What if the engine knew it was failing?

An explainable deep learning system that estimates remaining useful life and enables proactive aircraft maintenance

Deep Learning – Final Project
June 2026

Marian · Theo · Nour · Nuria · Ignacio · Alberto

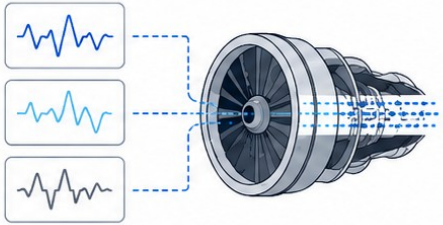
Fixed maintenance schedules leave airlines exposed to avoidable downtime

Aircraft engines already generate rich sensor data, but maintenance decisions often fail to reflect each engine's true degradation state.

1 Current reality

- Maintenance is planned around fixed intervals and inspections
- Engine wear varies by route, load, and operating conditions

2 Signals already exist



- Every flight cycle produces telemetry that reflects engine health
- The challenge is converting sensor signals into a usable maintenance view

3 Better decisions

- Estimate how many flight cycles remain before intervention is needed
- Schedule maintenance before risk turns into an aircraft-on-ground event

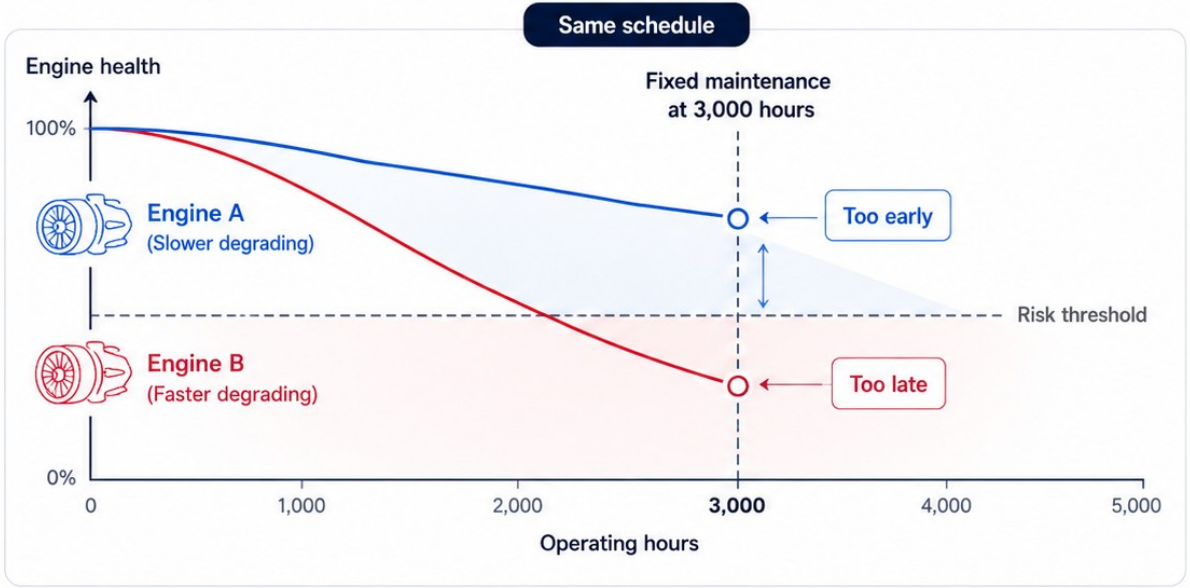
Unplanned failure
$500K+
Unplanned failure

Planned maintenance
$35K
Planned maintenance

Value at stake
$465K
Value at stake

Fixed intervals force airlines to service engines either too early or too late

A single 3,000-hour rule ignores the fact that engines wear differently across routes, loads, and operating conditions.



Unnecessary maintenance

Healthy engines are opened before needed, consuming labor, parts, and remaining useful life.

Failure exposure

At risk engines stay in service until the next fixed interval, increasing the chance of an aircraft on ground event.

The problem isn’t lack of data, but relying on one schedule for engines that wear at different speeds

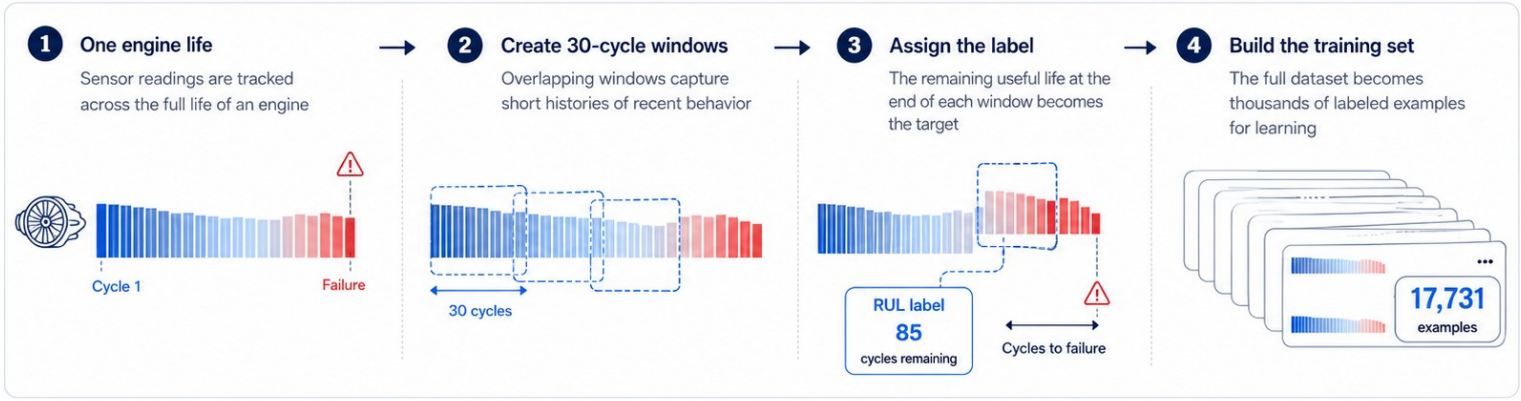
Run to failure hist histories give us the signal needed to predict remaining life

Our model follows 100 engines from first cycle to breakdown, turning each engine life into thousands of labeled examples.

100
Engines observed
Each engine is recorded from first cycle to failure

17,731
Training windows
A 30-cycle sliding window converts each engine life into many examples

17
Inputs per cycle
Model-ready features after filtering low-value sensors



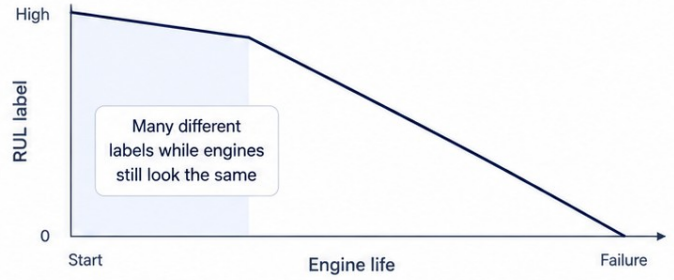
Because every engine ends in failure, the data captures both normal wear patterns and the warning signs that appear before breakdown.

Capping RUL at 125 cycles focuses learning on meaningful degradation

Early sensor histories look similar, so treating very high RUL values as equally healthy gives the model a cleaner target to learn from.

Without the cap

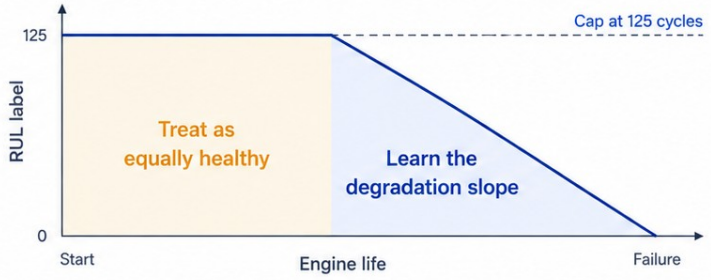
Raw target



- Early healthy states get different labels
- The model spends capacity on distinctions with little sensor signal

Training target we use

Piecewise capped target



- Above 125 cycles, the label stays flat
- Below 125 cycles, the model learns the part of life where degradation becomes visible

Example transformation

190 → 125	160 → 125	120 → 120	85 → 85

Why it helps

Less noise in the healthy zone

What the model learns

When degradation starts and how fast remaining life falls

The cap removes artificial precision from early life and concentrates learning on the signal that matters.

RUL prediction works best when the model remembers how the engine got there

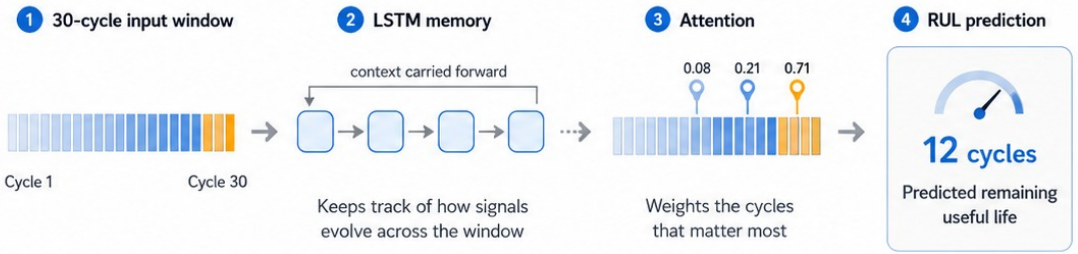
The model must read the full 30-cycle history, retain context across steps, and focus on the moments where degradation is visible.

Why a plain feed-forward model falls short

- Sees one cycle in isolation
- Cannot tell whether signals are stable, rising, or falling
- Misses the pattern that builds before failure

Why LSTM with attention fits the task

The model first remembers the sequence, then concentrates on the most informative moments.



Sequence awareness

Reads the full 30-cycle history instead of treating each cycle independently.

Memory

Retains how key sensor patterns changed from earlier cycles to later ones.

Focus

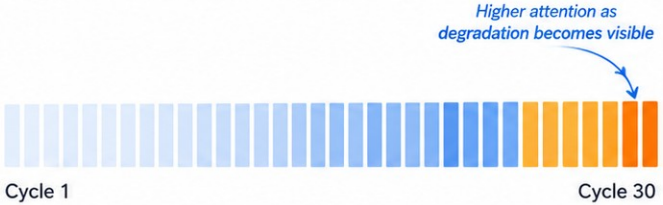
Attention emphasizes degradation cues instead of giving every cycle equal importance.

For engine RUL, failure risk as a trajectory, not a snapshot.

Attention makes each prediction easier to explain and trust

The model shows both where in the sequence it focused and which sensor signals contributed most.

Where the model looked

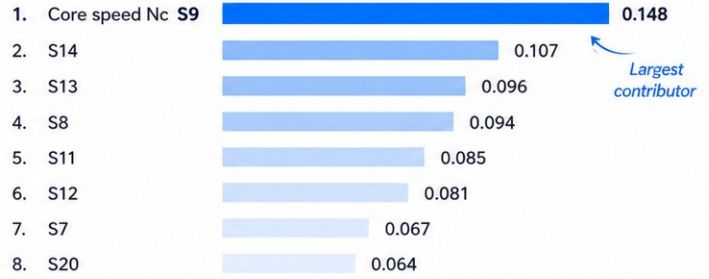


- Recent cycles carry more weight than early healthy history
- The model emphasizes the part of the sequence where wear accelerates
- Attention turns the 30-cycle window into an interpretable signal

What drove the estimate

Example prediction 12 cycles remaining

Top sensor drivers



Operational settings contribute little in this example

Time focus

We can see which part of the engine history the model relied on.

Signal drivers

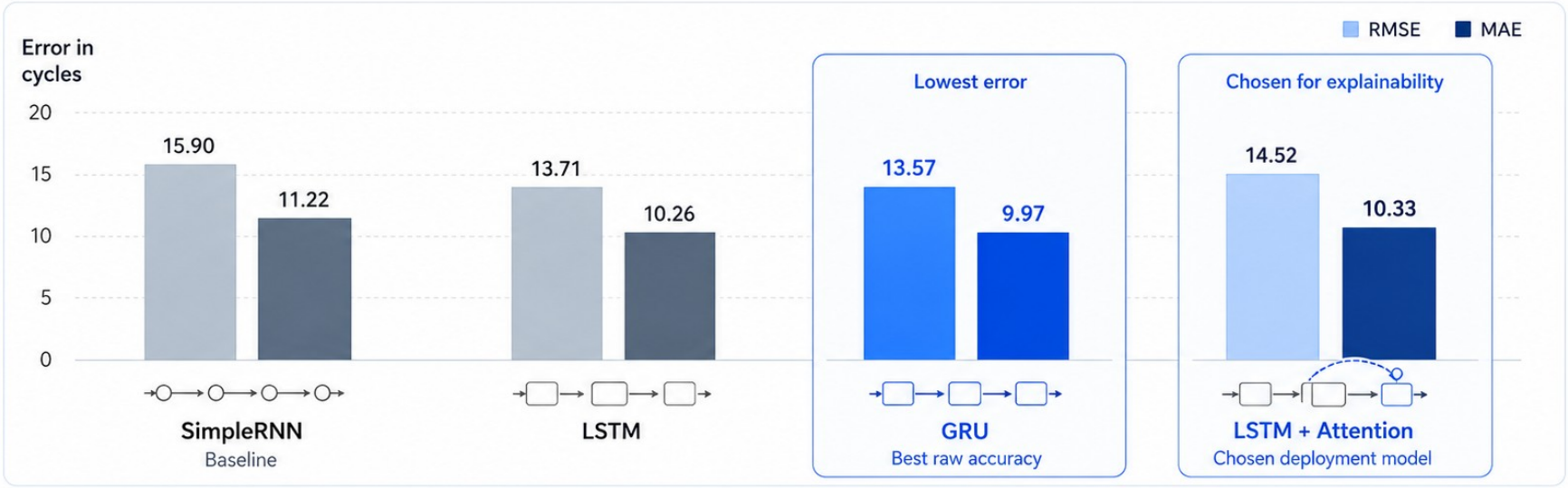
We can rank the sensors that most influenced a given prediction.

Decision confidence

This makes the alert easier for maintenance teams to review and trust.

Explainability turns a model output into a maintenance signal engineers can act on.

GRU achieved the lowest error while LSTM with attention offered the best balance



LSTM with attention was selected because it combines performance with an explainable prediction path.

Each modeling choice maps directly to a clear component in the repo

01 Optimization

Stabilize learning

`Adam` `ReduceLROnPlateau` `Huber loss`

Controls convergence, step size, and training loss.

Where it lives

`notebooks/03_model_training.ipynb`

02 Regularization

Control overfitting

`Dropout(0.2)` `BatchNorm` `EarlyStopping`

Keeps the model robust and focused on true signal.

Where it lives

`notebooks/03_model_training.ipynb`

03 Sequence models

Benchmark architectures

`SimpleRNN` `LSTM` `GRU`

Tests recurrent architectures on the same data and objective.

Where it lives

`notebooks/03_model_training.ipynb`

04 Attention

Explain the prediction

`DotProductAttention` `GlobalAveragePooling1D`

Turns the chosen model into an interpretable deployed prediction path.

Where it lives

`backend/predict.py`
`backend/model/lstm_model.keras`

The architecture is auditable because every capability is implemented in a named repo component.

Live demo

Deep Learning – Final Project
June 2026

Marian · Theo · Nour · Nuria · Ignacio · Alberto

Training choices reduce overfitting and penalize late mistakes more heavily

We monitored generalization during training and used a cost-aware error treatment.

Dropout 0.2
Reduces co-adaptation

Batch normalization
Stabilizes the dense head

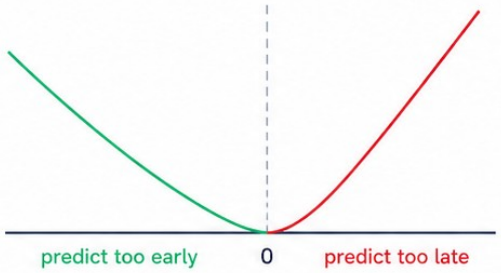
Early stopping
Keeps the best weights

Huber loss δ=10
Robust near outliers

We watched the validation curve

- Training and validation loss decline together, indicating stable learning
- We stop when validation improvement stalls instead of over-training

Being late is worse than being early



early exp(-d / 13) - 1

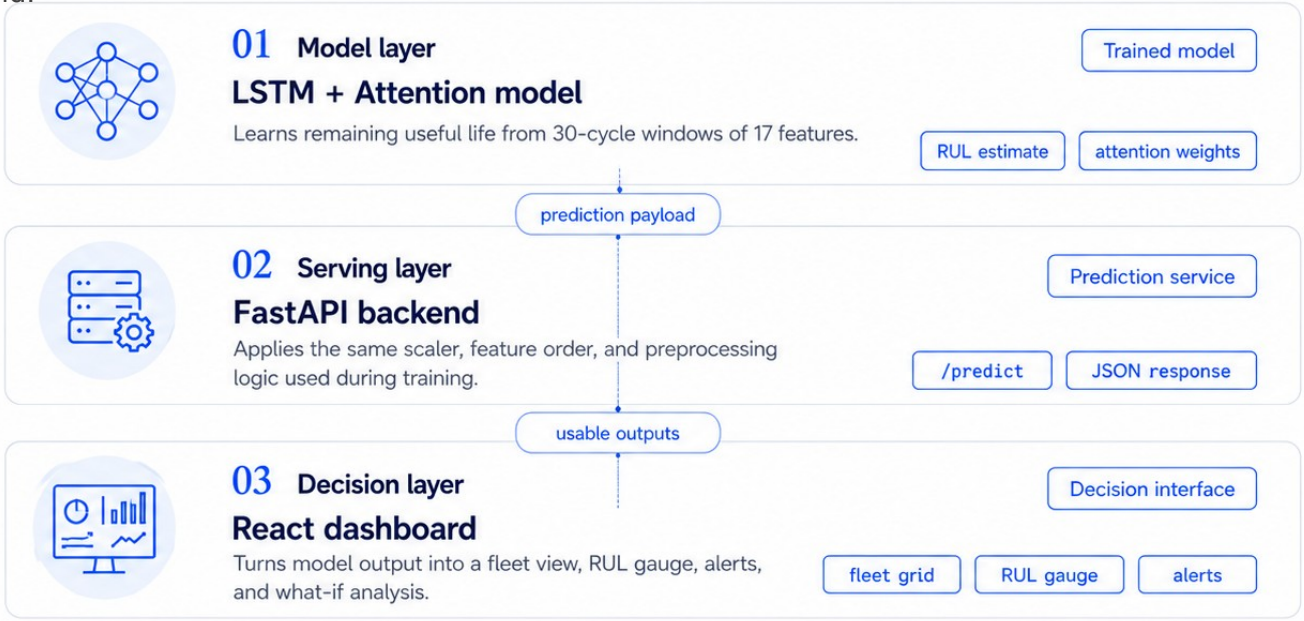
late exp(d / 10) - 1

- Slightly early maintenance creates cost but avoids operational risk
- Late predictions are penalized more because they can leave a degrading engine in service

The training process favors robust generalization and treats late prediction errors as the riskier mistake

Keeping the model, API, and dashboard in sync makes the prediction usable

The same preprocessing and feature logic flow form training to serving to the interface, so the result stays consistent end to end.



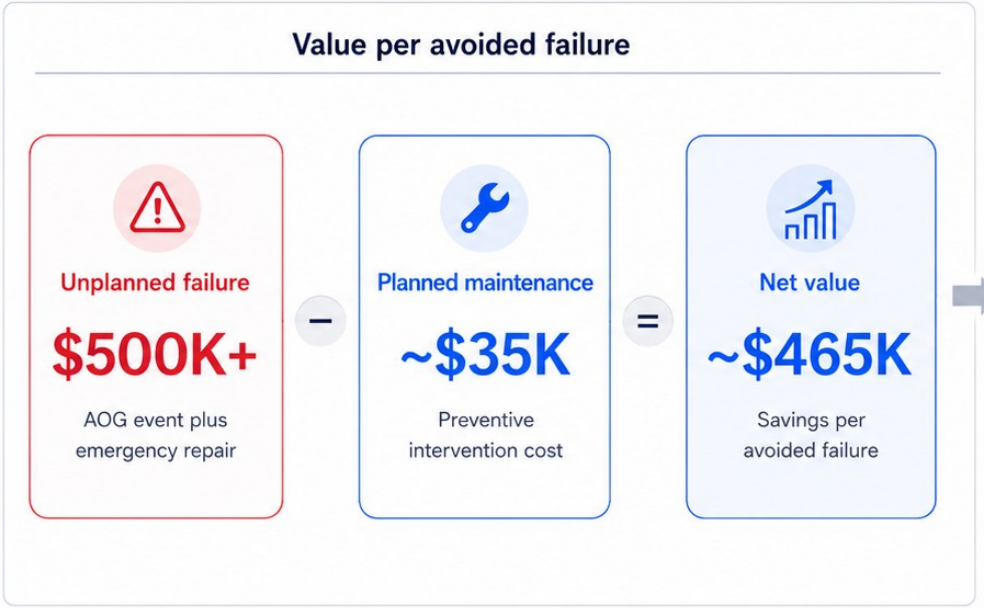
Why the layers must stay aligned

- One preprocessing path avoids mismatched predictions
- API responses carry the same logic the model learned
- The interface turns technical output into action

The architecture matters because one consistent pipeline turns a model output into an actionable goal.

A 50-engine fleet can save about $11.2M a year by avoiding unplanned failures

The economics are attractive because one avoided aircraft on ground event is worth far more than the preventive maintenance



The business case is driven by simple unit economics and turns predictive maintenance into measurable value

Thank you

Q&A

Deep Learning – Final Project
June 2026

Marian · Theo · Nour · Nuria · Ignacio · Alberto