

TECHNICAL, Functional and Graphical design

Classifying Arrhythmia in Fighter Pilots

APRIL 30, 2025

Group 3

Picture 147, Picture, Picture

Table of contents

[1. Graphical Design 2](#_Toc1651516395)

[2. Technical Design 4](#_Toc1083952533)

[2.1 Scope and Complexity 5](#_Toc1169070971)

[2.1.1 Problem Scope 5](#_Toc1808567496)

[2.1.2 Complexity Factors 5](#_Toc1873098751)

[2.2 Architecture Overview 5](#_Toc991391715)

[2.2.1 Data Layer 5](#_Toc989488945)

[2.2.2 Modeling Layer 5](#_Toc424110651)

[2.2.3 Deployment and interface Layer 6](#_Toc802094144)

[2.3 Tool Stack Justification 6](#_Toc1394795136)

[2.4 Scalability And Usability Considerations 6](#_Toc344227098)

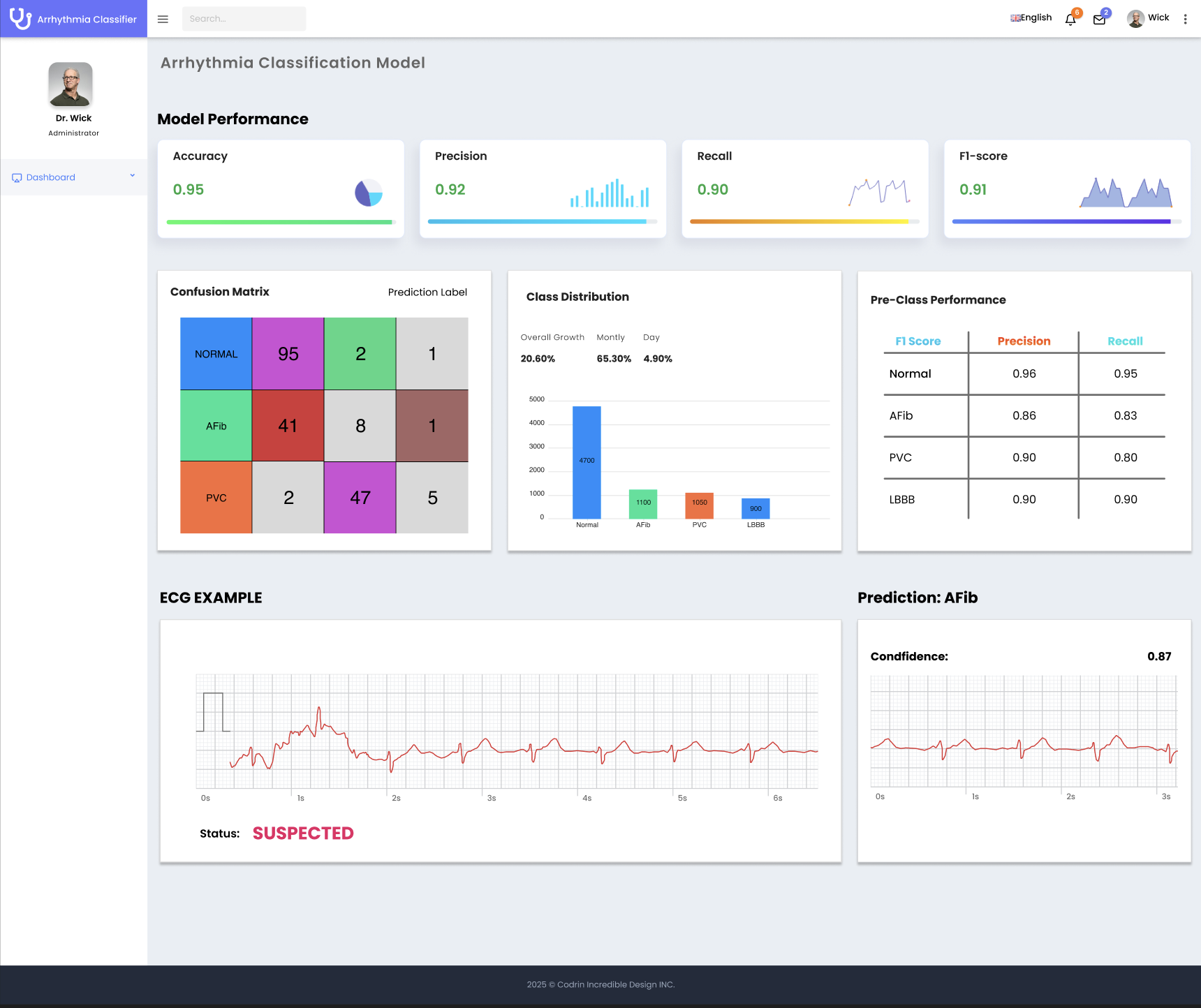
[2.5 Conclusion 7](#_Toc851327584)

[3. Functional design 7](#_Toc1241584284)

[3.1. Functional flow of the application 8](#_Toc320403286)

[3.2. System Architecture and Class Structure 10](#_Toc1782749879)

# 1. Graphical Design



(Design Concept)

This is a concept dashboard for an arrhythmia classification model. It shows the main ideas and features you would include when presenting how the model works and how well it performs.

The dashboard includes:

* **Model performance metrics** like accuracy, precision, recall, and F1-score to show how reliable the predictions are.
* A **confusion matrix** to compare the actual vs. predicted classes and spot where the model makes mistakes.
* A **class distribution chart** to see how many ECG samples belong to each type of heart rhythm (like Normal, Arrhythmia).
* A **per-class performance table** to understand how well the model performs for each type of arrhythmia.
* An example of an **ECG signal** with a highlighted part showing what the model focused on and the predicted arrhythmia type with a confidence score.

# 2. Technical Design

This proof-of-concept (PoC) aims to develop and demonstrate an arrythmia classification system that detects abnormal heartbeats such as premature or delayed ventricular contraction from ECG data. The system is designed for F-35 fighter pilots who may experience unexplained physiological events (UPEs) during high-stress, high-G maneuvers.

Due to the sensitive and domain-specific nature of the applications, the solution must not only be accurate but also generalizable across varied conditions and sensor types. The PoC will showcase the technical feasibility of ESG signal analysis using machine learning.

## 2.1 Scope and Complexity

### 2.1.1 Problem Scope

* Classify ESG signals into normal and arrhythmic categories using labeled physiological data.

### 2.1.2 Complexity Factors

* Significant class imbalance: Only 1.5% of samples are arrhythmic.
* Domain shift: Training data from hospital environments must generalize to in-flight conditions.
* Need for interpretability and model transparency.

## 2.2 Architecture Overview

### 2.2.1 Data Layer

* Input Source: Cleaned dataset derived from the MIT-BIH Arrythmia Database
* Preprocessing:
  + Bandpass filtering (0.5-40 Hz) to remove noise.
  + Z-score normalization.
  + Windowing around R-peaks.
  + Incomplete or out-of-bound segments skipped automatically
* Format: .csv file containing 216 float features per sample + 1 label.

### 2.2.2 Modeling Layer

* Training Models:
  + Primary: CNN + LSTM hybrid
  + Baseline: Random Forest – used to highlight overfitting risks.
* Evaluation Metrics:
  + Precision, Recall, F1-Score
  + Confusion Matrix
* Model Development Toolkits:
  + Scikit-learn
  + Numpy
  + Pandas
  + Matplotlib

### 2.2.3 Deployment and interface Layer

|  |  |  |
| --- | --- | --- |
| Component | Tool | Justification |
| Web Interface | Streamlit | Quick to build, fully Python-based, interactive |
| Model Serving | In-process within Streamlit | Lightweight for demo use |
| Visualization | Matplotlib, plotly, Streamlit widgets | Display ECG waveforms, model outputs, metrics |
| Interactivity | Tools from Streamlit | Allow testing with new ECG data, model comparisons |
| Deployment | Steamlit Cloud or Docker | Easy to share |

## 2.3 Tool Stack Justification

|  |  |  |
| --- | --- | --- |
| Layer | Tools | Justification |
| Preprocessing | Numpy, scipy, wfdb, pandas | Standard EG signal processing and statistical cleaning |
| Modeling | Scikit-learn | Compatible with imbalanced data handling |
| Visualization | Matplotlib, seaborn, plotly | For waveform display and metric analysis |
| Deployment | Streamlit, Docker | Rapid UI + scalable, reproducible container setup |

## 2.4 Scalability And Usability Considerations

* Scalability: While Streamlit is not optimized for concurrent production use, it’s well-suited for PoC. Future deployment could port to FastAPI with Kubernets for scalable microservices.
* Usability: Streamlit enables direct interaction with the model, making it ideal for:
  + Model debugging
  + Performance interpretation

## 2.5 Conclusion

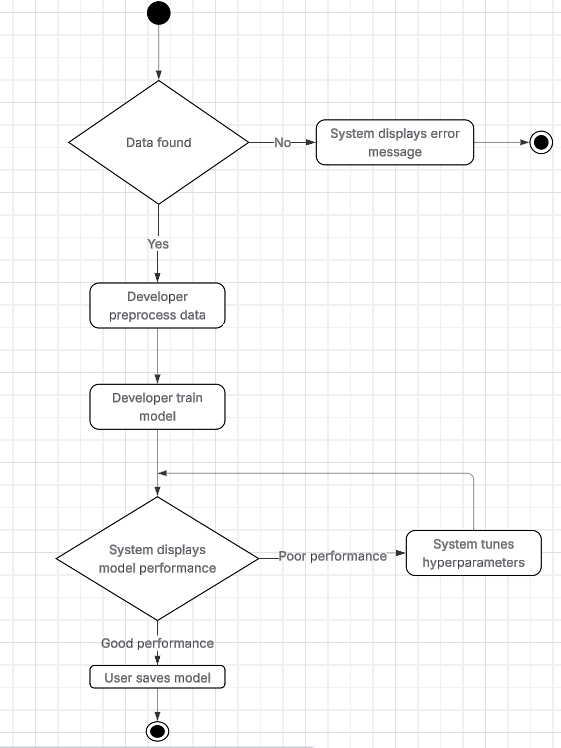
This technical design supports the creation of a robust, scalable, and interpretable PoC for arrythmia detection in high-G environments. The chosen tool stack ensures alignment with the problem’s scope and complexity while facilitating a smooth path to demonstration.

# 3. Functional design

This chapter outlines the functional design of the proof-of-concept application developed for arrhythmia detection in F-35 pilots. The proposed solution was conceptualized and structured using a range of UML diagrams to clarify both the user interaction and the internal system behavior. The aim is to ensure that the application design is transparent, logically structured, and functionally sufficient for the complexity of the problem at hand.

## 3.1. Functional flow of the application

The application logic begins with a check for available data. If no data is found, the system issues an error message and stops. If data is available, the process moves forward through several stages handled by both the developer and system components.



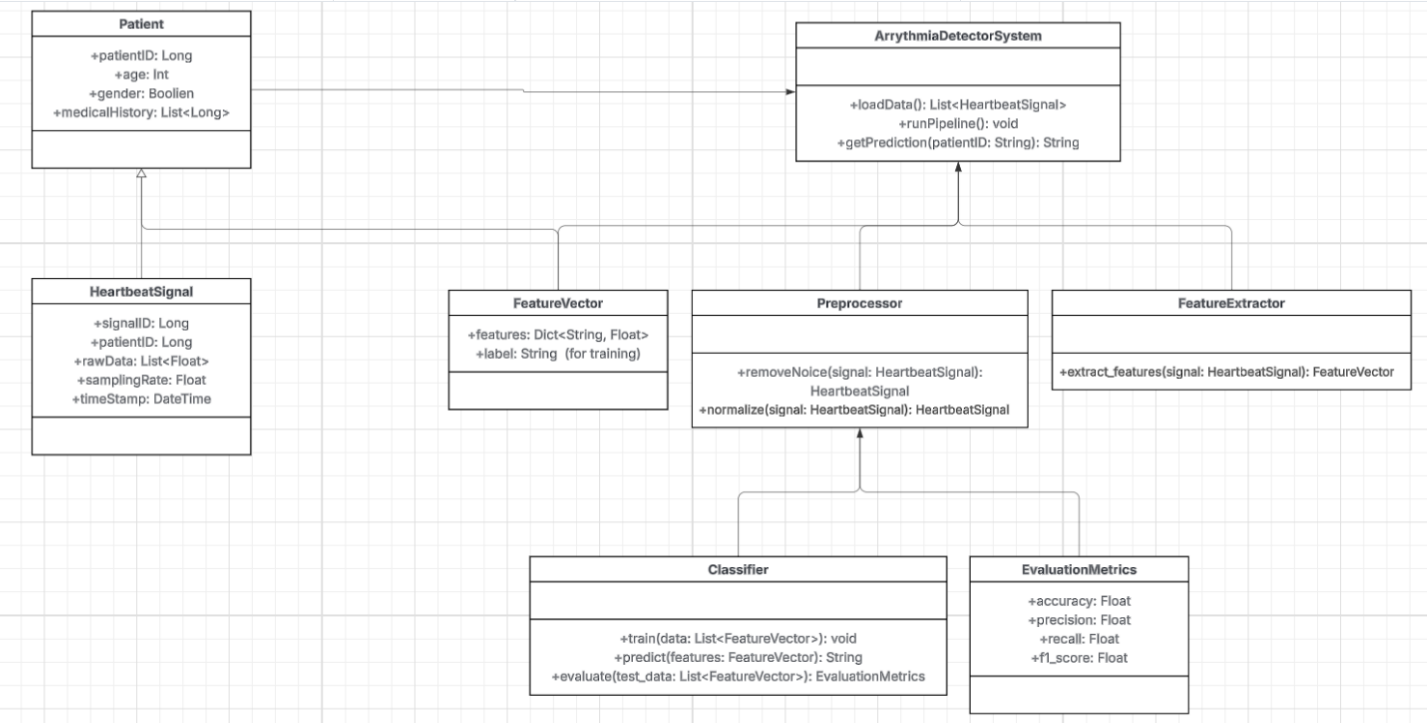
The process flow has been deliberately designed to ensure reliability, control, and iterative feedback, which are crucial in high-stakes environments like fighter aviation. The following design choices reflect these priorities:

* **Initial data check**  
  The process starts with a check to see if the data is found. This ensures that all downstream operations only run if valid ECG data exists. In high-reliability systems, failing early is critical, hence the system immediately issues an error and stops if no data is found.
* **Automated data preprocessing**  
  Once data is confirmed, the system automatically proceeds through predefined preprocessing steps. These include noise filtering and signal normalization, which are essential for handling raw ECG signals, particularly in high-G environments where motion artifacts and signal distortion are common. Automating this ensures consistency and reproducibility while reducing the risk of human error.
* **Training is distinct from tuning**  
   The training stage is kept separate from performance evaluation and tuning. This allows for isolation of model-building from model-assessment logic and aligns with common machine learning pipelines. It reflects a research-first mindset, where training and iterative evaluation are not automated until final validation.
* **Iterative performance check**  
  After training, the system evaluates model performance based on metrics like accuracy, recall, and F1-score. If the model underperforms, the system initiates an automated hyperparameter tuning cycle. This feedback loop continues until a satisfactory result is achieved. This design supports progressive improvement while preserving traceability of each training iteration.
* **User confirmation to save**  
   A final step lets the user (developer) explicitly choose to save the model if performance isgood. This manual checkpoint ensures traceability and verification, rather than allowing models to be saved automatically.

This flow strikes a balance between automated processing for efficiency and manual validation for safety, aligning with best practices in medical and defense-grade software systems.

## 3.2. System Architecture and Class Structure

The application includes multiple components for handling different functional aspects of the arrhythmia prediction system. The class diagram below presents the relationship between system entities such as patient data, ECG signals, preprocessing, and classification logic.



The system’s architecture reflects a pipeline-based, layered modular design, with components corresponding to real-world data and processing stages:

* Separation of Domain Data and Logic
  + *Patient* and HeartbeatSignal are modeled as independent data classes. This supports modular data loading, easy dataset expansion, and patient-level tracking.
  + *HeartbeatSignal* is decoupled from preprocessing, allowing raw data to be preserved for reproducibility and auditability which is important for clinical evaluation.
* Preprocessing as a Dedicated Class
  + *Preprocessor* handles both noise removal and normalization. Abstracting these tasks allows for swappable preprocessing strategies and testability. Keeping this logic separate avoids polluting the main pipeline (*ArrhythmiaDetectorSystem*) with low-level signal operations.
* Explicit Feature Engineering
  + The *FeatureExtractor* is separated from the classifier. This decision allows experimentation with different signal representation strategies (e.g., time-domain vs. frequency-domain features), supporting research flexibility.
* Classifier Encapsulation
  + *Classifier* houses all ML model logic (training, prediction, evaluation). This encapsulation isolates learning logic from other system concerns. It aligns with clean architecture practices and allows for easier testing and replacement of the model (e.g., swapping a CNN for a Random Forest).
* Evaluation as a Standalone Class
  + The *EvaluationMetrics* class captures output performance in a structured, reusable format. This abstraction allows consistent access to model accuracy, precision, recall, and F1-score, supporting both automated testing and manual inspection.
* Pipeline Controller (*ArrhythmiaDetectorSystem*)
  + This central controller manages the end-to-end flow: loading data, running the pipeline, and generating predictions. It orchestrates the other components without holding implementation logic, following the single responsibility principle.

This architectural structure ensures testability, replaceability, and readability, making the system suitable for iterative development and adaptation to new domains or datasets.