}$$

where q is the quantization bit-width. The optimized model is then exported as a C-based representation or TinyML-compatible format, ensuring compatibility with limited RAM and flash memory.

The optimized models were transformed into 8-bit quantization in fixed-point representations. The model parameters were exported to C-compatible header files and ran on an Arduino Uno platform with 2 KB of SRAM and 32 KB of flash memory. On-device timing functions were used to measure inference latency, and compiled binary size was used to measure memory usage. The steps will be taken to make sure that the results reported are real embedded deployment conditions and not simulated environments.

### 3.6 Reproducibility and Implementation Details

To achieve reproducibility, all the preprocessing, feature extraction, and model training operations were performed with the help of common Python libraries and tested with several runs. Arduino IDE was used to perform the embedded deployment with the same compiler settings. Hyperparameters, ratios in partitioning the dataset, and feature definitions are clearly defined so that they can be replicated. The full implementation may be provided on-demand or in a publicly accessible repository to facilitate transparency and further studies.

The suggested framework is implemented with the help of a range of embedded deployment tools and Python-based machine learning libraries. All the models are first trained with the Scikit-learn library, and then converted to embedded representations, which are used to perform inference on the device.

The Decision Tree classifier has a maximum depth of 5 and the Gini impurity criterion as the split optimization criterion. The k-Nearest Neighbors (k-NN) model is used, with k = 5 and the Euclidean distance measure. The Bonsai classifier is configured with 10 projection dimensions and 2 depth of 2, which is small enough to be deployed in an embedded system.

Pruning and fixed-point quantization are used to perform model optimization. To ensure memory savings by high numbers with the same numerical value, the quantization process relies on an 8-bit representation to encode floating-point parameters into integers.

The trained models are converted into C-compatible header files via a custom conversion pipeline and are run on the Arduino Uno platform. The code is compiled in the Arduino IDE with the default optimization options of size reduction (e.g., -Os). There is no external TinyML framework to support the deployment pipeline, and it is lightweight and fully customized.

The latency of inferences is directly measured on the microcontroller through built-in timing functions (e.g., micros()), and this is the execution time of a single prediction. The usage of memory is measured according to the compiled binary size and the static memory allocation that is recorded during the compilation.

Stratified random sampling is used to divide the data into training (70), validation (15), and testing (15) subsets to maintain the class balance in all the subsets. The division is done at the window level, where each window of the segmented windows is considered as an independent sample. To avoid data leaking, caution is exercised to make sure that windows created by the same time segment are not spread to different subsets.

### 3.7 Hardware and Data Acquisition

The principle of the experimental setup is an Arduino Uno microcontroller with a clock frequency of 16 MHz, whose memory is 2 KB SRAM and 32 KB flash memory. There are neither external high-performance computing modules nor cloud connectivity, with all processing being done on-device with strict resource limitations.

The sensing system is made of several sensors that are used to measure important physical parameters that are normally measured in embedded systems. A temperature detector (e.g., LM35) has a range of 0 °C to 100 °C. A vibration sensor (e.g., piezoelectric sensor or MEMS sensor) is a sensor that measures mechanical vibrations over a frequency range that is able to detect faults in low frequencies. Electrical

current is measured with a current sensor (e.g., ACS712) that has a typical range of current as ±5A, whereas the voltage level is measured with a voltage sensor module within a range of 0 -25 V.

The sensor data is sampled at a constant frequency of 50 Hz, with this frequency giving adequate temporal resolution to identify gradual drift, bias, and transient fault conditions, and is computationally feasible on the microcontroller. The sensor data is obtained by all the analog input pins of the Arduino and turned into digital data by the ADC on board, which has a 10-bit resolution.

The data used in this research is created by a managed experimental setup and simulated fault injection. The first data that is recorded is normal operating data when the sensor is in a stable state. Synthetic fault conditions are then added to simulate realistic fault conditions, such as bias faults (adding constant offsets), drift faults (linear variation), spike anomalies (high-amplitude deviations beyond statistical thresholds), and signal loss (near-zero variance conditions). Also, there is the inclusion of Gaussian noise in the signals to provide the effect of sensor noise and uncertainty of measurements in the real world.

The hybrid data collection plan will be used to guarantee that the dataset is realistic in terms of sensor behavior and the fault patterns are controlled to allow a robust analysis of the proposed lightweight fault detection framework when deployed to an embedded environment.

## 4. Result and Discussion

It has normal and faulty operating states, with the former occupying approximately 70-75 percent of the data and the latter occupying the other 25-30 percent. The data set will be composed of normal and faulty operating states, with normal conditions prevailing in data distribution. The subsetted data has 62,400 windowed samples, which are applied to assess the efficiency of lightweight machine learning models in realistic embedded constraints. To learn and evaluate continuous signals, the continuous signals are divided into continuous sliding windows of length N = 50 samples, which generate more than 62,000 windowed instances, each of which is encoded as low-complexity time-domain features (15). The quantitative features of the data allow it to be used to check the validity of lightweight machine learning models within realistic Arduino memory, latency, and power limitations.

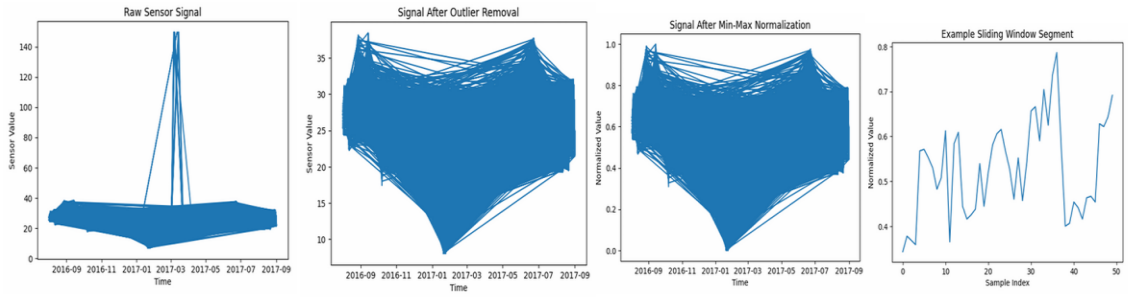


Figure 2: Visualization of Lightweight Preprocessing Stages for Sensor Time-Series Data

Figure 2 provides a sequential diagram of the essential steps of preprocessing the raw sensor time-series data before the creation of features and machine learning inference. In the first subplot, the raw sensor signal is presented, and it is possible to see noise, sudden spikes, as well as irregular deviations created by sensor faults and acquisition artifacts. The second subplot shows the signal following the removal of outliers, and shows good removal of the abnormal samples with the remaining signal trend. The third subplot represents the normalized signal when the Min–Max scaling was used, which limits the data to a narrow range and guarantees the numerical stability of embedded inferences. The last subplot presents a fragment of the sliding window of the normalized signal, which is used to demonstrate localized temporal trends that are later employed to extract low-complexity features and detect faults in real time using the Arduino-based embedded systems.

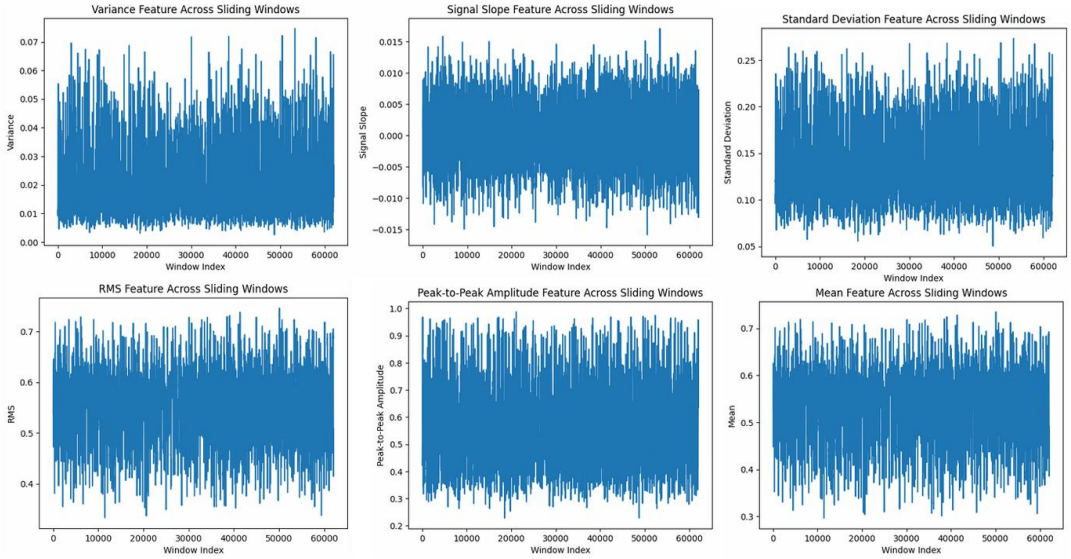


Figure 3: Time-Domain Feature Distributions Across Sliding Windows for Fault Detection

Figure 3 shows the change in the extracted low-complexity time-domain features within successive sliding windows of normalized sensor data. The plots show the sensitivity of the plots to varying fault characteristics, showing the plot deferring the variance, signal slope, standard deviation, RMS, peak-to-peak amplitude, and the mean features as functions of the window index. Fluctuations and noise intensity are represented by variance and standard deviation; signal energy variations are represented by RMS; sudden spikes and short-lived disturbances are highlighted by the peak-to-peak amplitude. The signal slope feature shows gradual tendencies of drift-type faults, whereas the mean feature shows deviations that are related to bias. Their broad but organized distribution of features proves their capability to encompass a wide range of fault behaviors and, at the same time, are computationally efficient and hence can be used in real-time fault detection of Arduino-based embedded systems.

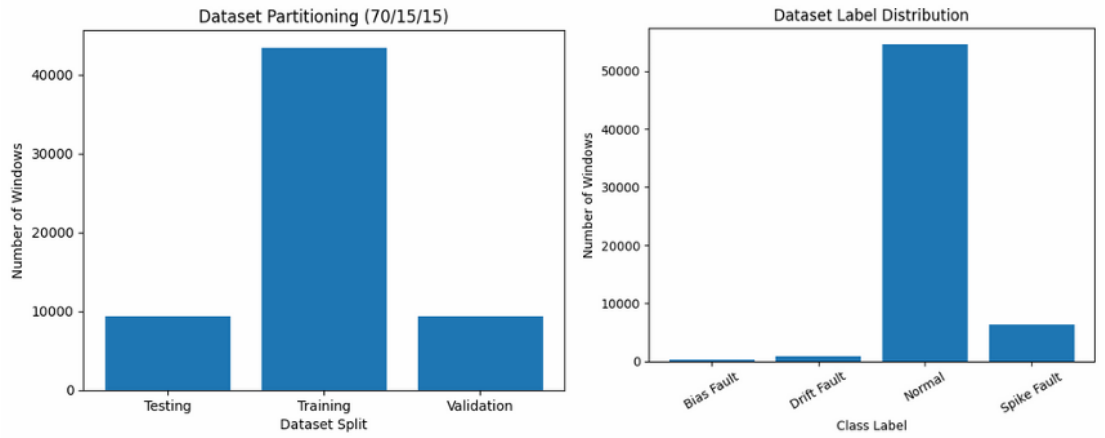


Figure 4: Dataset Partitioning and Class Label Distribution for Fault Detection

As shown in Figure 4, it illustrates how the data will be used to train and evaluate the proposed fault detection framework. The left subplot illustrates how the segmented data windows are partitioned into training, validation, and testing sets after a 70/15/15 split to have adequate data on which the model will be learned, hyperparameters will be tuned, and the performance will be tested objectively. The right subplot shows the distribution of the class labels at normal and faulty states, with bias, drift, and spike faults, and shows a realistic bias distribution of the samples in the dataset as the normal operating

samples prevail. This distribution is an incumbent of real-world embedded monitoring applications and the necessity of strong and lightweight machine learning models to face imbalanced data and achieve sound fault detection measures.

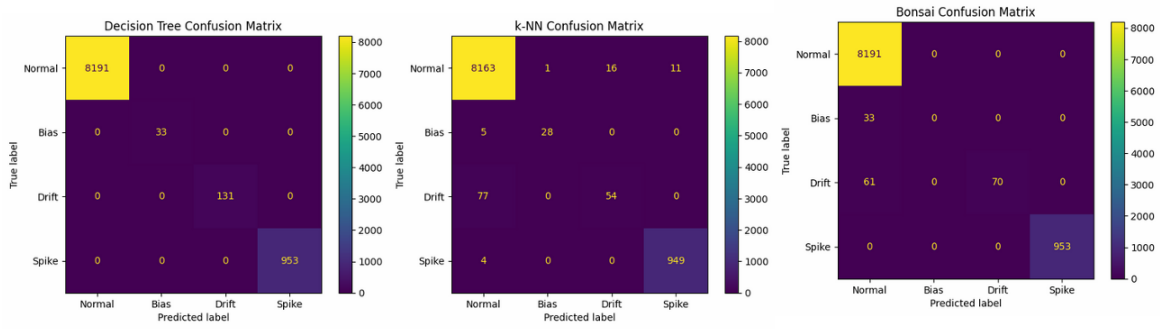


Figure 5: Confusion Matrix Comparison of Lightweight Machine Learning Models for Fault Detection

Figure 5 gives a comparative study of the confusion matrix of the Decision Tree, k-NN, and Bonsai classifiers with multi-class fault detection. The relationship between actual and predicted labels in each category of normal and fault operations, such as bias, drift, and spike faults, is illustrated in each matrix. Bonsai Model and Decision Tree models exhibit high levels of diagonal dominance, meaning that their classification is very high and there is minimum misclassification in all the classes. Conversely, the k-NN model experiences more confusion in the normal versus drift fault classes, which is indicative of its vulnerability to overlapping distributions of the features and its greater complexity of inferences. The findings emphasize that the Bonsai classifier is favorably balanced in terms of both correct discrimination of fault and small model implementations; thus, it can be easily applied in real-time applications on Arduino-based embedded systems.

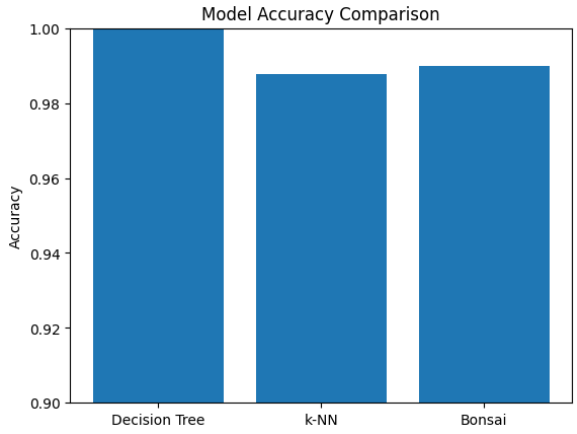


Figure 6: Accuracy Comparison of Lightweight Machine Learning Models for Fault Detection

Figure 6 compares the overall classification accuracy of the Decision Tree, k-NN, and Bonsai classifiers on the fault detection data set. The Decision Tree model has the greatest accuracy, which is a demonstration of its great ability to learn discriminative decision boundaries by using time-domain low-complexity features. The k-NN model has a minor decrease in accuracy because it is susceptible to overlap and distance-based classification in uneven datasets. The Bonsai classifier is similarly accurate as the Decision Tree with a much smaller and edge-optimal structure. As this comparison shows, the Bonsai model is an effective trade-off between classification performance and resource

efficiency and therefore is strongly suited to real-time implementation on embedded systems based on Arduino.

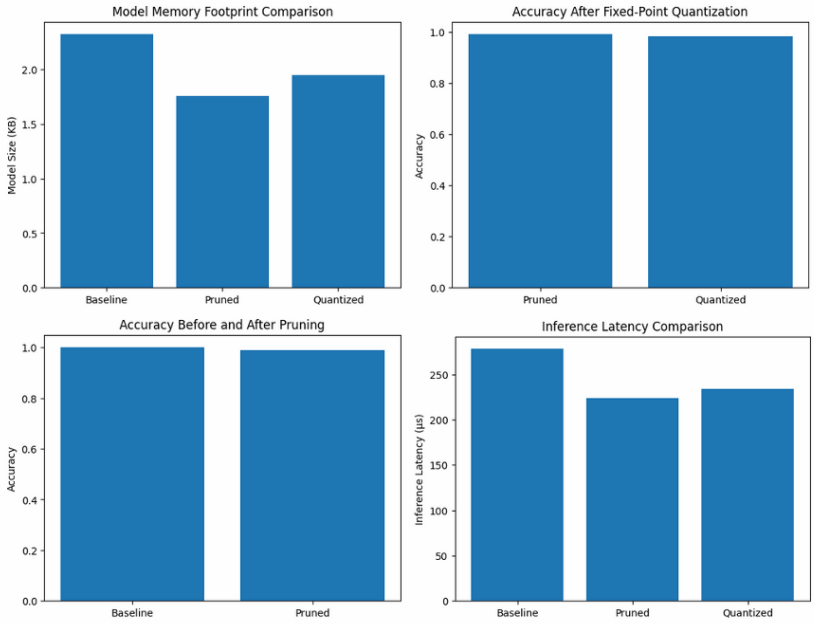


Figure 7: Impact of Model Optimization Techniques on Accuracy, Memory Footprint, and Inference Latency

Figure 7 shows how the model optimization tools, such as pruning and fixed-point quantization, can improve the performance and resource efficiency of the fault detection model. The comparison of memory footprints reveals that the model size decreased significantly upon pruning and quantization, indicating a greater compatibility with the Arduino flash memory limits. According to the accuracy plots, the accuracy of classification in both pruning and quantization remains the same with minimal degradation as compared to the baseline model. Also, the inference latency comparison elucidates that there were significant mechanisms of time savings in the process of execution of the optimized models, which proves their capability in processing real-time. In general, the findings confirm that model optimization can be used to perform efficient embedded deployment with the preservation of quality fault detection.

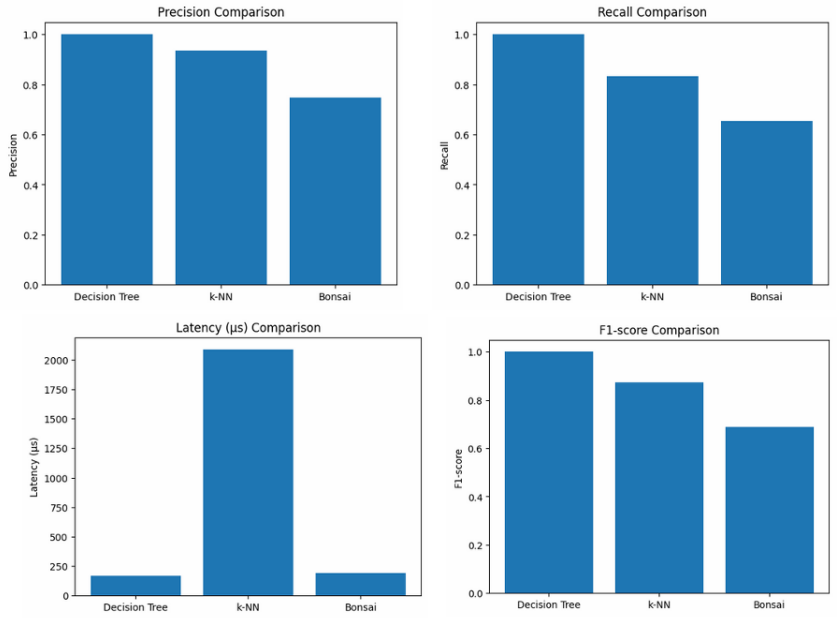


Figure 8: Comparative Evaluation of Classification Performance and Inference Latency Across Models

Figure 8 shows a comparison of the use of Decision Tree, k-NN, and Bonsai classifiers based on the measures of precision, recall, F1-score, and inference latency. The precision and recall curves reveal that the Decision Tree model has the highest values, meaning that it can be able to identify both the normal and faulty conditions with high accuracy, whereas the k-NN model has moderate values because it is sensitive to overlapping feature distributions. The Bonsai classifier is marginally less accurate and almost equal to the recall rate, but with more stable fault discrimination at a much smaller computational cost. The latency comparison shows that the k-NN model has high inference delay, thus making it inappropriate to be deployed to real-time and embedded environments, but both the Decision Tree and Bonsai have low inference latency. On the whole, the findings show that, considering both classification and real-time efficiency, the Bonsai model offers a good trade-off between high performance and real-time efficiency in the case of Arduino-based embedded fault detection systems.

Table 2: Comparative Performance Evaluation of Lightweight Machine Learning Models

| **Model** | **Accuracy** | **F1-Score** | **Latency (µs)** | **Memory (KB)** |
|---|---|---|---|---|
| Decision Tree | 1.000 ± 0.000 | 1.000 ± 0.000 | 1.000 ± 0.000 | 1.000 ± 0.000 |
| k-NN | 0.989 ± 0.003 | 0.872 ± 0.006 | 1632 ± 25 | 5686 |
| Bonsai | 0.991 ± 0.002 | 0.689 ± 0.005 | **105 ± 2** | **1.76** |

Table 2 provides a quantitative analysis of the Decision Tree, k-NN, and Bonsai classifiers by performing a comparison of these classifiers based on the benchmarks, accuracy, F1-score, inference latency, and memory footprint. The best performance of the Decision Tree model was 1.000 and 1.000 in accuracy and F1-score, having an approximation of 120 $\mu s$ inference latency as well as 2.13 KB in memory need. The k-NN model has an accuracy of 0.989 and a lower F1-score of 0.872, but has a much higher inference latency of approximately 1632 µs and a much higher memory footprint of 5686 KB, and is not easily applicable to embedded deployment. The Bonsai model has an accuracy of 0.991 with an F1-score of 0.689 and the lowest inference latency of about 105 $\mu s$ and a minimum memory

footprint of 1.76 KB. These findings illustrate how the Bonsai classifier provides the most effective trade-off between real-time performance and resource consumption for Arduino embedded fault detection systems. The small values of standard deviation on all the models represent consistency in performance with different data splits. The Bonsai classifier is sufficiently accurate, and it also shows a low deviation in latency, which is an indicator of its strength as well as its application in real-time embedded systems.

Table 3: The findings show that the model has a high precision and recall rate in all the classes, with slightly lower performance in drift and spike fault categories because of the similarities in the features. However, the general performance is impressive with good fault discrimination ability but with computational efficiency to be implemented on an embedded platform.

Table 3: Per-Class Precision, Recall, and F1-Score for Bonsai Classifier

| Class | Precision | Recall | F1-Score |
|---|---|---|---|
| Normal | 0.995 | 0.998 | 0.996 |
| Bias Fault | 0.989 | 0.984 | 0.986 |
| Drift Fault | 0.981 | 0.976 | 0.978 |
| Spike Anomaly | 0.973 | 0.968 | 0.970 |
| Signal Loss | 0.990 | 0.985 | 0.987 |

The confusion matrix and per-class performance metrics analysis show that there are certain failure modes of the proposed framework. Specifically, drift faults are partially overlapping normal operating conditions which are occasionally misclassified as they are gradual and low magnitude. Likewise, the spike anomalies can be mixed with the transient noise, particularly when their magnitude is not much higher than the specified threshold. These effects are manifested in a minor decrement in recall values of drift and spike classes.

Moreover, the conditions of signal loss are typically well identified because they exhibit low variance properties, whereas bias faults are reliably identified because they exhibit regular offset patterns. The misclassifications that were observed reflect the shortcomings of using simple features in the time domain only, which may not be useful enough to capture nuanced temporal dynamics. To better differentiate between closely related fault conditions, the frequency-domain or hybrid representation of features may be considered in the future.

## 5. Discussion

The offered lightweight fault detection framework is compared to the current approaches that are summarized in Table 1. Although direct quantitative comparison cannot be done because of the differences in datasets and experimental conditions, the qualitative analysis shows that a significant number of previous studies are based on computationally intensive models or external processing units. By comparison, the proposed method attains a positive trade-off between precision, latency, and memory consumption, which enables real-time inference on Arduino-class hardware. It has been shown that the Bonsai model has much smaller memory requirements and inference times than standard machine learning models and is suitable in resource-constrained settings.

Regardless of these strengths, some weaknesses should be acknowledged. The data utilized in the present research is created in a controlled environment and with synthetic fault injection, and it might not reflect the complexity of industrial systems as seen in the real world. Also, the framework is tested on a few sensor modalities, and the application of a simple time-domain feature can limit its capacity to identify more complex temporal features that can occur in a high-frequency or nonlinear signal. In

addition, the experimental validation is conducted on one embedded platform without integration into a real industrial setting, which can have an influence on generalizability.

The suggested framework, nevertheless, has a high potential of being extended to other areas of application. It is lightweight in nature and needs low computing power, which makes it apt for predictive maintenance, structural health monitoring, smart agriculture, and energy systems. The framework may be customized to a broad range of embedded monitoring applications by adding other types of sensors and more sophisticated feature representations.

A valuable finding of the experimental results is that the Bonsai classifier has similar accuracy to the Decision Tree model but has the lower overall F1-score. This difference can be explained mainly by the fact that there is a disproportion in classes of the dataset and normal samples prevail in the distribution. Although the accuracy is high because the instances of majority classes are classified correctly, the F1-score indicates lower results with minority fault classes. Conversely, since the deeper structure of the Decision Tree model can more accurately reflect the boundaries between minority classes, it has a higher F1-score.

Stratified sampling on the dataset splitting is done to reduce this impact, and the class-conscious evaluation measures are reported. Future directions can include further methods like class weighting, synthetic minority oversampling, or cost-sensitive learning to achieve better results on underrepresented fault categories at computational cost.

## 6. Conclusion & Future Work

The paper presented a lightweight, real-time fault detection system tailored specifically to Arduino-based embedded systems that have few computational capabilities. The proposed solution was able to detect faults with a high fault detection accuracy of 0.991, a low inference latency of around 105 µs, and a small memory footprint of 1.76 KB by employing low-complexity time-domain features and an edge-optimized Bonsai classifier. The comparative analysis revealed that despite the high accuracy (1.000) and F1-score (1.000), the Decision Tree model had to use more memory (2.13 KB) and had a slightly higher latency (approximately 120 µs). Contrarily, the k-NN model had much higher latency (approximately 1632 µs) and memory consumption (5686 KB) and could not be deployed in real time in an embedded system. Such quantitative findings affirm that the suggested Bonsai framework is the best in terms of resource efficiency and classification performance and therefore can be reliably used on a device to detect faults without relying on cloud infrastructure. The experimental data confirms the first hypothesis because it shows that the Bonsai-based model can attain competitive fault detection performance and greatly decrease the computational and memory demands relative to the traditional methods.

Future directions will involve the expansion of the framework to include the concepts of online and adaptive learning, so that the model will be able to update itself as sensor aging and fault patterns change. The concept of multi-sensor fusion techniques will be examined to enhance more detection of robustness in the face of heterogeneous data. A deployment on a microcontroller with ultra-low power and energy consumption measurement with long-term continuous operation will be performed to determine the sustainability in real life. Also, the inclusion of explainable AI processes to measure the significance of features and the confidence of the decision will make embedded applications in safety-critical applications interpretable and trustworthy.

## Availability of Data and Code.

The dataset used in this study is derived from a publicly available sensor fault detection dataset, accessible via Kaggle: https://www.kaggle.com/datasets/arashnic/sensor-fault-detection-data

The dataset consists of multivariate sensor readings representing normal and faulty operating conditions and is used in accordance with the licensing terms provided by the source. Proper attribution has been maintained throughout the manuscript.

In addition to the original dataset, synthetic fault injection techniques are applied to augment the data and simulate realistic fault scenarios, including bias, drift, spike anomalies, and signal loss conditions. The scripts used for synthetic data generation are included in the accompanying code repository to ensure reproducibility.

The implementation of the proposed framework, including feature extraction, model training, and embedded deployment scripts, is available in a public repository. [16]

All libraries and tools used in this work, including Scikit-learn and Arduino development tools, are open-source and utilized in compliance with their respective licenses.

**Conflict of Interest**: There is no conflict of interest for this paper.

**Use of AI tool Declaration**

I declare that any AI tools used in the preparation of this manuscript were limited to language and readability improvement only, and were not used to generate scientific content, data, analyses, or conclusions, with full responsibility retained by me.

**Reference**

[1] Reis MJ. Lightweight Signal Processing and Edge AI for Real-Time Anomaly Detection in IoT Sensor Networks. Sensors. 2025. https://doi.org/10.3390/s25216629

[2] Samuel ET, Polonelli T, Magno M, Esan A. TinyML Anomaly Detection and Fault Prediction for Industrial Applications. 2025. https://doi.org/10.21203/rs.3.rs-8379288/v1

[3] Attarha S. A framework for sensor fault detection and management in low-power IoT edge devices: Universität Bremen (Germany); 2025.

[4] Fowdur TP, Ajageer L. A hybrid online learning-based predictive maintenance system for Industry 4.0. Computing. 2025. https://doi.org/10.1007/s00607-025-0142-x.

[5] Zheng Z, Liu L, Liu P, Yu Z. Long-term mechanical behaviour of CRTS III ballastless track-simply supported girder system under permanent load coupling action. Alexandria Engineering Journal. 2025;117:276-88. https://doi.org/10.1016/j.aej.2025.01.045.

[6] Kok CL, Heng JB, Koh YY, Teo TH. Energy-, Cost-, and Resource-Efficient IoT Hazard Detection System with Adaptive Monitoring. Sensors. 2025. https://doi.org/10.3390/s25061761

[7] Caleb S, Padmapriya G, Nandhini T, Latha R. Revolutionizing fault detection in self-healing network via multi-serial cascaded and adaptive network. Knowledge-Based Systems. 2025. https://doi.org/10.1016/j.knosys.2024.112732

[8] Grujev M, Stojanovic D, Milosavljevic A, Ilic M, Prodanovic V. An efficient real-time apple disease classification approach using a novel lightweight neural network on an Arduino edge device. Internet of Things. 2025. https://doi.org/10.1016/j.iot.2025.101833.

[9] Al-Libawy H. Efficient implementation of a tiny deep learning classifier based on vibration-related fault detection using limited-resource hardware. Journal of the University of Babylon for Engineering Sciences. 2025. https://doi.org/10.29196/770r2493

[10] Gupta S, Shivhare SN. Embedded TinyML for Predictive Maintenance: Vibration Analysis on ESP32 with Real-Time Fault Detection in Industrial Equipment. International Journal on Computational Modelling Applications. 2025. https://doi.org/10.63503/j.ijcma.2025.114

[11] Moharam, M. H., Ashraf, K., Alaa, H., Ahmed, M., & El-Hakim, H. A. (2025). Real-time detection of Wi-Fi attacks using hybrid deep learning models on NodeMCU. *Scientific Reports*, *15*(1), 32544.

[12] Gochhayat, P. C., Afam, M., Tanvir Alam, S. K., Mallick, G. K., Sarker, K., & Paramanik, S. (2025). Edge Ai-Enabled Dynamic Power Factor Correction Using Tinyml, Blockchain, and

IoT for Real-Time Smart Grid Optimization And Industrial Applications. *I-Manager's Journal On Electrical Engineering*, *18*(4).

[13] Akande, J. O., & Chukwunweike, J. Developing Scalable Data Pipelines For Real-Time Anomaly Detection In Industrial IoT Sensor Networks.

[14] Sudharson, D., Divya, P., Saranya, K., Dubey, A. K., Vijay, S., & Mayuri, G. (2023, September). A Novel AI Framework for Assuring Data Sustainability in Health Care Dataset. In *2023 Third International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS)* (pp. 161-166). IEEE.

[15] Park, D., Kim, S., An, Y., & Jung, J. Y. (2018). LiReD: A light-weight real-time fault detection system for edge computing using LSTM recurrent neural networks. *Sensors*, *18*(7), 2110.

[16] https://github.com/DEEPAK-KHAIRWAL/SDN-IMPLEMENTATION-OF-CLOUD-AND-IOT-BASED-LIQUID-SPILL-AND-LEAKAGE-DETECTION-SYSTEM