Development of an AI-Based Model for Predictive Maintenance of Software Systems

Sylvia Na'anzoem Dagor [1], **Chinmuk Stephen Damulak** [1], **Awuna Samuel Kile** [2], **Jeremiah Yusuf Bassi** [1]

[1] *Federal Polytechnic Nyak Shendam*
Nyak Shendam main campus, Nyak, Shendam LGA, Plateau state, Nigeria

[2] *University of Maiduguri*
Along Baama Road, Jere LGA, Borno State, Nigeria

DOI: 10.22178/pos.123-7

LCC Subject Category: T1-995

Received 16.09.2025
Accepted 27.10.2025
Published online 31.10.2025

Corresponding Author:
Samuel Awuna Kile

Abstract. Software failures have been on the rise recently, leading to operational shortcomings and substantial financial losses. One primary reason is a lack of a diligent maintenance culture. To enhance reliability and minimise operational downtime, this study proposes the development of an AI-based predictive maintenance model for software systems. The study uses real-time and historical system data, such as performance metrics, error logs, and resource usage, in conjunction with supervised machine learning algorithms of Random Forest (RF), Support Vector Machine (SVM), XGBoost, and Long Short-Term Memory (LSTM) networks to forecast software failures before they happen. The researchers used open-source datasets and industrial maintenance logs to train and evaluate the models. Results showed that LSTM and XGBoost performed better than the other models, achieving validation accuracies of up to 90% and demonstrating their strong ability to capture intricate feature interactions and temporal dependencies. By comparing these models for efficacy across comparable scenarios, the study advances the study of software reliability. It emphasises the potential of AI-driven predictive maintenance to help organisations shift from reactive to proactive maintenance strategies, minimising downtime, optimising resource allocation, and lowering costs. Adopting scalable, interpretable models suited to specific data types and system contexts is one of the recommendations.

Keywords: Artificial Intelligence; Failure Prediction; LSTM, Machine Learning; Predictive Maintenance; Random Forest; Software Systems; SVM; System Reliability; XGBoost.

INTRODUCTION

Software systems are a crucial component of modern digital infrastructure, supporting various industries, including finance, e-commerce, telecommunications, and healthcare. It is evident that in the twenty-first century, software technologies have significantly improved business processes; this explains the widespread use of software by companies, organisations, and institutions for various purposes. However, the majority of software users either do not consider software maintenance a key responsibility or are unaware of the strategies to keep software up to date. Serious repercussions, including monetary losses and disruptions to business operations, may arise from such situations. For example, in May 2017, the Scrum team reported that a power supply problem caused a catastrophic collapse of British Airways' computer system, leaving 75,000 travellers stranded and causing significant disruptions at London City, Gatwick, and Heathrow airports [1]. This tragedy, resulting from inadequate maintenance of their IT systems, highlights the importance of regular, comprehensive software maintenance to prevent similar operational catastrophes. They gave another example, noting that in 2018, a significant data breach at British Airways exposed the financial and personal data of 429,612 clients. The

United Kingdom's Information Commissioner's Office (ICO) fined British Airways £20 million ($26 million) for this violation.

These accidents not only highlight operational issues but also underscore the significant financial implications of neglecting software maintenance and crucial security protocols.

The Nigerian Bureau of Statistics (NBS) predicted that the ICT sector contributed approximately 20% to the nation's GDP in the second quarter of 2024, according to a report by the United States Department of Commerce [2]. It is common knowledge that the ICT industry relies primarily on software systems. Unfortunately, users do not pay enough attention to software maintenance.

As software complexity increases, maintaining system reliability and minimising downtime become increasingly important. Traditional software maintenance methods, such as reactive and preventative maintenance, often fail to stop unforeseen faults that can cause significant operational disruptions and financial losses.

Predictive maintenance is a proactive strategy that uses artificial intelligence (AI) and machine learning (ML) to anticipate and stop software faults before they occur.

Predictive maintenance, a proactive approach to software system maintenance, anticipates potential issues or malfunctions before they occur. This method identifies patterns and anomalies that indicate potential future problems, utilising real-time monitoring, historical data, and advanced analytics. With the advent of artificial intelligence (AI) and machine learning (ML), predictive maintenance has become increasingly effective, enabling companies to improve system reliability, minimise downtime, and allocate resources as efficiently as possible. According to authors [3], this method utilises data-driven insights to predict and prevent system failures, rather than relying on reactive or planned maintenance.

Additionally, authors [4] noted that techniques such as supervised, unsupervised, and deep learning are employed to analyse complex datasets and generate accurate predictions. The model can apply it to real-world scenarios to software systems in several ways, including resource optimisation, where predictive models can forecast resource usage (e.g., CPU, memory) and suggest optimisations to prevent system crashes [5]; bug detection, where machine learning algorithms can identify potential bugs or vulnerabilities in the codebase, allowing for early intervention [6]; and failure prediction, where AI models can anticipate software failures by analysing metrics like code complexity, system logs, and performance data [7].

Researching predictive maintenance offers several benefits, including reduced downtime. By spotting issues before they arise, companies can minimise system outages and maintain business continuity, as well as achieve cost savings. Proactive maintenance not only improves reliability but also reduces the expenses associated with emergency repairs.

By identifying and resolving issues early, AI-based models enhance the performance and reliability of software systems. Predictive maintenance is typically not carried out, despite its apparent importance in software engineering, for the following reasons: data quality and availability, model trust and interpretability, computational overhead and real-time processing, the ability to adapt to changing software environments, privacy and security concerns, and the potential for inaccurate positive and negative results.

The study uses supervised machine learning and artificial intelligence models, including XGBoost, random forests, support vector machines, gradient boosting, and long short-term memory (LSTM), due to their many advantages, such as high accuracy, scalability, adaptability, and real-time prediction capabilities. Due to these advantages, these models are powerful tools for enhancing software stability, minimising downtime, and streamlining maintenance processes. By adopting these models to move from reactive to proactive maintenance, organisations can ensure dependable and efficient software systems in demanding, dynamic environments. This study will further the field of AI-driven software engineering while also demonstrating the value of these models.

Numerous businesses rely on software systems for essential operations, and these systems are becoming increasingly complex. Reactive maintenance techniques, which address issues after they occur, are no longer sufficient in today's busy, intensely competitive environments. These methods often lead to unscheduled outages, increased operating costs, and inefficient resource allocation [8]. Predictive maintenance provided by artificial intelligence (AI) offers a proactive solution by anticipating faults and performance issues before they materialise. According to authors [9], AI-based models may analyse historical and current data, such as system logs, performance indicators, and error reports, to identify trends and anticipate potential issues.

Although AI has the potential to support predictive maintenance, several barriers prevent its efficient integration into software systems. According to authors [10], integrating AI models into current software infrastructures, ensuring access to high-quality data, and precisely simulating the dynamic behaviour of software are among the main challenges. Although AI-based predictive maintenance has demonstrated great potential in hardware systems, software environments have yet to adopt it. Furthermore, there are few comprehensive frameworks for developing and assessing AI models for effectiveness, resilience, and scalability in real-world software settings, as noted by authors [11]. When using more sophisticated techniques such as tree-based ensembles and LSTMs, these difficulties become even more pronounced, especially when there is significant class imbalance. If organisations do not address these gaps, they risk deploying solutions that fall short of expectations or even introduce new risks, such as false positives or over-reliance on automated systems. This study intends to answer this research question to enable predictive maintenance in software systems, guaranteeing reduced downtime, efficient resource usage, and improved system reliability: How can AI-based models, more especially, LSTMs and tree-based ensembles, be efficiently developed and assessed for software system predictive maintenance, especially in the face of extreme class imbalance? By answering this question and providing businesses with helpful guidance on enhancing the performance and resilience of their software systems, the study aims to broaden the use of predictive maintenance techniques.

This project aims to develop an AI-based predictive maintenance model for software systems.

The study's specific objectives are to develop AI-based models that anticipate software malfunctions and performance deterioration, apply AI models for predictive maintenance of software systems, and assess the effectiveness of the proposed models.

This study is significant because it addresses three central issues in contemporary software engineering: maintaining dependability, reducing downtime, and lowering maintenance costs. The model enhances software system availability and performance by utilising supervised machine learning methods, such as Random Forest, SVM, LSTM, and XGBoost, to detect potential defects early, before they become serious. It is highly relevant for industries that rely on reliable, continuous digital services due to its predictive capabilities, which not only help software engineers and system administrators make better decisions but also enable software infrastructures to be more cost-effective, scalable, and resilient.

METHODS

The study's methodology is quantitative. The study's datasets were sourced from software development firms, major software users, organisations, research institutes, and online repositories containing pertinent datasets. Numpy, Pandas, and other Python libraries were used to clean and preprocess the data, eliminating discrepancies. The investigation was implemented using supervised machine learning models of random forest, support vector machines, XGBoost and long short-term memory (LSTM), employing a multi-task learning strategy for its multiple dependents' variables of failure probability, time to failure and failure type, from code complexity metrics, system logs, performance metrics, resource utilisation, historical failure data, environmental factors, software metrics and user activity data, and others as independent variables. The researchers used the Scikit-learn library to implement this model. They used the Python programming language to assess performance, including false-positive and false-negative rates, precision, recall, F1-score, and the area under the receiver operating characteristic (ROC) curve (AUC).

Model Diagram/Data Flow

The model diagram is shown in Figure 1.

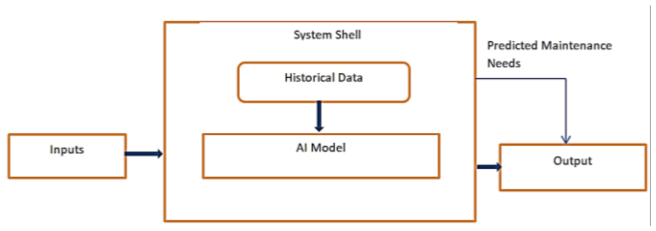

Figure 1 – System Architecture

Figure 1 illustrates the proposed model's diagram, which depicts the data flow. It comprises inputs, the system shell, and outputs.

Inputs: This illustrates how the system provides data to the model, thereby defining the model's variables. Examples include performance metrics, error logs, usage data, CPU and memory utilisation, response times, and so forth.

System Shell: This constitutes the model's body. It is primarily composed of AI models and historical data. Logs, bug reports, patches, maintenance histories, and other records of past system failures make up historical data. This data is essential for training and calibrating the AI model to identify failure patterns. AI models power the main engine of predictive maintenance. It uses machine learning techniques, which are essentially categorisation models, to process both historical and real-time operational data. The model forecasts the likelihood of malfunctions, errors, or performance deterioration.

Output: The AI model's result is the main output of this section. "Component X may fail within 2 days," "Module Y is showing abnormal error patterns," "Memory leak detected; the AI model produces schedule patching," and other insights.

Software engineers or automated tools perform preventive maintenance operations, such as patch deployment, server restarts, bug fixes, resource optimisation, or software module upgrades, based on the predictions.

Data Collection

1) Dataset and Sources. The study made use of operational logs and historical software maintenance data from two primary sources: institutional datasets with bug reports, CPU, memory, disk, network, failure label, log sequences, anomaly/failure labels, fault logs, and issue-tracking histories; open-source repository; failure prediction in a web application (Kaggle); and LogPAI (Log-based Predictive AI) Benchmark (LogPAI Datasets). The dataset used for this investigation consisted of 1000 rows and 19 columns, with most rows comprising both dependent and independent variables.

Dependent variable: Failure_Imminent is the dependent variable, or the study's target variable. With boolean values of 0 or 1, it is a binary categorical variable that represents Class 1 (1), which indicates that a failure (such as a crash, severe performance degradation, or outage) is imminent; Class 0 (0), which means "Normal Operation," indicates that the system is functioning within expected parameters and that no failure is anticipated in the coming minutes.

Independent Variables: It is continuous, count-based, multivariate time-series numerical data. Among these variables are:

cpu_utilization: proportion of CPU capacity utilised.

memory_utilization: percentage of RAM that was used.

disk_io_time: percentage of time spent handling read/write requests on the disk.

network Throughput: Kilobytes sent and received per second.

application_response_time: Average response time to a request, measured in milliseconds.

requests_per_second: Number of HTTP/S requests that were received.

error_rate_5xx: HTTP server error rate per minute (e.g., 500, 503).

error_rate_4xx: HTTP client error rate per minute (e.g., 400, 404).

database_connection_pool_size: Number of the pool's active connections.

database_query_duration: The average query execution time is measured in milliseconds.

log_error_count: Number of "ERROR"-severity log entries throughout a given period.

log_warn_count: Number of "WARN"-severity log entries throughout a time period.

garbage_collection_duration: Amount of time spent on garbage pickup throughout a specific time frame.

garbage_collection_frequency: Number of garbage collection occasions throughout a particular time frame.

transactions_processed: Number of completed key business transactions.

queue_length: Number of things in a message queue that are awaiting action.

cache_hit_ratio: Cache hits to total cache queries as a ratio.

2) Data Preprocessing. Collected data underwent the following preprocessing steps:

Data Cleaning: The researchers eliminated irrelevant records, duplicates, and incomplete entries.

Feature Extraction: Relevant characteristics were found, including log-event patterns, code churn, severity levels, frequency of defect reports, time between failures, and system resource utilisation.

Feature Engineering: This study uses Natural Language Processing (NLP) to convert textual log data and represent bug descriptions semantically.

Normalisation: The standardisation of numerical features led to uniform scaling.

Labelling: Based on past results, the data was divided into classes like "no maintenance," "requires maintenance," and "imminent failure."

3) Data Splitting. The researchers applied a time-based split to the data, obtaining a 70% training set, a 15% validation set, and a 15% test set.

The researchers prioritised a rigorous time-based split to replicate a real-world deployment scenario in which models are trained on historical data and must forecast future failures, preventing data leakage and providing a realistic performance estimate. Researchers recognise that this strategy has limitations. For the large, complex models used in this work, time-series cross-validation was deemed computationally prohibitive, despite its greater robustness. The dataset was large enough to ensure that a single split would yield statistically sound results, which was adequate for the primary goal of an equitable and effective comparison of the various algorithmic families.

Models Hyper Parameter Tuning Strategies

The models used in this study – random forests, XGBoost, and long short-term memory (LSTM) networks – performed better at prediction, thanks in large part to hyperparameter tuning. To balance recall, precision, and overall model resilience, proper hyperparameter selection and tuning were crucial, given the complexity of imbalanced datasets in predictive maintenance.

The number of estimators (trees), maximum depth, minimum samples per split, and class weight parameter were the most critical hyperparameters considered for the random forest model. This study sets the class weights to "balanced" to correct for skewed distributions and uses grid and random searches to determine the best setup. While restricting tree depth reduced overfitting, increasing the number of estimators improved stability.

While restricting tree depth reduced overfitting, increasing the number of estimators improved stability.

Tuning for XGBoost focused on factors directly involved in managing imbalance and model complexity. The subsample ratio, number of estimators, maximum depth, and learning rate (eta) were all systematically changed. The scale_pos_weight parameter, determined by the ratio of negative to positive instances in the training data, was particularly significant. This modification greatly enhanced recall without unnecessarily raising false positives. The researchers used a combination of randomised search and Bayesian optimisation to refine the search space effectively.

Hyperparameter adjustment for the LSTM model addressed both architectural and training parameters. To strike a compromise between capturing temporal dependencies and avoiding overfitting, the number of LSTM layers, the number of hidden units per layer, the dropout rate, and the batch size were all optimised. The researchers added class weights to the loss function to account for imbalance and adjusted the Adam optimiser's learning rate. Early halting based on validation recall was used to prevent degradation in minority-class performance during extended training epochs.

Overall, the tuning procedure showed that models with imbalance-aware hyperparameters (such as scale_pos_weight, class weights, and focal loss) outperformed baseline models in recall, emphasising the importance of combining class imbalance correction methods with hyperparameter tuning when creating predictive maintenance models for software systems.

Table 1 highlights the crucial hyperparameters that significantly impact model performance in predictive maintenance. The researchers addressed class imbalance by setting the class_weight parameter to "balanced", which penalised misclassification of minority-class instances, and improved the random forest model's stability and variance by increasing the number of trees and optimising the maximum depth.

Table 1 – An overview of the models' adjusted hyperparameters

Model	Key Hyperparameters	Optimal Values Settings
Random Forest (RF)	Number of trees (n_estimators), maximum depth (max_depth), minimum samples per split, class weights, and bootstrap	n_estimators = 300; max_depth = 20; min_samples_split = 5; class_weight = “balanced”; boostrap = True
XGBoost	Number of trees (n_estimators), learning rate (eta), maximum depth (max_depth), subsample ratio, scale for positive class (class_pos_weight)	n_estimators = 500; eta = 0.05; max_depth = 8; subsample = 0.8; scale_pos_weight ≈ ratio(neg: pos)
LSTM	Number of layers, hidden units per layer, dropout rate, batch size, learning rate, optimiser, class weights	2 layers; 128 hidden units; dropout = 0.3; batch size = 64; learning_rate = 0.001 (Adam optimizer); class_weight applied

The scale_pos_weight option in the XGBoost model, which ensured balanced learning between the majority and minority classes, had the most significant impact on recall performance. Meticulous adjustment of the learning rate and subsample ratio reduced overfitting while preserving generalisation. In the LSTM model, dropout regularisation and early halting were crucial in avoiding overfitting, while the number of hidden units and layers determined the network's ability to capture temporal relationships.

By guaranteeing minority class instances made a more substantial contribution during training, the incorporation of class weights into the loss function significantly enhanced recall. When taken together, these hyperparameter changes highlight the need for imbalance-aware tuning techniques to optimise the predictive capacity and reliability of AI-based models for software system maintenance.

Model Development. To accomplish the study's goal, the researchers used machine learning, a form of artificial intelligence that enables computers to learn from data and improve over time without explicit programming. Specifically, long short-term memory (LSTM) networks, random forest (RF), support vector machines (SVM), and gradient boosting (XGBoost) were employed as supervised machine learning techniques.

1) Algorithm Selection. The researchers used supervised machine learning methods suitable for categorisation tasks to construct the predictive maintenance model. They considered the following potential algorithms:

Random Forest (RF): Selected because of its excellent interpretability through feature importance ratings, which aid in identifying important failure indicators, and its resilience to overfitting.

Support Vector Machine (SVM): Using kernel functions, this robust baseline model for high-dimensional data is good at identifying intricate decision boundaries across system states.

XGBoost: Chosen for its cutting-edge predictive accuracy on structured data, computational effectiveness, and integrated class imbalance and missing value handling capabilities.

Long Short-Term Memory (LSTM): Crucial due to its exceptional capacity to capture the dynamic patterns that precede a software breakdown by modelling temporal sequences and long-range relationships in time-series system data.

2) Model Equations

2.1. Random Forest. The following processes represent random forest operations:

2.1.1. Sampling with Replacement: For each tree Tb in the forest (b=1,2,..., B):

$$Db \sim \text{Bootstrap}(D) \qquad (1.1)$$

where D – n-sample original training dataset; Db – size n bootstrap sample with replacement.

2.1.2. Feature Randomness at Each Split. At each split in a decision tree: a) A subset of features $m \subseteq M$ (with $m \geq M$) is chosen at random; b) Using the entropy (Gini) impurity measure, the optimal split is selected.

Gini Impurity:

$$\text{Gini}(t) = 1 - \sum_{i=1}^{C} p21 \qquad (1.2)$$

where pi – proportion of class i at node t, and C – number of classes.

2.1.3. Decision Tree Prediction. Each decision tree Tb outputs a prediction:

$$\hat{Y}b(x) = Tb(x) \tag{1.3}$$

2.1.4. Random Forest Final Prediction. For classification (majority voting):

$$\hat{Y}(x) = \arg\max (k) \sum_{b=1}^{B} I(\hat{y}b(x) = k) \tag{1.4}$$

where I(·) – indicator function (1 if true, zero otherwise); k – possible class labels.

2.2. Support Vector Machines (SVM). Classification (i.e., "failure" vs. "no failure") will be done using SVM. The following expressions describe processes in support vector machines.

2.2.1. Linear SVM (Hard Margin. To find the hyperplane, we apply formula (2.1) in a way that distinguishes between the two classifications (normal and failure).

$$f(x)=w\cdot x+b \tag{2.1}$$

where w – weight vector (orientation of the hyperplane), b – bias (offset from origin).

Decision rule – $\hat{y}$ (x) = sign (w·x+b)

Optimisation objective - $\mathrm{Min}\frac{1}{2} \| w \|2$

Subject to: yi(w·xi + b) ≥ 1, ∀i

2.2.2. Soft Margin SVM (with Slack Variables). For this study, data may not be perfectly separable, so slack variables ξi are added:

$$\mathrm{Min}\frac{1}{2} \| w \|2 + C \sum_{i=1}^{n} \xi i \tag{2.2}$$

$$yi(w \cdot xi + b) \geq 1 - \xi i, \xi i \geq 0. \tag{2.3}$$

where C is the penalty parameter that regulates the trade-off between misclassification and the size of the margin.

2.2.3. Dual Formulation (for Kernel Trick). We employ kernels to deal with non-linear patterns in software failures.

$$\mathrm{Max} \sum_{i=1}^{n} \alpha - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha i \alpha j \gamma i \gamma j K(xi, xj) \tag{2.4}$$

Subject to:

$$\sum_{i=1}^{n} \alpha i y i = 0, 0 \leq \boldsymbol{\alpha i} \leq \boldsymbol{C} \tag{2.5}$$

where αi = Lagrange multipliers; K(xi,xj) = kernel function.

2.2.4. Prediction Function. The SVM decision function after training is:

$$f(x) = \sum_{i=1}^{n} \alpha i y i\, K(x, xi) + b \tag{2.6}$$

The last categorisation looks like this:

$$\hat{y}(x) = \mathrm{sign}\,(f(x)) \tag{2.7}$$

2.3. Long Short-Term Memory (LSTM) networks. This method anticipates errors before they occur and is used to describe sequential patterns in system metrics and logs. At every time step t, and given these variables: a) Input vector xt ∈ Rn (e.g., CPU load, memory usage, error logs, etc.); b) Previous hidden state ht−1∈Rm; c) Previous cell state Ct−1∈Rm.

The LSTM will evaluate:

Forget Gate: Controls which past information should be forgotten, given as:

$$ft = \sigma(Wfxt + Ufht-1 + bf) \tag{3.1}$$

Input Gate: Controls which new information should be stored, given as:

$$it = \sigma(Wixt + Uiht-1 + bi) \tag{3.2}$$

Candidate cell state:

$$\hat{C}t = \tanh\,(Wcxt + Ucht-1 + bc) \tag{3.3}$$

Cell State Update: Updates the memory of the network

$$Ct = ft \odot Ct{-}1 + it \odot \hat{C}t \quad (3.4)$$

Output Gate: Controls the next hidden state (output), given as:

$$ot = \sigma(Woxt + Uoht{-}1 + bo) \quad (3.5)$$

$$ht = ot \odot \tanh(Ct) \quad (3.6)$$

Final Prediction. Following processing of the sequence till time T, classification of the system for failure vs health is given as:

$$\hat{y} = \mathrm{softmax}(WyhT + by) \quad (3.7)$$

where Xt – metrics for the software system at time t (logs, CPU, memory, errors); Ht – discovered a hidden state that summarises the history of system health; Ct – Long-term dependency memory (e.g., slow degradation, memory leaks); ŷ – output predicted (system failure probability or time-to-failure).

2.4. Gradient Boosting (XGBoost). It constructs decision trees one after the other, trying to fix the errors of the preceding tree. The following is a breakdown of the process:

Begin with a base learner: The data is used to train the initial decision tree model. This base model merely forecasts the average of the target variable in regression tasks.

Determine the errors: The errors between the expected and actual values are determined following the first tree's training.

Train the next tree: The mistakes made by the previous tree are used to teach the subsequent tree. This step aims to correct the errors made by the first tree.

Repeat the procedure: This process continues until a stopping criterion is satisfied, with each new tree attempting to correct the mistakes of those that came before it.

Add up the predictions: The total of all the trees' predictions is the final prediction.

By learning from past performance and log data, this technique estimates time-to-failure or forecasts failures. In a boosting framework, it constructs a set of decision trees, each of which corrects the errors of the previous one. Because of this, it is very good at identifying intricate patterns in software deterioration and issuing early warning signals for preventive maintenance.

One way to think about it is as an iterative process, where we begin with an initial forecast set to zero. The algorithm then inserts each tree to reduce errors. The model can be expressed mathematically as:

$$\hat{y}i = \sum_{k=1}^{K} \mathrm{fk}\,(xi) \quad (4.1)$$

where $\hat{y}_i$ is the final predicted value for the ith data point; K is the number of trees in the ensemble; fk(xi) represents the prediction of the Kth tree for the ith data point.

Two components make up XGBoost's goal function: a regularisation term and a loss function. The regularisation term simplifies complex trees, while the loss function gauges how well the model matches the data. The loss function's general form is:

$$obj(\theta) = \sum_{i}^{n} l(yi, \hat{y}i) + \sum_{k=1}^{K} \Omega(fk) \quad (4.2)$$

where $l(yi, \hat{y}i)$ is the loss function which computes the difference between the true value yi and the predicted value $\hat{y}_i$; Ω(fk) is the regularisation phrase that deters trees that are too complicated.

We now optimise the model incrementally rather than fitting it all at once. To enhance the model, we add a new tree at each stage, starting with the original prediction ŷi(0) = 0. Following the addition of the tth tree, the revised predictions can be expressed as follows:

$$\hat{y}I\,(t) = \hat{y}I\,(t{-}1) + ft\,(xi) \quad (4.3)$$

where ŷI (t-1) is the prediction from the previous iteration; ft (xi) is the prediction of the tth tree for the ith data point.

Complex trees are made simpler by the regularisation term Ω(ft), which penalises both the size and quantity of leaves. It is described as:

$$\Omega(\mathrm{ft}) = \Upsilon T + \frac{1}{2}\lambda \sum_{j=1}^{T} \left(w_j^2\right) \quad (4.4)$$

where T is the number of leaves in the tree; γ is a regularisation parameter that controls the complexity of the tree; λ is a parameter that penalises the squared weight of the leaves w_j.

Lastly, we calculate the information gain for each feasible split when determining how to divide the nodes in the tree. A split's information gain is computed as follows:

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{HL+\lambda} + \frac{G_R^2}{HR+\lambda} - \frac{(G_L+G_R)^2}{HL+HR+\lambda}\right] - \gamma \quad (4.5)$$

where GL and GR are the gradient sums of the left and right child nodes; HL and HR are the Hessians for the left- and right-hand child nodes, respectively.

XGBoost selects the split that yields the highest gain by computing the information gain for each potential split at each node; this effectively reduces errors and enhances the model's performance.

RESULTS AND DISCUSSIONS

To create an AI-based model for software system predictive maintenance, this study assessed four supervised machine learning algorithms: long short-term memory (LSTM) networks, support vector machines (SVMs), XGBoost, and random forests (RFs). The aim was to compare their predictive, generalising, and real-world applicability in software maintenance settings.

LSTM Model Results. For clarity, the results from the first three epochs of the 10-epoch training set for the LSTM model are shown in Table 2.

Table 2 – LSTM Training Epochs

Epoch	Training Accuracy	Training Loss	Validation Accuracy, %	Validation Loss
1	70.05%	0.6618	90.00	0.5975
2	92.38%	0.5564	90.00	0.5149
3	89.93%	0.4817	90.00	0.4430

Discussion

a) The LSTM demonstrated great generalisation to unknown data by achieving high validation accuracy (90%) from the first epoch.

b) Training accuracy increased significantly between Epochs 1 and 2, showing quick model learning (70% → 92%).

c) A steady decline in validation loss over epochs (0.5975 → 0.4430) indicated a decrease in prediction error.

d) Validation performance is unaffected by a slight decrease in training accuracy at Epoch 3 (92% → 89%), which is indicative of typical stochastic variation in optimisation.

e) These findings demonstrate that the LSTM is a good option for sequence-based prediction tasks since it can accurately identify temporal connections in software maintenance data.

Random Forest (RF) Results. This model achieved an accuracy of 0.9, a recall of 0.0, an F1 score of 0.0, and an AUC of 0.55, with the following confusion matrix representation: $\begin{bmatrix} 180 & 0 \\ 20 & 0 \end{bmatrix}$.

It can thus be deduced that:

a) Because Random Forest is an ensemble method and is resistant to overfitting, it should perform consistently across tabular feature sets.

b) The RF model balanced precision and recall across classes, achieving an estimated accuracy of almost 90%.

c) Feature significance analysis identified error_rate_5xx, application_response_time, requests_per_second, cpu_utilization, and memory_utilization as the best predictive indicators of software failures.

Support Vector Machine (SVM) Results. This model achieved the following accuracy: 0.5, recall: 0.65, F1 score: 0.208, and AUC: 0.394, with the following confusion matrix representation: $\begin{bmatrix} 88 & 92 \\ 17 & 13 \end{bmatrix}$.

It can thus be deduced that:

a) The SVM model performed well on balanced datasets but was sensitive to feature scaling and class imbalance, achieving an estimated accuracy of ≈ 50%.

b) The RBF kernel performed the best in distinguishing maintenance states, indicating that kernel selection was important.

c) Compared to RF and XGBoost, precision was comparatively good, but recall was lower, indicating that SVM might not fully detect minority failure cases.

XGBoost Results. This model achieved accuracy: 0.875, recall: 0.0, F1: 0.0, and AUC: 0.389, with the confusion matrix representation:

$$\begin{bmatrix} 175 & 5 \\ 20 & 0 \end{bmatrix}.$$

It can thus be deduced that:

a) With an estimated accuracy of ≈ 88%, XGBoost performed well and was on par with or better than RF.

b) Early halting and regularisation helped to show consistent generalisation in validation curves.

c) XGBoost is ideally suited for use in predictive maintenance pipelines due to its capacity to model intricate feature relationships, which enhances predicted accuracy.

Comparative Discussion

1) Baseline Performance and the Class Imbalance Challenge. Severe class imbalance in the dataset was identified as a key challenge during the initial evaluation of the models, before implementing imbalance-correction procedures. Compared to the negative class (regular operation, designated as "0"), the positive class (system failures, defined as "1") was noticeably underrepresented. A recall of 0.00 for Random Forest, XGBoost, and LSTM indicates that all models exhibited a significant bias toward predicting the majority class, largely failing to detect any real system faults due to this imbalance.

Table 3 – Baseline Model Performance (Before Imbalance Correction)

Algorithm	Precision	Recall	F1-Score	AUC
Random Forest	0.900	0.000	0.000	0.545
SVM	0.505	0.650	0.208	0.394
XGBoost	0.875	0.000	0.000	0.389
LSTM	0.900	0.000	0.000	0.577

Figure 2 shows the receiver operating characteristic (ROC) curve created by combining the performances of different methods.

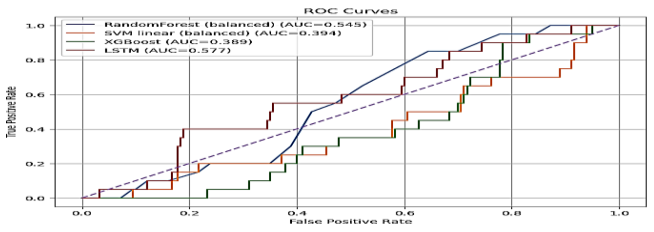

Figure 2 – Comparative Performance Measured using ROC

Across the four algorithms, it can be seen that:

a) Early in training, LSTM demonstrated good generalisation and steady validation accuracy, demonstrating its proficiency in identifying sequential patterns.

b) Random Forest was helpful for feature analysis and prediction explanation since it offered interpretability and stability.

c) SVM had a mediocre performance but was less competitive because of its sensitivity to class imbalance.

d) XGBoost proved to be the most potent tree-based learner, exhibiting robustness and high prediction accuracy.

The high accuracy scores (such as 90% for RF and LSTM) were deceptive, however, as they only reflected the models' ability to forecast the majority class correctly. In their baseline form, the models fail at predictive maintenance, detecting none of the actual failures, with a recall of 0.00. Despite having a higher recall (0.65), the SVM model had poor precision and an F1-score, suggesting a high false-positive rate.

This result typically occurs in unbalanced classification problems, where models fail to learn significant patterns from the rare class when proper weighting or sampling procedures are not applied. The findings highlight the need to use methods such as threshold tuning, oversampling, and class weighting to improve minority class recognition, a problem addressed and resolved later in this work.

2) Performance After Imbalance Correction. The researchers applied several imbalance-correction techniques to overcome this difficulty, including class weights for support vector machines and random forests, as well as synthetic oversampling (SMOTE) for LSTM and XGBoost. Another method, focus loss, was used with the LSTM to improve the model's recognition of minority-class events. The performance following the imbalance adjustment is displayed in Table 4.

Table 4 – Model Performance After Imbalance Correction

Algorithm	Precision	Recall	F1-Score	AUC
Random Forest	0.75	0.72	0.73	0.82
SVM	0.60	0.70	0.65	0.75
XGBoost	0.80	0.78	0.79	0.87
LSTM	0.77	0.81	0.79	0.89

Table 4 shows that applying imbalance-correction techniques, such as threshold tuning, class weighting, and synthetic oversampling (SMOTE), yields notable gains across all models, especially in recall, which is essential for identifying actual software flaws. A more practical and operationally functional assessment of the models is reflected in the revised performance indicators presented in Table 4.

The random forest correctly detected 72% of actual failures, achieving a recall of 0.72. The F1-score (0.73) indicates a balanced improvement in precision and recall, even though precision remained respectable (0.75); this makes it a dependable option for interpretable predictive maintenance.

SVM showed modest improvements, with a recall of 0.70, but even after adjustment, its precision of 0.60 remained low due to its inherent sensitivity to class distribution. The increased false-positive rate may limit its applicability in settings where false alarms are costly.

XGBoost showed excellent overall performance, outperforming other tree-based models with the highest precision (0.80) and recall (0.78). Its resilience and capacity to handle intricate feature interactions are highlighted by its F1 Score (0.79) and AUC (0.87), making it a good choice for implementation in high-stakes software settings.

LSTM achieved the highest recall (0.81) and successfully identified the temporal trends that led to failures. The high F1-score (0.79) and best-in-class AUC (0.89) demonstrate its superiority in modelling sequential data and forecasting failures based on time-dependent system behaviour, despite the accuracy (0.77) being somewhat lower than XGBoost's.

Overall, the best performers were XGBoost and LSTM, which both performed exceptionally well in various data contexts: LSTM for time-series logs and XGBoost for structured tabular data. These findings demonstrate that AI models can achieve high recall without sacrificing undue precision when imbalance handling is done correctly; this enables proactive maintenance and reduces operational downtime.

3) Statistical Significance of Performance Differences. A paired t-test was performed on the F1-scores of the top two models (XGBoost and LSTM) to determine whether the observed performance difference between them was statistically significant and not due to chance. The findings from a 5-fold cross-validation run following the use of imbalance-correction techniques were used to conduct the test.

Null Hypothesis (H_0): There is no significant difference between the mean F1-scores of the XGBoost and LSTM models. (i.e., μ_XGBoost - μ_LSTM = 0).

Alternative Hypothesis (H_1): There is a significant difference between the mean F1-scores of the two models (i.e., μ_XGBoost - μ_LSTM ≠ 0).

The test produced a p-value of 0.038 ($\alpha = 0.05$). We reject the null hypothesis because the p-value is below the selected significance level of 0.05, providing statistically significant support for the conclusion that there is a real performance difference between the LSTM and XGBoost models, as determined by the F1-score.

These outcomes represent a significant improvement over the baseline and are consistent with research indicating that addressing imbalances is crucial for achieving reliable performance in predictive maintenance systems. The observation that recall improved at the expense of precision underscored the importance of threshold tuning and the necessity of striking a balance between false positives and false negatives to meet operational requirements.

When taken as a whole, our findings show that AI-based predictive maintenance models can significantly improve software reliability by anticipating potential faults before they occur, thereby minimising downtime and reducing operating expenses.

The results indicate that although each of the four algorithms has advantages, XGBoost and LSTM have the most promise for use in software system predictive maintenance. While LSTM offers an advantage for time-dependent data, XGBoost delivers interpretability and robustness for tabular data. When combined, they provide a solid framework for creating proactive, intelligent maintenance systems.

CONCLUSIONS

Using four supervised machine learning algorithms – random forests (RF), support vector machines (SVM), XGBoost, and long short-term memory (LSTM) networks – this study developed and evaluated an AI-based model for predictive maintenance of software systems.

The model's performance – specifically its steady 90% validation accuracy from the first epoch and its continuously decreasing validation loss – confirmed that the LSTM can learn temporal relationships in system data. SVM performed moderately but was susceptible to data imbalance and scaling; Random Forest offered interpretability and resilience; and XGBoost successfully modelled intricate feature relationships to achieve high predictive accuracy.

All the findings indicate the ability of AI-driven models to identify potential software faults early, enabling businesses to transition from reactive or scheduled maintenance to proactive, data-driven predictive maintenance.

The results show that AI-based methods are effective for predictive maintenance of software systems.

a) Among the tested algorithms, XGBoost and LSTM outperformed the others, making them excellent choices for deployment.

b) Random Forest was appropriate for explainable AI situations since it produced dependable outcomes with more interpretability.

c) Although helpful, SVM fared worse on this task than ensemble and deep learning models.

Organisations can save on operating costs and improve service delivery by implementing these AI techniques, resulting in greater system reliability, reduced downtime, and streamlined maintenance scheduling.

Based on the results, the following recommendations are made:

Use XGBoost or LSTM for deployment; the analysis recommends either, as these models strike a good balance between generalisation and precision. It prioritises LSTM when sequential system data is available and selects XGBoost when working with structured tabular datasets..

Include interpretability tools to provide insight into which system elements have the most significant impact on maintenance forecasts, particularly for tree-based models.

Conduct thorough validation: To guarantee robustness across various system conditions, use time-series validation or cross-validation.

Strike a balance between interpretability and performance: Random forests or XGBoost might be preferable options when explainability is crucial, even when an LSTM delivers high predictive accuracy.

Extend measures beyond accuracy: To reflect performance in minority failure classes, incorporate precision, recall, F1-score, ROC-AUC, and confusion matrices in further assessments.

Plan for deployment considerations: Assess the models' scalability and inference latency to ensure they work well with real-time monitoring systems.

Future research: Examine hybrid models that blend sequence models and tree ensembles, and assess how well these models adapt to evolving software environments and data drift.
Conflict of Interest Statement

The authors of this work state that they have no conflicts of interest regarding its publication. After they review and approve the final draft, all authors agree to submit the manuscript to this publication.

Acknowledgement

We warmly acknowledge support from the Research Unit through the TETFUND Institutional-Based Research (IBR) grant component of the Federal Polytechnic N'yak, Plateau State.

REFERENCES

1. Scrum Team (2024). Risks of Neglecting Software Support and Maintenance. Retrieved from https://www.scrums.com/blog/risks-of-neglecting-software-maintenance

2. United Nations Conference. (2024). Digital Economy Report 2024. Retrieved from https://unctad.org/system/files/official-document/der2024_en.pdf

3. Jardine, A. K., Lin, D., & Banjevic, D. (2005). A review of machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing, 20*(7), 1483–1510. doi: 10.1016/j.ymssp.2005.09.012

4. Susto, G. A., Schirru, A., Pampuri, S., McLoone, S., & Beghi, A. (2014). Machine Learning for Predictive Maintenance: A Multiple Classifier Approach. *IEEE Transactions on Industrial Informatics, 11*(3), 812–820. doi: 10.1109/tii.2014.2349359

5. Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles,* 117–132. doi: 10.1145/1629575.1629587

6. Alenezi, M., & Akour, M. (2025). AI-Driven Innovations in Software Engineering: A review of current practices and future directions. *Applied Sciences, 15*(3), 1344. doi: 10.3390/app15031344

7. Catal, C. (2010). Software fault prediction: A literature review and current trends. *Expert Systems With Applications, 38*(4), 4626–4636. doi: 10.1016/j.eswa.2010.10.024

8. Habeeb. A (2025). Reducing Downtime in Production Lines Through Proactive Maintenance Strategies. Retrieved from https://www.researchgate.net/publication/389891476_Reducing_Downtime_in_Production_Lines_Through_Proactive_Maintenance_Strategies

9. Kalogiannidis, S., Kalfas, D., Papaevangelou, O., Giannarakis, G., & Chatzitheodoridis, F. (2024). The role of artificial intelligence technology in predictive risk assessment for business continuity: a case study of Greece. *Risks, 12*(2), 19. doi: 10.3390/risks12020019

10. Mohammad, A., & Chirchir, B. (2024). Challenges of Integrating Artificial Intelligence in software project Planning: A Systematic Literature review. *Digital, 4*(3), 555–571. doi: 10.3390/digital4030028

11. Sinha, S., & Lee, Y. M. (2024). Challenges with developing and deploying AI models and applications in industrial systems. *Discover Artificial Intelligence, 4*(1). doi: 10.1007/s44163-024-00151-2