Technical Design Document — Predictive Maintenance

Carlos Torres Sánchez
2025-09-04

Contents

1 1. Introduction

The objective of this project is to design, train, and deploy a predictive maintenance (PdM) solution able to forecast machine failures within a 14-day horizon.

The business motivation is to reduce unplanned downtime and associated costs by identifying machines at risk of failure before the event occurs. By prioritizing recall, the model minimizes the chance of missing true failures.

This document describes the full system: **architecture, data flow, components, implementation, and design decisions**, aligning with the deliverables defined in the Risk Specialist Technical Challenge.

2 2. Data

The project uses the public **Microsoft Azure Predictive Maintenance dataset** from Kaggle:

- **Machines**: static attributes (`machineID`, model, age).
- **Telemetry**: hourly sensor readings (volt, rotate, pressure, vibration).
- **Failures**: component-level failures with timestamps and types.
- **Errors**: logged error codes (error1–error5).
- **Maintenance**: scheduled component replacements.

2.1 Integration and Assembly

All tables are merged into a granular frame called `df_brute`:

- Telemetry acts as the temporal base.
- Failures contribute binary flags and labels.
- Errors and maintenance events are pivoted to one-hot columns and aggregated counts.
- Machines add static metadata.

2.2 Data Quality and Sanity Checks

- Required columns validated (`machineID`, `datetime`, sensor fields).
- Duplicate pairs avoided.
- Sensor ranges enforced: volt (90–270), rotate (100–800), pressure (40–200), vibration (0–100).
- Target `fail_next_14d` created using a forward 14-day rolling window.

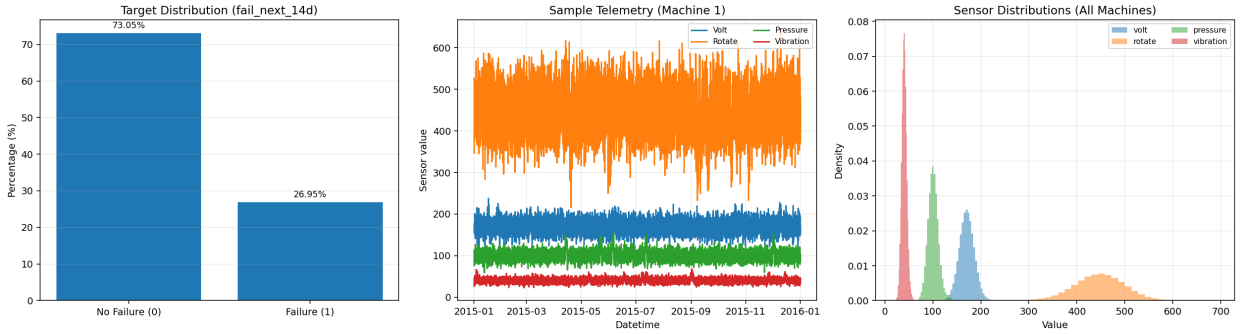

Figure 1: Data overview. Left: class balance for `fail_next_14d`. Middle: example telemetry signals (volt, rotate, pressure, vibration) for one machine. Right: sensor value distributions across the fleet.

3 3. Data Flow and Architecture

3.1 End-to-End Pipeline

The pipeline automates the ML lifecycle:

1. Dataset download from Kaggle via `kagglehub`.
2. Assembly of unified `df_brute`.
3. Feature engineering with 14-day rolling windows.
4. Feature selection: VIF stepwise + low-variance pruning.
5. (Optional) Winsorization of class 0 to stabilize outliers.
6. Train-test stratified split.
7. Logistic Regression training and evaluation.
8. KS-based threshold selection.
9. Export of metrics, plots, and artifacts.
10. Deployment through a FastAPI service.

3.2 Architecture Diagram

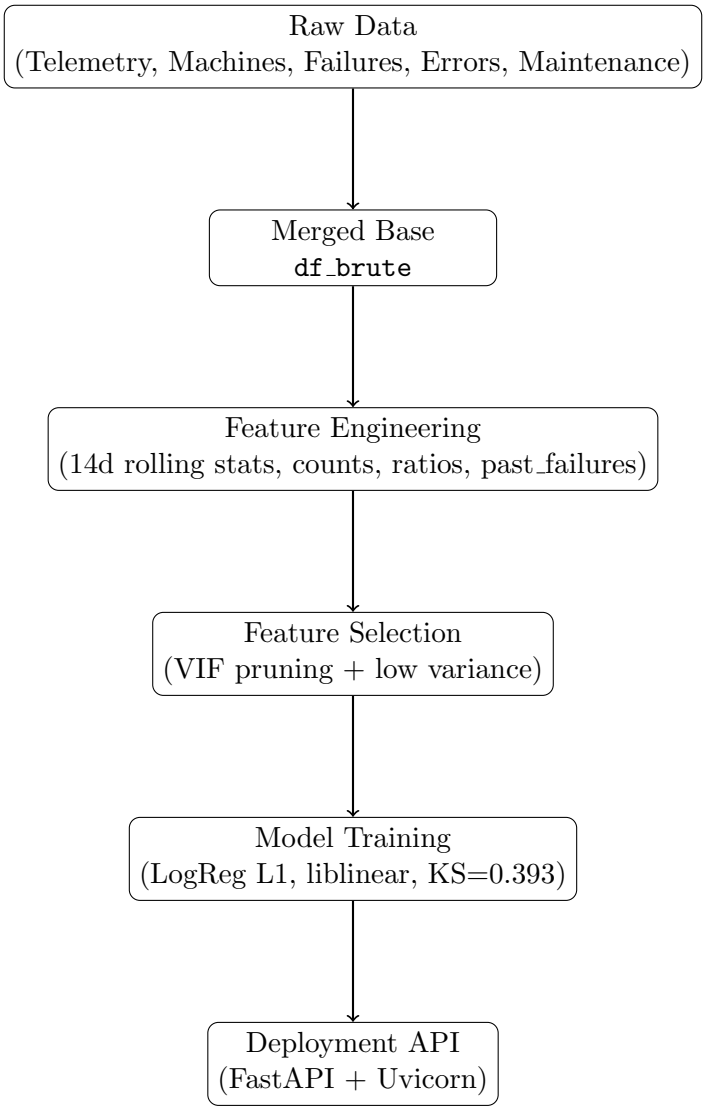

Figure 2: End-to-end pipeline: from raw data ingestion to API deployment.

3.3 MLOps Considerations

- Reproducibility: all steps scripted in `training_pipeline.py`.
- Automation: arguments control randomness, splits, hyperparameters.
- Versioning: artifacts and metrics stored under `artifacts/`.
- Deployment: API containerized via Uvicorn/FastAPI.

4 4. Components

4.1 Feature Engineering

- Rolling statistics: min, max, mean, std, range, rstd for each sensor.
- Aggregated counts: `error_count_14d`, `maint_count_14d`.
- Historical indicators: `age_days`, `past_failures`, `error_per_maint`.

4.2 Feature Selection

- Stepwise VIF elimination (threshold 5.0) to address multicollinearity.
- Low-variance pruning to remove quasi-constant variables.
- Final features: 10–12 numeric aggregates.

4.3 Model Training

- Algorithm: Logistic Regression.
- Parameters: `penalty=l1`, `solver=liblinear`, `max_iter=3000`, `class_weight=balanced`.
- Metrics: ROC-AUC, KS, confusion matrix, precision, recall, F1.

4.4 Threshold Selection

- KS statistic used to find the optimal separation threshold.
- Default: 0.39 (test set).

4.5 Serving Layer

- Framework: FastAPI with Uvicorn.
- Endpoints:
  - `/health` – check service availability.
  - `/metadata` – model version, feature list.
  - `/predict` – scoring endpoint returning probability, label, and message.
- Artifacts consumed: `model.joblib`, `imputer.joblib`, `scaler.joblib`, `selected_features.json`.

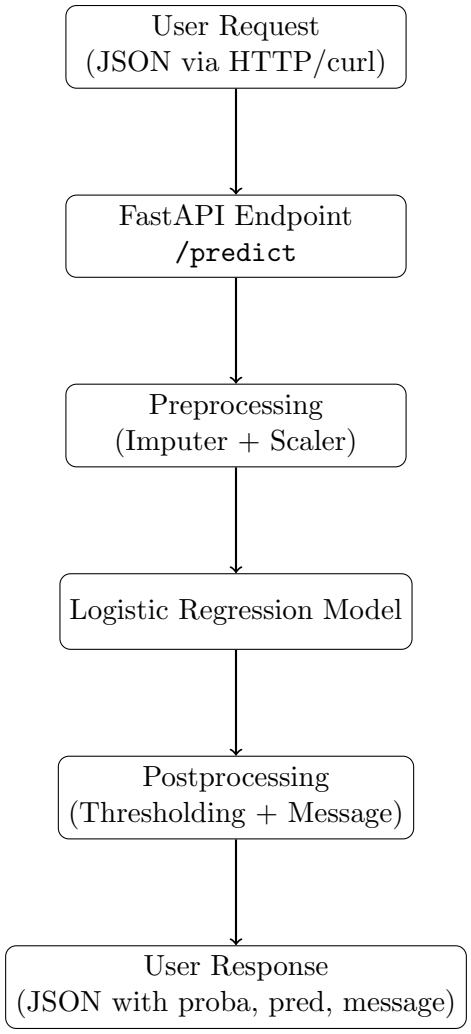

Figure 3: API serving flow: from user request to JSON response.

5 5. Key Design Decisions

- **14-day window:** balances statistical signal and sample size.
- **Model choice:** Logistic Regression over XGBoost/RandomForest for interpretability and calibration.
- **Threshold calibration:** KS ensures business-driven recall focus.
- **Imbalance handling:** class weights instead of resampling to preserve calibration.
- **Optional winsorization:** stabilizes extreme outliers in non-failure class.

6 6. Implementation Details

6.1 Repository Structure

```
toyota_pdm/
  artifacts/   (metrics.json, roc_curve.png, model.joblib, etc.)
  notebooks/   (EDA and feature exploration)
  reports/     (Short tech report, figures, design doc)
  src/         (training_pipeline.py, app.py)
```

6.2 Training Command

```
python src/training_pipeline.py \
  --output_dir artifacts/ \
  --test_size 0.2 \
  --random_state 42 \
  --class_weight balanced \
  --penalty l1 \
  --solver liblinear \
  --max_iter 3000 \
  --tol 1e-5
```

6.3 Serving Command

```
uvicorn src.app:app --host 0.0.0.0 --port 8000
```

7 7. Risks and Mitigations

- **Data leakage:** avoided by strict time-based target creation.
- **Multicollinearity:** handled by VIF pruning.
- **Imbalance:** mitigated by class weights.
- **Drift:** monitor via periodic retraining (monthly).
- **Reproducibility:** random seeds and artifacts ensure consistency.