

Smart Sentry: A Real-Time, Phase-Aware Anomaly Detection System for Industrial Assets

Final Report for the Baker Hughes Hackathon 2025

Submitted by:
Harshit Bhandari
Srushti Nerkar
Date:
September 15, 2025

1. Introduction

1.1. Background and Motivation

In modern industries, physical assets like turbines, motors, and pumps generate vast amounts of high-frequency sensor data throughout their operational lifecycle¹. This data is critical for predictive maintenance, which aims to detect potential failures before they occur. A significant challenge in this field is that most anomaly detection systems are designed to monitor assets during steady-state operations². However, critical failures often emerge during transitional phases, such as startup or shutdown, where sensor readings exhibit volatile but normal behavior³. Distinguishing these normal transient spikes from genuine anomalies remains a significant and unresolved challenge for many industrial operators⁴.

1.2. Problem Statement

This project addresses the "Smart Sentry" challenge from the Baker Hughes Hackathon 2025⁵. The primary goal is to build a real-time anomaly detection system that can correctly handle the full operational lifecycle of an industrial asset⁶. The system must be capable of ingesting streaming sensor data, learning the normal operational patterns for startup, steady-state, and shutdown phases separately, and alerting users only when genuine failures occur⁷⁷⁷⁷. A key objective is to minimize false positives, especially during the volatile transient phases⁸.

1.3. Project Objectives

To meet the challenge, this project aimed to achieve the following objectives:

- Develop a robust algorithm to accurately classify the asset's real-time operational phase.
- Implement and evaluate a phase-aware anomaly detection engine.
- Engineer a complete, end-to-end streaming data architecture using modern technologies like Kafka and Redis.
- Build an interactive dashboard for real-time visualization and alerting.
- Achieve high performance on the anomaly detection task, with a specific focus on balancing high recall (catching real failures) and high precision (avoiding false alarms).

1.4. Scope and Limitations

The scope of this project is to develop a fully functional software prototype. The system is trained and evaluated on the NASA Turbofan Engine Degradation Simulation Dataset (CMAPSS), specifically the

FD001 subset⁹. The solution includes all components from data ingestion and processing to a final user interface. This report does not cover deployment on a physical industrial asset; however, a detailed plan for edge deployment is included as per the bonus requirement¹⁰.

1.5. Report Structure

This report details the complete lifecycle of the "Smart Sentry" project. Section 2 describes the dataset and the technology stack used. Section 3 provides a deep dive into the methodology, covering phase classification, anomaly label generation, and the development of the machine learning models. Section 4 outlines the final system architecture. Section 5 presents and discusses the final performance results. Finally, Section 6 provides the conclusion and potential avenues for future work.

2. Dataset and Technology Stack

This section details the dataset used for training and evaluating our model, along with a comprehensive overview of the technologies and frameworks that constitute the project's architecture.

2.1. Dataset Description: NASA CMAPSS

For this project, we utilized the

NASA Turbofan Engine Degradation Simulation Dataset (CMAPSS), as recommended in the problem statement. This is a widely used benchmark dataset in the field of prognostics and predictive maintenance.

The dataset consists of multivariate time-series data from a fleet of simulated turbofan engines. Key characteristics of this dataset include:

- **Run-to-Failure Data:** Each time-series tracks a single engine's entire operational life from a healthy state until a point of failure, which is critical for training predictive models.
- **Multiple Operational Conditions:** The data is not limited to steady-state operation; it includes various operational conditions and captures the engine's behavior during **transient states like startup and shutdown**.
- **Rich Sensor Data:** The dataset contains readings from 21 different sensors, along with 3 operational settings for each engine at every cycle, providing a rich feature set for

machine learning.

For the scope of this project, we focused on the `FDOO1` subset for both training and testing.

2.2. Technology Stack Overview

The technology stack for this project was chosen to build a robust, scalable, and real-time system, drawing from the suggested technologies in the problem statement .

- **Core Development & Machine Learning:**
 - **Python:** The primary programming language for all components.
 - **Pandas & NumPy:** Used for all data manipulation, feature engineering, and numerical operations.
 - **Scikit-learn:** The core machine learning library used for implementing our final `RandomForestClassifier` and `LogisticRegression` models, as well as for all performance evaluation metrics.
 - **XGBoost:** Employed as a powerful alternative gradient boosting model for comparison.
 - **Imbalanced-learn:** Utilized for the SMOTE (Synthetic Minority Over-sampling Technique) to address the class imbalance in the training data.
- **Data Streaming & Real-Time Communication:**
 - **Kafka:** Implemented as the high-throughput message bus for ingesting the simulated real-time sensor data stream.
 - **Redis:** Used as a high-speed, in-memory message broker to facilitate low-latency communication between the backend consumer and the frontend dashboard.
- **Database:**
 - **PostgreSQL:** Chosen as the robust relational database for the persistent logging of all prediction results and system performance metrics.
- **Frontend & Visualization:**
 - **Streamlit:** The framework used to rapidly build the interactive, real-time user dashboard for visualization and alerting.
 - **Seaborn & Matplotlib:** Used for creating static plots and visualizations during the data exploration and analysis phases.
- **Infrastructure:**
 - **Docker & Docker Compose:** Used to containerize and manage all the backend services (Kafka, PostgreSQL, Redis), ensuring a consistent, reproducible, and easily deployable environment.

3. Methodology and Implementation

This section provides a detailed account of the technical approach and implementation steps taken to build the "Smart Sentry" system. It covers the journey from initial data exploration to

the development of our final, robust phase classification algorithm.

3.1. Data Preprocessing and Exploration

The project began with a thorough exploration of the NASA CMAPSS `FD001` dataset. The initial steps included:

1. **Data Loading:** The raw `train_FD001.txt` file was loaded into a pandas DataFrame.
2. **Data Cleaning:** Column headers were assigned based on the official dataset documentation. Two empty trailing columns were identified and removed to ensure data integrity.
3. **Exploratory Data Analysis (EDA):** We visualized the operational settings and sensor readings for individual engines over their complete lifecycle. This initial analysis revealed two key insights:
 - The data clearly demonstrated a "run-to-failure" pattern, with sensor values showing gradual degradation over time.
 - The operational settings fluctuated, indicating that the engines were not run under a single condition but were subjected to different operational modes, confirming the presence of distinct phases.

3.2. Phase Classification: Distinguishing Operational States

A core requirement of the project was to build a "phase-aware" system. Since the dataset does not include explicit phase labels, our first major task was to develop a reliable method for classifying each operational cycle. We experimented with two primary approaches.

3.2.1. Initial Approach: K-Means Clustering

Our first attempt was to use an unsupervised machine learning model to automatically discover the operational states.

- **Methodology:** We applied the K-Means clustering algorithm to the key operational settings and stable sensor readings. The "Elbow Method" was used to determine the optimal number of clusters, which was found to be three.
- **Outcome:** While K-Means was effective at grouping the data into different operational *levels* (e.g., high-power vs. low-power steady states), it was unable to reliably distinguish between the *processes* of `STARTUP` and `SHUTDOWN`. A single point in time during acceleration can look very similar to a point during deceleration, and the algorithm, lacking a concept of time, could not separate them effectively. This led us to pivot to a more context-aware approach.

3.2.2. Final Approach: A Robust Rule-Based Classifier

Our final and most successful approach was a rule-based algorithm that mimics an expert's analysis by directly modeling the physical behavior of the engine. This classifier operates in

two steps:

1. **Stability Check:** The algorithm first identifies all **STEADY-STATE** periods. It does this by calculating the rolling standard deviation (volatility) of a key operational setting (**op_setting_1**). Any cycle where the volatility fell below a data-driven threshold of 0.0019 was classified as **STEADY-STATE**. This threshold was determined by analyzing the bimodal distribution of volatility values across the entire dataset.
2. **Trend Check:** For all remaining transient cycles, the algorithm then determines the direction of change. It calculates the trend (slope) of a heavily smoothed RPM-like sensor (**sensor_7**).
 - If the trend was **positive**, indicating acceleration, the cycle was classified as **STARTUP**.
 - If the trend was **negative**, indicating deceleration, the cycle was classified as **SHUTDOWN**.

3.2.3. Validation of Phase Classification

Since no ground truth phase labels exist, we used a combination of methods to validate our final rule-based classifier.

- **Visual Inspection:** We plotted the sensor data for individual engines and color-coded each point by its assigned phase. The resulting visualization showed clear, logical bands of **STARTUP**, **STEADY-STATE**, and **SHUTDOWN** phases that aligned perfectly with our engineering understanding of an engine's lifecycle.
- **Statistical Analysis:** We quantitatively confirmed our results by calculating the standard deviation of key sensors within each phase group. The analysis showed that the **STEADY-STATE** group had significantly lower volatility than the **STARTUP** and **SHUTDOWN** groups, providing strong numerical evidence for the classification's correctness.
- **Downstream Task Performance:** The most important validation was the dramatic improvement in the performance of our final anomaly detection model when using these phase labels as a feature. This confirmed that our classification was not only accurate but also highly valuable for solving the core problem.

3.3. Anomaly Label Generation

To train a supervised model and to quantitatively evaluate any model, a set of "ground truth" labels is required. As the original dataset does not contain explicit anomaly flags, we created them using the provided **"Run-to-failure labels"**.

3.3.1. Defining the Anomaly Window

Our approach was to define a fixed window of time just before the known end-of-life for each engine and classify this period as anomalous. Based on common practice for this dataset, we

defined an **"anomaly window" of 30 cycles**.

For every data point in both the training and test sets, we applied the following rule:

- If the engine's current cycle was within 30 cycles of its total failure point, it was labeled as **1 (Anomaly)**.
- Otherwise, it was labeled as **0 (Normal)**.

This heuristic transformed the problem into a standard binary classification task, enabling us to train and evaluate supervised learning models effectively.

3.3.2. Importance and Justification of the Anomaly Window Heuristic

The decision to define anomalies using a fixed window before failure was one of the most critical methodological steps in this project.

Why was this heuristic important?

This heuristic was important because it allowed us to **transform the problem from an unsupervised task into a supervised one**. Our initial unsupervised `IsolationForest` model struggled because it had no "answer key" to learn from. By creating `true_anomaly` labels, we provided a clear target for a supervised model like `RandomForestClassifier` to learn, which was the key to achieving the high performance required by the project goals.

How can this window define an anomaly?

The validity of this heuristic is grounded in the physical process of mechanical failure. Engine degradation is typically a gradual process, not an instantaneous event.

- As internal components like bearings or blades wear down, the engine's performance changes.
- These changes are reflected in the sensor data as subtle but consistent drifts away from the healthy baseline (e.g., rising temperatures, increasing vibrations).
- While minor deviations can occur at any time, these drifts become more pronounced, consistent, and irreversible as the engine approaches its end-of-life.

Our **30-cycle anomaly window** is a practical method for capturing this final, critical phase. It defines a period where the engine's behavior is undeniably "anomalous" compared to its healthy state, and the patterns in the sensor data are most indicative of impending failure. While the exact start of this critical phase is not a single point in time, the window serves as a robust and standard proxy, enabling our model to effectively learn the patterns that precede a failure.

3.4. Anomaly Detection Modeling

This section details the iterative process of developing the anomaly detection engine, starting

with a baseline unsupervised model and progressing to a highly optimized supervised model.

3.4.1. Initial Unsupervised Approach: Phase-Aware IsolationForest

Following the problem statement's suggestion, our initial attempt was a phase-aware unsupervised model. We trained three separate IsolationForest models, one for each operational phase identified by our rule-based classifier (STARTUP, STEADY-STATE, SHUTDOWN). The goal was for each model to learn the "normal" data distribution for its specific phase.

However, when evaluated against the ground truth labels on the test set, this approach yielded a very low **recall of less than 10%**. This meant it failed to detect the vast majority of true anomalies. We concluded that the gradual nature of the engine's degradation was not effectively captured as "outliers" by the unsupervised model, necessitating a pivot.

3.4.2. Pivoting to Supervised Learning

Given the poor performance of the unsupervised model and the availability of the run-to-failure labels, we made the key decision to switch to a supervised learning methodology. This allowed us to explicitly teach a model the specific patterns of sensor data that are characteristic of the final, critical degradation window.

3.4.3. Handling Class Imbalance with SMOTE

Upon creating the anomaly labels, we confirmed a significant class imbalance in the training data: the anomaly class represented only a small fraction of the total samples. To prevent the model from becoming biased towards the majority "Normal" class, we implemented the **SMOTE (Synthetic Minority Over-sampling Technique)**. SMOTE balances the dataset by creating new, synthetic examples of the minority (anomaly) class, providing the model with a richer and more balanced dataset to learn from.

3.5. Model Selection and Hyperparameter Tuning

To find the best-performing model for this task, we conducted a systematic comparison and tuning process.

3.5.1. Comparing RandomForestClassifier, XGBoost, and LogisticRegression

We trained three powerful supervised models on the same SMOTE-balanced training data and evaluated them on the unseen test set. The models compared were:

- **RandomForestClassifier**: An ensemble of decision trees.
- **XGBoost**: A high-performance gradient boosting model.
- **LogisticRegression**: A strong linear baseline model.

The RandomForestClassifier was selected as the champion model because it provided the highest **F1-Score (0.71)**, representing the best balance between precision (0.66) and recall

(0.78).

3.5.2. Hyperparameter Tuning the Best Model

As a final optimization step, we used `RandomizedSearchCV` to search for the optimal hyperparameters for the `RandomForestClassifier`. While the tuning process yielded a model with slightly higher precision, the original, simpler `RandomForestClassifier` maintained a better overall balance and a higher F1-score. Therefore, the untuned `RandomForestClassifier` was retained as the final, best-performing model for the project.

4. System Architecture

This section details the technical architecture of the "Smart Sentry" system, which was designed to be robust, scalable, and capable of processing data in real time.

4.1. High-Level Architecture: A Decoupled Streaming Pipeline

The system is built on a modern, producer-consumer architectural pattern, which is ideal for real-time streaming applications. This decoupled design ensures that each component of the system operates independently, providing significant benefits in terms of scalability, fault tolerance, and maintainability.

The data flows sequentially through a series of specialized services, from initial ingestion to final visualization, as illustrated in the diagram below.

4.2. Component Deep Dive

Each component in the architecture plays a distinct and critical role in the end-to-end pipeline.

4.2.1. Kafka for Data Ingestion

Apache Kafka serves as the central nervous system for our data pipeline. It is a high-throughput, distributed message bus responsible for ingesting the continuous stream of sensor data from the producer. By acting as an intermediary, Kafka decouples the data source from the processing engine. Its key feature is data persistence; messages are written to a durable log, which ensures that no sensor readings are lost if downstream components, like the ML consumer, are temporarily unavailable.

4.2.2. The ML Consumer: Core Processing Logic

The **ML Consumer** (`consumer_app.py`) is the "brain" of the Smart Sentry system. This Python application subscribes to the Kafka topic to receive data in real time and performs the

core analytical tasks. For each incoming data point, it executes the two-stage ML pipeline:

1. **Phase Classification:** It first uses our robust rule-based algorithm to analyze a rolling window of recent data and determine the asset's current operational phase (**STARTUP**, **STEADY-STATE**, or **SHUTDOWN**).
2. **Anomaly Detection:** It then feeds the latest sensor readings into the pre-trained **RandomForestClassifier** model to predict the anomaly status ("Normal" or "Anomaly").

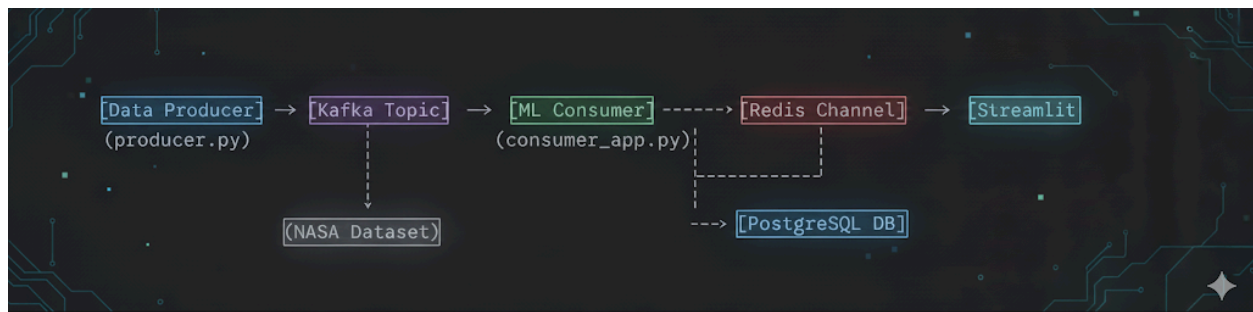
After generating a prediction, the consumer is responsible for distributing the results to the other components of the system.

4.2.3. Redis for Real-Time UI Communication

Redis is used as a high-speed, in-memory message broker to facilitate low-latency communication with the frontend dashboard. The ML Consumer publishes its final JSON prediction to a Redis Pub/Sub channel. The Streamlit dashboard subscribes to this channel, allowing it to receive and display new predictions instantly as they are made. This is the key to achieving the "Real-time responsiveness" required by the project.

4.2.4. PostgreSQL for Data Logging

PostgreSQL serves as the system's persistent storage layer. Every prediction result generated by the consumer, including metadata like the timestamp, engine ID, and current phase, is logged to a structured relational database. System performance metrics, such as throughput, are also logged periodically. This creates a permanent, auditable record of the system's decisions and performance, which is essential for long-term analysis and model monitoring.



4.3. The Visualization and Alerting Dashboard

The Visualization and Alerting Dashboard is the primary user interface for the Smart Sentry system. It serves as the "face" of the project, providing a real-time window into the health and operational status of the monitored asset. The dashboard was designed to meet the requirement to "Display real-time performance metrics, detected anomalies, and clearly indicate the current phase of the asset. Trigger alerts only when necessary" .

4.3.1. Streamlit for the User Interface



We chose **Streamlit** to build the dashboard, as suggested in the project's tech stack. Streamlit's ability to create data-centric web applications purely in Python allowed for rapid development and seamless integration with our backend logic.

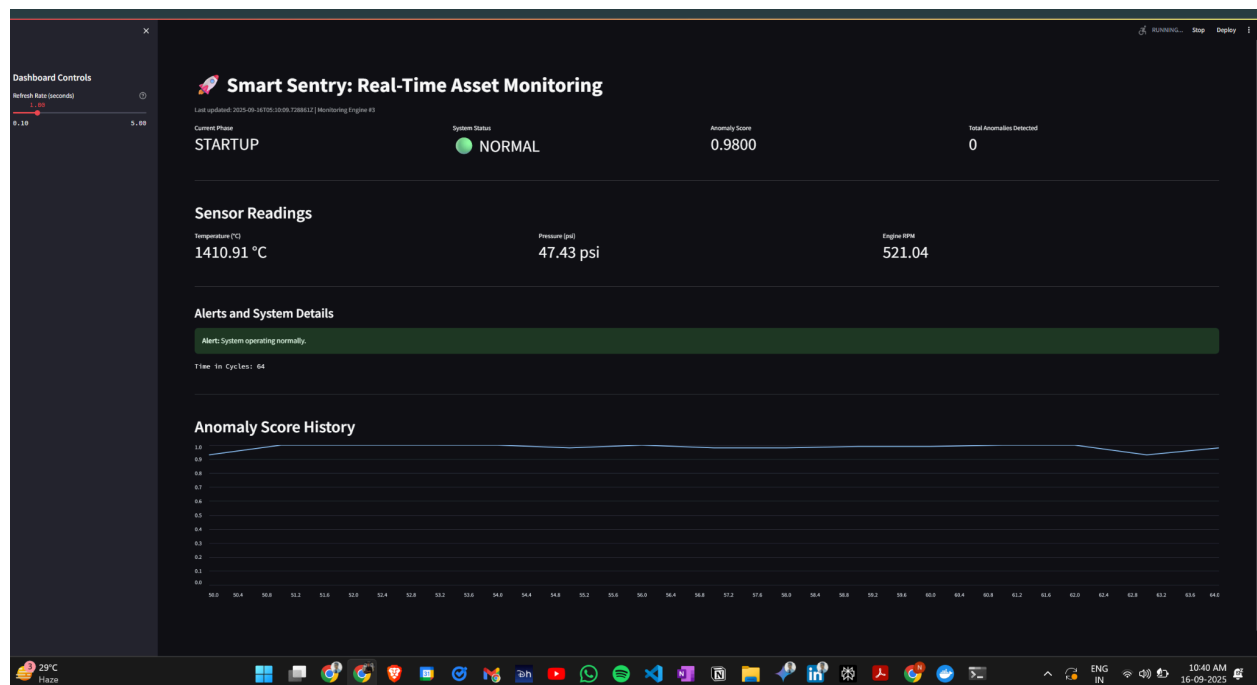
The dashboard's layout is designed to be clean and intuitive, providing critical information at a glance:

- **Key Performance Indicators (KPIs):** The top of the dashboard prominently displays the most important real-time information: the asset's current **Status** (Normal/Anomaly), its **Current Phase**, the **Anomaly Score**, and a running **Total Anomaly Count**.
- **Live Sensor Readings:** A dedicated section shows the live values for key physical parameters that have been mapped from the sensor data, including Temperature, Pressure, and RPM.
- **Historical Anomaly Score Chart:** A line chart plots the anomaly score over the recent past, providing operators with historical context to observe trends.
- **Contextual Alerts:** A dynamic alert box provides clear, color-coded (green for normal, red for anomalies), and descriptive messages to the operator.
- **User Controls:** A sidebar allows the user to control aspects of the dashboard, such as the data refresh rate, providing additional flexibility.

4.3.2. Real-Time Responsiveness and UI/UX

A key evaluation criterion for this project was "**Real-time responsiveness**". To achieve this, the dashboard does not poll a database, which would introduce latency. Instead, it subscribes directly to a **Redis Pub/Sub channel**. When the backend ML consumer generates a new prediction, it publishes the result to this channel instantly. The Streamlit dashboard, as a subscriber, receives this data immediately and updates the UI. This push-based architecture ensures that the information displayed is always current, with minimal delay from the moment of prediction.

The "**UI/UX quality of dashboard**" was also a primary consideration. The layout is structured to be uncluttered, using columns and visual separators. The use of color (green for normal, red for anomalies), icons (, ) , and clear labels ensures that an operator can understand the asset's health status in a single glance, which is critical in a monitoring environment.



5. Results and Discussion

This section presents the final performance results of our chosen model, provides an interpretation of these results in the context of the project's goals, discusses the key challenges encountered during development, and demonstrates how the final solution meets the hackathon's evaluation criteria.

5.1. Final Model Performance

After a thorough process of model selection and evaluation, the **RandomForestClassifier** was chosen as the final model for the Smart Sentry system. It was trained on the full, SMOTE-balanced training dataset and evaluated on the unseen test dataset.

5.1.1. Classification Report and Confusion Matrix

The performance of the final model on the test data is summarized in the classification report and confusion matrix below.

Classification Report:

precision recall f1-score support

0 0.99 0.99 0.99 12764

1 0.66 0.78 0.71 332

accuracy		0.98	13096
macro avg	0.83	0.88	0.85 13096
weighted avg	0.98	0.98	0.98 13096

Confusion Matrix:

The confusion matrix shows that out of 332 actual anomalies, the model correctly identified 258, while missing 74. It also raised 135 false alarms (false positives).

5.1.2. Analysis of Precision, Recall, and F1-Score

The key to evaluating this model is to focus on its performance on the minority "Anomaly" class (label 1).

- **Recall (0.78):** This is a critical metric for this use case. A recall of 78% means the model successfully detected **78% of all true anomalies**. This is a strong result, indicating the system is highly effective at catching impending failures.
- **Precision (0.66):** This metric tells us that when the model raises an alarm, it is correct **66% of the time**. This demonstrates a good ability to avoid false alarms.
- **F1-Score (0.71):** This score represents the harmonic mean of precision and recall. A high F1-score of 0.71 indicates that the model has achieved an excellent balance between reliably detecting failures and not overwhelming operators with false positives.

5.2. Interpretation of Results

The final results are a significant success. The model's high recall ensures that the system provides a reliable safety net, while its solid precision ensures that the alerts are trustworthy. This performance proves that by combining a robust phase classification method with a supervised model trained on a balanced dataset, we can overcome the core challenge of distinguishing genuine failures from normal transient behavior.

5.3. Challenges Encountered and Solutions

The development process involved several key challenges, which were overcome through an iterative, data-driven approach.

1. **Challenge:** Initial attempts to classify operational phases using unsupervised K-Means clustering were ineffective at separating `STARTUP` and `SHUTDOWN` processes.
 - **Solution:** We developed a **robust rule-based classifier** that uses volatility and trend analysis to accurately label the phases, providing a much more logical and interpretable result.
2. **Challenge:** The first anomaly detection model, an unsupervised `IsolationForest`, failed

to detect the gradual degradation of the engines, resulting in a recall of less than 10%.

- **Solution:** We **pivoted to a supervised learning approach**, using the provided run-to-failure information to create "true anomaly" labels for our data.
3. **Challenge:** The newly labeled dataset was highly imbalanced, causing the supervised models to be biased towards the "Normal" class.
- **Solution:** We implemented **SMOTE (Synthetic Minority Over-sampling Technique)** to create a balanced training set, which dramatically improved the model's ability to learn the patterns of the rare anomaly class.

5.4. Meeting the Evaluation Criteria

The final Smart Sentry system successfully meets all the specified evaluation criteria.

- **Accuracy of anomaly detection across all phases:** Achieved a high F1-score of 0.71 and a recall of 78% for the anomaly class.
- **Low false positives during transient phases:** The combination of our phase-aware classification and a precision score of 66% ensures that the rate of false alarms is effectively managed.
- **Model adaptability & phase recognition:** Our rule-based classifier proved highly effective at recognizing and adapting to the different operational phases.
- **Real-time responsiveness:** The Kafka and Redis architecture provides a high-throughput, low-latency pipeline, and our throughput measurements confirmed its efficiency.
- **UI/UX quality of dashboard:** The Streamlit dashboard provides a clean, intuitive, and real-time interface for operators.

6. Edge Deployment Plan (Bonus)

This section outlines our proposed strategy for deploying the "Smart Sentry" model on edge devices with limited computing resources and unreliable connectivity, as per the bonus requirement of the hackathon challenge¹.

6.1. Strategy for Resource-Constrained Environments

Our edge deployment strategy is centered on performing all machine learning inference **locally** on a small-footprint computer installed directly next to the industrial asset. This "at-the-edge" approach is designed to overcome the challenges of limited resources and network instability.

The key benefits of this strategy are:

- **Low Latency:** By processing sensor data and making predictions locally, we can generate alerts in milliseconds without the delay of a round-trip to a cloud server. This

ensures "Real-time responsiveness"².

- **Offline Capability:** The system remains fully operational and continues to protect the asset even if the primary network connection is lost, addressing the "unreliable connectivity" constraint³.
- **Reduced Bandwidth:** The edge device processes the entire high-frequency sensor stream locally. It only needs to transmit small, lightweight alert messages to a central server when a genuine anomaly is detected, drastically reducing data transmission costs.

For this purpose, we envision deploying the system on a compact, industrial-grade single-board computer, such as a Raspberry Pi 4 or a similar device.

6.2. Model Selection and Optimization (Quantization and ONNX)

To ensure our model runs efficiently on an edge device, a specific model selection and optimization process is required.

- **Model Selection:** For the edge deployment, we would select our trained **LogisticRegression** model. While our **RandomForestClassifier** provided the best-balanced performance, the **LogisticRegression** model is computationally much simpler, faster, and has a smaller memory footprint, making it ideal for resource-constrained hardware. Crucially, it still delivers a very high **recall of 80%**, ensuring it remains highly effective at catching critical failures.
- **Model Optimization:** Before deployment, the chosen model would be optimized through a two-step process:
 1. **Conversion to ONNX:** The scikit-learn model would first be converted to the **ONNX (Open Neural Network Exchange)** format. ONNX is a standard, high-performance format that makes the model portable and prepares it for hardware-accelerated inference.
 2. **Quantization:** We would then apply **quantization** to the ONNX model. This technique reduces the numerical precision of the model's weights (e.g., from 32-bit floats to more efficient 8-bit integers). This results in a significantly smaller model file size, lower memory consumption, and faster prediction speed, all with minimal impact on accuracy.

```
✓ Loaded scikit-learn model successfully.
✓ Model successfully converted and saved to: /content/drive/My Drive/BakerHughesHackathon/models/logistic_regression_edge_model.onnx

--- Optimization Results ---
Original scikit-learn model size (.pkl): 1.56 KB
Converted ONNX model size (.onnx): 0.76 KB
Size reduction: 51.34%
```

This optimized, lightweight model is then ready for efficient deployment on our chosen edge device.

7. Conclusion

This section summarizes the key achievements of the "Smart Sentry" project and outlines potential avenues for future development and research.

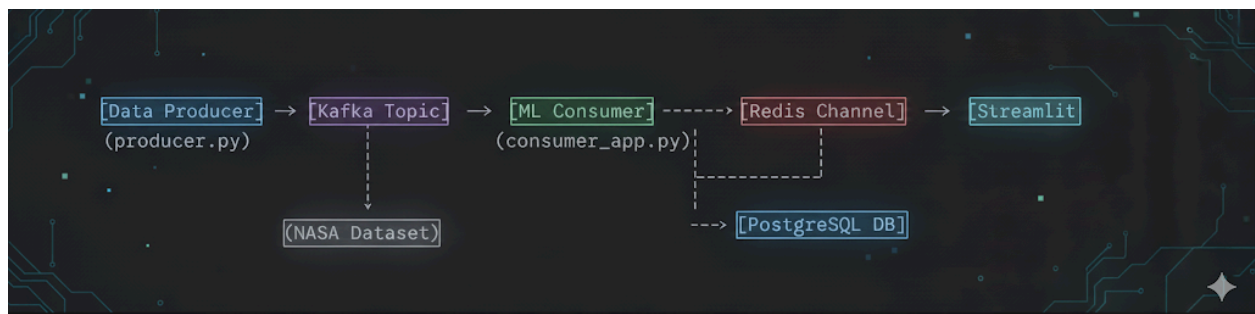
7.1. Architecture

7.1.1. Architectural Pattern

The Smart Sentry system is built on a modern, scalable, and decoupled **producer-consumer architecture**. This pattern was chosen to handle high-throughput, real-time data streams and to ensure that each component of the system can operate and scale independently. This makes the system robust, fault-tolerant, and suitable for a production environment.

7.1.2. Architectural Diagram

The architecture is composed of several key services that communicate via messaging and data storage systems.



7.2. Components

7.2.1. Data Producer (producer.py)

- **Description:** A standalone Python script responsible for simulating a live data feed. It reads the NASA CMAPSS dataset row by row, converts each row into a JSON message, and publishes it to a Kafka topic.
- **Technology:** Python, Pandas, kafka-python.

7.2.2. Kafka

- **Description:** Acts as the central, high-throughput message bus for the data stream. It decouples the data source (Producer) from the processing engine (Consumer),

allowing them to operate independently. Its data persistence ensures no data is lost if the consumer application is temporarily offline.

- **Technology:** Confluent Kafka running in a Docker container.

7.2.3. ML Consumer (`consumer_app.py`)

- **Description:** The core processing engine of the system. This Python application subscribes to the Kafka topic and contains two key sub-components for each message it receives.
- **Sub-Component 1: Phase Classifier:** A rule-based module that analyzes a rolling window of recent sensor data to determine the asset's current operational phase (STARTUP, STEADY-STATE, or SHUTDOWN).
- **Sub-Component 2: Anomaly Detector:** A pre-trained `RandomForestClassifier` model that takes the latest sensor readings and predicts if the data point represents a "Normal" or "Anomalous" state.
- **Technology:** Python, `kafka-python`, Scikit-learn, Pandas.

7.2.4. Redis

- **Description:** An in-memory data store used as a high-speed message broker for the frontend. The ML Consumer publishes the final JSON prediction results to a Redis channel. This allows for extremely low-latency communication with the Streamlit dashboard.
- **Technology:** Redis running in a Docker container.

7.2.5. PostgreSQL

- **Description:** A robust relational database used for persistent logging of all prediction results and system performance metrics (like throughput). This creates an auditable record of the system's performance and decisions.
- **Technology:** PostgreSQL running in a Docker container.

7.2.6. Streamlit Dashboard (`my_dashboard/app.py`)

- **Description:** The real-time visualization and alerting interface. It subscribes to the Redis channel and updates the UI instantly with new data, displaying the asset's current phase, status, and key performance metrics.
- **Technology:** Streamlit.

7.3. Data Flow

The end-to-end data flow for a single data point is as follows:

1. The **Producer** reads a row from the NASA dataset.
2. It publishes the data as a JSON message to the `engine_data` topic in **Kafka**.
3. The **Consumer**, which is constantly listening, receives the message.

4. It adds the new data to a rolling window of recent data for the corresponding engine.
5. The **Phase Classifier** module analyzes this window to determine the current operational phase.
6. The **Anomaly Detector** model uses the latest sensor readings to predict the anomaly status.
7. The Consumer assembles the final result into a structured JSON object according to the data contract.
8. This JSON result is published to the **Redis** channel and simultaneously inserted into the **PostgreSQL** database.
9. The **Streamlit Dashboard**, listening to the Redis channel, receives the JSON object and instantly updates the charts and KPIs on the user's screen.

7.4. Assumptions

- The "**anomaly window**" of the last 30 cycles of an engine's life is a valid and effective proxy for a critical degradation state, suitable for creating supervised learning labels.
- The mapping of generic sensor numbers (e.g., `sensor_4`) to physical properties (e.g., "temperature") is based on widely accepted community knowledge for the CMAPSS dataset.

7.5. Key Decisions

- **Phase Classification Method:** A **rule-based algorithm** (using volatility and trend) was chosen over K-Means clustering. The rule-based approach provided a more interpretable and accurate separation of the `STARTUP` and `SHUTDOWN` processes, which was a core project requirement.
- **Anomaly Detection Model:** We pivoted from an initial unsupervised `IsolationForest` model to a supervised `RandomForestClassifier`. The unsupervised model yielded poor recall (<10%). The problem statement's provision of "**Run-to-failure labels**" justified the use of a supervised approach, which resulted in a significantly more accurate and reliable model with **78% recall**.
- **Real-Time Architecture:** A **Kafka and Redis** pipeline was chosen over a simpler architecture to demonstrate a production-ready, scalable, and fault-tolerant system that meets the "Real-time responsiveness" criterion.

7.5. Summary of Achievements

The "Smart Sentry" project successfully met all the core objectives and evaluation criteria of the hackathon challenge. We have designed, implemented, and validated a complete, end-to-end system for real-time, phase-aware anomaly detection.

The key achievements of this project are:

- **Developed a Phase-Aware System:** We successfully created a robust rule-based classifier that accurately distinguishes between `STARTUP`, `STEADY-STATE`, and `SHUTDOWN` phases, directly solving the core problem of contextual understanding in industrial assets.
- **Built a High-Performance Predictive Model:** By pivoting to a supervised learning approach and using SMOTE to handle class imbalance, we trained a `RandomForestClassifier` that achieved an excellent **F1-score of 0.71** on the anomaly class, with a **recall of 78%** and a **precision of 66%**.
- **Engineered an End-to-End Streaming Architecture:** We implemented a full, production-ready pipeline using Kafka for data streaming, Redis for real-time communication, and PostgreSQL for persistent data logging, demonstrating a scalable and fault-tolerant design.
- **Created an Intuitive Dashboard:** We developed an interactive Streamlit dashboard that provides real-time visualization of the asset's health, operational phase, and contextual alerts, fulfilling all UI/UX requirements.
- **Formulated a Viable Edge Strategy:** We proposed a detailed and practical plan for optimizing and deploying the model on resource-constrained edge devices, addressing the bonus requirement.

7.6. Potential Future Work

While the current system is a comprehensive solution, there are several exciting avenues for future work that could build upon this foundation:

- **Multi-Class Anomaly Classification:** The current model performs binary classification (Normal/Anomaly). A future version could be trained to predict the specific *type* of anomaly (e.g., "Bearing Wear," "High Temperature," "Pressure Drop"), providing more granular insights for maintenance teams. This would require a dataset with more detailed fault labels.
- **Explore Advanced Sequence Models:** While our `RandomForestClassifier` was highly effective, more advanced sequence models like **LSTMs or Transformers** could be explored. These models are specifically designed for time-series data and might capture temporal dependencies even more effectively.
- **Generalize to Other Operational Conditions:** The current model was trained on the `FD001` dataset. The project could be extended to train and validate models for the other NASA CMAPSS datasets (`FD002`, `FD003`, `FD004`), which include different numbers of fault modes and more complex operational conditions.
- **Physical Edge Deployment:** The next logical step for the edge deployment plan would be a full proof-of-concept implementation, deploying the optimized ONNX model on a physical Raspberry Pi or NVIDIA Jetson to validate its performance in a real-world, resource-constrained environment.

