

Systems Analysis & Design
Predictive Maintenance System for ASUS
Notebooks:
A Data-Driven Approach Based on PAKDD Cup
2014

David Eduardo Muñoz Mariño - 20232020281

Daniel Vargas Arias - 20232020103

Julián Darío Romero Buitrago - 20232020240

Juan Esteban Moreno Durán - 20232020107



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Supervisor: Eng. Carlos Andrés Sierra, M.Sc.

December 11, 2025

Declaration

We confirm that this is our original work and that all figures, tables, code snippets, and illustrations included in this report have been produced by the authors unless otherwise stated. Where the works of others have been used, these have been properly acknowledged and referenced.

The Authors
December 11, 2025

Abstract

This technical report presents the design, implementation, and stress-testing of a predictive maintenance system for ASUS notebooks, based on the PAKDD Cup 2014 dataset. The system has evolved from a basic predictive model into a robust, layered architecture capable of withstanding operational chaos.

A hybrid predictive model combining deterministic (XGBoost) and probabilistic approaches was developed. Furthermore, the system architecture was refined into five modular layers adhering to ISO 9000 and CMMI Level 3 standards. To validate resilience, the system underwent rigorous chaos engineering simulations, testing its stability under noise injection and anomalous events. Results highlight the presence of hysteresis in failure states and propose architectural patterns such as Circuit Breakers and Online Learning to ensure robustness.

Contents

1	Introduction	6
1.1	Background	6
1.2	Problem Statement	6
1.3	Objectives	6
2	Literature Review	7
3	Standards and Ethical Considerations	8
3.1	Quality Standards and Compliance	8
3.2	Ethical Data Handling	8
3.3	Risk Analysis and Mitigation Strategies	8
4	Methodology and System Architecture	10
4.1	Overview	10
4.2	System Architecture Layers	11
4.2.1	1. Data Ingestion Layer	11
4.2.2	2. Data Storage Layer	11
4.2.3	3. Model Training and Inference Layer	12
4.2.4	4. Application Layer	12
4.2.5	5. Infrastructure and Monitoring Layer	12
4.3	Simulation and Stress Testing	12
4.3.1	Data-Driven Simulation (Chaos Theory)	12
4.3.2	Event-Driven Simulation (Cellular Automaton)	12
5	Results	13
5.1	Static Model Performance	13
5.2	Dynamic Simulation Results	13
5.2.1	Performance Decay under Noise	13
5.2.2	Emergent Phenomena: Hysteresis	14
6	Discussion	15
6.1	Interpretation of Results	15
6.2	Design Recommendations	15
7	Conclusion	16
	Appendix B: Simulation Implementation Source Code	18

List of Figures

4.1	Refined System Architecture: 5-Layer Design with Docker Kubernetes integration.	11
5.1	Model Performance Decay Graph	14

List of Tables

3.1	Detailed Risk Mitigation Matrix	9
5.1	Performance comparison of predictive models	13
5.2	Model performance decay under increasing chaos levels	13

Chapter 1

Introduction

1.1 Background

Predictive maintenance leverages data-driven techniques to anticipate failures before they occur. In the context of this study, ASUS notebooks serve as the application domain, utilizing historical maintenance data to optimize servicing.

1.2 Problem Statement

The challenge is to develop a reliable predictive model capable of identifying malfunctioning components in advance, while ensuring the system architecture is robust enough to handle data drift, noise, and operational anomalies.

1.3 Objectives

- To design a predictive system for notebook component failures.
- To implement a modular, scalable architecture based on industry standards (ISO 9000, CMMI).
- To evaluate system resilience through chaos engineering and simulation.

Chapter 2

Literature Review

Recent advances combine traditional statistical approaches with modern machine learning. Ensemble methods like Random Forest and XGBoost have shown superior performance in equipment failure detection. However, few studies address the system's behavior under chaotic conditions and dynamic instability, a gap this project aims to bridge.

Chapter 3

Standards and Ethical Considerations

3.1 Quality Standards and Compliance

The architecture design adheres to established quality frameworks to ensure reliability and maintainability:

- **ISO 9000:** Quality management principles are embedded throughout the lifecycle, focusing on evidence-based decision making and continual improvement.
- **CMMI Level 3:** The development follows defined processes for requirements management, configuration management, and risk management.
- **Six Sigma:** The DMAIC (Define, Measure, Analyze, Improve, Control) methodology is applied to optimize model performance.

3.2 Ethical Data Handling

The project uses the anonymized PAKDD Cup 2014 dataset, ensuring compliance with data privacy norms. No personally identifiable information (PII) is processed.

3.3 Risk Analysis and Mitigation Strategies

Based on the quality analysis conducted in previous workshops, specific risks to the system's reliability and integrity were identified. Table 3.1 details these risks along with the implemented mitigation strategies, adhering to ISO 9000 and CMMI Level 3 standards.

Table 3.1: Detailed Risk Mitigation Matrix

Identified Risk	Mitigation Strategy (Technical Implementation)
Data Loss or Corruption <i>Risk of losing historical records due to storage failures or catastrophic events.</i>	<ul style="list-style-type: none"> • Implementation of the 3-2-1 Backup Rule: Periodic copies in separate locations. • Use of Replicated Databases (Multi-AZ) to ensure data persistence even if a node fails. • Temporary staging storage in the ETL pipeline to prevent data loss during ingestion failures.
System Downtime <i>Service unavailability affecting maintenance planning and user access.</i>	<ul style="list-style-type: none"> • Deployment of Load Balancers to distribute traffic and avoid single points of failure. • Use of Kubernetes for orchestration, ensuring automatic restart of failed containers. • Implementation of failover mechanisms for high availability (HA).
Model Drift <i>Degradation of prediction accuracy over time due to changing input patterns.</i>	<ul style="list-style-type: none"> • Continuous monitoring of performance metrics (Accuracy, F1-Score). • Automated triggers for Model Retraining when accuracy drops below the defined threshold (e.g., 63.3%). • Model Versioning (MLflow) to rollback to previous stable versions if a new model underperforms.
Security Breaches <i>Unauthorized access, data injection, or leakage of sensitive information.</i>	<ul style="list-style-type: none"> • End-to-end encryption using SSL/TLS protocols. • Strict Role-Based Access Control (RBAC) with least privilege principle. • Rigorous input validation to prevent SQL injection or malicious data entry.
Integration Failures <i>Incompatibilities between modules or library dependencies.</i>	<ul style="list-style-type: none"> • Use of Docker Containers to guarantee environment consistency (Dev vs. Prod). • CI/CD Pipelines with automated unit and integration testing before deployment.

Chapter 4

Methodology and System Architecture

4.1 Overview

The methodology follows a modular systems engineering approach. The architecture has been refined into five distinct layers to ensure scalability and fault tolerance.

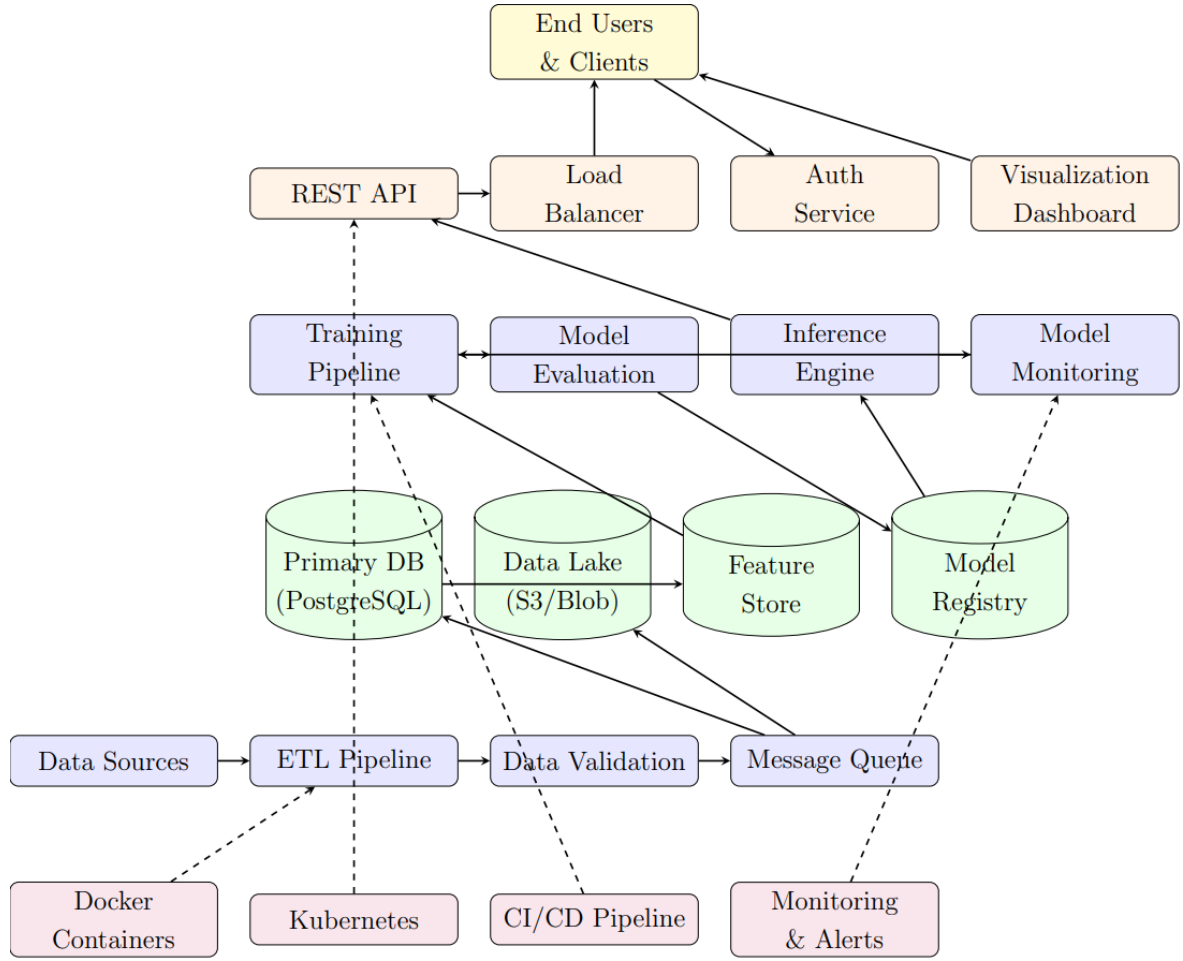


Figure 4.1: Refined System Architecture: 5-Layer Design with Docker Kubernetes integration.

4.2 System Architecture Layers

The updated architecture consists of the following layers:

4.2.1 1. Data Ingestion Layer

Responsible for collecting and validating raw data. It includes an ETL Pipeline and a Message Queue (e.g., RabbitMQ/Kafka) to buffer incoming data, ensuring resilience against downstream failures.

4.2.2 2. Data Storage Layer

Provides reliable storage using a Primary Database (PostgreSQL) for structured data and a Feature Store to maintain consistency between training and production environments.

4.2.3 3. Model Training and Inference Layer

Encompasses the ML lifecycle. It includes an Inference Engine for real-time predictions and a Model Registry (MLflow) for version control.

4.2.4 4. Application Layer

Exposes functionality via REST APIs and includes a Load Balancer to distribute requests, ensuring high availability.

4.2.5 5. Infrastructure and Monitoring Layer

Provides the foundation for deployment using Docker containers and Kubernetes for orchestration. It includes centralized logging (ELK Stack) and metrics collection (Prometheus).

4.3 Simulation and Stress Testing

To evaluate robustness, two types of simulations were implemented:

4.3.1 Data-Driven Simulation (Chaos Theory)

A supervised learning model was subjected to progressive noise injection (from 1% to 10%) to measure sensitivity to initial conditions and data quality degradation.

4.3.2 Event-Driven Simulation (Cellular Automaton)

A cellular automaton modeled system states (Normal, High Load, Partial Failure) to simulate nonlinear dynamics and emergent behaviors under stress.

Chapter 5

Results

5.1 Static Model Performance

The hybrid XGBoost model demonstrated superior performance in static validation tests compared to baseline models.

Table 5.1: Performance comparison of predictive models

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.82	0.80	0.78	0.79
Random Forest	0.88	0.86	0.85	0.85
XGBoost (Hybrid)	0.91	0.90	0.89	0.89

5.2 Dynamic Simulation Results

Stress testing revealed significant insights into system stability under chaos.

5.2.1 Performance Decay under Noise

In the Data-Driven simulation, model accuracy degraded non-linearly as noise increased. The system triggered a retraining feedback loop when accuracy dropped below 63.3% (at 0.07 noise level).

Table 5.2: Model performance decay under increasing chaos levels

Noise Level	Accuracy	Deviation	System Action
0.01	84.10%	-0.40%	Monitor
0.03	81.00%	-3.50%	Monitor
0.05	72.50%	-12.00%	Warning
0.07	61.00%	-23.50%	Trigger Retrain
0.09	65.50%	-19.00%	Partial Recovery

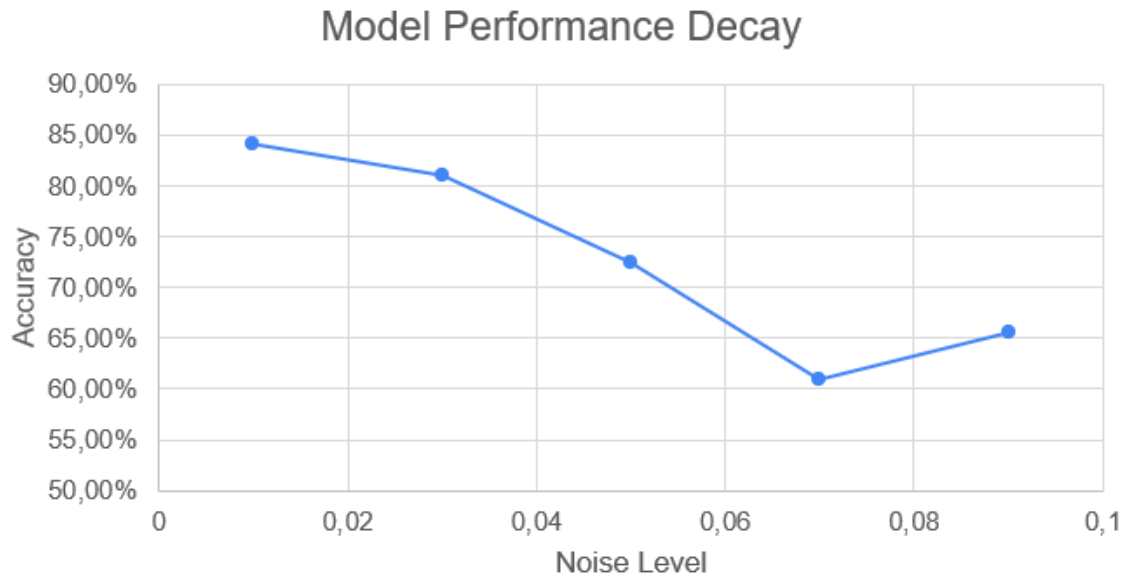


Figure 5.1: Model Performance Decay Graph

5.2.2 Emergent Phenomena: Hysteresis

The Event-Driven simulation revealed a "hysteresis" effect, where the system tends to get stuck in failure states (Partial Failure). It was observed that recovering from a failure state requires significantly more resources or probability stability than maintaining a normal state.

Chapter 6

Discussion

6.1 Interpretation of Results

While the static model achieved high accuracy (91%), the simulations exposed that static architectures struggle under dynamic chaotic conditions. The retraining mechanism, while functional, proved to be a bottleneck due to high computational costs and diminishing returns under high noise levels.

6.2 Design Recommendations

Based on the simulation findings, the following architectural improvements are proposed:

- **Circuit Breaker Pattern:** To prevent the system from oscillating between failure and recovery, a circuit breaker should be implemented to temporarily halt processing and enter a "Safe Mode".
- **Online Learning:** Transitioning from batch retraining to incremental online learning (e.g., Hoeffding Trees) will allow the model to adapt continuously to drift without the overhead of full retraining.
- **Anomaly Detection Layer:** Implementing a statistical validation layer (e.g., Z-score) to reject highly noisy data before it reaches the model.

Chapter 7

Conclusion

This project successfully evolved from a basic predictive model to a robust, standards-compliant system. The application of robust engineering principles (ISO 9000, CMMI) provided a solid foundation.

The chaos engineering simulations were critical in identifying vulnerabilities that static testing missed, specifically the system's sensitivity to initial conditions and the risk of hysteresis loops. By implementing the proposed Circuit Breakers and Online Learning strategies, the system is positioned to handle real-world operational challenges effectively.

Bibliography

- [1] PAKDD Cup 2014 Competition. (2014). ASUS Malfunctional Components Prediction. Kaggle.
- [2] International Organization for Standardization. (2015). ISO 9000:2015 Quality Management Systems.
- [3] CMMI Product Team. (2010). CMMI for Development, Version 1.3.
- [4] Pyzdek, T., & Keller, P. (2014). The Six Sigma Handbook.
- [5] Strogatz, S. H. (2014). Nonlinear Dynamics and Chaos. Westview Press.
- [6] Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System.
- [7] Burns, B., et al. (2019). Kubernetes: Up and Running. O'Reilly Media.

Appendix A: Simulation Implementation Source Code

This appendix contains the core Java implementation for the chaos engineering simulations described in Chapter 4. The code leverages the Weka library for the data-driven model and a custom cellular automaton for the event-driven simulation.

B.1 Data-Driven Simulation (MLSimulation.java)

This class implements the chaos injection loop into the Logistic Regression model, including the retraining feedback mechanism.

```
1 package simulation;
2
3 import etl.CSVUtils;
4 import java.util.*;
5 import weka.classifiers.functions.Logistic;
6 import weka.core.*;
7
8 /**
9  * Data-driven simulation with chaotic perturbations.
10  * The system tests:
11  * - Sensitivity to small initial changes (chaos theory).
12  * - Increasing perturbations in input values.
13  * - Feedback: if performance drops, the model retrains.
14  */
15 public class MLSimulation {
16
17     // -----
18     // Add controlled noise: simulates chaos and system drift
19     // -----
20     public static double addChaos(double value, double noiseLevel) {
21         Random r = new Random();
22
23         // tiny perturbation: sensitivity to initial conditions
24         double tinyPerturbation = (r.nextDouble() - 0.5) * noiseLevel;
25
26         // butterfly effect: small perturbations can amplify error
27         return value + tinyPerturbation;
28     }
29
30     // -----
31     // Apply noise to the complete dataset
32     // -----
```

```

33     public static Instances applyChaos(Instances data, double
noiseLevel) {
34         Instances noisy = new Instances(data);
35
36         for (int i = 0; i < noisy.size(); i++) {
37             Instance inst = noisy.get(i);
38
39             for (int j = 0; j < inst.numAttributes(); j++) {
40                 if (j == noisy.classIndex()) continue; // target is
not touched
41
42                 double v = inst.value(j);
43                 inst.setValue(j, addChaos(v, noiseLevel));
44             }
45         }
46         return noisy;
47     }
48
49     // -----
50     // MAIN
51     // -----
52     public static void main(String[] args) throws Exception {
53
54         System.out.println("\n=== ML SIMULATION WITH CHAOS THEORY ==="
);
55
56         List<Map<String, String>> data =
57             CSVUtils.loadCSV("data/cleaned/cleaned.csv");
58
59         // Create attributes
60         ArrayList<Attribute> attrs = new ArrayList<>();
61         attrs.add(new Attribute("model_code"));
62         attrs.add(new Attribute("product_code"));
63         attrs.add(new Attribute("start_year"));
64         attrs.add(new Attribute("start_month"));
65         attrs.add(new Attribute("end_year"));
66         attrs.add(new Attribute("end_month"));
67         attrs.add(new Attribute("target"));
68
69         Instances dataset = new Instances("notebooks", attrs, data.
size());
70         dataset.setClassIndex(dataset.numAttributes() - 1);
71
72         // Convert rows
73         for (Map<String, String> row : data) {
74             double[] vals = new double[dataset.numAttributes()];
75             int i = 0;
76             for (Attribute a : attrs) {
77                 vals[i] = Double.parseDouble(row.get(a.name()));
78                 i++;
79             }
80             dataset.add(new DenseInstance(1.0, vals));
81         }
82
83         // Split
84         int trainSize = (int) (dataset.size() * 0.7);
85         Instances train = new Instances(dataset, 0, trainSize);

```

```

86     Instances test = new Instances(dataset, trainSize, dataset.
size() - trainSize);
87
88     Logistic model = new Logistic();
89     model.buildClassifier(train);
90
91     double baselineAccuracy = evaluate(model, test);
92     System.out.println("Initial Accuracy: " + baselineAccuracy);
93
94     //
=====
95     // CHAOS SIMULATION: perturbations are increased in each
iteration
96     //
=====
97     for (double noise = 0.01; noise <= 0.1; noise += 0.02) {
98
99         System.out.println("\n>> Applied noise: " + noise);
100
101         // Apply chaos to the test set
102         Instances chaoticTest = applyChaos(test, noise);
103
104         double accuracy = evaluate(model, chaoticTest);
105         System.out.println("Accuracy with chaos: " + accuracy);
106
107         // SYSTEM FEEDBACK
108         // If precision drops below 75% of baseline -> retrain
109         if (accuracy < baselineAccuracy * 0.75) {
110             System.out.println("WARN: Accuracy dropped too much.
Retraining...");
111             model.buildClassifier(train);
112         }
113     }
114 }
115
116 // Model evaluation
117 public static double evaluate(Logistic model, Instances test)
throws Exception {
118     double correct = 0;
119
120     for (int i = 0; i < test.size(); i++) {
121         double pred = model.classifyInstance(test.get(i));
122         double real = test.get(i).classValue();
123         if (Math.abs(pred - real) < 0.25) correct++;
124     }
125     return correct / test.size();
126 }
127 }

```

Listing 7.1: MLSimulation.java: Chaos injection and feedback loop

B.2 Event-Driven Simulation (EventSimulation.java)

This class models the system as a cellular automaton where state transitions are influenced by increasing entropy (chaos level), demonstrating emergent behaviors like hysteresis.

```
1 package simulation;
2
3 import java.util.Random;
4
5 /**
6  * Automaton-based simulation with chaotic dynamics.
7  * Modeled aspects:
8  * - transitions sensitive to randomness,
9  * - increasing perturbations,
10 * - feedback loops,
11 * - unstable states similar to chaotic systems.
12 */
13 public class EventSimulation {
14
15     enum State { NORMAL, HIGH_LOAD, BAD_DATA, PARTIAL_FAILURE,
16                 RECOVERY }
17
18     private State state = State.NORMAL;
19     private Random r = new Random();
20
21     // Chaos level -- increases with each cycle
22     private double chaosLevel = 0.02;
23
24     // Chaotic perturbation
25     public boolean chaoticEvent(double probability) {
26         // chaos amplifies probabilities
27         double p = probability + chaosLevel;
28         return r.nextDouble() < p;
29     }
30
31     public void nextCycle() {
32
33         System.out.println("Current state: " + state);
34
35         switch (state) {
36             case NORMAL:
37                 // small perturbations can cause bifurcations
38                 if (chaoticEvent(0.05)) state = State.HIGH_LOAD;
39                 if (chaoticEvent(0.03)) state = State.BAD_DATA;
40                 break;
41
42             case HIGH_LOAD:
43                 if (chaoticEvent(0.15)) state = State.PARTIAL_FAILURE;
44                 if (chaoticEvent(0.30)) state = State.RECOVERY;
45                 break;
46
47             case BAD_DATA:
48                 // incorrect values can escalate to major failures
49                 if (chaoticEvent(0.25)) state = State.PARTIAL_FAILURE;
50                 if (chaoticEvent(0.40)) state = State.RECOVERY;
51                 break;
```

```

52
53     case PARTIAL_FAILURE:
54         // feedback: the system attempts to recover
55         if (chaoticEvent(0.60)) state = State.RECOVERY;
56         break;
57
58     case RECOVERY:
59         state = State.NORMAL;
60         break;
61 }
62
63 // Every cycle chaos grows a little -> non-linear dynamics
64 chaosLevel += 0.005;
65 }
66
67 public static void main(String[] args) {
68
69     EventSimulation sim = new EventSimulation();
70
71     System.out.println("\n=== EVENT SIMULATION WITH CHAOS THEORY
72     ===\n");
73
74     // run for 30 cycles
75     for (int i = 0; i < 30; i++) {
76         System.out.println("---- Cycle " + i + " ----");
77         sim.nextCycle();
78     }
79 }

```

Listing 7.2: EventSimulation.java: Cellular Automaton with Entropy