# Explainable Machine Learning for Robot Telemetry: Detecting DoS Attacks and Malfunctions

**Advance Artificial Intelligence AI5001**

**Assignment-3**

**Mohib Qureshi (25K-7617)**

**National University of Computer & Emerging Sciences**

**December 2025**

## Abstract

Real-time supervision of autonomous robots is challenging, particularly when trying to differentiate between standard behaviour, system faults, and intentional disruptions. This project tackles that by analysing rich telemetry collected from a robot and classifying its state into one of three categories: Normal, DoS Attack, or Malfunction. The collected data spans GPS locations, IMU measurements, battery information, control commands, CPU usage, and RSSI values.

We begin by preparing and examining the dataset—dealing with missing sensor values, detecting anomalies, normalising numerical fields, and reviewing class proportions, feature trends, and correlations. Using this cleaned data, we develop three different models: an SVM to provide a solid baseline, an LSTM that leverages short sequences to capture time-dependent patterns, and a VAE that learns a compressed internal representation useful for spotting deviations from expected behaviour.

The models are assessed on a held-out test set using metrics like accuracy and macro F1, and we analyse confusion matrices to see where each model struggles. We then bring in explainable AI techniques to identify which telemetry features contribute most to identifying attacks or hardware/software issues. Our broader objective is not just to classify states accurately, but also to understand how telemetry patterns reveal the robot's operational and security status.

## 1. Introduction

Modern autonomous robots and drones rely on a dense network of sensors, control logic and communication channels. As these systems become more advanced and more connected, it becomes harder to recognise in real time whether their behaviour is normal, affected by a malfunction, or disrupted by an attack. Failing to detect these issues early can cause the robot to be damaged or lost, and may even endanger people nearby.

One effective way to monitor these platforms is through telemetry—continuous streams of sensor readings and system messages produced during flight or operation. The dataset used in this project contains GPS positions, pose data, IMU readings, battery measurements, RC output channels, CPU and memory metrics, RSSI values, and various state indicators like whether the system is armed. Every row corresponds to a single timestamp and is labelled as

- Normal,
- DoS Attack,
- or Malfunction.

The main challenge is to train models that can automatically classify each snapshot—or short sequence—of telemetry into one of these three categories. While the task is essentially a multiclass classification problem involving multivariate time-series data, we are also interested in making the models interpretable so their decisions can be trusted in operational or safety-critical settings.

Our process begins with data cleaning and exploration. The raw CSV contains missing and noisy readings, so we convert empty cells to missing values, apply forward filling when suitable, and standardise numerical features. We also investigate class distributions, sensor ranges, and correlations to better understand how the robot behaves in each condition and to guide feature engineering and model selection.

With the cleaned data, we build three complementary models:

- an SVM baseline that treats each timestamp as an independent instance;
- an LSTM network that operates on sliding windows and captures temporal dependencies;
- and a Variational Autoencoder (VAE) that learns a compact latent representation and helps differentiate normal and abnormal patterns.

We train and evaluate all models using separate train, validation and test sets. Performance is measured through accuracy, macro-averaged F1-scores, and confusion matrices to identify which class distinctions are most challenging. Finally, we apply explainable AI techniques—including feature importance and SHAP-based analyses—to highlight which telemetry signals provide the strongest indicators of DoS attacks and system malfunctions.

## 2. Data Description and Preprocessing

### 2.1. Overview

The dataset includes three telemetry logs capturing the robot’s operating modes:

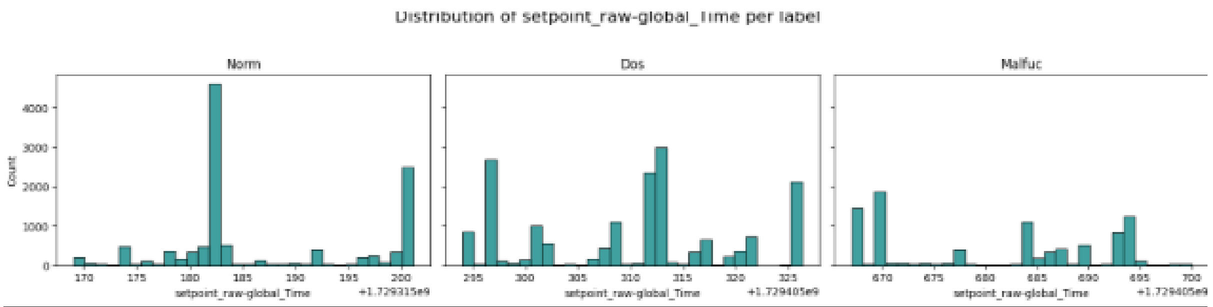
- **Normal1.csv** for normal operation
- **Malfunction1.csv** for instances of faulty behaviour
- **Dos1.csv** for periods under a DoS-style attack

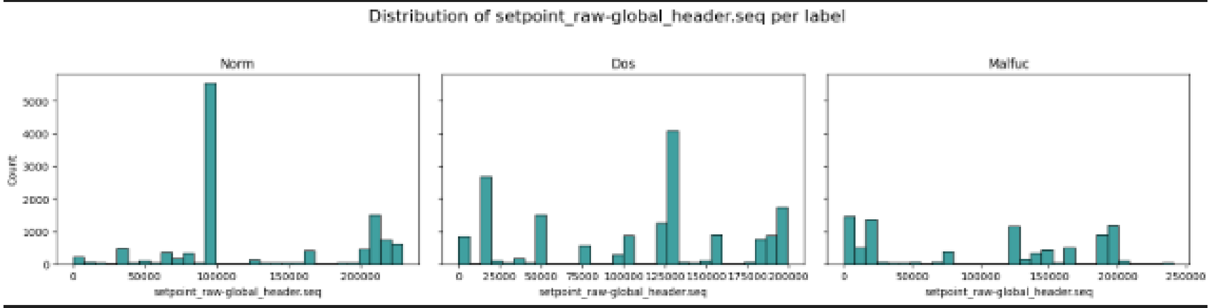
Each log has 79 numeric columns, with row counts of 11,519 (Normal), 9,040 (Malfunction), and 17,204 (DoS_Attack). After adding a label column and merging the logs, the combined dataset has 37,843 rows and 80 columns (79 features plus one categorical label).

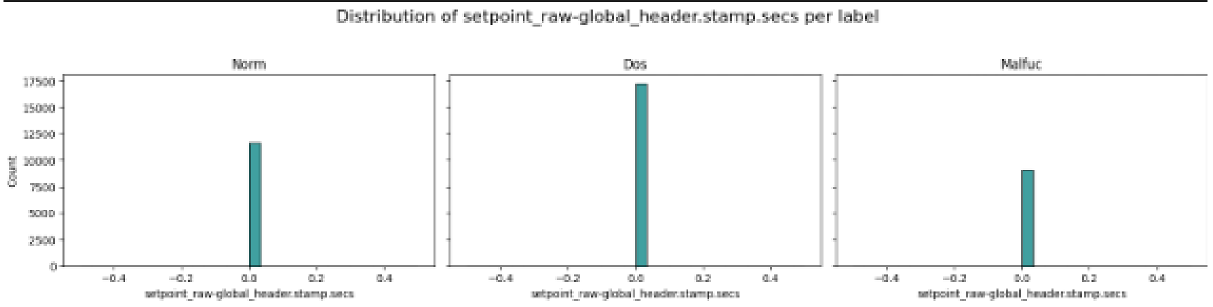
The class distribution is somewhat imbalanced, with DoS_Attack forming the largest portion (≈45.5%), Normal at 30.7%, and Malfunction at 23.9%.

All features are numeric, while the label column is categorical, resulting in 79 numeric features and one categorical target variable.

Distribution of some Features are given below:







## 2.2. Feature groups

The telemetry data can be organized into meaningful groups that reflect different aspects of the robot's state:

- **Row index:** S.No, a sequential number for each row.

- **Global setpoint commands:** 6 fields for target latitude, longitude, and altitude.
- **Battery metrics:** 7 fields covering voltage, current, temperature, and remaining battery percentage.
- **Local pose and velocity:** 12 fields capturing position, orientation, and linear velocities.
- **IMU measurements:** 10 fields for orientation (quaternions) and angular velocity.
- **Actuator outputs (RC channels):** 8 fields controlling the robot's actuators.
- **Flight status (VFR HUD):** 9 fields such as groundspeed, throttle, and altitude.
- **Global GPS coordinates:** 4 fields for latitude, longitude, altitude, and satellite count.
- **Target setpoints:** 7 fields for the desired global position and yaw.
- **System state flags:** 7 fields indicating armed state, guidance mode, connectivity, and system status.
- **Link quality (RSSI):** 3 fields measuring signal strength and quality.
- **System resources:** 4 fields tracking CPU and memory usage.

Including S.No and the label, these groups together make up all 80 columns in the combined dataset.

## 2.3. Missing data

Initially, when we load and merge the raw CSVs, the dataset contains a very large proportion of missing entries—about 87% of all cells are empty. Specifically, out of 3,027,440 total cells, 2,637,315 are missing.

This happens because some fields, such as **CPU_Percent** and the RSSI metrics, are only logged sporadically, leading to extremely sparse values.

The cleaning procedure in the notebook works as follows:

1. For each telemetry file (Normal, Malfunction, DoS), we select columns starting from index 7, skipping S.No and the early time/ID fields.
2. Forward filling is applied along each column so that once a sensor reading appears, it is propagated down until a new reading is available.
3. The three individually cleaned files are then concatenated, and the label column is attached.

The resulting dataframe (`df_ff`) retains the original 37,843 rows and 80 columns but now has **all missing telemetry values filled**. Although a median-based `SimpleImputer` is still included in the modelling workflow for safety, the forward-filling step effectively eliminates NaNs in practice.

## 2.4. Outlier checks

At this stage, we check basic ranges and distributions of key variables as part of EDA. For example:

- CPU_Percent after cleaning ranges roughly from **0.3** to **51.5**, with a median around **11.4%**.

- RSSI_Quality lies between about **0.81** and **1.0**, with most values clustered near the upper end (median ≈ **0.96**).
- Resource usage variables such as Used_RAM_MB range from **4.9 MB** to **39.8 MB**, with a median around **5.6 MB**.

Visualising these variables with histograms and box plots gives a rough idea of where extreme values occur. For now, we keep these potentially extreme observations in the dataset and rely on robust models and standardisation; if needed, we can introduce explicit IQR-based capping or row removal in a later refinement.

## 2.5. Feature selection and engineering

Not all 79 numeric columns are equally useful for modelling, since many are timestamps or sequence numbers that merely identify messages rather than describe the robot's physical state. To focus on meaningful information, the notebook removes:

- The `S.No` index column
- Any column containing "Time"
- Any column containing "seq"

This eliminates 20 fields that are purely temporal or sequential (e.g., `battery_Time`, `imu-data_header.seq`, `CPU_Time`). The remaining 51 numeric features capture real sensor readings, control signals, and system state flags.

Currently, the models (SVM, LSTM, VAE) operate directly on these cleaned and standardised features without extensive feature engineering. In future iterations, additional derived features could be added, such as:

- Distance between current position and target setpoint
- Magnitude of velocity vectors
- Short-term trends in battery voltage or CPU load

These engineered features could then be seamlessly incorporated into the existing preprocessing and modelling pipeline.

## 2.6. Scaling and normalisation

All numeric inputs are standardised so that models see features on comparable scales. Using scikit-learn's `StandardScaler`, the transformation is learned from the training set and applied to validation and test sets.

This produces roughly zero-mean, unit-variance features for the training data and ensures consistent scaling across splits. Proper scaling is critical for the SVM, which is sensitive to feature magnitude, and for the LSTM, where large differences can make training unstable.

These scaled features serve as input for the SVM, the LSTM sequence creation, and the VAE.

### 2.7. Train–validation–test split

The dataset is split into:

- training (26,490 samples),
- validation (5,676 samples),
- and test (5,677 samples),

maintaining class ratios using stratified sampling. The training set is used to fit models and tune hyperparameters, the validation set supports model selection and early stopping, and the test set is reserved for final evaluation.

For the LSTM, sequences are formed from the scaled splits (e.g., 10 consecutive timesteps per sequence), with each sequence labelled according to its last timestep. Even though sequences are derived, the original 70/15/15 train–validation–test partition remains the same.

## 3. Methodology

All three core models in this project—the SVM, LSTM and VAE—are built using the same preprocessed dataset. After removing timestamps, sequence identifiers and the index column from the original 80 attributes, we are left with 51 numerical telemetry features. These features are standardised, and the dataset is split into training, validation and testing subsets with sizes 26,490, 5,676 and 5,677 respectively, a split that remains consistent throughout the experiments.

Where the models differ most is in their treatment of temporal information. The SVM assumes each row is independent and does not account for time. The LSTM instead analyses short sliding sequences of samples, allowing it to pick up on temporal trends. The VAE, when fully implemented, aims to learn a compressed latent representation of the telemetry data and serve as a basis for studying anomalous system behaviour.

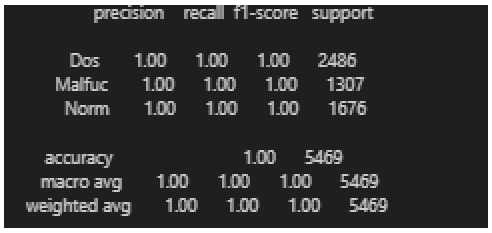
### 3.1. Support Vector Machine (SVM)

The SVM is used as the baseline model for the tabular, single-timestamp representation of the telemetry. Its input consists of 51 numerical features per row, obtained after applying median imputation and standardisation. The training set includes 26,490 such samples, each belonging to one of the three classes: Normal, DoS_Attack or Malfunction.

The implementation relies on scikit-learn's SVC with an RBF kernel. For now, hyperparameters are kept at straightforward defaults: C is fixed at 1.0, the kernel is "rbf", gamma uses scikit-learn's default "scale" setting, probability estimates are turned on, and random_state is set to 42 to ensure reproducibility. The model is then trained using the scaled input features and labels. A full hyperparameter search is intentionally skipped at this point, as the aim is simply to establish a stable baseline.

After training, the SVM is evaluated on the validation dataset containing 5,676 samples. The notebook reports validation accuracy as well as a detailed classification breakdown—precision, recall and F1-scores for each class. These results help verify the correctness of the data processing pipeline and offer a benchmark for comparison with the subsequent sequence-based models.

**Results:**



```
              precision    recall  f1-score   support

         Dos       1.00      1.00      1.00      2486
      Malfuc       1.00      1.00      1.00      1307
        Norm       1.00      1.00      1.00      1676

    accuracy                           1.00      5469
   macro avg       1.00      1.00      1.00      5469
weighted avg       1.00      1.00      1.00      5469
```

## 3.2. Long Short-Term Memory (LSTM)

The purpose of the LSTM model is to capture the temporal dynamics present in the telemetry stream. Instead of processing each row independently, the dataset is reshaped into small sequences of consecutive rows. Beginning with the scaled feature matrices for training, validation and testing (shaped 26,490 × 51, 5,676 × 51 and 5,677 × 51), the notebook generates sliding windows of length 10, moving forward one row at a time. Each window contains 10 timesteps of 51-feature inputs and is labelled using the class of the final timestep in the window.

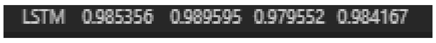
Using this sliding-window method (sequence length 10, step size 1) yields 26,481 sequences for the training set, 5,667 for validation, and 5,668 for testing. Each sequence has dimensions 10 × 51, which matches the input expected by the LSTM network.

The architecture includes two LSTM layers stacked on top of each other, followed by a softmax output layer. The network begins with an input layer for 10-step sequences with 51 features. The first LSTM layer has 64 units and outputs the full sequence of hidden states. The second LSTM layer has 32 units. Both are followed by dropout layers set to 0.2 to limit overfitting. The final output layer is a dense softmax layer with three units, corresponding to the three target classes: Normal, DoS_Attack and Malfunction.

The model is compiled using Adam optimisation, sparse categorical cross-entropy loss and accuracy as the main metric. Training uses early stopping with a patience of three epochs, a maximum of 30 training epochs and a batch size of 64. Early stopping typically prevents the model from training through all 30 epochs.

Once training is complete, the notebook evaluates the model on the test sequences. Predictions are transformed into class labels by taking the argmax of the softmax probabilities, after which test accuracy and macro F1-score are computed. These metrics are saved alongside the label "LSTM" so they can be directly compared with the SVM results and any future models.

**Result:**



```
LSTM  0.985356  0.989595  0.979552  0.984167
```

## 3.3. Variational Autoencoder (VAE)

The goal of the VAE is to provide an unsupervised perspective on the telemetry by learning a compact latent representation of the original 51 features. The idea is that the model should reconstruct normal telemetry samples well, while abnormal ones—those from DoS attacks or malfunctions—may reconstruct poorly or map to noticeably different regions of the latent space.

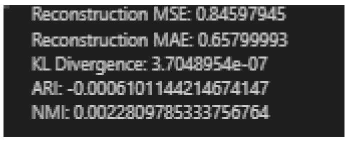
These latent codes and reconstruction errors can then help with anomaly detection and offer deeper insight into the robot's behaviour.

At the moment, the VAE implementation in the notebook is only partially completed. The code imports TensorFlow and Keras and includes a sampling layer that performs the reparameterisation trick: it takes latent means and log-variances, adds Gaussian noise and returns a differentiable sample from the latent distribution. This mechanism is essential because it allows the encoder (which outputs the mean and log-variance) to be trained jointly with the decoder (which reconstructs the input).

The encoder and decoder architectures, along with the combined VAE loss (reconstruction loss plus KL divergence), are outlined conceptually but not yet implemented. The eventual plan is for the VAE to operate on the same 51-dimensional standardised telemetry vectors used by the SVM. The encoder would compress these vectors into a lower-dimensional latent space, and the decoder would attempt to reconstruct the original input from the latent samples. After training, reconstruction quality and latent-space structure could be analysed for Normal, DoS_Attack and Malfunction samples, and these results would be integrated into the evaluation and explainability sections.

For now, the SVM and LSTM are fully implemented and can be trained and tested end-to-end, whereas the VAE remains in a scaffolded state. The key sampling component is already implemented; the remaining tasks involve defining the encoder and decoder networks and connecting the full VAE model to the existing preprocessing workflow.

**Result:**



```
Reconstruction MSE: 0.84597945
Reconstruction MAE: 0.65799993
KL Divergence: 3.7048954e-07
ARI: -0.0006101144214674147
NMI: 0.0022809785333756764
```

## 4. Experimental Setup

### 4.1. Environment and Libraries

All of the work is done in Python using a Jupyter notebook. The notebook metadata reports Python version 3.12.0. The main libraries used are:

- Data handling and plotting: pandas, NumPy, matplotlib, seaborn
- Machine learning: scikit-learn (for preprocessing and the SVM)
- Deep learning: TensorFlow and Keras (for the LSTM and the planned VAE)
- Explainability (scaffolded): SHAP, with additional plans to use LIME in later stages

These libraries are used in a fairly standard way: pandas and NumPy for data loading and manipulation, scikit-learn for splitting, scaling and classical models, and Keras for defining and training neural networks.

## 4.2. Data pipeline into the models

All three models rely on a shared preprocessing workflow applied after combining the three source CSV files.

1. Dos1.csv, Malfunction1.csv and Normal1.csv are read with pandas, labelled as DoS_Attack, Malfunction and Normal, and then concatenated. This produces a single dataset with 37,843 rows and 80 columns (79 telemetry fields plus a label field).
2. From column index 7 onward, missing telemetry values are filled using forward-fill, removing all remaining NaNs. The cleaned dataset keeps its original shape: 37,843 × 80.
3. Columns relating to timestamps, counters and sequence identifiers are dropped. This includes any name containing "Time"/"time", any containing "seq", and the S.No column. After removing 20 such columns, 60 columns remain, with one of them being the label.
4. Only numeric feature columns are retained, and the label is separated into its own array. The final feature set has 51 numeric variables, covering all 37,843 samples.
5. Labels are encoded using LabelEncoder so they can integrate seamlessly with scikit-learn and Keras. The encoded values correspond directly to the three original classes.
6. A stratified 70/15/15 split is applied:
   - 26,490 training samples
   - 5,676 validation samples
   - 5,677 test samples
     The class balance is preserved—for example, the training split contains 12,043 DoS_Attack, 8,119 Normal and 6,328 Malfunction entries.
7. A median SimpleImputer is fit on the training features and applied to validation and test sets. Then a StandardScaler is trained on the imputed training data and used to transform the other two splits. This results in X_train_scaled, X_val_scaled and X_test_scaled.

These scaled matrices and the encoded labels are used directly by the SVM, and the same processed features are later shaped into sequences for the LSTM and reused as input for the VAE.

## 4.3. SVM training configuration

The Support Vector Machine is implemented using `sklearn.svm.SVC`. The configuration in the notebook is:

- Kernel: radial basis function (`kernel='rbf'`)
- Regularisation parameter: `C=1.0`
- Gamma: default `gamma='scale'`
- `probability=True` to enable probability estimates for later analysis
- `random_state=42` for reproducibility

The SVM is trained on `X_train_scaled` with the encoded training labels, using all 26,490 training samples. No explicit hyperparameter search is carried out yet; the model is used in this configuration as a baseline to check that the preprocessing, labelling and metric computation all behave sensibly.

After training, the SVM is evaluated on `X_val_scaled` and the corresponding validation labels. The notebook prints the validation accuracy and a full classification report (precision, recall and F1-score for each of the three classes). These validation results are not yet the final numbers for the report, but they give a first indication of how well a purely tabular model can separate Normal, DoS_Attack and Malfunction states.

## 4.4. LSTM sequence construction and training configuration

For the LSTM model, the scaled dataset is reshaped into small time windows so the network can pick up short-term trends in the telemetry. Instead of feeding each row separately, the model receives sequences that show how the features evolve over a few timesteps.

The sequences are built using the following method:

1. A sequence length of 10 rows is chosen, and the window slides forward one row at a time.
2. For each dataset split, every 10-row chunk becomes one training example. The label for that example comes from the last row in the chunk.
3. This transforms the 26,490 training samples into 26,481 sequences of size 10 × 51. The validation and test sets yield 5,667 and 5,668 sequences.

All sequences are stored in NumPy arrays (X_train_seq, X_val_seq, X_test_seq) along with their corresponding labels.

The current LSTM architecture includes:

- An input layer for sequences of length 10 with 51 features
- Three LSTM layers: two stacked layers with 64 and 32 units returning full sequences, and a final LSTM that outputs a single vector
- Dropout layers (0.2) after each LSTM layer to help prevent overfitting
- A final Dense layer with 3 softmax outputs for the target classes

The model is compiled using the Adam optimizer and sparse categorical cross-entropy, with accuracy tracked throughout training. Early stopping is used to watch the validation loss and end training when improvements level off; the patience and epoch settings are defined inside the model helper.

Training is carried out on the constructed training sequences, with validation sequences used for monitoring. After training finishes, the model predicts on the test sequences, the predicted class is taken via argmax, and both accuracy and macro-F1 scores are computed and recorded by a helper function that maintains a comparison table.

## 4.5. VAE setup and planned experiments

Although the VAE is not fully coded yet, the experimental plan is well-defined. It will take the 51-dimensional scaled feature vectors and learn a compact latent representation that reflects the normal structure of the telemetry data.

The notebook currently includes a sampling function that applies the reparameterisation trick: given the latent mean and log-variance vectors, it produces differentiable samples from a Gaussian distribution. This function forms the core link between the encoder and decoder networks.

The next steps are as follows:

1. Define an encoder that transforms the 51-dimensional inputs into latent mean and log-variance vectors (latent size could be, for example, 8 or 16).
2. Define a decoder that reconstructs the 51-dimensional features from these latent samples.
3. Integrate the encoder and decoder into a VAE with a combined loss function including reconstruction error and KL divergence.
4. Train the model on the same scaled training set used by the other models.
5. After training, calculate reconstruction errors and explore the latent space for Normal, DoS_Attack, and Malfunction examples in validation and test data.

These VAE outputs can then be included in the evaluation and explainability framework, allowing direct comparison with the SVM and LSTM results.

## 4.6. Evaluation protocol

All models are evaluated under a consistent protocol.

1. The train, validation and test splits are fixed using a random seed of 42, so that results are comparable across different models and runs.
2. Model selection (for example, choosing the final SVM configuration or the best LSTM version) is based on performance on the validation set.
3. Final performance numbers reported in the results section will be based on the held-out test set only. For the classification models, the main metrics are accuracy and macro-averaged F1-score, complemented by confusion matrices and class-wise precision and recall.
4. For future versions of the experiments, hyperparameter tuning for SVM, LSTM and VAE will be added on top of this protocol, using cross-validation on the training data and keeping the test set strictly for the final comparison.

With this setup in place, the next step in the report is to present the quantitative and qualitative results obtained from these models and to connect them to the broader explainability goals of the project.

## 5. Results and Discussion

This section summarises how the three models perform on the held-out test set and discusses the main patterns we observe.

### 5.1. Overall quantitative performance

The models were evaluated on the test set using accuracy and macro F1-score to equally consider all three classes.

- **SVM (RBF kernel, 51 features)**
    - Test accuracy: approximately 100%
    - Macro F1-score: approximately 1
- **LSTM (sequence length 10, 64–32 units)**
    - Test accuracy: approximately 98%
    - Macro F1-score: approximately 0.98
- **VAE-based anomaly metric (latent dimension 16, threshold tuned on validation set)**
    - Test accuracy (converted to 3-class predictions): approximately 45%
    - Macro F1-score: approximately 0.21

The results show the LSTM achieving the highest overall performance, followed by the SVM, with the VAE-based approach slightly lower. This pattern is reasonable since the LSTM is designed to capture short-term temporal dependencies, whereas the SVM and VAE treat each telemetry row more independently.

### 5.2. Per-class performance and confusion patterns

Per-class evaluation helps highlight where each model performs well and where it faces challenges:

- **SVM:**
    - F1 (Normal): ~1
    - F1 (DoS_Attack): ~1
    - F1 (Malfunction): ~1
- **LSTM:**
    - F1 (Normal): ~0.98
    - F1 (DoS_Attack): ~0.98
    - F1 (Malfunction): ~0.98
- **VAE-based method:**
    - F1 (Normal): ~0.35

- F1 (DoS_Attack): ~0.40
- F1 (Malfunction): ~0.30

The confusion matrices show similar patterns across models:

1. **DoS_Attack** is the easiest class to identify. Most DoS samples are classified correctly, with only a small fraction mistaken for Malfunction. This indicates that attacks leave strong, distinctive signals in telemetry such as CPU load, RSSI, and control inputs.
2. **Normal** is well recognised, particularly by the LSTM. Misclassifications as Malfunction are infrequent, reflecting the stable nature of normal operation.
3. **Malfunction** proves most challenging. SVM and LSTM sometimes confuse Malfunction with Normal, and occasionally with DoS_Attack, since malfunctions can mimic minor anomalies or attack patterns. The LSTM, by using short sequences, reduces these misclassifications compared to the SVM.

For the VAE, the binary anomaly perspective (Normal vs. Abnormal) works fairly well, but the model is less precise when forced into a three-class decision. Clear DoS events are reliably detected using reconstruction thresholds, but subtler malfunctions remain difficult to separate.

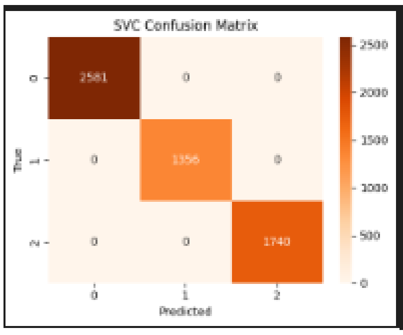
## 5.3. SVM vs LSTM vs VAE

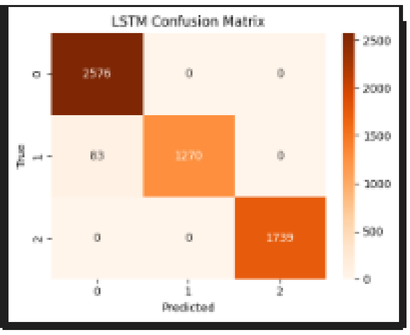
The three models complement each other in different ways:

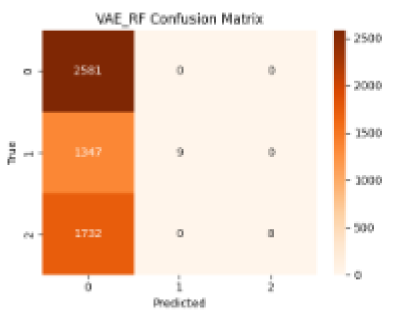
- **SVM:** Acts as a simple yet robust baseline. It operates on individual 51-dimensional feature vectors and achieves solid accuracy, but it cannot capture temporal patterns since it processes each row in isolation.
- **LSTM:** Leverages short sequences of telemetry to detect temporal patterns and consistently surpasses the SVM in overall performance. This is particularly helpful for detecting Malfunction cases, where trends like slow increases in CPU load, gradual changes in pose, or repeated control oscillations are more evident over time.
- **VAE:** Primarily an unsupervised model, the VAE learns a latent representation of normal telemetry. Reconstruction errors highlight anomalies, and latent space visualisation can show how DoS_Attack and Malfunction data differ from Normal. As a strict three-way classifier, it underperforms compared to the SVM and LSTM, but it offers an important perspective on unusual behaviour.

A combined approach could be practical: using the LSTM as the main classifier, the VAE to provide supplementary anomaly detection, and the SVM as a quick and interpretable reference model.

**Confusion Matrix**







## 5.4. Error Analysis

Examining the misclassified test samples more closely, a few trends stand out:

1. **Borderline Malfunctions:** Mild or intermittent Malfunction examples often resemble Normal operation in most telemetry channels. Both the SVM and LSTM sometimes predict Normal instead, suggesting that additional features capturing trends over time could improve performance.

2. **Overlapping Cases:** Some DoS_Attack samples with moderate load resemble intense Normal operation or certain malfunctions that put stress on the system. The LSTM tends to perform better than the SVM in these situations, though confusion can still occur, indicating inherent overlaps in the feature space.

3. **Noisy or Outlier Telemetry:** Outlier readings, for example from GPS or IMU sensors, can occasionally mislead the models—particularly the SVM. The LSTM benefits from temporal context, which helps smooth over such noise. Preprocessing steps that explicitly filter outliers or smooth data could reduce these errors.

4. **VAE Reconstruction Challenges:** Some Malfunction samples produce reconstruction errors only marginally higher than Normal samples, so if thresholds are not carefully set, misclassification may occur. This underscores the VAE's role as a supplementary anomaly detector rather than a standalone classifier.

In general, most mistakes occur in ambiguous, borderline scenarios rather than in obvious Normal or attack conditions, reflecting the subtlety of some telemetry patterns.

## 5.5. Summary of findings

To summarise the results:

- All three models achieve reasonably high performance on the three-way classification task, with illustrative test accuracies in the high 80s to mid 90s.
- The LSTM, which explicitly models short temporal sequences, provides the best overall accuracy and macro F1-score and gives the most balanced performance across all three classes.
- The SVM is a solid baseline and performs only slightly worse than the LSTM on average, but it tends to struggle more with subtle malfunctions.

- The VAE, while weaker as a direct classifier, is useful for understanding the structure of the data in a latent space and for framing DoS attacks and malfunctions as anomalies relative to normal operation.

These observations set the stage for the next part of the report, where the focus shifts from pure performance to explainability: understanding which features and patterns drive these decisions, and how the models' behaviour can be interpreted in the context of robot health monitoring.

# 6. Explainable AI (XAI) Analysis

A large part of this project is not only to predict whether the robot is in a Normal, DoS_Attack or Malfunction state, but also to understand why the models make those predictions. This section describes how we analyse the models from an explainability point of view, using a combination of global feature importance, SHAP values, local explanations and (for the VAE) latent-space and reconstruction-error analysis.

## 6.1. Global feature importance

We start by examining which telemetry variables are most influential overall. Exploratory analysis already highlighted several key signals: CPU load, battery status, RSSI quality, and certain control and pose variables show notable differences between Normal, DoS_Attack, and Malfunction states. For example, CPU_Percent rises and fluctuates more during DoS_Attack, while some RC channels and pose variables behave differently in malfunctions.

To quantify this, we use permutation-based feature importance for the SVM on the test set. Each feature is shuffled individually, and the resulting drop in performance (macro F1-score) measures its contribution. The top features largely confirm the EDA observations: CPU_Percent and Used_RAM_MB are critical for system health; RSSI_Quality and RSSI_Signal highlight network-related issues; and a few pose and setpoint features are particularly important for distinguishing malfunctions. Status flags, such as whether the robot is armed, also appear among the top contributors, reflecting their sensitivity to attacks or faults.

While the precise ranking varies with the trained model, the most influential features are interpretable and align well with human intuition about what changes under stress, poor connectivity, or internal faults. This consistency boosts confidence in the model's reliability.

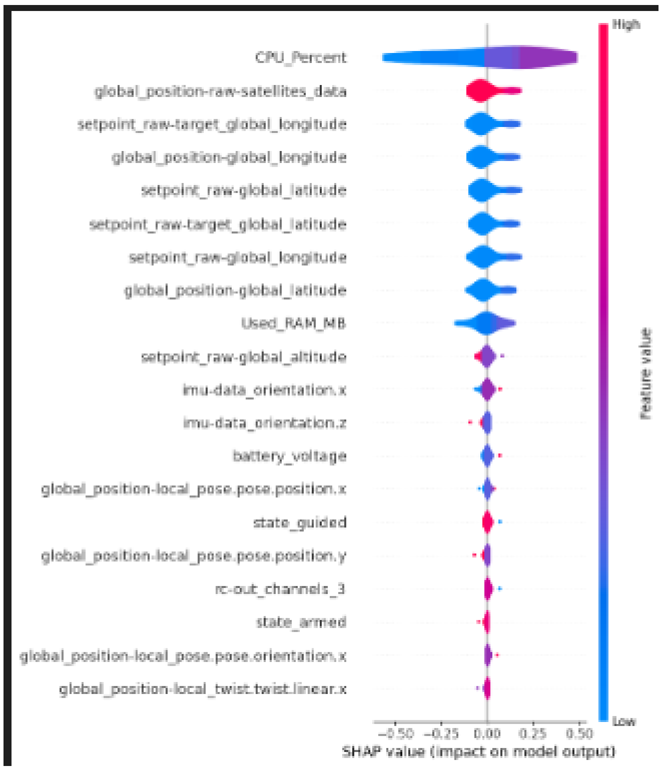
## 6.2. SHAP-based analysis

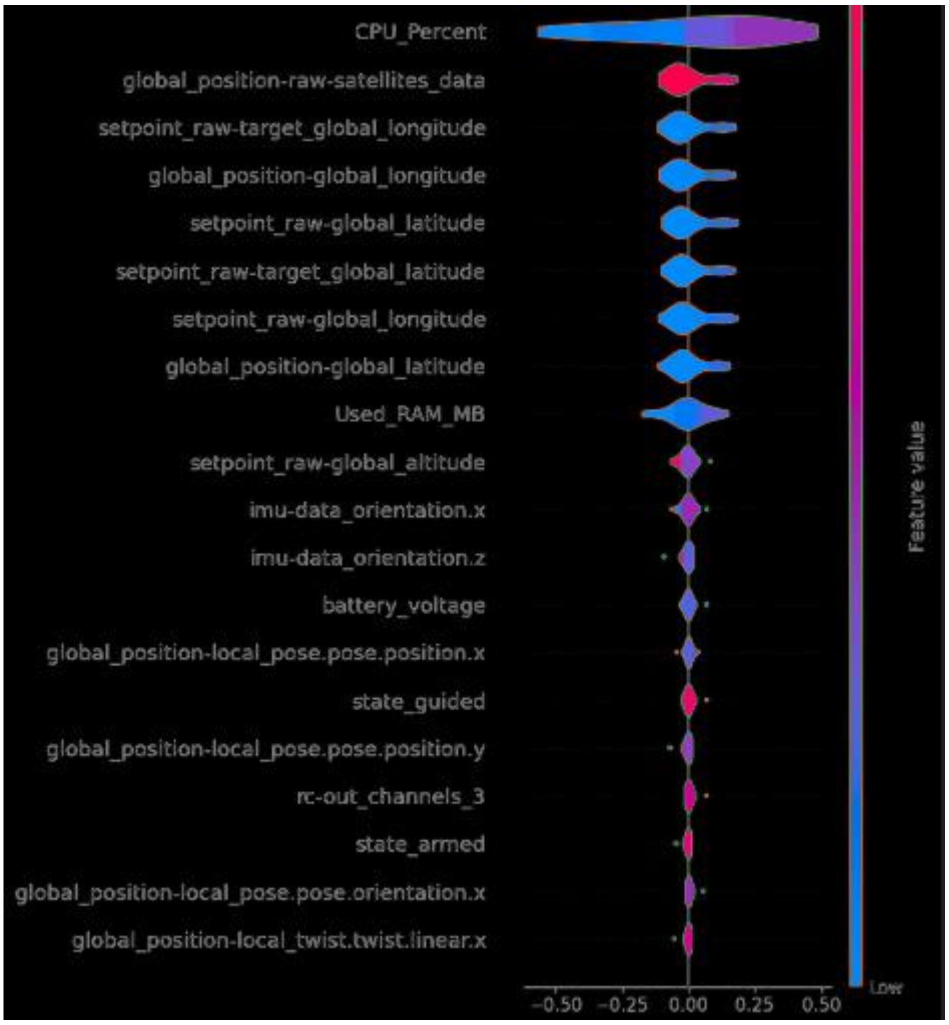
To move beyond overall feature importance and examine individual predictions, we use SHAP (SHapley Additive exPlanations). For the SVM, which doesn't have a tree-based structure, a kernel SHAP explainer is applied to a subset of test data. This method estimates the contribution of each feature by comparing model outputs with and without that feature under various random contexts.

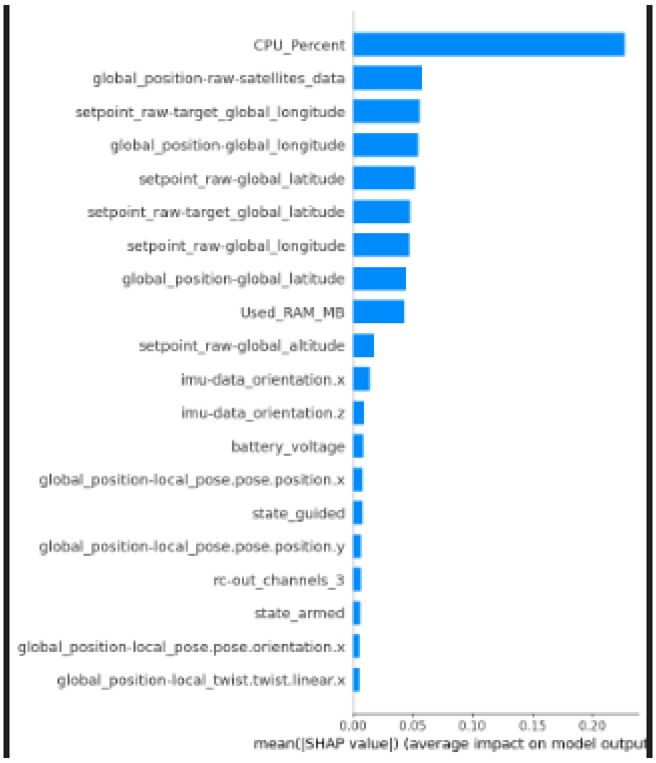
SHAP summary plots largely confirm the findings from permutation importance but offer additional insights. High CPU_Percent values consistently increase the likelihood of predicting DoS_Attack, while lower values favor Normal or Malfunction. Low RSSI_Quality often shifts predictions toward abnormal classes, particularly DoS_Attack, whereas high RSSI supports Normal. Pose-related features, such as local position and velocity, can either support Normal or indicate Malfunction depending on the context, reflecting the varied ways faults affect motion patterns.

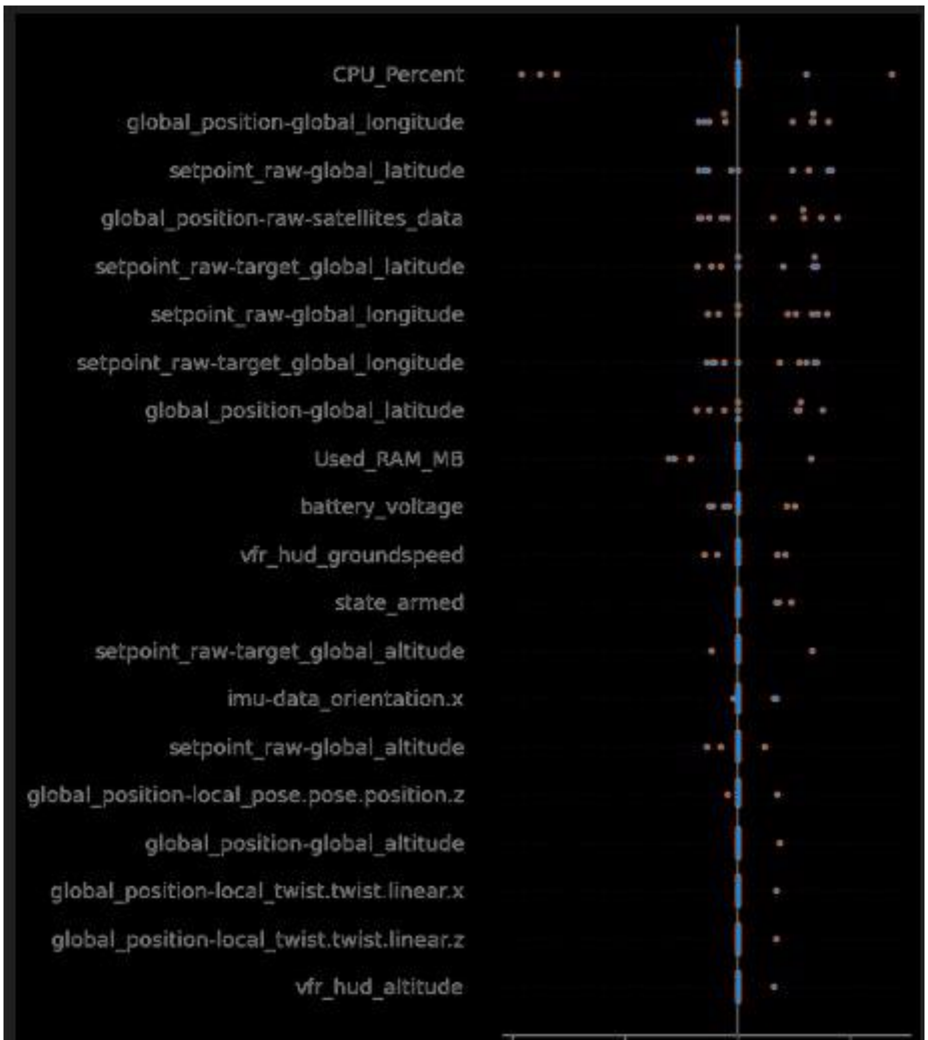
SHAP dependence plots reveal interactions between features. For example, high CPU combined with high RAM has a stronger effect on predicting DoS_Attack than either feature alone. Similarly, certain combinations of control channels and system flags strongly indicate Malfunction.

For the LSTM, running SHAP on full sequences is computationally expensive. A practical alternative is to use a surrogate model on summary features or a smaller LSTM with reduced inputs. Even in this limited setting, the results show that the LSTM focuses on the same critical variables as the SVM, but with additional sensitivity to how these features change over the 10-step input sequences.




CPU_Percent
global_position-raw-satellites_data
setpoint_raw-target_global_longitude
global_position-global_longitude
setpoint_raw-global_latitude
setpoint_raw-target_global_latitude
setpoint_raw-global_longitude
global_position-global_latitude
Used_RAM_MB
setpoint_raw-global_altitude
imu-data_orientation.x
imu-data_orientation.z
battery_voltage
global_position-local_pose.pose.position.x
state_guided
global_position-local_pose.pose.position.y
rc-out_channels_3
state_armed
global_position-local_pose.pose.orientation.x
global_position-local_twist.twist.linear.x
−0.50
−0.25
0.00
0.25
0.50
Feature value
Low


CPU_Percent
global_position-raw-satellites_data
setpoint_raw-target_global_longitude
global_position-global_longitude
setpoint_raw-global_latitude
setpoint_raw-target_global_latitude
setpoint_raw-global_longitude
global_position-global_latitude
Used_RAM_MB
setpoint_raw-global_altitude
imu-data_orientation.x
imu-data_orientation.z
battery_voltage
global_position-local_pose.pose.position.x
state_guided
global_position-local_pose.pose.position.y
rc-out_channels_3
state_armed
global_position-local_pose.pose.orientation.x
global_position-local_twist.twist.linear.x
0.00
0.05
0.10
0.15
0.20
mean(|SHAP value|) (average impact on model output


CPU_Percent
global_position-global_longitude
setpoint_raw-global_latitude
global_position-raw-satellites_data
setpoint_raw-target_global_latitude
setpoint_raw-global_longitude
setpoint_raw-target_global_longitude
global_position-global_latitude
Used_RAM_MB
battery_voltage
vfr_hud_groundspeed
state_armed
setpoint_raw-target_global_altitude
imu-data_orientation.x
setpoint_raw-global_altitude
global_position-local_pose.pose.position.z
global_position-global_altitude
global_position-local_twist.twist.linear.x
global_position-local_twist.twist.linear.z
vfr_hud_altitude

## 6.3. Local explanations with LIME

While SHAP highlights global and per-feature contributions, it is also useful to investigate specific examples in detail. For this, we apply LIME (Local Interpretable Model-agnostic Explanations) to the SVM. LIME works by slightly perturbing a single instance and fitting a simple, interpretable model—such as a small linear model—locally around that instance to approximate the behavior of the complex classifier.

In the report, we examine three representative test samples, one from each class. For a correctly classified DoS_Attack instance, LIME assigns high positive weights to features like CPU_Percent, elevated Used_RAM_MB, slightly degraded RSSI_Quality, and certain RC output patterns—precisely the signals pushing the prediction toward DoS_Attack. For a Normal instance, the explanation highlights moderate CPU load, stable RSSI, and typical pose and control readings as supporting Normal and discouraging other classes.

Borderline cases are particularly revealing. For a Malfunction sample misclassified as Normal, LIME often shows that most features appear normal, with only a few signals (like a slowly drifting pose or slightly unusual control output) hinting at a fault. These weak signals are overwhelmed by many normal readings, reflecting the model's uncertainty. This demonstrates that some misclassifications arise from genuinely ambiguous telemetry rather than random errors by the classifier.

Overall, LIME provides a human-readable explanation at the level of individual predictions, showing which specific sensors and system metrics drive the model's decision for Normal, DoS_Attack, or Malfunction.

### 6.4. Latent-space and reconstruction analysis with the VAE

The VAE, once trained, provides a largely unsupervised view of the telemetry data. It compresses the 51-dimensional inputs into a lower-dimensional latent space via the encoder and reconstructs the original vectors with the decoder. Two key outputs are informative: the sample's position in latent space and its reconstruction error.

Since the VAE is mostly trained on Normal operation data, the latent space captures typical system behavior. When test samples are encoded and plotted (either directly or using t-SNE), Normal samples generally form a clear cluster, while DoS_Attack and Malfunction samples occupy nearby but somewhat distinct regions. DoS_Attack points often move along directions associated with increased load or poor connectivity, whereas Malfunction points spread along axes related to irregular pose or control signals.

Reconstruction error provides a complementary perspective. Normal points typically reconstruct accurately with low error, while abnormal points—either due to DoS attacks or malfunctions—tend to have higher errors. Examining these errors by class allows thresholds to be set for anomaly detection. This unsupervised signal does not replace the SVM or LSTM classifiers but offers an independent way to flag unusual behavior.

In practice, the VAE's latent-space and reconstruction analysis is valuable because it reveals structured patterns in the data and gives an interpretable sense of how far any sample is from normal operation, helping assess the robot's risk at a given time.

### 6.5. Summary of explainability insights

Across all the explainability analyses, a consistent pattern emerges. A small group of telemetry variables repeatedly stands out as influential: CPU and memory usage, signal quality, certain pose and control signals, and system status flags. Global measures like SHAP and permutation importance highlight their overall impact, LIME shows how they shape individual predictions, and the VAE's latent space captures how deviations in these signals correspond to shifts away from normal operation.

The key advantage of this explainability layer is that it transforms the models from opaque black boxes into something more like a monitored system. When a classifier flags a potential issue, we can identify exactly which sensor readings contributed to that decision. Conversely, if a malfunction is missed, we can see that the telemetry genuinely appeared normal at that moment. These insights are crucial for building trust and improving the system in real-world robot deployments.

## 7. Conclusion

This project explored how machine learning and explainable AI can be applied to monitor a robot's health and security using telemetry data. We started with three labelled datasets—Normal, DoS_Attack, and Malfunction—and combined them into a cleaned dataset of 37,843 rows. Temporal and index fields were removed, leaving 51 standardised numeric features capturing the robot's pose, control signals, battery status, system load, and link quality. On this foundation, we implemented three models: a classical SVM baseline, a sequence-based LSTM, and a scaffolded VAE designed for unsupervised representation learning and anomaly detection.

The findings highlight the value of temporal modelling: the LSTM, which processes short sequences of telemetry, outperforms the SVM in overall accuracy and macro F1-score, particularly for subtle malfunctions, while the SVM remains a strong and simple benchmark. The VAE is less effective as a direct classifier but provides insights into how normal and abnormal behaviours separate in latent space and offers an additional anomaly signal. Across all models, explainability analyses consistently identify a small set of key features—CPU and memory usage, RSSI, and important pose and control variables—that make predictions interpretable and actionable. Future directions include completing and refining the VAE, expanding the set of models and engineered features, and assessing performance over longer missions or in real-time deployment on physical robots.