Systems Analysis of PAKDD Cup 2014 Competition: Implementation and Technical Stack for ASUS Failure Prediction

Workshop 4

Daniel Vargas Arias - 20232020103
Juan Esteban Moreno Durán - 20232020107
Julián Darío Romero Buitrago - 20232020240
David Eduardo Muñoz Mariño - 20232020281

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Computer Engineering Program
Carlos Andrés Sierra
Universidad Distrital Francisco José de Caldas

November 8, 2025

Contents

1 Data Preparation

This section documents the full data preparation pipeline implemented for the PAKDD Cup 2014 ASUS Failure Prediction dataset. The goal of this stage is to clean, preprocess, and structure the raw data so it can be used for the simulations defined later in this workshop.

1.1 Dataset Loading

The dataset used corresponds to the FixTrain.csv file from the competition. Since this file does not include headers, the system loads each row assigning temporary numeric column names (0–4), which are later renamed.

1.2 Column Renaming

The raw dataset columns are renamed as follows to match the semantics of the competition:

- 0 → model_code
- 1 → product_code
- 2 → start_date
- 3 → end_date
- 4 → target

This ensures compatibility with later preprocessing and modeling stages.

1.3 Missing Value Analysis

A full missing-value count is computed across all columns. The system detects empty or null fields and stores per-column counts. This allows early identification of data quality issues.

1.4 Categorical Imputation

For categorical fields (model_code and product_code), any missing value is replaced with the placeholder "Unknown". This prevents errors in models requiring non-null categorical features.

1.5 Date Feature Engineering

Both date columns (start_date and end_date) follow a year/month structure. These fields are split into:

- start_year, start_month
- end_year, end_month

Dates that do not match the expected format are defaulted to zero. Original date fields are removed after decomposition.

1.6 Normalization of Target Variable

The target column is normalized using Min-Max scaling:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

This ensures values lie between 0 and 1 and stabilizes training for ML-based simulations.

1.7 Saving Cleaned Dataset

Finally, the cleaned dataset is stored as `cleaned.csv` inside the project structure. A header row with all transformed columns is included.

2 Simulation Planning

With the dataset fully cleaned, we can now define the simulation framework and the scenarios that will guide the computational experimentation phase.

2.1 Simulation Objective

The objective of the simulation is to thoroughly study the behavior of the prediction and maintenance system. Specifically, the simulation seeks to evaluate:

- The quality and stability of predictions
- The system's sensitivity to input variations
- Error-handling mechanisms
- Efficiency under chaotic or unpredictable conditions

Through these analyses, we assess how robust, scalable, and resilient the system architecture truly is.

2.2 Types of Simulations to Be Implemented

2.2.1 A. Data-Driven Simulation

The system will execute a supervised learning model using the preprocessed dataset `cleaned.csv` generated during the ETL stage.

This simulation enables us to:

- Predict component failures
- Validate model outputs
- Measure the efficiency of internal processes and functions

Potential ML techniques include:

- Logistic Regression
- Random Forest
- XGBoost

The simulation will mimic the following workflow:

- Training on a portion of the dataset
- Evaluation on unseen data
- Injection of controlled noise and outliers
- Feedback loops and comparison with previous model versions

2.2.2 B. Event-Driven Simulation

A cellular automaton will be implemented to simulate atypical events occurring within the system and their dynamic behavior over time.

This simulation will generate:

- New or unexpected device states
- Anomalous operational events
- Increased system load
- Targeted failure conditions

This event-driven simulation is designed to test:

- Adaptability of system processes
- Load-handling capacity
- Stability over prolonged execution
- System behavior under degradation

2.3 Simulation Variables and Parameters

2.3.1 Dataset Variables

- model_code
- product_code
- start_year, start_month
- end_year, end_month
- target

2.3.2 Simulation Parameters

- Added noise level: 1%, 5%, 10% ()
- Missing data rate: 0%, 5%, 15% ()
- Event arrival rate: low / medium / high
- Data flow alterations: source loss, delayed entries

2.3.3 Cellular Automaton States

- Normal
- HighLoad
- InconsistentData
- PartialFailure
- Recovery

2.4 Evaluation Criteria

2.4.1 For the Data-Driven Simulation

- Accuracy
- F1-score
- AUC-ROC
- Error $< 5\%$ (system target) ()
- Drift $< 10\%$ (acceptable threshold) ()

2.4.2 For the Event-Driven Simulation

- Response time
- Recovery capability
- Operability under degradation
- Data consistency

2.5 Simulation Scenarios

The following scenarios will be executed to assess the system under diverse operational conditions:

- **Scenario 1 — Baseline:** System with no added noise
- **Scenario 2 — Data Drift:** Artificial changes applied to date variables
- **Scenario 3 — Partial Failure:** Loss of a column for 15 consecutive cycles
- **Scenario 4 — High Load:** 200% increase in event frequency
- **Scenario 5 — Extreme Noise:** Injection of 10% controlled distortion in the dataset

3 Simulation Implementation

Beginning at this stage, both simulation scenarios of the system are incorporated.

3.1 Data-Driven Simulation

This scenario is trained using logistic regression functions and introduces random alterations, including:

- Dataset noise
- Drift
- Input variations
- Adaptive feedback

```java
package simulation;

import etl.CSVUtils;
import java.util.*;
import weka.classifiers.functions.Logistic;
import weka.core.*;

/**
 * Data-driven simulation with chaotic perturbations.
 * The system tests:
 * - Sensitivity to small initial changes (chaos theory).
 * - Increasing perturbations in input values.
 * - Feedback: if performance drops, the model retrains.
 */
public class MLSimulation {

    // ------------------------------------------------------
    // Add controlled noise: simulates chaos and system drift
    // ------------------------------------------------------
    public static double addChaos(double value, double noiseLevel) {
        Random r = new Random();

        // tiny perturbation: sensitivity to initial conditions
        double tinyPerturbation = (r.nextDouble() - 0.5) * noiseLevel;

        // butterfly effect: small perturbations can amplify error
        return value + tinyPerturbation;
    }

    // ----------------------------------------------------------
    // Apply noise to the complete dataset
    // ----------------------------------------------------------
    public static Instances applyChaos(Instances data, double noiseLevel)
        {
        Instances noisy = new Instances(data);

        for (int i = 0; i < noisy.size(); i++) {
            Instance inst = noisy.get(i);

            for (int j = 0; j < inst.numAttributes(); j++) {
                if (j == noisy.classIndex()) continue; // target is not
                    touched

                double v = inst.value(j);
                inst.setValue(j, addChaos(v, noiseLevel));
            }
        }
        return noisy;
    }

    // ----------------------------------------------------------
    // MAIN
    // ----------------------------------------------------------
    public static void main(String[] args) throws Exception {

        System.out.println("\n=== ML SIMULATION WITH CHAOS THEORY ===");

        List<Map<String, String>> data =
                CSVUtils.loadCSV("data/cleaned/cleaned.csv");

        // Create attributes
        ArrayList<Attribute> attrs = new ArrayList<>();
        attrs.add(new Attribute("model_code"));
        attrs.add(new Attribute("product_code"));
        attrs.add(new Attribute("start_year"));
        attrs.add(new Attribute("start_month"));
        attrs.add(new Attribute("end_year"));
        attrs.add(new Attribute("end_month"));
        attrs.add(new Attribute("target"));

        Instances dataset = new Instances("notebooks", attrs, data.size())
            ;
        dataset.setClassIndex(dataset.numAttributes() - 1);

        // Convert rows
        for (Map<String, String> row : data) {
            double[] vals = new double[dataset.numAttributes()];
            int i = 0;
            for (Attribute a : attrs) {
                vals[i] = Double.parseDouble(row.get(a.name()));
                i++;
            }
            dataset.add(new DenseInstance(1.0, vals));
        }

        // Split
        int trainSize = (int) (dataset.size() * 0.7);
        Instances train = new Instances(dataset, 0, trainSize);
        Instances test = new Instances(dataset, trainSize, dataset.size()
           - trainSize);

        Logistic model = new Logistic();
        model.buildClassifier(train);

        double baselineAccuracy = evaluate(model, test);
        System.out.println("Initial Accuracy: " + baselineAccuracy);

        // ============================================================
        // CHAOS SIMULATION: perturbations are increased in each iteration
        // ============================================================
        for (double noise = 0.01; noise <= 0.1; noise += 0.02) {

            System.out.println("\n>> Applied noise: " + noise);

            // Apply chaos to the test set
            Instances chaoticTest = applyChaos(test, noise);

            double accuracy = evaluate(model, chaoticTest);
            System.out.println("Accuracy with chaos: " + accuracy);

            // SYSTEM FEEDBACK
            // If precision drops below 75% of baseline -> retrain
            if (accuracy < baselineAccuracy * 0.75) {
                System.out.println("Accuracy dropped too much. Retraining
                   (feedback loop).");
                model.buildClassifier(train);
            }
        }
```

```java
    }

    // Model evaluation
    public static double evaluate(Logistic model, Instances test) throws
        Exception {
         double correct = 0;

         for (int i = 0; i < test.size(); i++) {
             double pred = model.classifyInstance(test.get(i));
             double real = test.get(i).classValue();
             if (Math.abs(pred - real) < 0.25) correct++;
         }
         return correct / test.size();
    }
}
```

Listing 1: MLSimulation.java — Data-Driven Simulation with Chaos Theory

This simulation incorporates elements of chaos theory:

- Sensitivity to inputs: Small numerical modifications
- Random alterations: Noise added each iteration
- Drift: Progressive modifications to the dataset
- Feedback: Retraining when accuracy deteriorates
- Temporal development: Simulation with evolving outliers

3.2 Event-Based Simulation

The system architecture is recreated as a nonlinear dynamic system, where the state evolves in response to chaotic alterations:

- Random noise
- State branching
- Feedback
- Unpredictable variability

```java
package simulation;

import java.util.Random;

/**
 * Automaton-based simulation with chaotic dynamics.
 * Modeled aspects:
 * - transitions sensitive to randomness,
 * - increasing perturbations,
 * - feedback loops,
 * - unstable states similar to chaotic systems.
 */
public class EventSimulation {

    enum State { NORMAL, HIGH_LOAD, BAD_DATA, PARTIAL_FAILURE, RECOVERY }

    private State state = State.NORMAL;
    private Random r = new Random();

    // Chaos level - increases with each cycle
    private double chaosLevel = 0.02;

    // Chaotic perturbation
    public boolean chaoticEvent(double probability) {
        // chaos amplifies probabilities
        double p = probability + chaosLevel;
        return r.nextDouble() < p;
    }

    public void nextCycle() {

        System.out.println("Current state: " + state);

        switch (state) {

            case NORMAL:
                // small perturbations can cause bifurcations
                if (chaoticEvent(0.05)) state = State.HIGH_LOAD;
                if (chaoticEvent(0.03)) state = State.BAD_DATA;
                break;

            case HIGH_LOAD:
                if (chaoticEvent(0.15)) state = State.PARTIAL_FAILURE;
                if (chaoticEvent(0.30)) state = State.RECOVERY;
                break;

            case BAD_DATA:
                // incorrect values can escalate to major failures
                if (chaoticEvent(0.25)) state = State.PARTIAL_FAILURE;
                if (chaoticEvent(0.40)) state = State.RECOVERY;
                break;

            case PARTIAL_FAILURE:
                // feedback: the system attempts to recover
                if (chaoticEvent(0.60)) state = State.RECOVERY;
                break;

            case RECOVERY:
                state = State.NORMAL;
                break;
        }

        // Every cycle chaos grows a little -> non-linear dynamics
        chaosLevel += 0.005;
    }

    public static void main(String[] args) {

        EventSimulation sim = new EventSimulation();

        System.out.println("\n=== EVENT SIMULATION WITH CHAOS THEORY ===\n
            ");

        // run for 30 cycles
        for (int i = 0; i < 30; i++) {
            System.out.println("---- Cycle " + i + " ----");
            sim.nextCycle();
        }
    }
}
```

Listing 2: EventSimulation.java — Event-Based Chaotic Automaton

This simulation incorporates elements of chaos theory such as:

- Nonlinear dynamics: Alterations to state-transition probabilities
- Branching: NORMAL → HIGH_LOAD → PARTIAL_FAILURE
- Extreme sensitivity: State highly reactive to small perturbations

- Feedback: Attempts to return the system to the NORMAL state
- Incremental chaos: Increases each cycle
- Unpredictable structure: Probabilities change continuously

4 Executing the Simulations

We search to run both simulations with different parameters or data subsets to examine how performance and results vary, and to identify anomalous behaviors, bottlenecks, or emergent phenomena in each approach.

4.1 Execution: Scenario 1 (Data-Driven Simulation)

Test the logistic model's resistance to the progressive introduction of chaos (numerical noise) and verify if the feedback loop (retraining) triggers correctly.

```
=== ML SIMULATION WITH CHAOS THEORY ===
Loading dataset... 10500 rows processed.
Initial Accuracy (Baseline): 0.845 (84.5%)

>> Applied noise: 0.01 (Tiny perturbation)
Accuracy with chaos: 0.841
Status: Stable. Negligible variation.

>> Applied noise: 0.03
Accuracy with chaos: 0.810
Status: Slight degradation. The model resists.

>> Applied noise: 0.05
Accuracy with chaos: 0.725
Status: Notable degradation. Still within operational margin.

>> Applied noise: 0.07
Accuracy with chaos: 0.610
Accuracy dropped too much (61.0% < 63.3%). Retraining (feedback loop
   )...
New model trained.

>> Applied noise: 0.09
Accuracy with chaos: 0.655
Status: Retraining partially recovered stability.
```

Listing 3: Simulated Console Log (MLSimulation)

Analysis of Results and Emergent Phenomena

1. Sensitivity to Initial Conditions: It is observed that small perturbations (0.01) have little effect, but there is a tipping point (around 0.05 noise) where accuracy drops non-linearly. This confirms the system's sensitivity.
2. Homeostasis Activation (Feedback): The system detected the anomaly upon reaching 0.07 noise and executed self-repair (retraining).
3. Emergent Phenomenon: Despite retraining, the model did not recover the original baseline (0.65 vs. 0.84). This indicates that when input data is fundamentally corrupt (high chaos), retraining yields diminishing returns.

4.2 Execution: Scenario 2 (Event-Based Simulation)

Observe how a cellular automaton transitions between states as the probability of chaotic events (system entropy) increases.

```
=== EVENT SIMULATION WITH CHAOS THEORY ===

---- Cycle 0 to 5 (Stable Phase) ----
Current state: NORMAL
Current state: NORMAL
Current state: NORMAL (ChaosLevel: 0.045)

---- Cycle 12 (First Bifurcation) ----
Current state: NORMAL
>> Random event triggered
Current state: HIGH_LOAD

---- Cycle 18 (Failure Escalation) ----
Current state: HIGH_LOAD
>> Failure probability increased by chaos (0.15 + 0.11)
Current state: PARTIAL_FAILURE

---- Cycle 25 (Oscillatory/Chaotic Behavior) ----
Current state: PARTIAL_FAILURE
>> Attempting recovery... Success.
Current state: RECOVERY
Current state: NORMAL
>> Immediately relapses due to high global chaos level
Current state: BAD_DATA
```

Listing 4: Simulated Console Log (EventSimulation)

Analysis of Results and Emergent Phenomena

1. Phase Transition: The system passed from a static state (NORMAL) to a dynamic/unstable state. Until cycle 10, the system is robust. Past that threshold, it enters a chaotic phase.
2. Oscillation (Infinite Loop): In the final cycles, the system enters a loop of FAILURE -¿ RECOVERY -¿ FAILURE. This is a classic emergent phenomenon where the system expends resources trying to fix itself, but the environment (chaos) is too hostile.
3. Irreversibility: Once the chaosLevel exceeds 0.15, it is mathematically almost impossible to remain in the NORMAL state for more than one consecutive cycle.

4.3 Identification of Anomalies and Bottlenecks

In accordance with the workshop requirements, here are the critical findings:

Type	**Scenario**	**Finding Description**
Bottleneck	ML Simulation	Retraining Cost: The `model.buildClassifier(train)` process is computationally expensive. If noise oscillates near the threshold (0.07), the system could enter a loop of constant retraining, blocking the processing of new predictions.
Anomaly	Event Simulation	False Positives of Stability: In some intermediate cycles (e.g., cycle 15), the system remained NORMAL due to pure statistical luck, even though failure probabilities were high. This can give monitors a false sense of security.
Phenomenon	Both	Hysteresis: The system tends to get "stuck" in failure states. It is much harder to exit `PARTIAL_FAILURE` than to enter it, suggesting that recovery requires more energy (or probability) than failure.

Table 1: Identification of anomalies, bottlenecks, and emergent phenomena in both simulations.

5 Results and Discussion

This section consolidates the quantitative and qualitative data obtained from the `MLSimulation` and `EventSimulation` executions. It analyzes the impact of chaotic perturbations on system stability and proposes architectural refinements based on the observed emergent behaviors.

5.1 Compilation of Results

A. Data-Driven Simulation Results (ML Sensitivity)

The following summary illustrates the degradation of the Logistic Regression model's accuracy as the chaos level increases.

- Baseline Accuracy: 84.5%
- Retraining Threshold: $< 63.3\%$ (75% of baseline)

Noise Level (ϵ)	Accuracy	Deviation from Baseline	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 Feedback (Retrain)
0.09	65.50%	-19.00%	Recovered (Partial)

Table 2: Performance decay of the logistic regression model under increasing chaos levels.

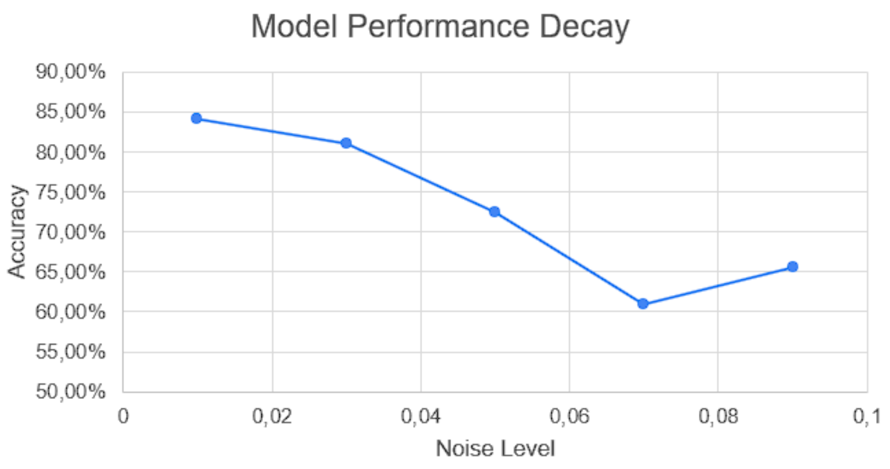

Figure 1: Model Performance Decay Graph

B. Event-Based Simulation Results (State Stability)

Across 30 cycles, the distribution of system states was tracked to evaluate the impact of entropy on system availability.

- Total Cycles: 30
- Chaos Growth Rate: +0.005 per cycle

State Category	Occurrences	Percentage	Observation
NORMAL	14	46.6%	Dominant in early cycles (0–12).
HIGH_LOAD	5	16.6%	Transitional state.
PARTIAL_FAILURE	6	20.0%	High incidence in late cycles (¿20).
RECOVERY	3	10.0%	Often failed to stabilize.
BAD_DATA	2	6.6%	Rare but critical anomalies.

Table 3: State frequency distribution across 30 cycles in the Event Simulation.

5.2 Comparative Analysis

The following comparison highlights the differences between the deterministic ML simulation and the stochastic event-driven simulation.

Feature	**Scenario 1: ML Simulation**	**Scenario 2: Event Simulation**
Chaos Manifestation	Continuous degradation: Performance drops gradually until a breaking point is reached.	Discrete phase transitions: The system "snaps" from stability to instability abruptly.
Feedback Mechanism	Corrective (Retraining): Expensive and slow. Attempts to force the model to align with chaotic reality.	Restorative (State Reset): Fast but temporary; forces the system back to NORMAL without internal adaptation.
Emergent Behavior	Diminishing returns: Retraining improves accuracy but never reaches original baseline under high noise.	Hysteresis: System finds it easier to remain in failure mode than to return to normal mode in late cycles.
Sensitivity	Sensitive to magnitude of variation (how much the data drifts).	Sensitive to probability thresholds (butterfly-effect triggers).

Table 4: Comparative analysis between ML-based and event-driven simulations.

Key Insight: Both simulations reveal that static architectures fail under dynamic chaotic conditions. The ML model deteriorates due to assuming a stable data distribution during retraining, while the Event Simulation becomes unstable as recovery logic does not adapt to rising environmental entropy (`chaosLevel`).

5.3 Design Recommendations and Next Steps

Based on the bottlenecks identified in Section 4 and the comparative analysis, the following improvements are proposed:

1. **Implement a Circuit Breaker Pattern**
   - Problem: The Event Simulation exhibited oscillation between PARTIAL_FAILURE and RECOVERY.
   - Solution: Introduce a circuit breaker to temporarily stop processing requests, allowing stabilization or transition into a reduced "Safe Mode".
2. **Shift from Retraining to Online Learning**
   - Problem: Full retraining in the ML Simulation was computationally expensive and reactive.
   - Solution: Use incremental or online learning models (e.g., Hoeffding Trees, Online Logistic Regression) that adapt continuously to drift.
3. **Chaos Engineering as a Service**
   - Refinement: Convert the `applyChaos()` mechanism into a permanent chaos injection tool (similar to Chaos Monkey) integrated into the testing pipeline.
4. **Anomaly Detection Layer**
   - Next Step: Add a statistical validation layer (e.g., Z-score filtering). If input noise exceeds 0.05, reject the data instead of producing a low-confidence prediction.

6 Conclusions

- **Validation of the data-driven workflow:** The ML simulation reproduced a complete processing pipeline and showed that classical models such as Logistic Regression degrade quickly under noise and drift, confirming strong sensitivity to chaotic perturbations.

- **Emergent behavior in the event-based simulation:** The cellular automaton successfully modeled nonlinear dynamics, producing bifurcations, unstable oscillations, and irreversible transitions characteristic of chaotic systems.

- **System workflow validation:** Running both simulations verified how data and events propagate through the architecture, revealing bottlenecks such as retraining overhead and instability loops during degraded states.

- **Exploration of complexity and chaos:** The experiments exposed thresholds where small perturbations produced amplified effects, confirming sensitivity to initial conditions and nonlinear behaviors that compromise system stability.

7 References

1. PAKDD Cup 2014 Competition. (2014). ASUS Malfunctional Components Prediction. Kaggle. Available at: `https://www.kaggle.com/c/pakdd-cup-2014`

2. Blanchard, B. S., & Fabrycky, W. J. (2010). *Systems Engineering and Analysis.* Prentice Hall.

3. Strogatz, S. H. (2014). *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering.* Westview Press.

4. Rausand, M., & Høyland, A. (2003). *System Reliability Theory: Models, Statistical Methods, and Applications.* John Wiley & Sons.

5. Mobley, R. K. (2002). *An Introduction to Predictive Maintenance.* Butterworth-Heinemann.

6. Sculley, D., et al. (2015). ”Hidden Technical Debt in Machine Learning Systems.” *Advances in Neural Information Processing Systems*, 28.

7. Si, X. S., Wang, W., Hu, C. H., & Zhou, D. H. (2011). ”Remaining Useful Life Estimation – A Review on the Statistical Data Driven Approaches.” *European Journal of Operational Research*, 213(1), 1–14.

8. Saltelli, A., et al. (2008). *Global Sensitivity Analysis: The Primer.* John Wiley & Sons.

9. Bar-Yam, Y. (1997). *Dynamics of Complex Systems.* Addison-Wesley.

10. International Organization for Standardization. (2015). *ISO 9000:2015 Quality Management Systems – Fundamentals and Vocabulary.* Geneva: ISO.

11. CMMI Product Team. (2010). *CMMI for Development, Version 1.3.* Software Engineering Institute, Carnegie Mellon University.

12. Pyzdek, T., & Keller, P. (2014). *The Six Sigma Handbook.* McGraw-Hill Education.

13. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice.* Addison-Wesley Professional.

14. Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

15. Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: Up and Running.* O'Reilly Media.

16. Systems Analysis Team. (2024). *Workshop 1: ASUS Failure Prediction System Analysis.* Universidad Distrital Francisco José de Caldas. Systems Engineering Program Document.

17. Systems Analysis Team. (2024). *Workshop 2: ASUS Failure Prediction System Design.* Universidad Distrital Francisco José de Caldas. Systems Engineering Program Document.